A History of Software Measurement at Rome Laboratory

Contract Number F30602-89-C-0082
(Data & Analysis Center for Software)

25 November 1996

PREPARED FOR:

Rome Laboratory
RL/C3C
Griffiss Business Park
Rome, NY 13441

PREPARED BY:

Elaine Fedchak and Robert Vienneau
Data & Analysis Center for Software (DACS)
Griffiss Business and Technology Park
775 Daedalian Drive
Rome, New York 13441-4909
Robert Vienneau - rob@dacs.dtic.mil
Elaine Fedchak - efedchak@dacs.dtic.mil
A History of Software Measurement at Rome Laboratory

Table of Contents

1. INTRODUCTION
   1.1 BACKGROUND
   1.2 MODELS, METRICS AND DATA
   1.3 ROME LABORATORY ORGANIZATION

2.0 BUILDING A MEASUREMENT TECHNOLOGY FOUNDATION
   2.1 MODELS
      2.1.1 Reliability Models
      2.1.2 Reliability Model Validation Studies
      2.1.3 Cost Models
      2.1.4 Complexity Models
   2.2 QUALITY METRICS
   2.3 DATA
      2.3.1 The Data & Analysis Center for Software
      2.3.2 Tools for Data Collection
      2.3.3 Data Collection Studies
   2.4 MODERN PROGRAMMING PRACTICES
      2.4.1 The Structured Programming Series
      2.4.2 MPP Evaluation Studies

3.0 CONTEMPORARY WORK
   3.1 THE RL SOFTWARE QUALITY TECHNOLOGY TRANSFER CONSORTIUM
   3.2 RECENTLY COMPLETED CONTRACTS

4.0 REFERENCES

Appendix A: ACRONYMS
Appendix B: THE FRAMEWORK

LIST OF FIGURES
Figure 1-1: THE EVOLUTION OF ENGINEERING DISCIPLINES
Figure 2-1: ROME LABORATORY SOFTWARE QUALITY FRAMEWORK
1.0 INTRODUCTION

Problems associated with software began to receive wide notice in the late 1960s. Software is often late, over budget, and fails to perform as expected. This set of problems has come to be known as the "software crisis." In 1968, the IEEE sponsored the first International Conference on Software Engineering, at Garmisch, West Germany. It was at this conference that the need for software engineering as an autonomous discipline was first recognized (Booch 87). The discipline that would be imposed on the software development process by engineering software rather than just creating it would lead to solutions for the problems of the software crisis.

These problems and needs were especially acute in the Department of Defense. Even then software was a growing component of defense systems, and farsighted individuals predicted that software would become a driving force in system cost and functionality (Boehm 73a). Furthermore, much DOD software is developed for embedded systems, which raises additional issues of cost and complexity.

Implementation of a disciplined engineering approach to software development, however, requires the existence of an adequate supporting software engineering technology. A key element of that technology is measurement. Measurement technology supporting software engineering did not exist when software engineering was first conceived. Models needed to be created to characterize and describe software usefully. Metrics needed to be identified to define the dimensions along which software could be measured. Data needed to be collected to validate the emerging models and metrics and to demonstrate that they provide useful guidance to the software engineer and manager. A body of empirical data needed to be built which could be used to quantitatively assess the effects of new techniques and methodologies.

This document describes software measurement activities conducted by Rome Laboratory (RL) from the early seventies to the present. This history is based mainly on the written record as reflected in RL Technical Reports, Data & Analysis Center for Software (DACS) reports, and documents in the DACS Software Engineering Bibliographic Database. This report is not intended to be comprehensive. Rather, its purpose is to present highlights of RL involvement in measurement technology development, and in so doing provide a corporate memory for Rome Laboratory in this area.

This document contains an introductory discussion of the relationships between software measurement technology and software engineering, and a brief look at how Rome Laboratory has evolved during this time period. The bulk of the paper presents the RL-sponsored measurement research, beginning with the foundation-building efforts from the early 1970s through the mid-1980s, and concluding with the current industry partnerships approach. The foundations section categorizes the early efforts into model, metric, and data research, and includes the programming practices studies that applied measurement research results to assess process improvement. A complete bibliography of measurement-related RL Technical Reports is included as well.
1.1 BACKGROUND

Today, more than 25 years after it was invented, software engineering remains more a label for a desirable discipline than a widely accepted and successful body of techniques for producing software solutions to practical problems.

Mary Shaw of the Software Engineering Institute (SEI) has proposed a scheme describing how engineering disciplines evolve (Shaw 90), which is diagrammed in Figure 1. In this theory, engineering disciplines evolve out of crafts. During the craft stage, work is performed in an ad hoc manner, and depends on the personal capabilities of each practitioner. There are no recorded standard solutions, although folklore may develop about some practices. The software industry had these characteristics in the 1960s when most programs were small and depended on individual capabilities.

Crafts, however, can only be produced on a small scale. The needs of large scale production lead to a codification of standard procedures into commercial practice. The commercial stage of the software discipline includes "life cycle models, routine methodologies, cost-estimation techniques, documentation frameworks, configuration-management tools, quality assurance techniques, and other techniques for standardizing production activities" (Shaw 90). These practices allow software projects to be managed more or less well, and they can be refined pragmatically. But software professionals typically do not draw on any well-established science for analytical techniques.

The evolution of an adequate software engineering discipline depends on the prior and parallel development of an adequate science. Engineering applies science to practical problems. Some elements of the science needed for software engineering have begun to appear. They include developments such as:

- Demonstration that any algorithm can be expressed by a limited set of control constructs
- Techniques for formally expressing the syntax and semantics of a language
- Formal specification techniques for abstract data types
- A theory of computational complexity for analyzing well defined processes such as sorting and searching

These elements, however, are only the beginning of the science needed to support software engineering. True engineering requires measurement. Development of software measurement technology is, therefore, a necessary step in the evolution of software engineering. Measurements are the means by which science transitions to engineering, and through which scientists communicate with engineers. It is how replication of results and standardization of practices become possible. Engineering design requires trade-offs between competing goals, and needs a feedback mechanism for determining the effects of trade-off decisions. The science accompanying software engineering needs to specify the dimensions along which software can be measured. Resources should be specifiable in quantified terms: How big will the system be? How much will it cost? How long will it take? Software project goals should be specified along many dimensions: reliability, functionality, performance, etc.
Management theory suggests that all development projects are faced with trade-offs among three dimensions of cost, schedule and product. Only two of the three can be maximized for any one effort. If you want it cheap and fast, you have to settle for lower quality, or fewer functions. If you want solid gold with all the bells and whistles, you must pay for it in dollars or development time, or both. But even after software project managers determine what levels of cost, schedule and functionality are appropriate, they need measurement technology to implement their plans and to monitor the process.

![Figure 1-1: THE EVOLUTION OF ENGINEERING DISCIPLINES](image)

With sufficient measurement technology available, a software engineer would be able to design a development process to ensure specified quantifiable goals are met. For example, it would be possible to know what the impact on the reliability of a software product will be when the degree of formality in the development process is varied. It should also be possible to determine whether or not the resulting product meets its pre-established goals.
A History of Software Measurement at Rome Laboratory

1.2 MODELS, METRICS AND DATA

Measurement technology consists of three interrelated technical areas: models, metrics and data. No one of these can advance very far without support from and progress in the others. From the beginning of the age of software engineering, RL has sponsored work in all three areas.

Early software engineering work concentrated on the coding and debugging phases of the software life cycle, because that's where software happened. Stemming from Edsger Dijkstra's criticism of the use of the GOTO statement in 1968 (Dijkstra 68), structured programming was invented as a way to reduce software problems by changing the way it is coded. Structured programming later evolved into modern programming practices (MPP) that address a larger set of development activities. At the same time, there was much interest in increasing the reliability of software because of the enormous time and effort that was being spent on testing and debugging.

Most of the early research in software measurement, and specifically model development was, therefore, in the areas of cost, errors and complexity. Cost is a management concern, and so cost model research has usually had management support. Cost is also measured in terms of effort and productivity, which include elapsed time, or schedule, in their definition. Errors and reliability were fruitful areas to attack because the testing and debugging phases were long, costly, and difficult. Size and complexity became early targets for research when their effect on both cost and testing efficiency were recognized.

A model is an abstract (usually mathematical) representation of the relationships among two or more variable attributes of an entity, e.g., that the cost to develop a piece of software is related to the number of lines of code it contains. A good model will incorporate the influence of all variables that affect the outcome. A useful model will have predictive capabilities given the values of some attributes we know, or can estimate reasonably well, it will determine the future values of other attributes with an acceptable degree of uncertainty.

But models require data, because an essential step in the development of useful models is validation. Validation involves applying the model to a set of historical data, and comparing the model's predicted results to the actual results recorded in the data. Consequently, in order to validate models, such sets of historical data must be available. The catch is, the historical data set must include values for each variable attribute in the model, and they must be consistent with respect to units of measure (or be able to be normalized). But how could the person(s) that recorded the historical data know what variables, and units of measure, etc. these as-yet-undefined models would require?

Metric development research provides the link between the needs of model developers and the needs of data collectors. Metrics define concrete, quantifiable attributes of software. For tangible, three-dimensional objects we have traditional, well-known, and in many cases ancient metrics, along with standard units of measure. To the question of "How big is it?" we can answer 5 feet tall, or 300 pounds, or 1.5 liters, etc. But for something as abstract as software, research is needed to define both what to measure and how to measure it. What is the nature of the data to be collected? What are the units
of measurement, or the measurement interval? How do you describe software in ways that are meaningful to researchers, developers, and managers?

Measures of software size, for example, are gradually becoming standardized. Lines of code is now the most commonly used size metric for software, although person-hours of effort and function points are also useful. Previously, the only available answers to "How big is the software?" were management type responses, such as a $5 million development, or a two-year project. Without sufficient measurement technology to really engineer software, however, even those responses were just estimates, not accurate predictions, and one person's $5 million effort was not necessarily the same size as another's.

Software metric development must also include determination of appropriate units for the attributes to be measured, and operational definitions of how to apply the metrics. Lines of code, for example, does not provide sufficient information to ensure consistency between different data collectors. How do you define a line of code? Which lines of code are to be included in the count: source? object? assembly? HOL? comment lines?

Applying a metric to a product results in a data item. Applying a set of metrics to a product, and applying them throughout the course of its development, results in a data set. A model uses metric data as input, and produces another metric as output. Model development research drives metric definition in this sense, but work in development of generic software metrics has progressed independently and comprehensively, such that now model developers can design their models to incorporate existing metrics.

As software metrics become standardized and are applied to software projects and products, the software engineering research community should have access to more complete and more consistent data for validating their models. Yet the availability of software data remains a problem, for several reasons. To get data, you need not only metrics to define it and samples to take it from, but also sufficient motivation or justification for collecting it to outweigh the added costs that data collection adds to a development effort. Once data has been collected it needs to be stored so that it retains its usefulness. The data must be retrievable, understandable, and relevant. The validity of the data itself needs to be established, as well. When data collected by different people or at different times is to be compared or combined, care must be taken to normalize the data items with respect to each other, so that the numbers retain their meaning.

Data collection adds to the cost of a project development, although the relative cost is decreasing due to the use of automated tools for data collection. The long-term benefits of investing in data collection are beginning to be demonstrated and accepted, too. Storage of software measurement data has not posed a significant hurdle, because the amount of data that has been collected is not large in relation to existing storage capacities.

Software process improvement is currently a high-interest topic. Software development organizations are using the SEI's Capability Maturity Model (Humphrey 87) to assess their software process capability levels. The first step of the assessment, that of understanding the current status of the development process, requires data collection, and so the number of organizations that have established software data collection programs is increasing.

Issues surrounding the exchange and release of data have not been completely resolved. Software measurement data is often considered proprietary, especially when cost and productivity information is present or derivable from the data. Companies do not want to release such information for competitors to
A related issue that was identified during the development of the STARS Data Collection Forms is concern about the potential for the software data collected to be in conflict with official reports of a project's progress or status. Many software development organizations now have in-house databases of project experience data. But the only true national software data repository that maintains and distributes software data for research use is the DACS Software Lifecycle Empirical Database (SLED) (Chruscicki 93).

The goals of software measurement technology development have remained constant since its inception: to define metrics that will govern the collection of software data against which models can be validated. By applying validated models to software, researchers are able to:

- Learn more about the nature of software
- Get better control of software development projects
- More accurately predict costs, effort, and schedules
- Develop better software, that meets specified quality requirements
- Prove the worth of new tools and techniques
- Improve the software development process using feedback from measurement results
1.3 ROME LABORATORY ORGANIZATION

This document describes software measurement activities conducted by Rome Laboratory (RL) from the mid-seventies to the present. This section of the report presents a brief introduction to the current and previous organizations of the Lab.

Rome Laboratory, located at Griffiss Air Force Base in Rome, NY, is now one of the four Air Force super-labs. Unlike the other three super-labs which are tied to particular Air Force programs, RL's computer science and technology research and development program is chartered to advance generic computer technology.

At the beginning of the time period covered by this report, Rome Laboratory was called the Rome Air Development Center (RADC). RADC was both an Air Force center and a research laboratory, and thus had two chains of command. As a center, RADC reported to Electronic Systems Division (ESD). As a laboratory, RADC reported to Air Force Systems Command (AFSC). ESD, now the Electronic Systems Center (ESC), is located at Hanscom AFB in Bedford, MA.

Early in the history of software engineering, RL perceived that there was a need to develop and transition software engineering technology, and that addressing this need was within the scope of its mission. Ever since the early 1970s, therefore, RL has been studying, developing, and promoting software engineering and software measurement technology.

The RADC mission statement has evolved to keep pace with both advancements in technology and changes in the center's role with respect to the Air Force. Until 1976, the RADC mission statement described RADC as "the principal AFSC organization charged with planning and executing the USAF exploratory and advanced development programs for information sciences, intelligence, command, control and communications technology...Primary RADC mission areas are...information system technology..." The revised mission statement reflected an updated view of C3 as an application area, rather than a distinct technology, "RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C3) activities, and in the C3 areas of information sciences and intelligence. The principal technical mission areas are...information system technology..." The current mission statement states, "Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESC Program Offices (POs) and other ESC elements to perform effective acquisition of C3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including...computational sciences and software producibility..."
identified as Technical Program Objectives (TPOs). The TPOs encompass more narrowly defined sub-programs and thrusts. The individual efforts and tasks undertaken by the Lab are in support of a particular TPO's sub-programs and/or thrusts. Much of the early work was in support of TPO 11, Software Sciences Technology, which later became TPO 5, Software Cost Reduction. Some of the sub-areas associated with this TPO were Quality Measurement, Quality Control, Software Data Collection and Analysis, and System Availability (Hardware/Software).

The organization of the Laboratory into directorates, divisions, and branches mirrors the decomposition of the TPOs into programs, sub-programs and thrusts. When RADC began research in software measurement technology, the primary organization involved was the Information Sciences Division, IS. Within the IS division, the Information Management Sciences Section, ISIM, and the Software Sciences Section, ISIS, of the Information Processing Branch, ISI, had responsibility for the measurement-related research programs.

After a reorganization at the beginning of the 1980s, most of the software measurement research was the responsibility of the Command and Control Division, CO, which later became a directorate. As a division, CO had three branches: Battle Management Systems Technology, COA; Command & Control Systems Technology, COT; and Command & Control Software Technology, COE. As a directorate, CO consisted of two divisions: the Applied C3 Systems Division, COA; and the Command & Control Software Technology Division, COE.

For most of the research presented in this paper, the organizations at RL that sponsored the work were under the Command and Control Division/Directorate, CO, and mostly the Command & Control Software Technology Branch/Division, COE. Some measurement work was sponsored from within the Directorate of Reliability & Compatibility, RB, in the Microelectronics Reliability Division, RBR. The Directorate of Intelligence & Reconnaissance, IR, although more strictly applications-oriented, has also participated in measurement technology research.

Within COE, the two sections were the Software Engineering Section, COEE, and the Knowledge Engineering Section, COES. COEE was the primary section that sponsored measurement technology research.

After RADC became Rome Laboratory, there was another reorganization. The COE Directorate was transformed into the Computer Science and Technology Directorate, C3C. The Reliability & Compatibility Directorate, formerly RB, became ER.

The computer science and technology directorate, C3C, is currently the organization that is responsible for the computer science and technology program. This directorate is divided into three divisions, corresponding to the program's three subprograms: Software Engineering, C3CB, Computer Systems, C3AB, and Artificial Intelligence, C3CA. (DiNitto 92)

Current research related to measurement technology falls mostly under the software engineering sub-program. The technology areas under software engineering are: system/software support environments, system definition technology, software and system quality, and software for high performance computers.

It should be noted that, to at least some extent, all of the software technology research currently underway at RL depends on the software measurement foundation built by the research catalogued in this report.
The many contractors and researchers who have made major contributions in software measurement technology under RL sponsorship are identified in the following sections, as each effort is catalogued and placed in context.
2.0 BUILDING A MEASUREMENT TECHNOLOGY FOUNDATION

This section presents RL research in measurement technology and related efforts from the 1970s through the mid-1980s. The work documented here formed the basis on which current software engineering techniques depend. For example, the early work in reliability modeling and error classification has not only led to better models, but has also provided insights into the creation of software and into how the process can be altered to improve the product. The application of quality metrics to software proved that quality is not something that can be "injected" into software, no matter how careful or prolonged the testing is. This result helped shift the focus of software engineering research and methodology development away from coding and testing, and back toward the earlier phases of the lifecycle. Similarly, as a result of the extensive research in cost models, at RL and elsewhere, it has been shown that the cost to correct an error during system integration is up to fifty times more than if it were caught during requirements analysis (DiNitto 92).

The efforts presented in this section are organized by RADC contract number, and are grouped according to their primary research objective: model development and validation; metric definition; software data collection; and modern programming practices. Within the models section, the efforts are furthered classed as reliability, cost, and complexity model research.

The individual parts of this section are referred to as "catalogues," because the same type of identifying information is included for each contract. Because the intent of this paper is not to discuss the technical details of the research, what has been included instead is information that shows how the various efforts are related, and how the technology has evolved through continued RL sponsorship.

For each effort, the catalogue entry includes an identifying name, the time period, the name of the contractor, the contract number, a list of publications, and a brief summary of the project. The publications listed include both RL technical reports and any other related documents that were found in the DACS Software Engineering Bibliographic Database which include that contract number in the citation. All of the reports identified here are also listed in the References section. The project summaries are taken from the document abstracts, where applicable.
2.1 MODELS

Early RL studies in software measurement emphasized the creation of models. Very few software models existed in the early 1970s, and a foundation was needed for further work in software engineering. RL studies were central in creating models for software reliability, software cost and software complexity. Many of the models created by RL research have become widely referenced and the foundation for later research and practice. This section does not include software quality models; RL's software quality research is instead presented in conjunction with the framework development research in the metrics section of this paper.

A model is an equation (or set of equations) that takes software metric data as input parameters and calculates some other parameter. The output from a model is therefore a software metric, too, and may be used as input to another model. For example, a key input parameter for many cost and some reliability models is the software size. When these models are used to predict the cost or reliability of yet-to-be-developed software, the software size is not known but can be estimated by feeding project estimates into a sizing model.

This section is divided into three sub-sections, one for each major category of software model created and investigated by RL-sponsored research: reliability, cost and complexity. Although some efforts (and models) may have straddled categories, they have been placed where the area of primary research interest lay. Model research efforts documented here include both those that derived new models to describe software behavior and those that applied models against software data to validate them.
2.1.1 Reliability Models

Models have been developed to measure, estimate and predict the reliability of computer software. Software reliability has received much attention because reliability has always had obvious effects on highly visible aspects of software development: testing prior to delivery, and maintenance. Early efforts focused on testing primarily because that is when the problems appeared. As technology has matured, root causes of incorrect and unreliable software have been identified earlier in the life cycle. This has been due in part to the availability of results from measurement research and/or application of reliability models.

Reliability is the probability that a system will operate without failure for a given time in a given environment. Note that reliability is defined for a given environment. Since the test and operations environments are generally different, model results from test data may not apply to an operations environment. The "given time" in this definition may represent any number of actual data items, such as number of executions, number of lines of code traversed, or wall clock time. The use of a model also requires careful definition of what a failure is. Reliability models can be run separately on each failure type and severity level. One of the most frequently used failure classification schemes was developed by TRW in a RL-sponsored study (Thayer 74, 75, 76)

Reliability models are mathematically intense, incorporating stochastic processes, probability and statistics in their calculations, and relying on maximum likelihood estimates, numerical methods (which may or may not converge) and confidence intervals to model their assumptions.

Software reliability analysis and modeling includes several areas of investigation, all of which have some impact on reliability. Some of these are illustrated in the RL software quality framework, where the component criteria for reliability are consistency, accuracy and simplicity. Because simplicity, which is the inverse of complexity, is one of the component criteria of reliability, the distinction between reliability and complexity models often becomes fuzzy.

The reliability factor itself is alternately identified as error tolerance. Fault tolerance issues relate to reliability in that a fault tolerant system can still work even if it experiences a failure. Similarly, a large part of software reliability research has been devoted to error analysis, partly because error data was more readily available, but mostly because errors are a good metric for reliability. Other issues involved in the study of software reliability raise questions such as: How good is good enough? and How do you determine when you're done?

The Data & Analysis Center for Software prepared a Software Engineering Research review, Quantitative Software Models, in 1979 (DACS 79a), and an update in 1987 (Iuorno 87) that listed and summarized the most popular and important models for software reliability, complexity, sizing and cost. Of the thirty reliability models listed in these two reports, approximately one-third were developed or refined as a result of RL-sponsored research efforts.
The remainder of this sub-section contains a catalogue of RL efforts in reliability model research, organized by contract number, covering both development-oriented and validation oriented studies.

**Study Name: Software Modeling Studies**

Martin Shooman and colleagues at the Polytechnic University of New York performed software reliability modeling and other research from 1974 to 1979 under two RL contracts. Although Shooman's Exponential Model predates these investigations, the RL-sponsored work refined and validated that model. Other reliability models associated with these contracts are the Shooman-Natarajan model (bug generation), the Shooman-Trivedi Markov model, the Rudner model (seeding) and the Shooman Micromodel.

The series of Summary of Technical Progress, Software Modeling Studies reports describe objectives and progress at several points throughout the contract period. Research areas included: software error, reliability, and availability models (Markov availability models, bug tagging estimates of initial error counts and error models incorporating error generation); a micro reliability model, incorporating representative features of the internal program structure, involving path (module) traversed frequencies and times and path failure probabilities parameters; test models and techniques (determination of the number of the tests necessary to execute all program paths and statistical test models); and complexity models (component measures like testedness and natural language theory measures). Other research was undertaken in areas such as acceptance test models and automatic programming techniques. (Ruston 74, 75a, 75b, 76a, 76b, 77a, 77b, 78).

The models known as Shooman-Trivedi are many-state Markov models which were developed for the purpose of providing quantitative reliability criteria for computer software where the software system under consideration is large enough that statistical deductions become meaningful and initially contains an unknown number of bugs. The basic models provide estimates and predictions for a quantifier that represents the state of debugging of the system and which is generally the most probable number of software errors that will have been corrected at a given time in the operation of the software system based upon preliminary modeling of the error occurrence rate and the error correction rate. The models also provide predictions for the availability and for the reliability of the system. As part of this effort, the differential equations corresponding to the basic many-state Markov models were solved for verification and demonstrative purposes. Manufacturer's data were obtained on the performance of system software for a medium-sized software operating system. These data were analyzed to obtain frequency distributions of the random variables representing the time to close software error reports. The data were then used for application of the basic many-state Markov model. Also studied in the course of this effort were error data collection and software error documentation practices (Shooman 75).

The Shooman-Natarajan model built on earlier software error models by Shooman and Jelinski-Moranda which related the number of errors in a large software system to the rate of error removal. In this effort, expressions for the number of remaining errors as the software undergoes debugging were formulated and additional assumptions were made to relate the number of residual errors to the operational system reliability. One of the key assumptions of the earlier models was that the sum of the errors removed and those remaining in the program is constant. This work added a major refinement to the above models by introducing the possibility of error generation during debugging. In this refinement the error generation terms are modeled in several different ways: proportional to the number of detected errors, corrected errors, the number of remaining errors, or some function of these effects. The correction rate is assumed
to be a function of the manpower deployed on the project, thus permitting the use of the model to investigate optimum manpower deployment strategies. The effects on the economics of debugging due to error growth were also analyzed (Natarajan 76).

The Rudner models estimate the number of software errors based on the number of errors either inserted deliberately in a program (seeded) or found by debugging (tagged), the number of errors found by a debugger unaware of the first set, and the number of errors appearing in both sets. Three models were developed. The mean and mean-squared error of a maximum likelihood estimate and a modified maximum likelihood estimate were derived for all three models. These quantities were found to vary with certain relations among the total number of errors, size of tagged or seeded set, and size of accompanying sample set. A procedure for determining optimum values for size of tagged or seeded set and number found by the second debugger was defined. Finally, multi-trial estimates for parameters were found and compared with single-trial estimates (Rudner 77).

Another effort investigated the application of concepts of statistical language theory (Zipf's Laws) to the derivation of formulas for measuring program and language complexity. Experimental data from several different programs and programming languages, such as PL/I, assembly and FORTRAN, was used to verify the necessary underlying assumption and to verify formulas for program length by comparison with actual statistics. Finally, the derived formulas were then compared with those of Software Physics derived by Halstead (Laemmel 78).

Study Name: **Software Modeling Studies**

The principal topics investigated in this effort were: software test models and implementation of automated test drivers to force-execute every program path; methods for developing programs with low error content; development of new measures of program complexity based upon information theory; models of software management and organizational structure; and statistical measures relating the probability of finding a program error to the testing of that program (Ruston 79, 81).

The error data work included analysis of a set of error data gathered from field trials of a medium-sized real-time computer system. The software was 50K in size, and ran on a microcomputer which performed measurements on a batch of samples and issued reports. The reliability model used assumed that the failure rate was constant and proportional to the number of remaining errors. The analysis showed how hardware, software, and operation failure rates interact to yield system reliability, and results included recommendations for how MTBF goals should be chosen (Shooman 79b).

The use of flowchart graphs, adjacency matrices, and zero-one linear programming to find the minimum number of tests necessary to execute every segment of a computer program at least once were investigated. The methods of Lipow were used as the basis for determining the maximum incomparable set. Methods were defined for finding the maximum incomparable set for loopless and some elementary looping flowcharts (Popkin 78).

The research in software test models and associated test drivers defined five broad classes of software tests: Type 0, Type 1, Type 2, Type 3 and Type 4. In a Type 0 test, all instructions are exercised at least once. In a type 1 and 2 test, all flowchart paths are exercised at least once. Type 1 is performed by forced traversal and Type 2 by natural execution. Types 3 and 4 correspond to an exhaustive interaction of all input and stored data. Types 3 and 4 were determined to be unfeasible, therefore only a strategy lying between Type 1 and 2 can effectively be implemented. Since enumeration of all the paths in a given
program is required for Type 1 and 2 tests, this research established the lower and upper bounds on the number of paths as a function of the number of deciders, described a manual decomposition procedure to cut a graph into smaller subgraphs, and proposed an algorithm to machine-identify all paths. A complete Type 1.5 driver system for forced path traversal was implemented in PL/1, along with suggestions on how to extend these techniques to other languages. A typical program was analyzed manually, tested with data, and run through the system. Some evaluation of the usefulness of the system was performed in the light of the accumulated experience (Baggi 80).

Other work performed under this sponsorship included development of a tutorial on Software Cost Models. Contents of the tutorial included a discussion of the Rayleigh Model for the rate of expenditure of project costs over the development life cycle. Application of the model to field data, and estimation of the two model parameters was discussed. The next topic was the estimation of total project cost. A classification scheme for total cost estimation models was developed, and seven different models were introduced and used to estimate the cost of developing a program for the same problem. In addition, an example of an incentive/penalty contract was given, and basic studies of the underlying risk analysis were referenced. The concept of an incentive/penalty based acceptance test was introduced, as was the concept of warranty contracts. The appendix provided a statistical justification for why we should expect increased accuracy with a bottom-up estimation method (Shooman 79a).

Study Name: Bayesian Software Prediction Models

At Syracuse University, Dr. Amrit Goel and Kazuhiro Okumoto and colleagues have been doing reliability model and other research for RL since 1976. This research has produced two sets of reliability models, the Goel-Okumoto Bayesian model and the Goel-Okumoto Non-Homogeneous Poisson Process model.

The Bayesian software prediction models, known as the Goel-Okumoto Bayesian models, include the Imperfect Software Debugging Model, a stochastic model for software failure phenomena which assumes errors are not corrected with certainty. By assuming the initial number of errors, the probability of successfully correcting an error, and the constant error occurrence rate are all known, formulas for such quantities as the distribution of time to completely debugged software, the distribution of time to a specified number of remaining errors, and the expected number of errors detected by time t can be derived. The Imperfect Debugging Model was then extended to incorporate error correction time, estimation of model parameters, and development of a Bayesian model.

Other research under this effort included work toward developing bivariate software reliability models, where system errors are classified as serious and non-serious, for the operational phase of a software system. The models incorporate the uncertainty of error removal and the time spent in correcting errors. The Bayesian Software Correction Limit Policies were designed to determine the optimum time value that minimizes the long run average cost of debugging at two levels - correction action undertaken by the programmer (Phase I) and action undertaken by a system analyst or system designer, if the error is not corrected in Phase I (Phase II). One of the models assumes the cost of observations of error occurrence and correction time, prior to implementation of the optimum policy, is negligible; another incorporates the cost of observations. Other work included development of empirical models for software error data; development of software reliability demonstration plans for making accept/reject decisions for software packages; and investigations of the effects of changes in prior distributions and/or model parameters on quantities of interest (Goel 77, 78).
Study Name: **Software Reliability Models**

This software reliability modelling and estimation research resulted in two new models of very general applicability. A new methodology for determining when to stop testing and start using software was developed, and a new model for analyzing the operational performance of a combined hardware-software system was developed, even though it was not a part of the original research plan (Goel 82).

This effort led to the development of the Goel-Okumoto NonHomogeneous Poisson Process model, a parsimonious model whose parameters have a physical interpretation, and which can be used to predict various quantitative measures for software performance assessment. With this objective, the behavior of the software failure counting process \( N(t), t > 0 \) was studied. It was shown that \( N(t) \) can be well described by a non-homogeneous Poisson process (NHPP) with a two parameter exponentially decaying error detection rate. Several measures, such as the number of failures by some prespecified time, the number of errors remaining in the system at a future time, and software reliability during a mission, were proposed. Models for software performance assessment were also derived. Two methods were developed to estimate model parameters from either failure count data or times between failures. A goodness-of-fit test was also developed to check the adequacy of the fitted model. Finally, actual failure data were analyzed from two DOD software systems: a large command and control system and a Naval data analysis system (Goel 80a).

The problem of determining the optimum time when testing can stop and the system can be considered ready for operational use was considered. This decision, of course, depends on the model for the software failure phenomenon and the criterion used for evaluating system readiness. With the time dependent error detection rate model of Goel and Okumoto, two criteria are investigated: software reliability and total expected cost. Based on the cost criterion, an optimum release policy can be derived and its sensitivity to the model parameters studied (Goel 80).

Study Name: **Software Reliability Assessment**

Recognizing that there existed within DOD a need for specifying software reliability requirements in RFPs and for performing reliability assessments of developed software to insure these requirements are being met, RL initiated a long term effort to develop a handbook for software reliability assessment, similar to MIL-HDBK-217, that could be used by appropriate Air Force personnel in specifying and assessing software reliability requirements. The nonhomogeneous Poisson process software reliability prediction model formed the basis of this initial guidebook (Sukert 80).

The completed handbook contained: a summary and evaluation of most of the available models for software reliability assessment; a discussion of software quality, software testing and software reliability; brief descriptions of times between failures and failure count software reliability models and of fault seeding and input domain models; assumptions, limitations and applicability of the models; and a step by step procedure for developing a software reliability model (Goel 83).

Study Name: **Bayesian Software Prediction Models**

This experiment investigated the effect of FORTRAN and Ada languages on program reliability. The experimental design employed was a 2 by 2 full factorial design, for example, a design in two variables, each at two levels. The problems used in the experiment was the Launch Interceptor Program (LIP), a simple but realistic anti-missile system. Reliability comparisons between Ada and FORTRAN programs
were based on the total number of errors as well as on errors found during various testing phases. Some comparisons were also based on error density, the number of errors per 100 non-comment lines of code. It was found that on the average, the Ada programs had about seventy percent less errors than the FORTRAN ones. If errors during unit testing were excluded, the Ada programs had about seventy-eight percent less errors. Similar differences were found for data based on error causes and error types (Goel 88).

Study Name: **Software Reliability Study**

The objectives of this study were to: (1) Determine what software structural and development characteristics are available for analysis and which of these characteristics are relevant to the description (or prediction) of software reliability; (2) Define improved methods for collecting reliability data; (3) Based on error histories seen in the data, define sets of error categories, both causative and symptomatic, to be applied in the analysis of software problem reports and their closures; (4) Recommend changes in: i) development techniques to enhance the error-freeness (reliability) of the coded product and ii) test techniques to make it possible to find more errors earlier; (5) Perform a survey of existing software reliability models; (6) Extend Nelson's Mathematical Theory of Software Reliability (MTSR) and apply it to data collected on an ongoing software development project. These objectives were achieved principally through analysis of empirical data collected during software testing.

This effort involved the study of software error types, including techniques for locating them, and developing recommendations for improvement of reliability. A survey of then-current software reliability models was performed, and TRW work in this area was summarized. Data collection and analysis schemes were developed for the subject data sets. Data from four large software systems was analyzed to determine the types of errors found in software during testing. The objective was principally to recommend new development or test techniques for the detection and prevention of software errors, but they also attempted to model software reliability. One result of the Software Reliability Study was an illustration of the benefits of software reliability data collection and analysis. Recommendations for identifying data that need collecting were also made (Thayer 74, 75, 76).

Lipow's paper, Prediction of Software Failures, discusses the aspects of the research which used empirical data collected during development and operational use of one large (>300K machine language instructions) software system spanning a period of two years. These data were examined in light of their predictive utility, and techniques for analyzing the data were developed and implemented. The applicability of these predictors in a universal sense, i.e., to more than one system, was considered in light of the results of the sample application (Lipow 77).

Study Name: **Statistical Prediction of Programming Errors**

In this study statistical predictions of programming errors were made using multiple linear regression analysis. Programming errors were predicted from linear combinations of program characteristics and programmer variables. Each of the program characteristic variables were considered to be measures of the program's complexity and structure. Two distinct data samples comprising 783 programs with approximately 297,000 source instructions written for command and control software applications were analyzed. As a result of the analysis, prediction equations, known as the Motley-Brooks model, were derived. The predictability of errors and error rate in each sample was investigated, and recommendations for further research were made (Brooks 77).
Study Name: **Software Reliability Estimation**

In this effort, a model was developed and programmed for estimating the reliability of large software packages. The model incorporated features found in realistic testing environments. The model was analyzed for deterministic data, and run against simulated random test data. The algorithm estimates the model parameters from software error data, and then computes running estimates of the mean-time-to-failure and the number of software errors remaining (Crawford 79).

Included in this effort was an examination of measures for evaluating the performance of algorithms for single instruction stream - multiple data stream (SIMD) machines. The measures included execution time, speed, parallel efficiency, overhead ratio, processor utilization, redundancy, cost effectiveness, speed-up of the parallel algorithm over the corresponding serial algorithm, and an additive measure called "price" which assigns a weighted value to computations and processors (Siegel 82).

Study Name: **Methodology for Software Reliability Prediction**

The objective of this research and development effort was to develop a methodology for predicting and estimating software reliability throughout the life cycle of a software project. A framework which spans the life cycle of a software system was employed, which built on the RL software quality framework research in metrics and models as techniques to aid in the prediction and estimation of reliability during the software development process. Within the framework, the specific data needed to measure software reliability and the utility of the measurements to help make sound software engineering decisions was studied. Included in the effort was a comprehensive data collection effort to investigate software reliability, and development of a database.

A Methodology was developed in the form of a Guidebook for predicting and estimating software reliability based on the findings of this study. The Guidebook provided procedures for the preparation of software reliability predictions and estimations for embedded and separately procured computer systems. These procedures include those for developing a software reliability program plan, for performing software reliability prediction, and for performing software reliability estimation. Data collection procedures which describe what data must be collected to use the software reliability prediction and estimation procedures were provided. Metric data collection worksheets were also included in the Guidebook (Bowen 87).

Study Name: **DACS Studies**

The objective of the Data Parameters study was to identify data requirements for software reliability modeling. Brief descriptions of the models and the data needed to exercise the models are presented. The paper concludes with a list of recommendations for future data collection (Donahoo 79).

The purpose of the research review study was to disseminate information on the models and methods that encompass software life-cycle costs and productivity, software reliability and error analysis, and software complexity, and the data parameters associated with these models/methods, through preparation of the report. A unique contribution of this report was the inclusion of matrices which denoted the data parameters for each method. The matrices not only revealed the areas and types of data the methods provide and require but also furnished a common basis for the comparison of models and the definition of data parameters (DACS 79a).

The third study also investigated data requirements for software reliability modelling. The major
assumptions of the selected models were determined, along with a brief description of their uses and the
data needed to exercise the models. Methodologies for evaluating failure databases were developed, and
a sample evaluation was performed to determine the adequacy of the data for making comparisons across
a wide variety of projects, and to determine if the database contains the data elements required by the
various models (Donahoo 80).
Software Modeling Studies

Dates: 1974 - 1978
Contractor: Polytechnic University of New York
Contract #: F30602-74-C-0294

Tech Reports:
- RADC-TR-75-169, Computer Software Reliability: Many-State Markov Modeling Techniques
- RADC-TR-76-400, Effect of Manpower Deployment and Bug Generation on Software Error Models
- RADC-TR-75-246, Summary of Technical Progress Software Modeling Studies
- RADC-TR-75-245, Summary of Technical Progress Software Modeling Studies
- RADC-TR-76-143, Summary of Technical Progress - Software Modeling Studies
- RADC-TR-76-405, Summary of Technical Progress - Software Modeling Studies
- RADC-TR-77-88, Summary of Technical Progress, Software Modeling Studies
- RADC-TR-78-4, Software Modeling Studies Summary of Technical Progress Statistical (Natural) Language Theory and Computer Program Complexity

Related Pubs: Summary of Technical Progress, Software Modeling Studies

Software Modeling Studies

Contractor: Polytechnic University of New York
Contract #: F30602-78-C-0057

Tech Reports:
- RADC-TR-79-188, Summary of Technical Progress, Investigation of Software Models
- RADC-TR-78-229, On the Number of Tests Necessary to Verify a Computer Program
- RADC-TR-80-45, Software Test Models and Implementation of Associated Test Drivers
- RADC-TR-81-183, Software Modeling Studies, Executive Summary

Related Pubs:
- Tutorial on Software Cost Models Software Reliability Data Analysis and Model Fitting
Bayesian Software Prediction Models

Dates: 1976 - 1978
Contractor: Syracuse University
Contract #: F30602-76-C-0097

Tech Reports:
- RADC-TR-77-112, Summary of Technical Progress on Bayesian Software Prediction Models (Goel 77)
- RADC-TR-78-155, Bayesian Software Prediction Models Summary of Technical Progress - Vol V (Goel 78)

Software Reliability Models

Dates: 1978 - 1982
Contractor: Syracuse University
Contract #: F30602-78-C-0351

Tech Reports:
- RADC-TR-80-179, A Time Dependent Error Detection Rate Model or Software Performance Assessment with Applications
- RADC-TR-82-263, Software Reliability Modeling and Estimation Techniques

Related Pubs:
- A Guidebook for Software Reliability Assessment
- Optimum Release Time for Software Systems Based on Reliability and Cost Criteria
- When to Stop Testing and Start Using Software?

Software Reliability Assessment

Dates: 1981 - 1983
Contractor: Syracuse University
Contract #: F30602-81-C-0169

Tech Report:
- RADC-TR-83-176, A Guidebook for Software Reiability Assessment
Bayesian Software Prediction Models

Date: 1988  
Contractor: Syracuse University  
Contract #: F30602-81-C-0193

Tech Report:  
- RADC-TR-88-213, An Experimental Investigation Into Software Reliability

Software Reliability Study

Dates: 1974 - 1976  
Contractor: TRW  
Contract #: F30602-74-C-0036

Tech Reports:  
- RADC-TR-74-250, Software Reliability Study  
- RADC-TR-76-238, Software Reliability Study Final Report

Related Pubs:  
- Understanding Software Through Empirical Reliability  
- Analysis Prediction of Software Failures

Statistical Prediction of Programming Errors

Dates: 1976 - 1977  
Contractor: IBM FSD  
Contract #: F30602-76-C-0213

Tech Report:  
- RADC-TR-77-175, Statistical Prediction of Programming Errors

Software Reliability Estimation

Dates: 1978 - 1979  
Contractor: University of Utah  
Contract #: F30602-78-C-0025

Tech Report:  
- RADC-TR-79-230, Software Reliability Estimation Under Conditions of Incomplete Information
  Final Report
Methodology for Software Reliability Prediction

Date: 1987
Contractor(s): SAIC, SoHar, IITRI
Contract #: F30602-83-C-0118
Tech Report:
  ● RADC-TR-87-171, Vols I & II, Methodology for Software Reliability Prediction

DACS Studies

Dates: 8/78 - 8/81
Contractor: IIT Research Institute, Computer Sciences Corporation
Contract #: F30602-78-C-0255
Related Pubs:
  ● Software Engineering Research Review Quantitative Software Models
  ● Quantitative Software Reliability Models - Data Parameters: A Tutorial
  ● Data Needs for Software Reliability Modeling
2.1.2 Reliability Model Validation Studies

These efforts focused mainly on the validation and evaluation of reliability models, rather than on development of new models.

Study Name: **Software Reliability Modeling**
Date: 1976
Organization: Rome Laboratory
Tech Report: RADC-TR-76-247, A Software Reliability Modeling Study

This study was initiated to apply several software reliability error prediction models from the literature to error data from a large command and control software project. This error data details the complete error history of this project from the start of validation testing through the end of demonstration testing, and also includes actual operational error data. The study involved obtaining predictions from the various models used, and comparing the predicted results with the actual error data history. From these comparisons, analysis to determine the reasons why the predicted results did or did not agree with actual results was performed. This study included comments on overall trends and possible future extensions of this work (Sukert 76).

Study Name: **Analysis of Discrete Software Reliability Models**
Dates: 1978 - 1980
Contractor: IBM FSD
Contract #: F30602-78-C-0346

Discrete (binomial and Poisson) software reliability models, previously developed by IBM, were examined for their validity using historical data provided by RADC from seven development projects. Simulated error data that closely agreed with the model assumptions were also used for validation. The results indicated that the models provide reasonable fits to the historical error data. A secondary result was the recommendation that future studies of the models be accomplished with error data that are properly collected according to the model data requirements and assumptions (Brooks 80).

Study Name: **Validation of Software Reliability Models**
Dates: 1977 - 1979
Contractor: Hughes
Contract #: F30602-77-C-0219
Tech Report: RADC-TR-79-147, Validation of Software Reliability Models

This study investigated software reliability models. The purpose was to investigate the statistical properties of selected software reliability models, including the statistical properties of the parameter estimates, and to investigate the goodness of fit of the models to actual software error data. The results
indicated that the models fit poorly, generally due to problems with the data rather than shortcomings of the models (Angus 79).

Study Name: **Reliability Model Demonstration Study**
Dates: 1980 - 1983
Contractor: Hughes
Contract #: F30602-80-C-0273
Related Pub: Combined Hardware/Software Reliability Models

This purpose of this study was to determine the use and applicability to Air Force software acquisition managers of six quantitative software reliability models to a major command, control, communications, and intelligence (C3I) system. The scope of the study included the collection of software error data from an ongoing C3I project, fitting six software reliability models to the data, analyzing the predictions provided by the models, and developing conclusions, recommendations, and guidelines for software acquisition managers pertaining to the use and applicability of the models. The second volume of the study report contained the software data used in evaluating the models (Angus 83).

Also under this effort, a theory for combining well-known hardware and software reliability models was developed around the principles of Markov processes. The use of steady-state availability as a reliability/maintainability measure was shown to be misleading for systems exhibiting both hardware and software faults (Angus 82).
2.1.3 Cost Models

Software cost modeling is now a mature technology. Well established models exist, although they are not as widely employed as they should be. The creation of new models is no longer a concern for basic research. Barry Boehm's exposition (Boehm 81) of the Constructive Cost Model (COCOMO) was crucial for bringing about this state of affairs. COCOMO built on previous models, one of which was developed under RL sponsorship.

The Doty Model (Herd 77) was constructed based on an extensive data analysis effort. It was one of a number of early cost models of the same form as later popularized by COCOMO. An initial estimate of effort is found as an exponential function of size:

\[ MM = aI^b \]

where MM is man-months and I is the size. Size can be measured by either object or source code. This initial estimate is then modified to reflect the influence of certain multipliers or cost drivers. The Doty Model was a step along a road leading to the dominance of cost models of this structure.

Rome Lab sponsored efforts that looked exclusively at cost models follow:

- **Study Name:** Software Cost Estimation Study
- **Dates:** 1976 - 1977
- **Contractor:** Doty Associates
- **Contract #:** F30602-76-C-0182
- **Tech Report:** RADC-TR-77-220, Vols I & II,
  Software Cost Estimation Study Results
  Guidelines for Improved Software Cost Estimating

The study identified factors that have an adverse effect on software cost estimates, determined their impact on software cost estimates, discussed methods for controlling the effect of these factors, and developed an overall methodology for estimating the costs of software development. In addition to a generalized model for estimating software development costs, separate models were generated for estimating the development cost of command and control, scientific, utility, and business software (Herd 77).

This effort included the development of guidelines for use in arriving at estimates of computer software cost. The guidelines gave first consideration to the initial program estimate which is often made with a paucity of supportive data. Adjustments were presented for modifying the estimate given the availability of additional data. Procedures were defined for assessing the affordability of the resulting estimates. Emphasis was placed on developing a conservative but reasonable best estimate for purposes of program budgeting. Separate consideration was given to steps that should be taken to bring the program in at or below budget. Frequently recurring problems were identified and summarized in their time-phased order.
This effort evaluated nine software cost estimating models to determine if they satisfied Air Force needs. The evaluation considered both the qualitative and quantitative aspects of the model's outputs. Air Force needs for cost estimates were as established by the Major Weapon System Acquisition Process. Associated with the different development phases are five cost estimating situations. Decisions that are made early in the Acquisition Process require software cost information that includes the entire life cycle for complete software systems, subsequent decisions require more detailed cost information. The study compared the outputs of the nine test models with the requirements established by the five cost estimating situations. The results indicated that the evaluated models were able to satisfy only the needs of the earliest phase of the Acquisition Process. Estimating accuracy was measured by comparing outputs with actual experience using data from three organizations representing 45 software development projects (Thibodeau 81).

Study Name: **C3I Software Cost Estimation**

Dates: 1983, 1984

Contractor: Eddins-Earles

Contract #: F30602-83-C-0184

F30602-84-C-0154

Tech Reports: RADC-TR-84-156, Cost Estimation Techniques for C3I System Software

RADC-TR-87-194, C3I Software Cost Estimation Model Development

Cost Estimation Techniques for C3I System Software: The first phase of this research developed the concept and computer programming requirements for a software life cycle cost estimating system which included methodology for the establishment of a database for Command, Control, Communications and Intelligence (C3I) system sizing. The proposed system used the cost estimating relationships of the COCOMO model and generic files of baseline C3I software designs for aid in sizing the number of source instructions required for a new design. The computer programming requirements were developed for a user-friendly interactive program. They permitted computer program configuration items to be designed by choosing computer program components (CPCs) from a stored library of functionally structured computer program modules. They permitted CPCs to be designed by choosing generic modules of code from a similar stored library. They permitted interactive life cycle cost estimates to be made at each level of the software breakdown structure with automatic re-computation with input changes (Eddins-Earles 84).

In the second phase of this research, an implementation of the COCOMO software cost estimation model defined in the first phase, was created. This model was specifically tailored for the Command Control Communications Intelligence (C3I) software. A set of benchmark C3I software Sizing and Productivity
data was included with the program. The program can be used to estimate software life cycle cost during the conceptual phase with what is called a top level cost estimating methodology, and during full scale development with what is called build up cost estimating methodology. The more detail that is gained on the structure of the software to be developed, the more accurate the results of the cost estimation methodology (Eddins-Earles 87).
2.1.4 Complexity Models

Complexity is a major driver of the cost, reliability, and functionality of software systems. Both the inherent complexity of the problem and any additional complexity of the implementation are important aspects. Software size is included in this interpretation of complexity because, unless special techniques are employed to combat it, the complexity of software increases with its size.

To control complexity, one must be able to measure it, and researchers have proposed a wide range of complexity metrics. Complexity measures that have been successfully transitioned into practice, for example, Source Lines of Code (SLOC) and McCabe's Cyclomatic Complexity metric, focus on the complexity of a single Computer Software Unit (CSU), module, or subprogram.

With the exception of SLOC, McCabe's Cyclomatic Complexity metric (McCabe 76) is the most widely used and most well known complexity metric. Interestingly enough, RL sponsored research that developed a complexity metric very close to the McCabe metric (Sullivan 75). Apparently unknown to Thomas McCabe, his famous paper comes fairly close to duplicating the content of this previous research. McCabe published his work in a much more accessible source and in a very readable fashion. Nevertheless, if McCabe had not published his work, his metric might be known to posterity as the Sullivan Complexity Metric.

Like McCabe, Sullivan began by considering graphical representations of programs. Sullivan looked at both control flow and data structure, and used the formal algebraic structures in graph theory to represent these graphs. The number of paths through a control flow graph is then taken as a fundamental measure of complexity. Sullivan immediately has to confront the practical difficulties raised by loops, where no static upper bound can always be found for the number of paths. He decided to count two paths for loops, one executing the body of the loop once and one completely bypassing the loop. This was also McCabe's solution.

Another problem is created by the systems comprising many subroutines. Sullivan, like McCabe later, decided to represent the complexity of the system as the sum of the complexities of the control graphs of the subsystems. These decisions result in the following McCabe-like definition of complexity:

**Def.** The C2 complexity at any node of an elementary scheme is one less than the number of paths from the start node to that node, not counting paths where any node occurs more than x times, where x = 2 unless otherwise stipulated. Again, the complexity of the entire elementary scheme [control flow graph] is defined as the local complexity at the terminal node. The complexity of a composite scheme is defined as the sum of the complexities
of elementary subschemes.

McCabe immediately used his definition of complexity to investigate structured programming. He found that control flow graphs constructed out of sequence, alternation, and iteration alone minimizes cyclomatic complexity in a certain formal sense. Sullivan previously explored the same lines of research:

**Theorem (C2 - Structure).** A scheme S is structured in the (strict classical) control sense if and only if the C2 complexity measure of all elementary subschemes is 0 or 1...

**Corollary.** If a program is written (in an appropriate language) using simple sequence, IF-THEN-ELSE for alternation, WHILE-DO for iteration, and no other control flow constructions, the C2 measure is just the number of IFs and WHILEs.

There are differences between Sullivan's RL-sponsored research and McCabe's later paper. Sullivan references Edsger Dijkstra for his guidance on structured programming, while McCabe refers to Donald Knuth. Unlike Sullivan, McCabe applies his ideas to guide the development of test cases. On the other hand, Sullivan immediately examines the complexity of data structures. Nevertheless, the parallels between Sullivan's mostly unknown research and McCabe's later results are quite astonishing.

As systems grow larger, aspects of system-level complexity, reflected in relationships between CSUs and higher level components, become more important. Although there are certainly commercial tools for measuring system complexity, this aspect of software complexity is still a research issue or, at least, a technology transfer problem (Card 90).

Uncontrolled complexity is very apparent during operations and maintenance. Attempting to change one subroutine in a poorly designed system will require a change to further subroutines with changes rippling through the entire system. RL sponsored several research efforts in this area in the late 1970s and early 1980s, performed by Stephen Yau and associates at Northwestern University (Yau 79, 80a, 80b, 84). Under these efforts, a methodology was developed for software maintenance, oriented around the five phases of

- Determining the maintenance objective
- Understanding the program
- Generating a particular maintenance proposal
- Accounting for logical and performance ripple effects
- Testing the changed program

These phases were associated with particular quality attributes. An emphasis was placed on stability, an attribute that minimizes the ripple effects of program modifications. Some quantitative metrics were developed for stability, along with a tool to automatically compute these measures from source code.

Although these metrics are still not widely used, later work on software design measures frequently references this RL-sponsored research and builds on it. Understanding and improving software
maintenance remains a current research issue. In addition, more recent RL research has revisited the question of software design measures.

A listing of RL efforts related to complexity studies and model development follows:

**Study Name: Engineering of Quality Software Systems**
Dates: 1/73 - 6/73
Contractor: MITRE Corporation
Contract #: F19628-C-73-0001 (Customer RADC/ISIS)

Tech Report:
- RADC-TR-74-325, Vol I - VIII

Related Pubs:
- Introduction Effects of Management Philosophy on Software Production
- Software First Concepts
- Towards an Analysis of the LISP Programming Language
- Measuring the Complexity of Computer Software
- Some Case Studies in Structured Programming
- A Software Error Classification Methodology
- Software Reliability Modeling and Measurement Techniques

Of the eight volumes in this series, two are relevant to model research. Sullivan's paper, Measuring the Complexity of Computer Software, as indicated above, presents several measures of computer program complexity, defined by him as comprehensibility or intellectual manageability. The measures consider the program as an abstract process, and are independent of programming language or implementation details (Sullivan 75).

The reliability model research, in Software Reliability Modeling and Measurement Techniques, included both software reliability and error and reliability estimation, and included a detailed description of six models and techniques (Lapadula 74).

**Study Name: Algorithmic Complexity**
Dates: 1979 - 1982
Contractor: University of Rhode Island
Contract #: F30602-79-C-0124

Tech Report:
- RADC-TR-82-152, Vols I & II, Algorithmic Complexity

The objective of this study was to conduct applied research directed toward understanding the relationship between the complexity or efficiency of algorithms and the overall quality of computer software. The effort included review of previous RL work on software quality metrics, emphasizing measures concerned with the time and storage efficiency of programs. Next, a review of the field of algorithm analysis and computational complexity was undertaken. New and unresolved issues concerning the relationships between programming languages, computer architecture, and the performance of algorithms on computer systems were identified. Problem application areas considered
included: computational algebra, used for algorithm analysis and computational complexity; development of a systematic approach to the analysis of algorithms; an experimental analysis of a fast, new sorting method called DPS (distributive partitioning sorting); application of order statistics to investigate the expected quality of several approximation algorithms for the Euclidean traveling salesman problem, known to be NP-complete; a survey of data base access methods for both univariate and multivariate range queries; and an experimental evaluation of the frame memory model of a data base structure (Anderson 82).

Study Name: **Software Complexity Model Studies**
Dates: 1976 - 1979
Contractor: Northwestern University
Contract #: F30602-76-C-0397

Tech Reports:
- RADC-TR-79-127, On Software Reliability Modeling
- RADC-TR-79-128, Dynamic Monitoring for Linear List Data Structures
- RADC-TR-79-129, Performance Considerations in the Maintenance Phase of Large-Scale Software Systems
- RADC-TR-80-55, Performance Ripple Effect Analysis for Large-Scale Software Maintenance
- A Handbook: Part I, Logical Ripple Effect Analysis
- A Handbook: Part II, Performance Ripple Effect Analysis

Related Pubs:
- Ripple Effect Analysis of Software Maintenance
- A User-Oriented Software Reliability Model
- Assertion Techniques for Dynamic Monitoring of Linear List Data Structures

In this research the possible effect of program modifications during the maintenance phase on the performance of large-scale software systems was analyzed. Mechanisms for the propagation of performance changes from one part of the system to another were identified, and the relationship among these mechanisms, performance attributes, critical program sections and performance requirements was also investigated. Also covered were dependency relationships between virtual performance attributes and performance attributes. Maintenance techniques for predicting which performance requirements in the system may be affected by a proposed modification were developed. These techniques enable maintenance personnel to incorporate performance considerations in their criteria for selecting the type and location of software modifications to be made, and identify which performance requirements must be verified after the modification in order to insure that they have not been violated by the modification (Yau 79, 80a). A figure of merit was proposed to estimate the complexity of program modification. This figure could be used as a basis upon which various modifications can be evaluated (Collofello 78).

This research in the area of developing effective techniques for large-scale software maintenance, including those for the design, implementation, validation, and evaluation of reliable and maintainable software systems with a high degree of automation resulted in the development of a handbook. The handbook presents a ripple effect analysis technique for software maintenance from the logical or
functional perspective. The handbook included: discussion of capabilities and restrictions of the logical ripple effect analysis technique; an outline of the technique, consisting of a lexical analysis phase, and a tracing phase; and an in-depth description of each step in the technique. The second part of the handbook discussed the technique from a performance point of view. A discussion of the capabilities and restrictions of the technique was followed by considerations of user interface. An outline of the performance technique was given, and each step in the two phases (lexical analysis and tracing) was expanded. Finally, integration of the logical and performance ripple effect analysis processing steps was explained (Yau 80c).

The contributions that dynamic monitoring can make in the area of software maintenance were also investigated. New techniques that enable monitoring of most array-implemented linear list data structures were studied. The main advantage of these assertion techniques is the ability to construct a loop around a group of simple assertions. With this construct, the programmer can explicitly define the record traversal scheme for a linear list data structure implemented with either sequential or linked-list allocation. The practicality of the techniques, together with some performance figures were considered. The techniques were implemented in JOVIAL (Ramey 79).

Also as part of this research effort, various software reliability models were examined, along with the basic philosophy of their development, their characteristics, and derived figures of merit. These models' strengths and weaknesses were evaluated, and ways to validate the models were identified (MacGregor 79).

A user-oriented reliability figure of merit was defined to measure the reliability of a software system with respect to a user environment. The effects of the user profile, which summarizes the characteristics of the users of a system, on system reliability was considered. A simple Markov model was formulated to determine the reliability of a software system based on the reliability of each individual module and the measured inter-modular transition probabilities as the user profile. Sensitivity analysis techniques were developed to determine modules most critical to system reliability. The applications of this model to develop cost-effective testing strategies and to determine the expected penalty cost of failure were examined, and future refinements and extensions of the model proposed (Cheung 80).

Study Name: **Software Complexity Model Studies**

Contractor: Northwestern University
Contract #: F30602-80-C-0139

Tech Report:
- RADC-TR-83-262, Methodology for Software Maintenance

Related Pubs:
- A Method for Estimating the Execution Time of Arbitrary Paths in Programs
- Design Stability Measures for Software Maintenance
- An Approach to Incremental Program Modification
- An Evolution Model for Software Maintenance
- An Integrated Life-Cycle Model for Software Maintenance

Further research was performed under this second contract in the areas of developing effective
methodologies for software maintenance. This effort focused on refining, expanding and automating the software maintenance concepts and techniques developed under the previous contract. Topics addressed in the effort included techniques for specifying and realizing software modification proposals, logical ripple effect analysis and module revalidation after modification. These techniques and the performance ripple effect analysis technique developed during the previous contract period were demonstrated using a DEC VAX 11/780 computer. In addition, a number of software metrics related to modifiability, such as measures for logical and performance stability, module strength and coupling, were developed (Yau 84).

One measure of program performance is the execution time of the program. A technique based on a self-metric approach for estimating the execution time of program paths was developed. Estimates were obtained for each of the operations of a programming language. A program was then used to analyze the program to be measured, by inserting additional program instructions to obtain statistics regarding the execution time. This work was done to assist in the analysis of performance ripple effect during program modification. In this application, information may be needed about each execution of specific paths with critical timing constraints. The particular paths to be measured, and the type of statistics to be provided are determinable by the user. This technique was implemented and used for experiments with PASCAL programs running on a DEC VAX 11/780 computer (Carvalho 81).

An approach to incremental program modification, which assists the programmer to modify only the relevant portion of a program, was developed in which a syntax-directed program slicer was used to locate the sections of code to be modified and a syntax-directed program editor was used to perform the modifications. The integration of these tools was achieved by sharing a formal representation of the program, based on the Backus-Naur Form (BNF) notation and representing both the executable statements and the symbol tables of programs as trees with certain semantic links. This approach is applicable to various block-structured, procedural programming languages, such as Pascal, Algol 60, PL/I and Ada, and is useful for programs under development or maintenance. This approach was demonstrated using Pascal on a DEC VAX-11/780 computer (Chang 83).

A software life cycle model was developed for use in a software maintenance environment to represent information about the development and maintenance of software systems. It emphasized relationships between different phases of the software life cycle, and provided the basis for automated tools to assist maintenance personnel in making changes to existing software systems. This model is independent of particular specification, design and programming languages because it represents only certain basic semantic properties of software systems: control flow, data flow, and data structure. The software development processes by which one phase of the software cycle is derived from another are represented in the model by graph rewriting rules, which indicate how various components of a software system have been implemented. This modeling approach permits analysis of the basic properties of a software system throughout the software life cycle (Nicholl 86, Liu 88).
A software metric is a "meaningful measure of the extent or degree to which an entity possesses or exhibits a particular characteristic" (DACS 79b). Metrics have now been defined to measure almost any conceivable attribute of software. In the beginning, attributes describing productivity, reliability and cost received the most attention. As more systematic approaches to software measurement were developed, however, such individual metrics were no longer sufficient. For example, it became apparent that lifecycle costs are often raised by uncontrolled non-functional attributes like reliability, maintainability, and testability. The ability to measure software quality required first the development of an understanding of what is meant by software quality. By creating a usable system of metrics, it would be possible to get a more complete picture of the characteristics of a software product, and therefore assess its overall quality.

Even today, despite the abundance of metrics for measuring individual aspects of software quality, there are few comprehensive frameworks that treat all aspects of software quality. Those now in existence have been strongly influenced by the Rome Laboratory Software Quality Framework (RLSQF). The RLSQF was created by a RL-sponsored project performed by Gene Walters and James McCall, then with General Electric (McCall 77b). It was based on previous research performed by Dr. Barry Boehm.

Boehm's study, performed for the National Bureau of Standards' Institute for Computer Sciences and Technology, sought to identify a set of characteristics of quality software and, for each characteristic, to define a metric such that: 1) Given an arbitrary program, the metric provides a quantitative measure of the degree to which the program has the associated characteristic, and 2) Overall software quality can be defined as some function of the values of the metrics (Boehm 73b).

The RLSQF is a hierarchical definition of factors affecting software quality. The factors are composed of criteria, which are composed of questions. The questions are the metrics. The list of factors and criteria, and some examples of the metrics are included in the Appendix. An example of the relationship between a quality factor and its criteria is shown in Figure 2. By answering all the questions, a software engineer creates a complete set of data for a software project. The metrics are designed to quantify and objectify the desired attributes, so that anybody, including an automated tool, would give the same answers to the questions for the same piece of software.

The framework provides a structure for identifying characteristics and attributes of software, and defining metrics which can be used to quantify them. The framework supports research in model development by increasing and standardizing the data available for validating and applying the models. The framework itself has often been referred to as a software quality model, but its primary application and contribution has been in defining software quality metrics.
By defining metrics for software quality, the RLSQF defines what is meant by quality software. Therefore, one use of the RLSQF has been to provide early management indicators of software quality.

Another significant application of the framework is in the specification of quality requirements for software acquisitions. Traditionally, software development managers were motivated to track and control development costs and similar factors, and were less likely to be concerned with long-term issues and consequences, such as re-use and maintenance. This tendency was especially acute in contracting environments, when the organization responsible for the system development is not the same as the organization responsible for the Post Deployment Software Support (PDSS). Why spend extra resources during development on things that will provide no benefit to the developer? From an acquisition program officer's point of view, however, the need to minimize complete lifecycle costs creates a willingness to invest resources early in the lifecycle to achieve a payoff of reduced maintenance costs and/or of having software that can be re-used in future system developments. The RLSQF provides language, tools, and guidance to support an acquisition manager in specifying quality factor goals which can reduce overall lifecycle costs.

Rome Laboratory has sponsored numerous efforts related to the definition, refinement and application of the RLSQF, which are catalogued below. The Laboratory continues to sponsor research efforts to refine, apply and automate the RLSQF metrics.

The early work in defining the RLSQF was accompanied with promotion of the approach. Gene Walters and James McCall presented this work in several forums, receiving feedback from several audiences. These presentations included:

- A panel session at the second NASA/SEL Software Engineering Workshop (McCall 77a)
- The 1978 Software & Quality Assurance Workshop (Cavano 78), sponsored by the ACM Special Interest Group on Software
- Engineering (SIGSOFT)
- The Annual Reliability and Maintainability Symposium (Walters 78)
- IEEE Transactions of Reliability (Walters 79)

The paper presented at the Reliability and Maintainability Symposium won the P. K. McElroy Award for best paper.

Although rarely adopted unchanged in its entirety, the RLSQF has been widely influential even outside of RL sponsorship. A small selection of organizations adopting quality measurement techniques based on the RLSQF includes:
- Metriqs, a consulting firm headed by Mr. Gerald Murine
- Air Force Operational Test and Evaluation Center (AFOTEC)
- Air Force Quality Indicators defined by Headquarters Air Force Systems Command (AFSC 87)
- Hughes Aircraft (Deutsch 88)
- The Army STEP program (DA 92)

Inclusion of the framework in the STARS Software Quality Evaluation Data Collection Form (DCF) also generated much feedback from the software engineering community during the STARS Measurement DCF development task review process (IITRI 85).

This section focuses on quality metrics for two reasons. Most of the efforts involving metrics research at RL have been concerned with defining and measuring software quality, and are therefore related to the RLSQF. The second reason is that the framework encompasses most other attributes for which metrics have been defined. Software complexity measures, for example, are after the same software characteristic as the framework criterion simplicity, (although in the inverse). Simplicity is a criterion shared by the reliability, maintainability and testability factors.

A catalogue of Rome Lab-sponsored efforts related to the framework
A History of Software Measurement at Rome Laboratory

A catalogue of Rome Lab-sponsored efforts related to the framework follows:

Study Name: **Factors in Software Quality**

This first study resulted in the definition of the framework: the quality factors, the criteria, and metrics for measuring the criteria. The hierarchical definition of factors affecting software quality was compiled after an extensive literature search. The definition covers the complete range of software development, broken down into non-software-oriented and software-oriented characteristics. For the lowest level of the software-oriented factors, metrics were developed that would be independent of the programming language. These measurable criteria were collected and validated using actual Air Force data bases. This effort also included development of a handbook for use by Air Force acquisition managers in specifying the overall quality of a software system (McCall 77b).

Study Name: **Software Quality Metrics Enhancements**

This effort was initiated in response to RADC TP05, Software Cost Reduction, in the area of Quality Measurement. A significant result of the study was the verification that the metrics developed for Air Force Command and Control applications were also applicable to Army Management Information System applications. The software metrics which predict software quality were refined and enhanced. The metrics were classified as anomaly-detecting, predictive, or acceptance metrics. The purpose of the study was to develop a complete set of procedures and guidelines for introducing and utilizing then-current software quality measurement techniques into a quality assurance program associated with large scale software system developments. These procedures and guidelines identify: how to identify and specify software quality requirements; how and when to apply software metrics; and how to interpret the information obtained from the application of the metrics (McCall 80).

Problems encountered in defining software quality and the approach taken to establish the framework for the measurement of software quality were also documented as a result of this effort (Cavano 78).

Study Name: **Software Interoperability and Reusability**

In this effort, the software metrics which predict software quality were extended from the previous research for two quality factors: interoperability and reusability. Aspects of requirements, design, and source language programs which could affect these two quality factors were identified and metrics to measure them were defined. These aspects were identified by theoretical analysis, literature search, interviews with project managers and software engineers, and personal experience. A Guidebook for Software Quality Measurement was produced to assist in setting quality goals, applying metrics and making quality assessments (Bowen 83a). The metrics for predicting interoperability were validated using data from three aerospace projects (Presson 83).

Study Name: **Quality Metrics for Distributed Systems**
The purpose of this Quality Metrics for Distributed Systems effort was to provide methodology and technical guidance on software quality metrics to Air Force Software acquisition managers. The objective was to conduct exploratory development of techniques to measure system quality with a perspective on both software and hardware from a life cycle viewpoint. The effort was to develop and validate metrics for software quality on networked computers and distributed systems; i.e., systems whose functions may be highly distributed over microprocessors or specialized devices such as database machines. At the same time, the effects hardware has on software were studied, as were the trade-offs between hardware, firmware, and software. This effort applied the quality metrics to distributed systems and developed a guidebook for AF acquisition managers based on the study results which provides guidance for specifying and measuring the desired level of quality in a software product. This effort also included a qualitative study of distributed system characteristics, reasons for selection, design strategies, topologies, scenarios, and trade-offs. The study led to the changes in the framework, and to the validation of models (Bowen 83b).

The framework was reviewed as a theoretical model for measurement of Software Quality from a system perspective, and found to require modification through addition of new quality factors, criteria, metrics and weighting factors. The optimal degree of software quality is unique to a given product. It can be derived from analysis of consumer demand for the product and the cost of developing the desired quality factor profile in a product. The existing RLSQF quality factors were classified as consumer oriented and software production oriented. The consumer oriented factors are incorporated in a product and the production oriented factors can be used to predict the eventual consumer oriented quality of the software. Examples were developed that showed that even though the measurement framework accurately measures and predicts software quality, it may not be adequate to select the best among competing subcontractors because the software development approaches are not comparable (Lawler 81).

Study Name: Specification of Software Quality Attributes

This effort resulted in recommendations for integrating the RLSQF into the Air Force software acquisition management process and for changing Air Force acquisition documentation. This effort also proposed changes to the baseline framework and defined a specification methodology. This study investigated how a software acquisition manager would specify software quality requirements to be consistent with needs. Factor interrelationships and tradeoffs among factor quality levels in terms of relative costs were considered. An example for a command and control application was developed. The effort included development of procedures and techniques for evaluating achieved quality levels, including worksheets for use in metric data collection by software life cycle phases and scoresheets for scoring each factor. Procedures for assessing compliance with the specified requirements based on an analysis of data collected by these methods were defined as well (Bowen 84).

Study Name: Software Quality Measurement Methodology Enhancements Study

This effort consisted of four tasks: a critical analysis of the RADC Software Quality Measurement Methodology; determination if AFSC Management and Quality Indicators could be integrated into the RADC methodology; review of the completeness of Methodology Traceability through software life cycle phases; and creation of a cross reference guide between Methodology lifecycle phase metrics and applicable DOD-STD-2167A DID paragraphs. Results included recommendations that the specification process should be modified, because of the complexity of the methodology. Three of the AFSC Quality Indicators were found to be integrable into the framework. Traceability was found to be present, in
general. A cross reference matrix between the RLSQF and DIDs was developed; however, many problems were identified, and recommendations were made to alleviate them (Lasky 90).

Study Name: **Quality Metrics Study**

The objective of this effort was to demonstrate the application of software quality metrics to measure and predict the software quality factor portability. A portability experiment was designed and implemented to transport an Automated Metric Tool from the RADC HIS 6180 computer to a micro computer. Metrics data on the conversion and enhancement process were collected to perform the software quality analysis (IITRI 81).

Other relevant metrics research that incorporated the RLSQF includes work done in support of the Measurement Area in the original Software Technology for Adaptable, Reliable Systems (STARS) program. The STARS program included measurement as one of its six research thrusts, recognizing that measurement has practical benefits as a management, development and contractual tool for both developers and maintainers, and that measurement also supports scientists in the development of quantitative models. The STARS program sought to advance measurement technology and to facilitate its use among a wider group of software practitioners (Dunham 83).

Study Name: **STARS Measurement Tasks**

The Data Collection Forms Development task resulted in the definition of six forms for collecting different aspects of software measurements. The RSLQF forms the basis of the Software Evaluation Report. The other forms in this set are the Software Resource Expenditure Report, the Software Characteristics Report, the Software Test Information Report, the Software Problem/Change Report, and the Software Environment Report (IITRI 85).

Additional research for the STARS Measurement Area included a survey of the software community to identify existing measurement data bases, models and tools (Sunderhaft 86).
2.3 DATA

As increasing costs of software development fostered research in software methodology, project productivity and program reliability, the lack of standard, reliable data for an adequate sample of software projects upon which to base conclusions became acute. Accurate, valid data is needed to provide credible evaluations of proposed innovations in software development methodology, and to provide deeper insight into the software development process.

Early RL-sponsored work in data collection was in direct support of reliability model development research. Rome Laboratory proposed the establishment of a repository in which software productivity, cost and reliability data could be accumulated. The recognition of software data collection as a more general research subject, along with the development of more general metrics, led to more comprehensive data collection efforts.

The software data research efforts catalogued in this section are grouped into three categories: those concerned with definition and initial operation of the DACS, those concerned with developing tools for data collection, and those that actually collected data.

2.3.1 The Data & Analysis Center for Software

The creation of the Data & Analysis Center for Software by RL stemmed from the recognition of the data problems associated with software data collection, and the determination to attack those data problems which could be solved. Several of these issues were identified and addressed by the software data collection study that preceded the founding of the DACS. They were:

- Measurement problems, including definition of useful measurements, instrumentation issues, and interpretation of the data
- Instrumentation effects, where the act of collecting data affects the data collected
- Unreliability of measures, due to lack of standardization
- Reluctance to release data, especially performance (by workers) or cost (by management)
- Cost of data collection, when no quantifiable benefit is apparent
- Other problems, including such things as time delays, filtering and summation effects, forecasting inefficiencies, and instability of the project being measured

The Data & Analysis Center for Software (DACS) was chartered by RADC ISIS in 1976. The detailed plans and objectives for the DACS were developed by RADC from the results of two parallel efforts in 1975 and 1976.
2.3.3 Data Collection Studies

At the time the DACS was chartered, acquisition of software data for the repository was extremely
difficult. Despite the survey results which showed agreement among software engineering professionals
about the need for the repository, and their professed willingness to support it, there was very little data
in existence that could be deposited at the DACS. Meanwhile, RL also needed reliability (error) data for
on-going reliability research. RL therefore sponsored five data collection efforts to collect software error
data, which was incorporated in the DACS SLED. The TRW Reliability Study listed in the Models
Section of this paper was one of these efforts, and provided the error classification scheme which became
the de facto standard. The other four efforts are catalogued below:
A History of Software Measurement at Rome Laboratory

Section 2.2 Studies

Factors in Software Quality

Dates: 8/76 - 7/77
Contractor: General Electric Company/Command & Information Systems
Contract #: F30602-76-C-0417
Tech Report:
  ● RADC-TR-77-369, Vol I - III
  ● Concepts and Definitions of Software Quality
  ● Metric Data Collection and Validation
  ● Preliminary Handbook on Software Quality for an Acquisition Manager

Related Pubs:
  ● Metrics for Software Quality Evaluation and Prediction (McCall 77a)

Software Quality Metrics Enhancements

Dates: 6/78 - 7/79
Contractor: General Electric Company/Command & Information Systems
Contract #: F30602-78-C-0216
Tech Report:
  ● RADC-TR-80-109, Vols I & II
  ● Software Quality Metrics Enhancements
  ● Software Quality Measurement Manual

Other Pubs:
  ● A Framework for the Measurement of Software Quality

Software Interoperability and Reusability

Dates: 1980 - 1983
Contractor: Boeing Aerospace Company
Contract #: F30602-80-C-0265
Tech Report:
Quality Metrics for Distributed Systems

Dates: 1980 - 1983
Contractor: Boeing Aerospace Company
Contract #: F30602-80-C-0330
Tech Report:
- RADC-TR-83-175, Vols. I - III
- Software Quality Measurement for Distributed Systems
- Guidebook for Software Quality
- Distributed Computing Systems: Impact on Software Quality
Other Pubs:
- System Perspective on Software Quality

Specification of Software Quality Attributes

Dates: 1982 - 1984
Contractor: Boeing Aerospace Company
Contract #: F30602-82-C-0137
Tech Report:
- RADC-TR-85-37, Vols. I - III
- Specification of Software Quality Attributes
- Software Quality Specification Guidebook
- Software Quality Evaluation Guidebook

Software Quality Measurement Methodology Enhancements Study

Dates: report date 1/90
Contractor: Rochester Institute of Technology
Contract #: F30602-81-C-0193
Tech Report:
Quality Metrics Study

Date: 1981
Contractor: IIT Research Institute
Contract #: F30602-81-C-0280
Related Pubs:
- Quality Metrics Study

STARS Measurement Tasks

Dates: 1984 - 1986
Contractor: IIT Research Institute
Contract #: F30602-83-C-0026
Related Pubs:
- Interim Software Data Collection Forms Development
- STARS Measurement Survey Summary
The detailed plans and objectives for the DACS were developed by RADC from the results of two parallel efforts in 1975 and 1976.

Study Name: **Software Data Collection Study**

This study focused on issues surrounding the collection of software data. The relevant RADC TPO was 3.V.A.1.4, Quality Control. This effort succeeded in defining, classifying and categorizing data parameters in support of productivity, reliability and cost analyses, and in developing data collection forms that could be applied to diverse software projects and could evolve along with metric and model research results.

This study was conducted to generate recommendations concerning a data collection system for the software data repository that would form an historical data base to support research and analyses requirements within RADC. The objectives of the study were to investigate: (1) Data collection problems; (2) Data requirements for productivity, software reliability and cost studies; (3) Data entry/data management interface; and (4) Specifications for a software data collection and reporting system (Finfer 76).

In fulfillment of the Data Collection Study objective to recommend a set of parameters to be collected for the RADC Software Data Repository, past and present data collection systems and the results of a survey of the literature were examined and data parameters from contract award to software system installation were identified. A classification of data parameters was created from examination of significant software development factors. The first class consists of project environment factors including contract and customer relations data, organizational and personnel characteristics, hardware and support facilities parameters, and overall attributes of the software product itself, such as size, complexity, etc. The second class of data is project performance information reflecting the amount and quality of work performed for the duration of the project period. Class three data consists of automatically generated product measurements, which demonstrate the structure and behavior of the product through the application of analysis tools and testing procedures.

Another result of this effort was the development of recommended data collection forms, along with a suggested data base structure, and a summary list of definitions of the data parameters. A modular design approach was taken to the construction of the forms and the suggested data base structure. Data item priorities, collection frequency, and project size and complexity requirements were taken into account in defining modules. The system was deemed flexible enough to meet the needs of future changes and expansions to the RADC repository operations.

Study Name: **Software Data Repository Study**

This study provided a definition of and design for a Software Data Repository, which was implemented as the DACS. The effort was "part of a program, the goal of which is the implementation of a central repository for data and technical information for computer software technology and engineering that will
provide tools and facilities to store, analyze, and disseminate authoritative scientific and technical information concerning all aspects of computer software and the software development process" (Duvall 76, p. viii) and places the effort in concert with the RADC Technology Plan's TPO 11, Software Sciences Technology.

In terms of measurement, the primary objective of the DACS was to provide a source of historical data on the production of software. The data would be made available for use by software development researchers in the construction and validation of models and in the evaluation of alternative methodologies and tools, and would, through the collection, analysis, and dissemination of software development experience, help upgrade the software development process.

The repository was defined in terms of inputs, processes, and outputs. The input processing and the requirements of an information system for storing and processing the data was examined, and recommendations for the repository including the development and operation of a pilot facility and the expansion of this facility into a fully operational center were made.

The plans and objectives for the repository were publicized within the software community. At the 1977 IEEE Computer Software Applications Conference, emphasis was placed on how the repository would serve the government/university/industrial community as a focal point for the acquisition, analysis, and dissemination of software experience information (Duvall 77a).

The results of the survey on the exchange of software development experience information were presented at the 1977 Computer Related Information Systems Symposium. The survey indicated that members of the software community were interested in the establishment of a center for information sharing and were willing to share their experiences and data on developing computer software. Interest was shown in the results of analysis on determining the effects that different developmental and testing philosophies have on the cost and productivity of software development (Duvall 77b).

Study Name: **Baseline Software Data System**

This effort resulted in the definition of a Baseline Software Data System for the storage and retrieval of data collected in several software error data collection efforts and used for reliability modeling. Also implemented under this effort were a summary database and a software productivity database (Duvall 79). As more data became available, this system was expanded, and became the DACS Software Lifecycle Empirical Database (SLED).

Study Name: **Data & Analysis Center for Software Pilot**

The pilot study for implementing the DACS began in August 1978. When fully implemented and operational, the DACS was to provide a centralized source for current and readily-usable data and information concerning software technology. During the initial development of the DACS, descriptions of the software engineering computer database and the technology information base were developed. Other functions of the DACS included development of products and providing responses to technical inquiries relating to software technology (Duvall 80, Caron 81).

The DACS studied and assessed the state-of-the-art in software engineering data collection. As a result of the needs recognized for improved data collection and analysis, a role for the DACS in establishing realizable guidelines or standards for software engineering data collection was defined (Gloss-Soler 80).
The DACS planned a series of data publications dealing with software development data and its use to predict software cost and reliability in future software design, development, testing and maintenance efforts. The first of these provided specific information on 29 software development projects monitored by NASA/SEL during the 1976-1977 time frame. This information included: descriptions of data collection techniques, statistical and graphical data summaries, analysis of data quality and completeness and a discussion of the potential application of the data in statistical and modeling studies. This Data Compendium does not include exhaustive listings of dataset contents; these kinds of listings are available from the DACS in hard copy or machine readable format (Brement 82).

As the DACS gained more experience with data collection, and in monitoring the state of the art in software engineering, the need for data to support more than just productivity, cost and reliability research became apparent. Software conversion, for example, was a relatively new area of interest to software engineers but one that promised to present a whole new range of problems.

The economics and logistics involved with software inventory conversion were to become major procurement issues. In many cases the cost of the software conversion equals or exceeds the cost of the new hardware. When a large multi-site hardware upgrade is involved, conversion can take years and require a sizable dedicated programming staff to complete. Little was known about the types of problems likely to occur during a conversion, the frequency with which they occur, their severity, and how they may be corrected or prevented. By collecting data on past and present conversion efforts, the DACS proposed to compile such information which would be used for identifying problems and their demographics. A database of conversion data would help to conduct feasibility studies, estimate costs for performing conversions, identify conversion cost drivers, and help to establish cost-benefit relationships for conversion aids and tools. The DACS sought to establish such a database which would have been available to the software engineering community, and defined data collection forms to be used during software conversion (DACS 81).

**Study Name: Data & Analysis Center for Software Transition**

The DACS was operated on a pilot basis for its initial 36-month period, then entered a 15-month transition period. During the transition period, products and services were provided to the software engineering community. The Software Technology Information databases and the Software Lifecycle Empirical Database were maintained and enhanced (Gloss-Soler 83).

During this period the DACS continued to publicize its efforts to acquire software data for all phases of the software life cycle. A 1982 issue of the DACS Bulletin contained a review of data collection as applied to software development, discussed reasons for and advantages of developing a data collection methodology for software maintenance, and reviewed tools and techniques that could be utilized for data collection during software maintenance (Cornwell 82).

The DACS acquired seven sets of data collected from completed software developments during its initial operation periods which is maintained in the DACS in the Software Lifecycle Empirical Database. The subsets of the data available to users were: software reliability data, software data from avionics projects, data collected under several independent verification and validation contracts, error data set, operations and maintenance data, and baseline software data set representing six defense software projects (Turner 82).

The statement of work for each of the DACS contracts has contained tasks that specifically relate to
measurement. One requires the DACS contractor to maintain and augment the SLED. Another core DACS task is to perform analyses of the SLED data, and document the results in a series of data analysis reports. Additionally, the DACS provides access to SLED data in the form of standard and customized datasets as one of the many products and services available to DACS users.


A History of Software Measurement at Rome Laboratory

2.3.2 Tools for Data Collection

From the beginning, automation has been seen as a way to both cope with the burden imposed by data collection and increase the consistency of the data items. Many RL efforts have been directed at developing software tools to help automate the data collection process. Automation and tool support for data collection is still a topic of research at Rome Lab. Early research in development of tools to support software data collection is catalogued here:

Study Name: **Data Management System Testing and Methodology Validation**
Dates: 1971 - 1972
Contractor: PRC Information Sciences Company
Contract #: F30602-71-C-0318

This effort involved measurement of the performance of two data management systems. The intent of this effort was to validate a Test Methodology by actually employing the recommended measurement techniques. Additionally, it was expected that this exercise would illustrate the strengths and weaknesses of the selected techniques and, perhaps, further refine the developed test methodology. The techniques selected included analysis, numerical scoring, benchmark programs and software monitors. The software monitor technique could not be employed to measure one of the selected systems because its documentation was inadequate. The techniques evaluated were found to be valid when employed in a manner consonant with their aims. Software monitoring was found to be worthy of additional development (Ferguson 72).

Study Name: **Imbedded Software Monitors and Data Collection**
Dates: 4/73 - 4/74
Contractor: PRC Information Sciences Company
Contract #: F30602073-C-0198

The contractor developed a set of data collection programs to provide operational data about a Data Management System (DMS) in the General Comprehensive Operating System (GCOS) environment. The programs (software monitors) collected data on response times, input/output times, job throughput and frequency of DMS module use. Additionally, the contractor developed analysis procedures to reduce the data so that the monitored system could be evaluated, compared to other systems, or fine-tuned for better performance. The contractor developed the software monitor package to search the core image of user program modules at execution time and find calls to subroutines that are to be monitored. The original calls to the subroutines were replaced by calls to the monitor software which collected relevant
data before and after each DMS routine executed before final control was passed back to the user program. The software monitoring tool was designed to be independent of the systems within the environment and security transferable for monitoring other systems (Navarro 74).

Study Name: **Static FORTRAN Analyzer**  
Date: 1975  
Organization: Rome Laboratory  
Tech Report: RADC-TR-75-275  
- Static FORTRAN Analyzer

The National Bureau of Standards Static FORTRAN Analyzer, which samples FORTRAN programs and collects statistics on the utilization of predetermined FORTRAN syntactic constructs, was adapted to operate under the FORTRAN-Y compiler of RADC's GCOS operating system. After the conversion, a subsequent analysis of 258 sample programs, consisting of approximately 22,000 lines of source code, was performed. The statistical results of this effort provided relevant to concurrent FORTRAN language study and standardization efforts which address language and compiler design, optimization, and subsetting (Slavinski 75).

Study Name: **A BASIC Statistics Collector**  
Date: 1976  
Organization: Rome Laboratory  
Tech Report: RADC-TR-76-9  
- A BASIC Statistics Collector

The Beginner's All-Purpose Symbolic Instruction Code (BASIC) is an algebraic problem oriented programming language. This project designed and implemented a tool that could be used to collect and analyze data that would help identify the manner in which the BASIC language is used by programmers in writing programs. The effort considered the types of data collected, the method in which the data was collected, current analysis methods, and data sources (White 76).

Study Name: **Software Implementation Monitor**  
Date: 1976  
Contractor: MITRE Corporation  
Contract #: F19628-76-C-0001  
Tech Report: RADC-TR-76-288  
- Specifications for SIMON, A Software Implementation Monitor

Specifications were defined and a prototype developed for SIMON, a tool that would provide technical and managerial visibility during software development. Possible extensions of the prototype were also planned, along with some uses of the system (Corrigan 76).

Study Name: **A Statistical Collection Package for the Jovial J3 Programming Language**  
Date: 1977  
Organization: Rome Laboratory  
Tech Report: RADC-TR-77-293  
- A Statistical Collection Package for the Jovial J3 Programming Language
RADC developed a software package in-house to measure counts, averages, and percentages relative to the usage of constructs and features of the JOVIAL J3 high order language by programmers. The numbers of occurrences of certain language features are obtained by processing the input JOVIAL J3 sources program in a manner similar to that employed by the front end of a compiler. This data is then used to calculate other quantities, averages, and percentages. Hash coding identifiers in the input program and any associated compools, and a symbol table in which information about these identifiers is recorded are used to assist in the statistics collection process. This statistics collector is expected to be a valuable tool in the development of JOVIAL J3 and other programming languages by providing guidance relative to (1) more effective methods of programming; (2) implementation of compilers with greater efficiency; and (3) possible language changes. This effort also collected data on the syntax, semantics, and computer system interface errors made by the implementor in the process of development of the software package. The visibility provided by this information was expected to increase understanding of the nature, causes, and methods of avoidance of software errors (Stover 77).
Software Data Collection Study

Dates: 6/75 - 6/76
Contractor: System Development Corporation
Contract #: F30602-75-C-0248
Tech Report:
- RADC-TR-76-329, Vol I - VIII
- Summary and Conclusions
- Analysis of Software Data Collection Problems and Current Capabilities
- Data Requirements for Productivity and Reliability Studies
- Data Management System Interface
- Survey of Project Managers
- Proceedings of Data Collection Problem Conference
- Compendium of Procedures and Parameters
- Glossary

Software Data Repository Study

Dates: 6/75 - 8/76
Contractor: IIT Research Institute
Contract #: F30602-75-C-0257
Tech Report:
- RADC-TR-76-387, Software Data Repository Study

Related Pubs:
- The Design of a Software Analysis Center
- A Survey of the Software Community on the Exchange of Developmental Information
Baseline Software Data System

Dates: 2/77 - 8/78  
Contractor: IIT Research Institute  
Contract #: F30602-77-C-0052  
Tech Report:  
  ● RADC-TR-79-185, Vol I & II  
  ● System Description  
  ● Database Reference Manual

Data & Analysis Center for Software Pilot

Dates: 8/78 - 8/81  
Contractor: IIT Research Institute  
Contract #: F30602-78-C-0255  
Tech Report(s):  
  ● RADC-TR-80-204, Data and Analysis Center for Software  
  ● RADC-TR-81-385, Establishment of the Data and Analysis Center for Software

Related Pubs:  
  ● The Role of An Information Analysis Center in Software Engineering Data Collection  
  ● DACS Data Compendium Series: NASA/SEL Software Development Data  
  ● DACS Conversion Data Collection Forms

Data & Analysis Center for Software Transition

Dates: 1981 - 1983  
Contractor: IIT Research Institute  
Contract #: F30602-81-C-0280  
Tech Report: ]  
  ● RADC-TR-83-132, The Data & Analysis Center for Software: An IAC in Transition

Related Pubs:  
  ● Data Collection During the Software Operations and Maintenance Phase  
  ● The DACS Data Compendium
2.3.3 Data Collection Studies

At the time the DACS was chartered, acquisition of software data for the repository was extremely difficult. Despite the survey results which showed agreement among software engineering professionals about the need for the repository, and their professed willingness to support it, there was very little data in existence that could be deposited at the DACS. Meanwhile, RL also needed reliability (error) data for on-going reliability research. RL therefore sponsored five data collection efforts to collect software error data, which was incorporated in the DACS SLED. The TRW Reliability Study listed in the Models Section of this paper was one of these efforts, and provided the error classification scheme which became the de facto standard. The other four efforts are catalogued below:

Study Name: **Software Error Data Acquisition**

Date: 1977  
Contractor: Boeing Aerospace Company  
Contract #: F30602-76-C-0162  
Tech Report: RADC-TR-77-130

- Software Error Data Acquisition

Software error data was collected from a large DOD system development project. The errors were analyzed and put into a predefined set of categories. As part of the effort, the times to find and fix the errors were calculated, and the phase of the development project in which the errors arose was determined. Study results were also compared to results of a similar type of study performed by a second contractor who performed analysis of data from another DOD software project. Data recorded during this effort included descriptions of the hardware and software systems and the software development process. Also included in this effort was an investigation of categorization methods and the derivation of other contractually required data items (Fries 77).

Study Name: **Software Data Collection and Analysis**

Dates: 4/76 - 1/77  
Contractor: IBM Federal Systems Division  
Contract #: F30602-76-C-0161  
Tech Report: RADC-TR-77-192

- Software Data Collection and Analysis

This effort was initiated in response to a need for software error data. The work was in support of RADC TPO 5, Software Cost Reduction (formerly TPO 11, Software Sciences Technology). One objective was to be able to compare the error data acquired during this project with error data from other real-time ground based DOD software development projects. Another was to be able to contrast it with error data extracted from non-real-time projects. Both objectives were in support of software error prediction (reliability) model development (Baker 77).

Study Name: **Software Systems Reliability History**
Software Systems Reliability: A Raytheon Project History

This project involved collecting software data from the development records of a large Department of Defense ground-based system. Data recorded included a description of the subject system’s software development process, characteristics, tools, and test methods. Qualitative and quantitative data were gathered from configuration management files, and statistical summaries of this data were created. The study included recommendations for the use of the data as well as for the future collection of such data. The data consists of three files: (1) Module Description File; (2) Software Problem Report File; and (3) Error Category File. Each problem report was assigned an error category from the fault taxonomy and the data was cross correlated and summarized (Beauregard 77).

Software Systems Development History

Date: 1977
Contractor: Charles Stark Draper Laboratory
Contract #: F30602-76-C-0151
Tech Report: RADC-TR-77-213

Software Systems Development: A CSDL Project History

The contractor provided data to RADC for inclusion in the Software Data Repository. The data consists of a complete history of software modifications to the APOLLO on-board flight software for the period 1967 through 1971. The documentation delivered with the data included background material on the project that was the source of the data, tabular and graphic summaries of the data, and some recommendations for future work (Bamberger 77).

Software Data Baseline Analysis

Dates: 1978 - 1979
Contractor: General Electric Company
Contract #: F30602-78-C-0022

Software Data Baseline Analysis

This effort provided an analysis of software error data supplied by the Information Sciences Division of Rome Air Development Center. The data consisted of the software problem histories of five large-scale software developments, individually collected by the development contractors and supplied to RADC. The problems were classified by a previously developed error typology. The purpose of the analysis was to investigate the existence of any consistencies in the occurrence of errors utilizing the five development efforts. The analysis included consideration of the error typology, rate of occurrence, time of occurrence, time to fix, and module size. Results of the analysis isolate methodological problems in the gathering of software error data and suggest that positive incentives be provided to development team members involved in the data collection effort (Fish 79).
2.4 MODERN PROGRAMMING PRACTICES

Concurrent with early research in measurement technology, Rome Laboratory sponsored a series of efforts to first define and then evaluate what were becoming known as Modern Programming Practices (MPP). These efforts are included as part of this history of measurement technology research at RL because they depended on software data to be able to assess the effects of the practices.

The same dependency between methodology research and measurement technology exists today. The SEI model of process capability, for example, is completely dependent on measurement. Baseline measurements of existing practices are needed to determine where you are, before the effects of any improvement attempt can be quantified, and degrees of progress gauged.

2.4.1 The Structured Programming Series

Dijkstra's "GO TO" article (Dijkstra 68), which generated considerable controversy, was a major formative influence on the structured programming revolution, a result of which a whole generation of programmers learned new techniques for specifying, designing, implementing, testing, and maintaining computer software. C. A. R. Hoare, Donald Knuth, N. Wirth, and Harlan Mills were other influential early revolutionaries. A continual creation of new methodologies, tools, and programming languages has arisen since the early seventies, drawing on the ideas first formulated as part of structured programming. For example, Object Oriented Programming and formal methods, both issues of current interest, can be seen as outgrowths of structured programming (Vienneau 93).

So structured programming was viewed since its inception as an exciting idea among software developers. But what exactly is structured programming? To find out, RADC invited Harlan Mills to brief them on IBM's work on structured programming at the IBM Federal Systems Division. The awarding of a contract in February 1974 to IBM to define structured programming was an outgrowth of that briefing. The Air Force Systems Command, through RADC, sponsored IBM to develop a detailed set of guidelines for how to implement structured programming in a comprehensive environment. The U. S. Army Computer Systems Command soon offered an amendment and co-sponsorship to this contract (Naughton 75).

The result of this contract was a fifteen volume report, the Structured Programming Series, RADC-TR-74-300. This work took an expansive view of Structured Programming Technology. Structured Programming itself, the restriction during design and implementation to the three programming constructs of sequence, alternation, and iteration, was considered only one component of structured programming. Other important elements included language design, the design and use of precompilers, Top Down Programming, the Program Support Library, Program Design Language, and Chief Programmer teams. Management reporting, cost estimating, Verification and Validation, and software tools were also studied in the definition effort.

This study was very influential, not just at RADC or throughout the Air Force, but among software
researchers and practitioners in general. This set of reports provided the definition of Structured Programming for many. No textbook presentation of Harlan Mills' ideas on structured programming was available until 1979 (Linger 79), and even that book does not discuss the broader issues of the environment in which structured programming operates, and the issues involved with managing teams who are applying structured programming techniques, that are considered in the RADC study.

Study Name: Structured Programming Series
Dates: 1974 - 1975
Contractor: IBM Federal Systems Division
Contract #: F30602-74-C-0186

- VOL I: Programming Language Standards (Kessler 75a)
- VOL II: Precompiler Specifications (Tinanoff 75)
- VOL III: ANS COBOL Precompiler Program Documentation (Campbell 74)
- VOL IV: Data Structuring Study (Trimble 75)
- VOL V: Programming Support Library (PSL) Functional Requirements (Luppino 74a)
- VOL VI: Programming Support Library (PSL) Program Specifications (Luppino 74b)
- VOL VII: Documentation Standards (Ortega 74)
- VOL VII - Addendum: Documentation Standards (Bennett 74)
- VOL VIII: Program Design Study (Kraly 75)
- VOL IX: Management Data Collection and Reporting (Smith 74)
- VOL X: Chief Programmer Team Operations Description (Barry 75)
- VOL XI: Estimating Software Project Resource Requirements (Smith 75a)
- VOL XII: Training Materials (Kessler 74)
- VOL XIII: Final Report (Naughton 75)
- VOL XIV: Software Tool Impact (Kessler 75b)
- VOL XV: Validation and Verification Study (Smith 75b)

In addition to defining modern programming practices, two of the volumes addressed measurement-related issues, thus providing further support for measurement research efforts. By including measurement in the list of recommended practices, progressive software development organizations were encouraged to collect and use software data.

Volume IX, Management Data Collection and Reporting (Smith 74), contained the results of analysis performed on the question, "What data should be collected on programming projects, and how should it be reported to assist management planning, controlling and directing programming activities?" The question was applied to system development phases - definition, design, implementation and evaluation. Related questions of how to collect the data required for project management, including project cost data, and what is the cost of collecting this data, were also addressed.

Volume XI, Estimating Software Project Resource Requirements (Smith 75a), contained results of an analysis of techniques currently used for estimating software resource requirements and addressed the effect structured programming technology would have on these techniques. Techniques addressed
covered all phases of software development - definition, design, implementation and evaluation. The two major conclusions of the study were: there was no widely accepted methodology for estimating resource requirements for a software development project; and structured programming technology would not introduce new techniques for estimating but would support the mechanisms for obtaining a base of historical data to provide better estimates due to the creation of programming support libraries.
2.4.2 MPP Evaluation Studies

Given the results of the MPP definition study, RADC/ISIM sponsored a set of six efforts at different firms to assess the effects of modern programming practices on software system development projects. The relevant RADC program was TPO V/3.4: "to describe and assess software production and management tools and methods which significantly impact the timely delivery of reliable software."

Study Name: **Software Production Data**
Dates: 2/76 - 3/77
Contractor: Computer Sciences Corporation
Contract #: F30602-76-C-0163
Tech Report: RADC-TR-77-177

- Software Production Data

The objective of this research task was to assess the effects of modern programming practices on selected CSC software system development projects. Specific MPPs used on the selected projects were identified and defined. After the assessment was completed, the identified practices were compared with the techniques described in the IBM Structured Programming Series reports (Donahoo 77).

Study Name: **Modern Programming Practices Study**
Date: 1977
Contractor: Sperry Corporation
Contract #: F30602-76-C-0136
Tech Report: RADC-TR-77-106

- Modern Programming Practices Study Report

This effort showed that for the surveyed program developments: (1) Top-down program development resulted in a cost savings of 15 percent. (2) Regardless of the method for development used labor was generally distributed as follows: (a) 10 percent for analysis; (b) 30 percent for design; (c) 35 percent for coding and debugging; and (d) 25 percent for testing. (3) Program testing was automated using real-time, on-line simulation, scenario control, data recording, and reduction. (4) Software quality assurance and configuration management were formal disciplines. (5) Through system design, operating system software provided real-time, on-line casualty recovery of system functions from hardware failures. (6) Through system design, operating system software allowed for on-line maintenance testing of hardware equipments while continuing real-time operation. This study also evaluated the effectiveness of software tools used on the surveyed programs (Branning 77).

Study Name: **Impact of Modern Programming Practices on System Development**
Date: 1977
Contractor: TRW
Contract #: F30602-76-C-0095
Tech Report: RADC-TR-77-121
Impact of Modern Programming Practices on System Development

This effort evaluated the impact of modern programming practices (MPP) applied to TRW's Ballistic Missile Defense Systems Technology Program software development. Resulting information included a reconstructed chronology of significant events, a description of the software development environment, and details of the process used to identify, select and define appropriate practices to be evaluated. The objectives of the study were: to investigate the impact of the selected MPP on software development cost, schedule and quality; and to define and apply techniques and tools that could provide a means for comparing TRW practices with those used by other contractors on other projects (Brown 77).

Study Name: **BCS Software Production Data**
Dates: 1976 - 1977
Contractor: Boeing Computer Services
Contract #: F30602-76-C-0174
Tech Report: RADC-TR-77-116

The purpose of this study was to assess the impact of modern software development techniques on the cost of developing computer software. The five in-house projects selected for study varied in size type of application, and computing environment. The collection of practices found to have the most beneficial impact on software development were, in order of their impact: Project Organization and Management Procedures, Testing Methodology, Configuration Management and Change Control, and Design Methodology. Existing military standards and specifications were found to be sufficiently comprehensive to encourage the use of beneficial practices; however, modification of certain standards and specifications would make their applicability to software procurements more pertinent (Black 77).

Study Name: **Impact of Structured Programming Standards on Small Government Contractors**
Date: 1977
Contractor: W. W. Gaertner Research, Incorporated
Contract #: F30602-76-C-0390
Tech Report: RADC-TR-77-162 Vol I & II

The purpose of this effort was to investigate the impact of structured programming standards on small government contractors, to determine how they could reasonably comply with such requirements within the normal environment of the small contractor (Gaertner 77).

Study Name: **PAVE PAWS Modern Programming Data Collection System**
Dates: 1977 - 1979
Contractor: Raytheon Corporation
Contract #: F30602-77-C-0141
Tech Report: RADC-TR-79-137

As part of the overall PAVE PAWS development effort, RL requested documentation of the software development technologies used on the PAVE PAWS project, an assessment of the effectiveness of those
techniques, and descriptions of the techniques which were implemented to collect data to support on-going independent technology studies (Hall 79).

Study Name: **Advanced Systematic Techniques for Reliable Operational Software**  
Dates: 1977 - 1980  
Contractor: General Electric Company  
Contract #: F30602-77-C-0194  
Tech Report: RADC-TR-80-6Vol I & II  
- A Matched Project Evaluation of Modern Programming Practices  
- RADC-TR-80-28, Evaluation of Software Life Cycle Data from the PAVE PAWS Project

Related Pubs:  
- Measuring Software Development Methodologies  

The matched project evaluation effort was part of a jointly sponsored RL and Space and Missile Test Center plan for improving software development called Advanced Systematic Techniques for Reliable Operational Software (ASTROS). The objective of the plan was to provide guidelines for applying modern programming practices to software development. To test the utility of these techniques, two development projects were chosen for a quasiexperimental comparison. The Launch Support Data Base was developed under the guidelines of the ASTROS plan, while the Data Analysis Processor was developed using conventional techniques (Curtis 79, Milliman 80).

In the second project under this effort, data was collected over the development cycle of the PAVE PAWS software development project. This project was designed to be a technology demonstration of modern programming practices. Data available for evaluating the project included personhours, trouble reports, compiler summaries, code progression and durability charts. Only personhours and trouble reports were collected throughout the project. Conclusions about productivity, prediction of project outcomes, and recommendations for future data collection were also made (Curtis 80).

Philip Milliman presented this case study on measurement and evaluation of modern programming practices in a panel session at the 1978 Summer Software Engineering Workshop (Milliman 78). This research was also publicized at the 1979 Digital Avionics Systems Conference (Curtis 79).

Study Name: **Analysis of PAVE PAWS Ops Data**  
Dates: 1980 - 1982  
Contractor: IIT Research Institute  
Contract #: F30602-80-C-0223  
Tech Report: RADC-TR-82-169  
- Analysis of PAVE PAWS Ops Data

The purpose of this study was to establish a baseline software maintenance experience database and to determine relationships which exist between the use of Modern Programming Practices and Software Engineering Tools on the ease of software maintenance of the PAVE Phased Array Warning System. To achieve this purpose the study team performed a literature search, conducted on-site interviews, collected anomaly and change data, established continuing data collection and analysis procedures, developed a machine readable software maintenance experience database and performed analysis of the anomaly and
change data (IITRI 82).
3. CONTEMPORARY WORK

Rome Laboratory research in software measurement has adapted to the evolving maturity of the field. When the field of software engineering was begun, problems revolved around what characteristics of a software development project could and should be measured. RL addressed these problems by sponsoring research defining software models and metrics, many of which have become widely referenced. In a series of research projects, RL developed the first comprehensive suite of quality metrics, the RL Software Quality Framework.

Later developments were centered on issues of technology transfer, particularly the collection of data to support metric research and to validate the usefulness of software measurement. Here too RL was on the forefront of research, particularly with the founding of the Data & Analysis Center for Software (DACS). In addition, RL sponsored the development of a series of tools to ease the burden of data collection.

From the beginning of their measurement program, RL was concerned with the use of these metrics and models in demonstrating the improvements claimed by advocates of Modern Programming Practices. In a very influential study, RL defined these MPPs. A series of efforts studied the effects of MPPs by use of software measurement.

Current research continues these trends, but at a level more suitable for the advanced stage of software measurement. Issues in software measurement are no longer focused on individual metrics. Rather questions revolve around suites of software metrics and demonstrated usefulness. Many companies have now acquired experience with software metrics, but it is unclear what we have learned. A need exists to consolidate these results and to share findings among researchers and practitioners. How do results vary across environments? How can metrics be combined to control all dimensions of a software project? Particularly important is a demonstration that software measurement is cost effective. How much does software measurement cost? What specific quantitative benefits have been gained by software measurement?

RL has institutionalized exploration into these issues in several novel ways. First, RL now sponsors an annual Software Quality Workshop (SQW 89, SQW 90, SQW 91, SQW 91, SQW 92). These workshops bring together an eclectic collection of individuals from industry, government, and academia to discuss their experiences with software measurement. The most recent workshop in this series, Cooperstown I Workshop: Creating a National Vision and Force in Software through Software Measurement, had a somewhat different thrust. Held in Cooperstown, NY, from 30 August to 1 September 1993, Cooperstown I convened an illustrious group to advise Mr. Lloyd Mosemann II, Deputy Assistant Secretary of the Air Force (Communications, Computers, and Support Systems), on the formation of a national software council and the creation of a national database repository of software measurement data. Work emerging from that workshop is ongoing.
Another important institution recently created by RL is the Rome Laboratory Software Quality Technology Transfer Consortium (RLSQT2C). The RLSQT2C is a joint, applied research and development initiative between RL and several corporations from the U.S. defense industry. RL formed the Consortium as a mechanism to transfer software engineering measurement technology into the U.S. defense industry. The Consortium promotes cooperative application of software quality methods, tools, and procedures produced by RL and the exchange of experiences and data associated with these applications among participating organizations. The RLSQT2C aims at improving software quality with a focus on the application of IEEE and ISO standards for adoption and reporting of software quality information.

The Consortium is funded by a Cooperative Research and Development Agreement (CRDA), a written agreement between a Federal laboratory and a non-Federal organization which allows each participant to provide personnel, services, facilities, or equipment towards a joint research project. Currently the Consortium Support Team is composed of Mr. Gerard Murine (Metriqs); Dr. Ronald Gulezian (Drexel University); Mr. Walter Ellis; Kaman Sciences Corporation; and Software Productivity Solutions, Incorporated. Consortium members include CTA, Incorporated; Frontier Engineering, Incorporated; Grumman Aerospace Corporation; Hughes Aircraft of Canada, Limited; Kaman Sciences Corporation; and SoHar, Incorporated. Software quality measurement applications sponsored by the Consortium are oriented around the Quality Evaluation System (QUES), a tool implementing the RL Software Quality Framework (RLSQF).
3.2 RECENTLY COMPLETED CONTRACTS

- In addition, RL continues to sponsor cutting-edge research into software quality. Recent contracts
- Have enhanced to the RLSQF, including the development of QUES and a methodology for specifying software quality
- Replicated and extended an experiment examining the effectiveness of software testing and analysis techniques
- Developed a set of design and sizing metrics for Ada and a Total Quality Methodology for software.

Details on these efforts follow.

Study Name: **Quality Evaluation System**
Dates: December 1987-September 1991
Contractor: Software Productivity Solutions, Inc.
Contract #: F30602-88-C-0019
Tech Report: Quality Evaluation System (QUES)

This study designed and implemented the Quality Evaluation System (QUES), a software system which automates the process of establishing, maintaining, and applying a software quality evaluation framework. The main functions of the QUES include framework creation, framework tailoring, project creation, data collection, and report generation. Features are provided for integrating automated data collection tools into QUES. Source analysis tools for Fortran and Ada were interfaced to QUES as part of this project. QUES operates on a DEC VAX and Sun Sparcstations under a Graphical User Interface. (Dyson 91)

The RLSQF was implemented under QUES. The second volume of the final report for this project describes the framework as implemented in QUES. This is currently the version treated as a baseline within Rome Laboratory.

Study Name: **Conflict Resolution (CORE) for Software Quality Factors**
Dates: May 1990-September 1992
Contractor: Rochester Institute of Technology
Contract #: F30602-88-D-0026
Tech Report: Conflict Resolution (CORE) for Software Quality Factors

This study formulated the software quality problem as maximizing quality goals within the constraints of cost, schedule, and technical feasibility. Multiaattribute optimization problems, such as this one, are characterized by the presence of multiple, conflicting goals along with a large candidate solution space. Software quality factors in software development inherently conflict, another typical feature of such problems. This study developed a methodology and prototype tool which provides support for specifying quality using the RLSQF. Conflict Resolution (CORE) determines achievable software quality factor
Study Name: **Software Reliability, Measurement, and Testing**
Dates: September 1986-December 1989
Contractor: Science Applications International Corporation
Research Triangle Institute
Contract #: F30602-86-C-0269
Tech Report: Software Reliability, Measurement, and Testing

This study integrated software reliability, measurement, and test techniques in terms of prediction, estimation, and assessment. Experiments compared six testing techniques and measured the effect of software product and process variables on software reliability. These experiments extended previous work performed by Dr. Richard Selby of the University of Maryland, conducted in association with the NASA Software Engineering Laboratory. The experiments under this study were structured under a formal statistical experimental design and analyzed with statistical techniques. Error/anomaly and code reviews were the testing techniques found to be the most effective at the unit level. Branch testing and code reviews were the most effective at the CSC level. This finding of the effectiveness of code reviews, as compared with later testing techniques, replicates Dr. Selby's most startling result in a different environment. A guidebook based on these findings was produced to help program managers control and manage software reliability and testing. (McCall 92)

Study Name: **Mission Oriented Investigation and Experimentation (MOIE) Program**
Dates: October 1989-December 1991
Contractor: The MITRE Corporation
Contract #: F19628-89-C-0001
Related Pubs:
- Estimating Ada System Size During Development
- A Guide to Total Software Quality Control (Two volumes)
- Procedures for Applying Ada Quality Prediction Models

This research explored the estimation of software quality based on Ada designs. Motivating this research was the need for technology to analyze designs, when they are first represented, for their likely effect on quality factors. Multivariate models were constructed relating design characteristics and environmental factors to reliability and maintainability. Alternative representations of reliability and maintainability and representations of Ada design structure were examined. Software project data was used to statistically test hypotheses concerning the effects of design structure on reliability and maintainability. (Agresti 92)

Four models were developed for estimating the size of Ada systems at intermediate points in a software development products. The models use successively more detailed design and implementation data, thus taking advantage of current information about the evolving Ada product. The models were developed empirically using multivariate regression analysis with data from 21 Ada subsystems. Models with more detail explain a greater fraction of the variation in lines of code, as measured by the coefficient of determination. (Evanco 92)

Under this study, a guide was produced for program offices. The guide provides strategies and techniques for controlling the quality of software for mission-critical systems. It describes techniques to
prevent, detect, and correct defects, and to make improvements in the processes and resources used to
develop software. The guide recommends the use of a Total Software Quality Control Plan and shows
how to create one prior to start of system development. (Clapp 92)
A History of Software Measurement at Rome Laboratory

4.0 REFERENCES


(DACS 81) DACS Conversion Data Collection Forms, Data & Analysis Center for Software, 1981.


(Donahoo 77) John Donahoo, Susan Carter, Robbie Farquhar and James Hurt, Software Production Data, RADC-TR-77-177, Rome Air Development Center, July 1977.


(IITRI 81) Staff Author, Quality Metrics Study, Data & Analysis Center for Software, 1981.


(James 82) L. E. James, J. E. Angus, J. B. Bowen, and J. McDaniel, Combined Hardware/Software Models, RADC-TR-82-68, Rome Air Development Center, 1982.


(Ruston 77b) Henry Ruston and Martin L. Shooman, Summary of Technical Progress, Software


(Slavinski 75) Richard T. Slavinski, Static FORTRAN Analyzer, RADC-TR-75-275, Rome Air Development Center, 1975.


(Soistman 84) E. C. Soistman and K. B. Ragsdale, Combined Hardware/Software Reliability Prediction Methodology, Rome Air Development Center, December 1984.


(SQW 90) Second Annual Software Quality Workshop; Data & Analysis Center for Software, Rochester Institute of Technology, and Rome Air Development Center; Rochester, NY; 14-16 August 1990.


(SQW 92) Fourth Annual Software Quality Workshop; Data & Analysis Center for Software and Rome Air Development Center; Alexandria Bay, NY; 2-6 August 1992.


(Yau 80a) S. S. Yau and J. S. Collofello, Performance Ripple Effect Analysis for Large Scale Software Maintenance, Northwestern University, RADC-TR-80-55, Rome Air Development Center, March 1980.


Appendix A.

ACRONYMS

AFOTEC Air Force Operational Test & Evaluation Center
AFSC Air Force Systems Command
AMS Automated Measurement System
AMT Automated Measurement Tool
ANSI American National Standards Institute
ASQS Assistant for Specifying Quality Software
ASTROS Advanced Systematic Techniques for Reliable Operational Software
BASIC Beginner's All-Purpose Symbolic Instruction Code
C3 Command, Control, and Communications
C3I Command, Control, Communications and Intelligence
C3C Rome Lab Software Organization; also C3AB, C3CA, C3CB, COA, COE, COEE, COES
COCOMO Constructive Cost Model
CPC Computer Program Component, now called Computer Software Component (CSC)
CRDA Cooperative Research and Development Agreement
CSC Computer Software Component
CSC Computer Sciences Corporation
CSDL Charles Stark Draper Laboratory
CSU Computer Software Unit
DA Department of the Army
DACS Data & Analysis Center for Software
DCF Data Collection From
DID Data Item Description
DOD Department of Defense
DPS Distributive Partitioning Sorting
DMS Data Management System
ESC Electronic Systems Center, formerly ESD - Electronic Systems Division
FSD Federal Systems Division (IBM)
GCOS General Comprehensive Operating System
GRC General Research Corporation
HDBK Handbook
HIPO Hierarchy Input Process Output
HOL Higher-Order Language
IAC Information Analysis Center
IBM International Business Machines Corporation
IEEE Institute of Electrical and Electronics Engineers
IITRI Illinois Institute of Technology (IIT) Research Institute
ISIS RADC Software Sciences Section, also ISI, ISIM
MIL Military
MIS Management Information Systems
MPP Modern Programming Practices
MTBF Mean Time Between Failures
MTSR Mathematical Theory of Software Reliability
NASA/SEL National Aeronautics and Space Administration Software Engineering Laboratory
NHPP Non-Homogeneous Poisson Process
PAR Pattern Analysis Recognition
PAWS Phased Array Warning System
PDL Program Design Language
PDSS Post Deployment Software Support
PRC Planning Research Corporation
PSL Programming Support Library
QUES QUality Evaluation System
RB Rome Lab Reliability Organization, also RBR; now ER
R&D Research and Development
R&M Reliability and Maintainability
RADC Rome Air Development Center
RFP Request For Proposal
RL Rome Laboratory
RLSQF Rome Laboratory Software Quality Framework
RLSQT2C Rome Laboratory Software Quality Technology Transfer Consortium
SAIC Science Applications International Corporation
SEI Software Engineering Institute
SIMD Single Instruction Stream - Multiple Data Stream
SIMON Software Implementation MONitor
SLCSE Software Life Cycle Support Environment
SLED Software Lifecycle Empirical Database, also called Software Lifecycle Experience Database
SLOC Source Lines of Code
SOW Statement of Work
SPT Structured Programming Technology
STARS Software Technology for Adaptable, Reliable Systems
STD Standard
STEP Software Test & Evaluation Panel
TDP Top Down Programming
TPO Technology Program Objective
V&V Verification and Validation
Appendix B.

THE FRAMEWORK

The eleven distinct user-oriented quality concerns ("factors") in the framework are:

- Correctness: Does it do what I want?
- Efficiency: Will it run on my hardware as well as it can?
- Flexibility: Can I change it?
- Integrity: Is it secure?
- Interoperability: Will I be able to interface it with another system?
- Maintainability: Can I fix it?
- Portability: Will I be able to use it on another machine?
- Reliability: Does it do it accurately all of the time?
- Reusability: Will I be able to reuse some of the software?
- Testability: Can I test it?
- Usability: Can I run it?

The RLSQF provides software-oriented criteria, such as Modularity, Self-Descriptiveness, and Simplicity to measure the factors. One or more criteria are provided for each factor, with criteria overlapping for several factors. Criteria, in turn, are decomposed into metrics. The criteria in the framework are:

- Access Audit
- Access Control
- Accuracy
- Communications Commonality
- Communicativeness
- Completeness
- Conciseness
- Consistency
- Data Commonality
- Execution Efficiency
- Expandability
- Generality
Instrumentation
Machine Independence
Modularity (Shared by five of the factors.)
Operability
Self-Descriptiveness (Shared by five of the factors.)
Simplicity (Note: Inverse of complexity.)
Software System Independence
Storage Efficiency
Traceability
Training

The metrics are quantitative, and they are calculated based on checklists filled out at each phase. The structure ensures that metrics, criteria, and factors will all take on values between zero and one, inclusive. Higher values indicate better quality. The metrics in the framework either ask yes/no questions or request a numeric quantity. For example, the self-descriptiveness criterion, which is part of the Maintainability, Flexibility, Testability, Portability and Resuability factors, includes the following questions:

- Number of lines of comments.
- Number of non-blank lines of comments.
- Are non-standard HOL statements commented?
- Are the variable names descriptive of the physical or functional property they represent?