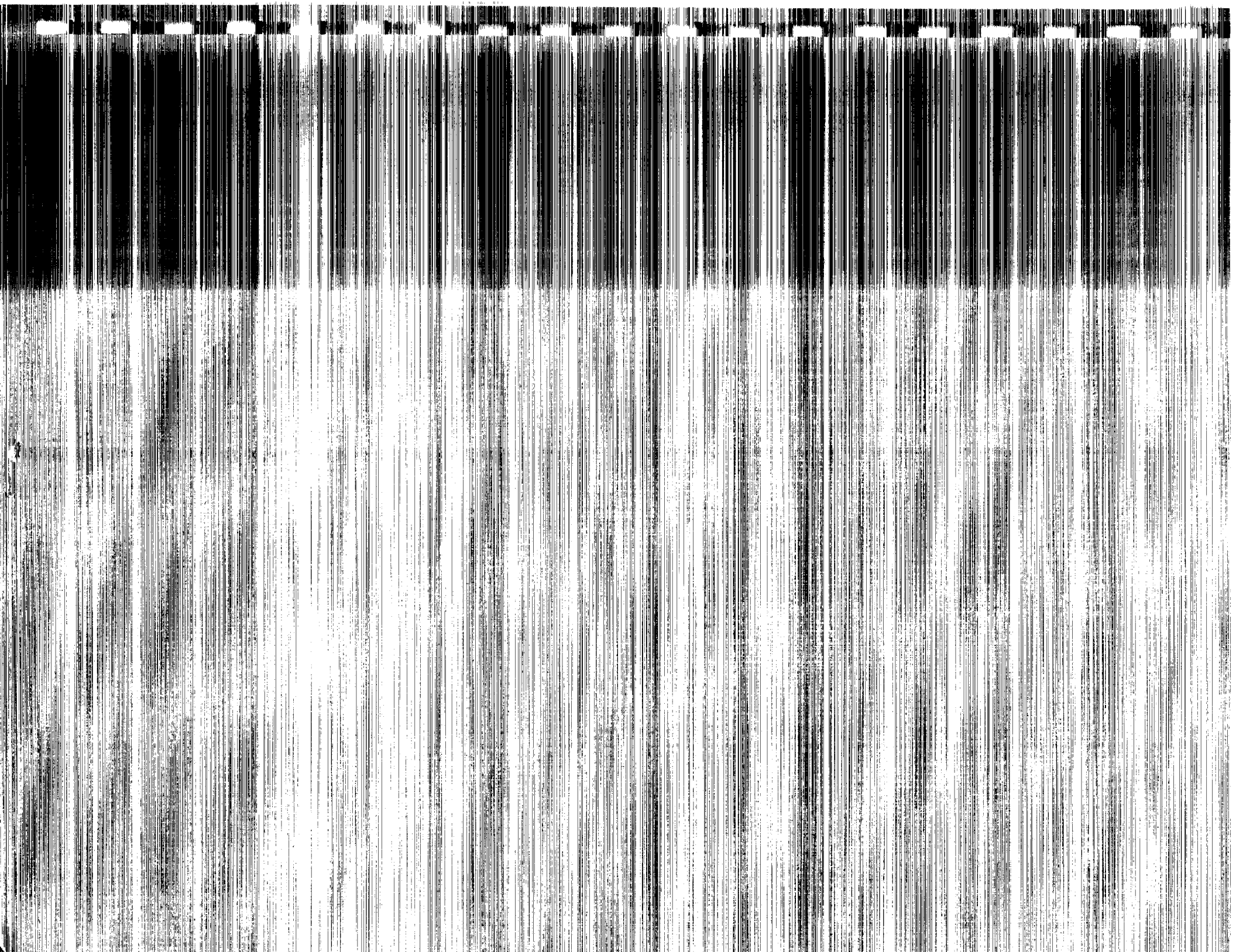
IMPLEMENTATION OF A PRODUCTION ADA PROJECT: THE GRODY STUDY

SEPTEMBER 1989

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771



SOFTWARE ENGINEERING LABORATORY SERIES SEL-89-002

IMPLEMENTATION OF A PRODUCTION ADA PROJECT: THE GRODY STUDY

SEPTEMBER 1989

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

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

NASA/GSFC, Systems Development Branch

The University of Maryland, Computer Sciences Department

Computer Sciences Corporation, Systems Development Operation

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

The major contributors to this document are

Sara Godfrey (GSFC)
Carolyn Brophy (University of Maryland)

Single copies of this document can be obtained by writing to

Systems Development Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

ABSTRACT

The use of the Ada language and design methodologies that encourage full use of its capabilities have a strong impact on all phases of the software development project life cycle. At the National Aeronautics and Space Administration/ Goddard Space Flight Center (NASA/GSFC), the Software Engineering Laboratory (SEL) conducted an experiment in parallel development of two flight dynamics systems in FORTRAN and Ada. This document describes the differences observed during the implementation, unit testing, and integration phases of the two projects and outlines the lessons learned during the implementation phase of the Ada development. Included are recommendations for future Ada development projects.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

TABLE OF CONTENTS (Cont'd)

LIST OF ILLUSTRATIONS

Figure

LIST OF TABLES

EXECUTIVE SUMMARY

During the past few years, a study has been conducted to determine the applicability of Ada for software development in the flight dynamics environment at the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) in Greenbelt, Maryland. The primary objectives of this study are to determine the cost-effectiveness and feasibility of using Ada and to assess the effect of Ada on the flight dynamics environment. The study consists of parallel efforts to develop the Gamma Ray Observatory Dynamics Simulator software, with one team of developers using FORTRAN and another team using Ada. A third team collects and assesses data from the two development efforts. The study is a joint project with participants from NASA/GSFC, Computer Sciences Corporation (CSC), and the University of Maryland.

This document concentrates on the implementation phase of the development, including coding, unit testing, and integration, during which the following conclusions were reached concerning the use of Ada as a development language for flight dynamics applications:

- The proper use of Ada's strong typing requires some training. An abstract type analysis should be incorporated into the design process to help control the number of types used during implementation.
- Ada's tasking feature was difficult to implement and test. It is recommended that the use of tasking be restricted to applications that really require its use.
- The excessive use of nesting in Ada increases the amount of compilation necessary during implementation and complicates unit testing.

- New programming skills are required to realize the potential benefits of some Ada features such as tasking, exception handling, and strong typing. Training alone does not seem to provide these skills and some on-the-job experience is necessary.
- Ada tools such as the compiler and the debugger were found to be quite useful, but are still immature.
- It required about two-and-one-half times as much Ada code to produce the functionality provided by FORTRAN code.
- The percentage of effort expended during each life-cycle phase of the Ada development was not significantly different from that expended during the FORTRAN development.
- The error rates and the change rates were similar for the Ada and FORTRAN developments. A higher percentage of errors was discovered in the Ada project during the implementation phase than in the FORTRAN project.

Further study is continuing to determine the validity of these conclusions with other Ada projects.

SECTION 1 - INTRODUCTION

1.1 BACKGROUND

This document is the third in a planned series describing various aspects of developing a dynamics simulator to be used as part of the ground support system for the Gamma Ray Observatory (GRO) satellite. This project, the GRO Dynamics Simulator in Ada (GRODY), is significant because a corresponding version of this simulator (GROSS) has been developed in FORTRAN. Analysis of the two projects will provide insight into the implications of developing flight dynamics software in Ada rather than FORTRAN--the usual development language in the past. This document will concentrate on the experiences of the GRODY team during implementation, unit testing, and integration testing. In addition, experience during the implementation of another Ada project, Flight Dynamics Analysis System (FDAS), will be included for comparison.

1.2 OBJECTIVES

1.2.1 OBJECTIVES OF DOCUMENT

There has been considerable interest in the potential of Ada and its associated methodologies since their introduction. The Ada language design supports such commonly accepted, highly regarded software engineering practices as information hiding, abstraction, modularity, and localization. It has been hoped that these design features would lead to great improvements in many aspects of software development when Ada is used. One area where improvement has been needed is in the size and overall complexity of the language (Reference 1).

Among the claims for Ada's potential are higher productivity, easier maintainability, generation of more reusable

software, and more reliability of software. Many of these claims have been based on subjective information, since the amount of reported quantitative data from actual projects is quite small. There have been some reports of significant productivity gains when Ada has been used (References 2 and 3).

One of the objectives of this document will be to examine these claims for Ada's potential in the light of two production-type systems developed in Ada--GRODY and FDAS. Much quantitative data has been collected during these projects to gain insight into some of the questions that arise when considering moving forward into an Ada software development environment. For instance:

- How does Ada measure up to other languages in productivity, reliability, reusability, efficiency, and maintainability?
- How does an existing software development environment make the transition into an Ada software development environment?
- Is Ada mature enough to satisfy production requirements? Are production-quality compilers and supporting development tools available?
- What lessons have been learned from these early experiences in Ada?

GRODY, a major Ada development effort that has recently been completed, will be analyzed to gain some of this insight. Additional experiences from FDAS, also developed in Ada, will be included to provide a wider experience base.

1.2.2 OBJECTIVES OF GRODY VERSUS GROSS

The overall goal of the Ada GRODY/FORTRAN GROSS software development project is to gain insight into the applicability

of the Ada development methodology and language in the NASA software environment. Several objectives have been established as mechanisms for attaining this goal (Reference 4). The primary ones are to determine the cost-effectiveness and feasibility of using Ada to develop flight dynamics software and to assess the effect of Ada on the flight dynamics environment. Related objectives are to determine whether present methodologies in use within the flight dynamics environment are suitable and to investigate other methodologies related to the use of Ada. For example, is the standard development life cycle (see Section 1.3.3) used on the FORTRAN development equally suitable for an Ada development?

Because reusability is an important goal for cost-effective software development, this experiment will also try to develop approaches for maximum reusability when Ada is being used for implementation. A major portion of the software developed in the flight dynamics environment is reused; because Ada is designed to facilitate reusability, the methods developed should maximize this feature.

Other factors being assessed throughout the GRODY project are the differences in reliability and maintainability between an implementation in FORTRAN and one in Ada. Obviously, a system that is more reliable--that is, one that has fewer errors per 100,000 source lines of code (SLOC)--will cost less to maintain.[1] Similarly, an implementation that is easier to correct and enhance will be less costly to maintain. This is of particular importance since the annual cost of maintenance usually ranges from 10 to 35 percent of the original development cost in staff hours (Reference 5).

[1]SLOC refers to any 80-byte record of code, including comments, blanks, declarations, and executable lines.

Looking to the future, Ada has been chosen as the implementation language for the Space Station Freedom project, which will be an extremely large, complex, long-term project. In order to plan effectively for the use of Ada on projects such as the Space Station, a good set of software measures needs to be developed. For instance, how can we make size estimates for Ada implementations? What is the expected productivity when Ada is used for implementation of a scientific application? GRODY can provide much useful information in these areas.

Portability is another area of interest for the longer-term projects. The rehosting of software to a new computer system is a likely occurrence with a long-term project, and a more portable implementation would certainly reduce rehosting costs. Does an Ada implementation provide more portability? Do the methods of implementation influence the portability of the final product?

1.3 PROJECT DESCRIPTIONS

1.3.1 GROSS AND GRODY

Both the GROSS and the GRODY projects are developing dynamics simulators for the GRO mission. GROSS has been developed in FORTRAN using the standard methodology in the flight dynamics environment (References 5 and 6) and is considered the operational software that will actually be used for GRO mission support. GRODY is developing a functionally identical software system using modified design techniques (References 4 and 7) and the Ada development language. (The portion of GROSS that is not included in GRODY is to be integrated into a real-time piece of software being developed by another group for simulation purposes.) Both the GROSS and GRODY development teams consist of members from NASA/GSFC and CSC. A study team is composed of members from NASA/GSFC, CSC, and the University of Maryland and has been

collecting data on both projects throughout their development. Figure 1-1 shows the organization of the two development projects and the study.

Work on both dynamics simulators began in January 1985. The FORTRAN team began with a typical development cycle, while the Ada team began with a training phase. Acceptance testing of the FORTRAN dynamics simulator was completed by June 1987. The Ada dynamics simulator began system testing in the fall of 1987 and completed system testing in June 1988. Formal acceptance testing was not conducted, but acceptance tests were run on the Ada system as part of the training received by another task preparing to develop another Ada dynamics simulator.

The FORTRAN development effort was carried out on a Digital Equipment Corporation (DEC) VAX-11/780; the Ada development was done on a DEC VAX-8600. The completed FORTRAN project is approximately 45,000 SLOC; the Ada project is approximately 128,000 SLOC.

1.3.2 FDAS

FDAS is a software development tool written in Ada and designed to provide an integrated support framework for flight dynamics research. FDAS supports research in the areas of orbit and attitude determination and control as well as mission planning and analysis. Research activities under FDAS will include the development of new models and algorithms that will be substituted for existing software or the reconfiguration of existing software to solve new problems. Basically, it provides a simplified, standardized approach to software modification by allowing the user to automate the building of software from available libraries of applications software. The software modules contained in the libraries can be thought of as pieces of a puzzle: just as different pieces of the same shape could be interchanged

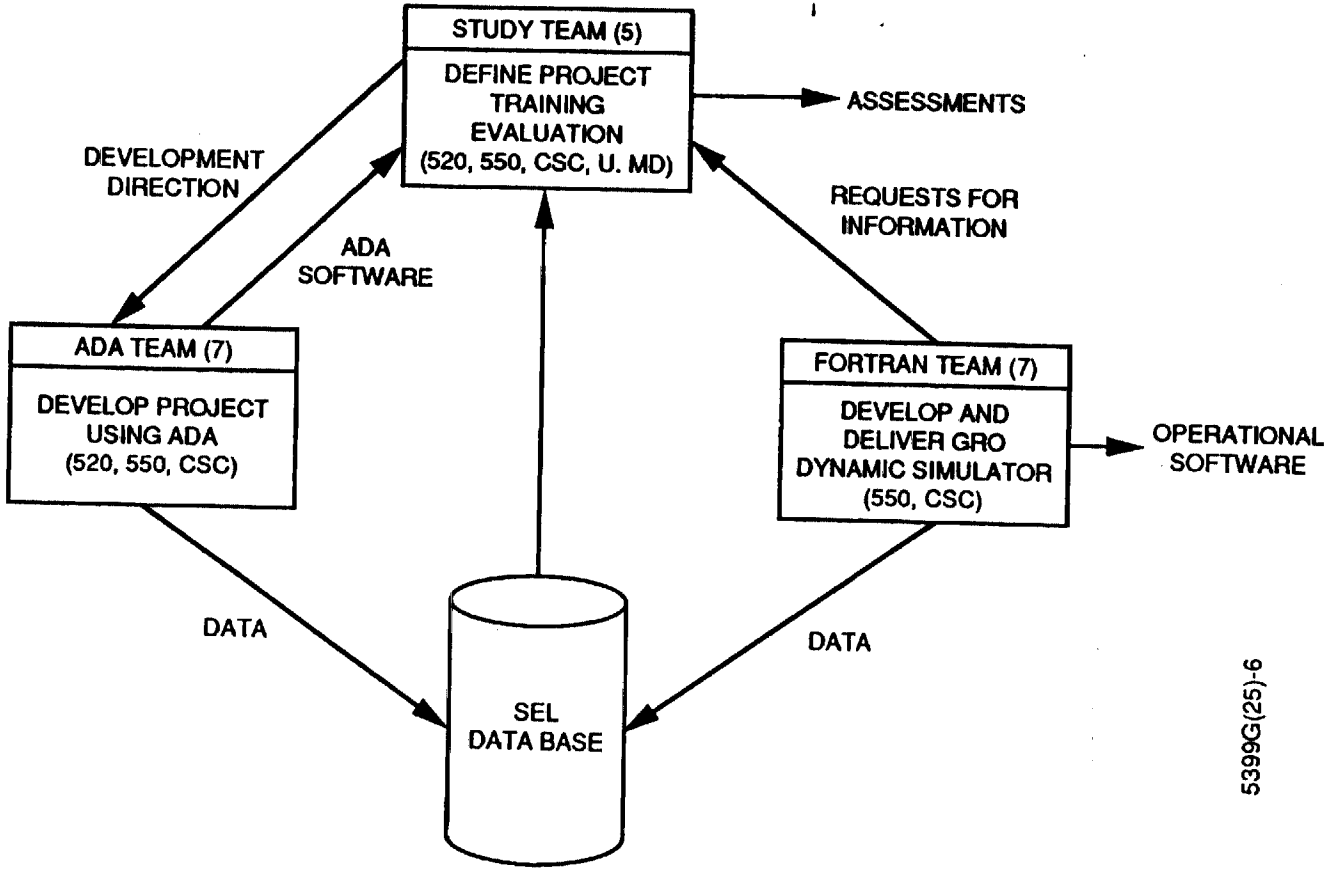


Figure 1-1. Ada Experiment Organization

to show different pictures, so modules that perform the same function could be interchanged to build a new piece of software.

FDAS has a long development history beginning with a research study and the building of a prototype to help establish the validity of the FDAS concept and to clarify the requirements for such a concept. This prototype was developed in FORTRAN on the DEC VAX-11/780 and was completed in December 1983. A period of prototype evaluation and requirements definition followed, with the development of FDAS beginning in January 1985. The decision to implement in Ada was not made until well into the design phase, even though FDAS was being designed to take full advantage of Ada. The implementation has progressed in a series of builds with increasing levels of capabilities. Build 3 has recently been completed and is currently under evaluation. The present capabilities allow a user to substitute "puzzle pieces" that are already in object module form under FDAS control.

The current version of FDAS has also been developed on the DEC VAX-11/780 and is approximately 30,000 SLOC. The program has been chosen by the Software Engineering Institute of Carnegie Mellon University as an educational vehicle to aid in teaching software engineering practices and as a basis for experiments on maintenance, enhancement, configuration management, and testing. Copies of FDAS are available through the NASA Computer Software Management and Information Center (COSMIC).

While this document does not intend to provide extensive information on the implementation of FDAS, some experiences are included here as a basis for comparison with experiences during GRODY implementation. The types of application problems addressed by GRODY and FDAS are radically different since FDAS could be described as a large executive or a

mini-operating system to manipulate other software, while GRODY is a "number cruncher"--that is, a program requiring an extremely large number of mathematical calculations. Thus, it is interesting to see how (or whether) the application type affected such considerations as the approach or methods used during implementation and the success of these methods.

1.3.3 SIMULATOR ENVIRONMENT/LIFE CYCLE

A general description of a dynamics simulator is included here to acquaint the reader with the type of implementation problem that confronted the GROSS and GRODY teams. The purpose of a dynamics simulator is to test and evaluate the onboard attitude control logic under conditions that simulate the expected inflight environment as closely as possible. The simulator can be considered a control system problem, beginning with an onboard computer (OBC) model that uses sensor data to compute an estimated attitude. Control laws are then modeled to generate commands to the attitude hardware (actuators) to reduce the attitude error. A truth model portion of the simulator simulates the response of the attitude hardware and generates a true attitude for the spacecraft. Sensor data corresponding to the true attitude are produced by the truth model and sent back to the OBC model (Figure 1-2).

The standard software development life cycle used in the flight dynamics environment is described in detail in Reference 6. A brief description of the phases of this life cycle are included here to acquaint the reader with the activities normally performed during each phase. Later in this document, the suitability of this life cycle for the Ada development will be discussed.

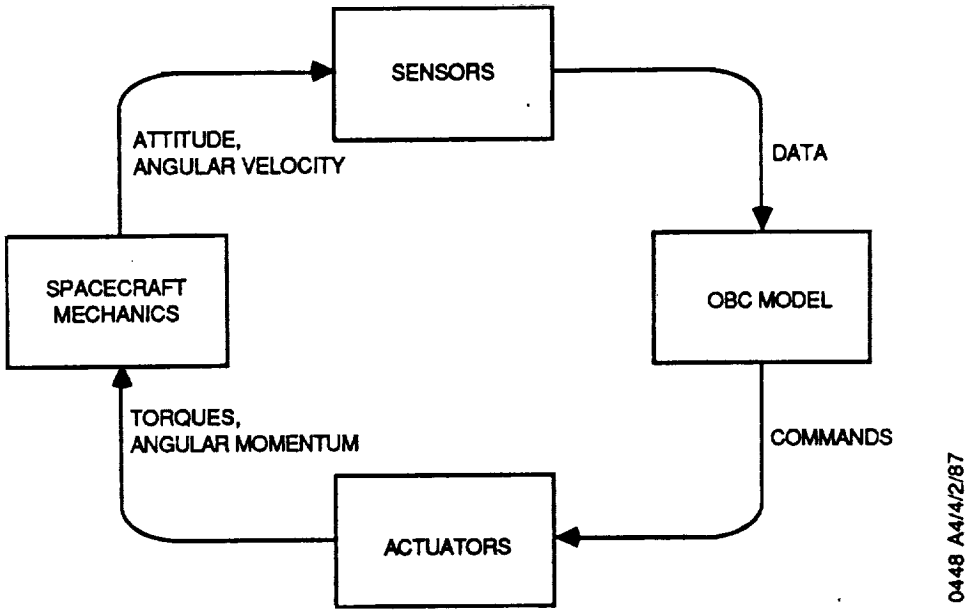


Figure 1-2. A Dynamics Simulator

The standard life cycle can be divided into the following seven sequential phases:

1. Requirements analysis--During this phase, the developer analyses a document that contains the functional specifications and requirements to assess the completeness and feasibility of the requirements and to make an initial estimate of the required resources. The results of this analysis are summarized in a requirements analysis report.

2. Preliminary design--In this phase, the design process is begun when the requirements are organized into functional capabilities and distributed into subsystems.

3. Detailed design--In this phase, the design that was outlined during the preliminary design phase is expanded to describe all aspects of the system.

4. Implementation--This phase consists of coding new modules from the design specifications, revising old code to meet new requirements, and unit testing to ensure that each module functions properly.

5. System testing--During this phase, the completely integrated system produced during implementation is tested according to a test plan (also generated during the implementation phase) to verify that all the required system capabilities function properly.

6. Acceptance testing--An independent team tests the system to ensure that it meets all requirements.

7. Maintenance and operation--At this point, the software becomes the responsibility of a maintenance and operations group that implements any further enhancements and any error corrections that might be necessary.

Implementation activities generally begin immediately after the critical design review (CDR), at the end of the detailed

design phase, unless serious problems surfaced during the review. Usually, the implementation is developed in stages or builds according to a plan for development that was prepared during the detailed design phase. Individual developers code and test the modules identified as belonging to a particular build. (A module is defined as a portion of a subsystem or system that performs certain designated functions.) Once these modules are unit tested (whether they are entirely new, reusable, or modified reusable), they are placed in a project-controlled library where the process of integrating the modules for a particular build begins. At this point, the source code for each module is placed under strict configuration control and any further changes to the module must be documented and approved by the development team leader before the changes can be incorporated into the controlled library.

When all the modules belonging to a particular build are contained in the controlled library, a series of integration tests is executed on the completed build. These integration tests are generally developed early in the implementation phases and are designed to test the functionality of the build. Meanwhile, developers not involved in the integration testing will continue to code and unit test modules belonging to the next build. This procedure continues until all the modules are completed and the system is ready for system testing.

A number of implementation methodologies are considered standard within the flight dynamics environment. These include the following:

1. Coding standards and structured code--Implementers should use only basic structured constructs which are specified in a module's program design language (PDL). Standards of coding are specified for each project and should be

enforced through quality assurance procedures. Principles of structured programming for FORTRAN implementations are described in Reference 8. For Ada implementations, coding standards and a guide to the use of Ada's constructs are included in the Ada Style Guide (Reference 9).

2. Code reading--As an implementer finishes coding a routine and successfully compiles it, another member of the development team reviews or "reads" the code to verify that it performs the function specified in the design. Many common coding errors are discovered during the code reading, before any testing has occurred.

3. Top-down implementation--Each system or subsystem build is implemented in a top-down fashion following the hierarchical structure, so that the higher-level controlling modules that are pictured at the top of the baseline or object diagram are implemented first. Implementation continues downward to the lower-level controlled modules. Modules that are not yet coded exist in the subsystem as stubs, or fully executable modules that acknowledge their execution only by writing out a message and then returning to the calling module.

4. Formal test plans--As the growing subsystem evolves in the controlled library, integration testing begins. When the full build capability is contained in the controlled library, this integration testing follows a formal test plan that specifies the functional capabilities to be tested and outlines the criteria for determining the success of the tests. Test plans for each build or release typically are developed early in that build or release. During the final release of a system, a system test plan is developed to test the complete end-to-end capabilities of the system.

Documentation produced during the implementation phase includes the test plans for both integration testing and system testing, a user's guide, and a system description. The user's guide and system description are usually produced in draft form during this phase so that the final versions, produced during acceptance testing, will reflect any changes that were necessary during the testing phases.

1.3.4 STAFFING

The GROSS and GRODY implementation teams were approximately the same size with about nine members each. However, the GRODY team members were more experienced, in general, with more years of software development experience and a wider range of application experience. In addition, they were familiar with more programming languages--an average of seven compared to three for the GROSS team. On the other hand, the GROSS team members were more experienced in the development of dynamics simulators. About two-thirds of the GROSS team had previously developed a dynamics simulator, compared to about two-fifths of the Ada team (Table 1-1). Very few of the GRODY team were experienced in any sort of real-time programming. This type of experience was considered useful for implementing and testing Ada tasking.

As the GRODY team members were unfamiliar with the Ada development language and its methodologies, they received extensive training in both. Each member of the Ada team received 2 months of full-time training spread over a 6-month period. This training began with a series of sessions during which videotapes on Ada specifics were viewed and discussed. These sessions were supplemented by reading and coding assignments as well as lectures on Ada methodologies. The final aspect of the training was the actual design and implementation of a practice problem consisting of nearly 6,000 SLOC. More information on the Ada training program can be obtained from Reference 10.

Table 1-1. Team Profiles

CHARACTERISTIC	FORTRAN TEAM	ADA TEAM
NUMBER OF LANGUAGES KNOWN (MEDIAN)	3	7
TYPES OF APPLICATION EXPERIENCE (MEDIAN)	3	4
YEARS OF SOFTWARE DEVELOPMENT EXPERIENCE (MEAN)	4.8	8.6
TEAM MEMBERS WITH DYNAMICS SIMULATOR EXPERIENCE (%)	66	43

0448 A4/4/7/87

The FDAS implementation team consisted of five members who, like the GRODY team, were not previously experienced with Ada. However, the FDAS team did not undergo any sort of formal Ada training; they learned Ada by reading textbooks and by on-the-job training. Consultants from the University of Maryland were available to aid the learning process.

1.3.5 TIMELINES

Figure 1-3 shows the development cycles of both the FORTRAN and the Ada projects against a time line, to emphasize that the FORTRAN development was conducted much earlier and on a much more compressed schedule. It is important to note that effort levels differ considerably over the life cycles of the two projects. Effort comparisons will be discussed in Section 4.

1.4 APPROACH

1.4.1 INFORMATION COLLECTION

The collection of information for this document (see Figure 1-4) was accomplished by using forms, surveys and interviews, observation, and code analysis.

Using forms, data on the GROSS and GRODY projects as well as FDAS have been collected through all the projects and stored on a data base by the Software Engineering Laboratory (SEL). The SEL is a joint effort of NASA/GSFC, CSC, and the Computer Sciences Department of the University of Maryland. It has been collecting detailed software development data through complete project life cycles for the past 10 years. During this time, the SEL has converged on a standard set of forms that contain information of general interest. These forms were used to collect information on all three projects, but for GRODY and FDAS a modified version of the forms was used to capture some Ada-specific information not included in the original forms. Examples of the forms used to

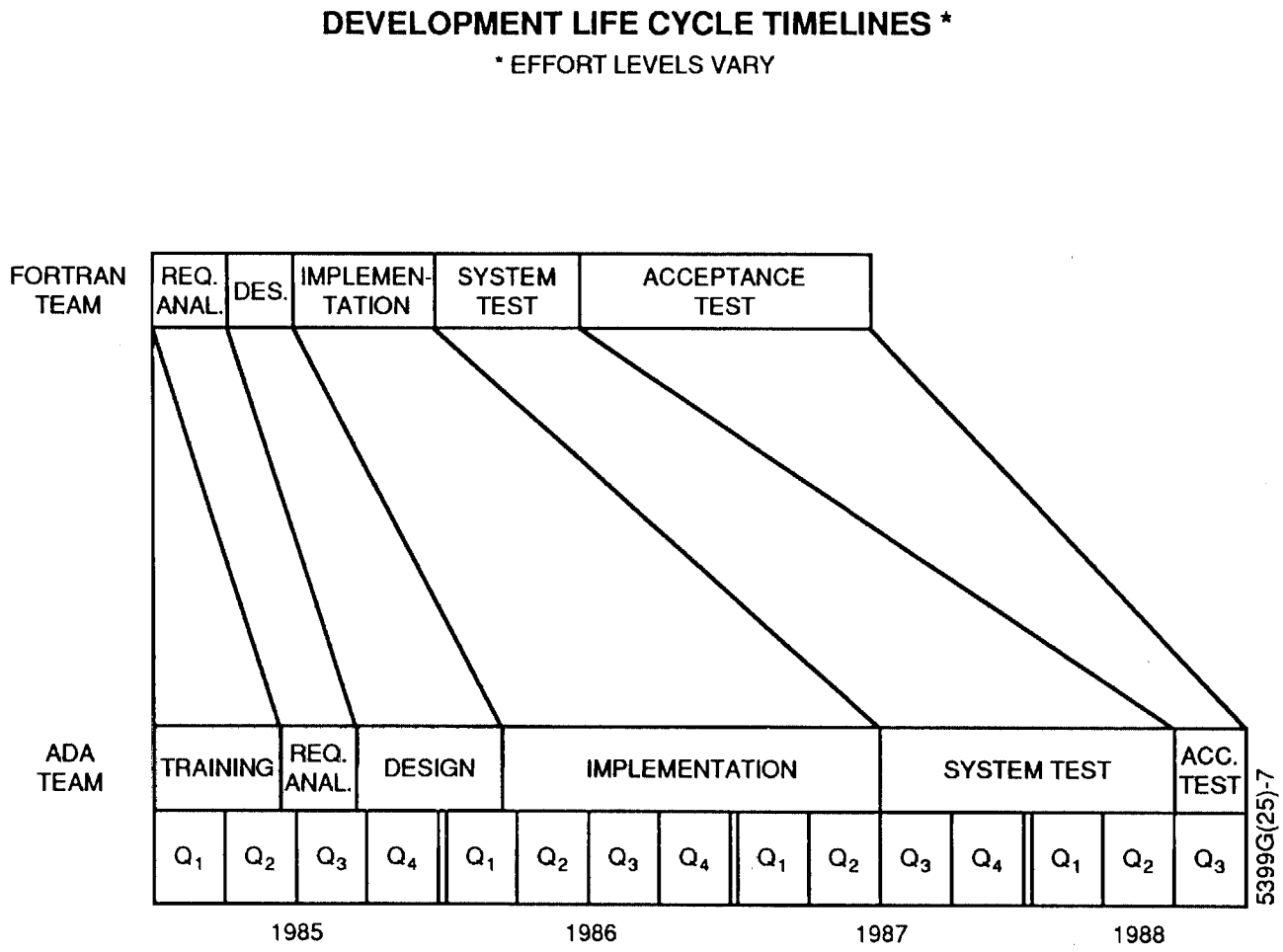


Figure 1-3. Development Life Cycle Timelines

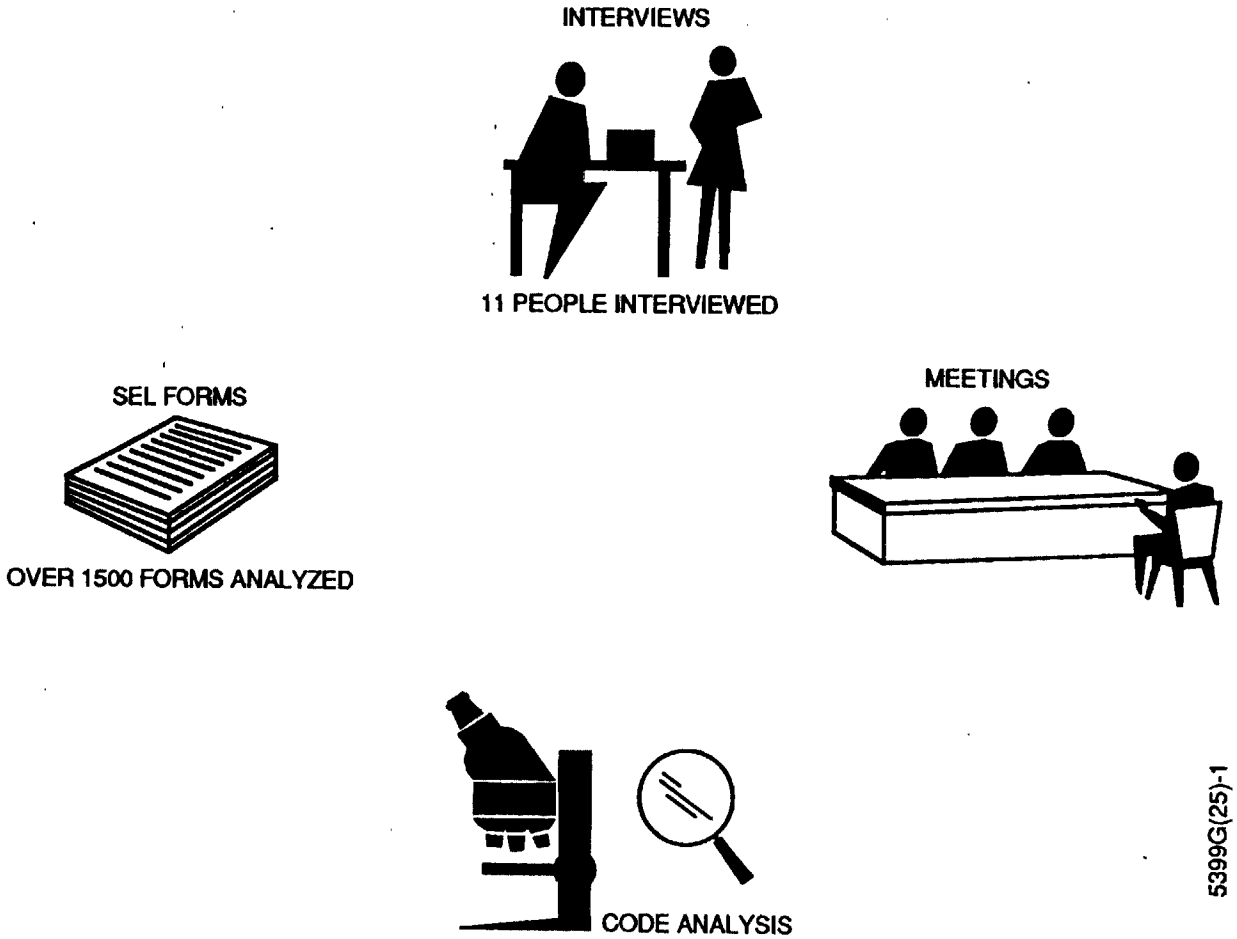


Figure 1-4. Information Collection

collect error and change data are shown in Figures 1-5 and 1-6.

Using surveys and interviews, the study team on the GRODY project collected a great quantity of information on the methods used and their success, the problems encountered during implementation and testing and their solutions, opinions on tools, and suggestions for improvements that could be made for future development efforts. Most of this information was collected during personal interviews conducted near the end of the integration phase. In order to supplement the interview information, to clarify some of the data, and to quantify opposing viewpoints, written surveys were distributed to team members. Team members from both GRODY and FDAS were included in this data collection process.

The study team carefully monitored the development progress in several ways. They attended design reviews given by the development team and, for GRODY, attended many of the implementation meetings held every 2 or 3 weeks. The purpose of these meetings was to discuss project status, to resolve any problems involving more than one implementation area, to share knowledge gained by experience, and to coordinate implementation efforts. The development progress was also monitored by tracking the weekly accounting information on computer usage and program size.

The completed code of the projects was analyzed by a program that provides statistics on the detailed characteristics of the source code. In addition to this automatic analysis, software developers performed a careful, detailed analysis of the GRODY code as a learning and training experience. As a result of this analysis, many observations were made concerning the quality of the GRODY code. These results are contained in Reference 11.

CHANGE REPORT FORM

Project Name:_____ Current Date:_____

Programmer Name:_____ Approved by:_____

Section A – Identification

Describe the change: (What, why, how) _____

Effect: What components (or documents) are changed? (Include version)_____

Effort: What additional components (or documents) were examined in determining what change was needed?_____

Location of developer's source files_____

	month	day	year
Need for change determined on:			
Change completed (incorporated into system):			

Check here if project is in Ada ☐
(If so, complete questions on reverse side)

	1 hr/less	1 hr/1 dy	1 dy/3 dys	>3 dys
Effort in person time to isolate the change (or error):				
Effort in person time to implement the change (or correction):				

Section B – All Changes

Type of Change (Check one)

- ☐ Error correction
- ☐ Planned enhancement
- ☐ Implementation of requirements change
- ☐ Improvement of clarity, maintainability, or documentation
- ☐ Improvement of user services
- ☐ Insertion/deletion of debug code
- ☐ Optimization of time/space/accuracy
- ☐ Adaptation to environment change
- ☐ Other (Explain on back)

Effects of Change

Y	N	
☐	☐	Was the change or correction to one and only one component?
☐	☐	Did you look at any other component?
☐	☐	Did you have to be aware of parameters passed explicitly or implicitly (e.g., common blocks) to or from the changed component?

Section C – For Error Corrections Only

Source of Error (Check one)

- ☐ Requirements
- ☐ Functional specifications
- ☐ Design
- ☐ Code
- ☐ Previous change

Class of Error (Check most applicable)*

- ☐ Initialization
- ☐ Logic/control structure (e.g., flow of control incorrect)
- ☐ Interface (internal) (module to module communication)
- ☐ Interface (external) (module to external communication)
- ☐ Data (value or structure) (e.g., wrong variable used)
- ☐ Computational (e.g., error in math expression)

*If two are equally applicable, check the one higher on the list

Characteristics (Check Y or N for all)

Y	N	
☐	☐	Omission error (e.g., something was left out)
☐	☐	Commission error (e.g., something incorrect was included)
☐	☐	Error was created by transcription (clerical)

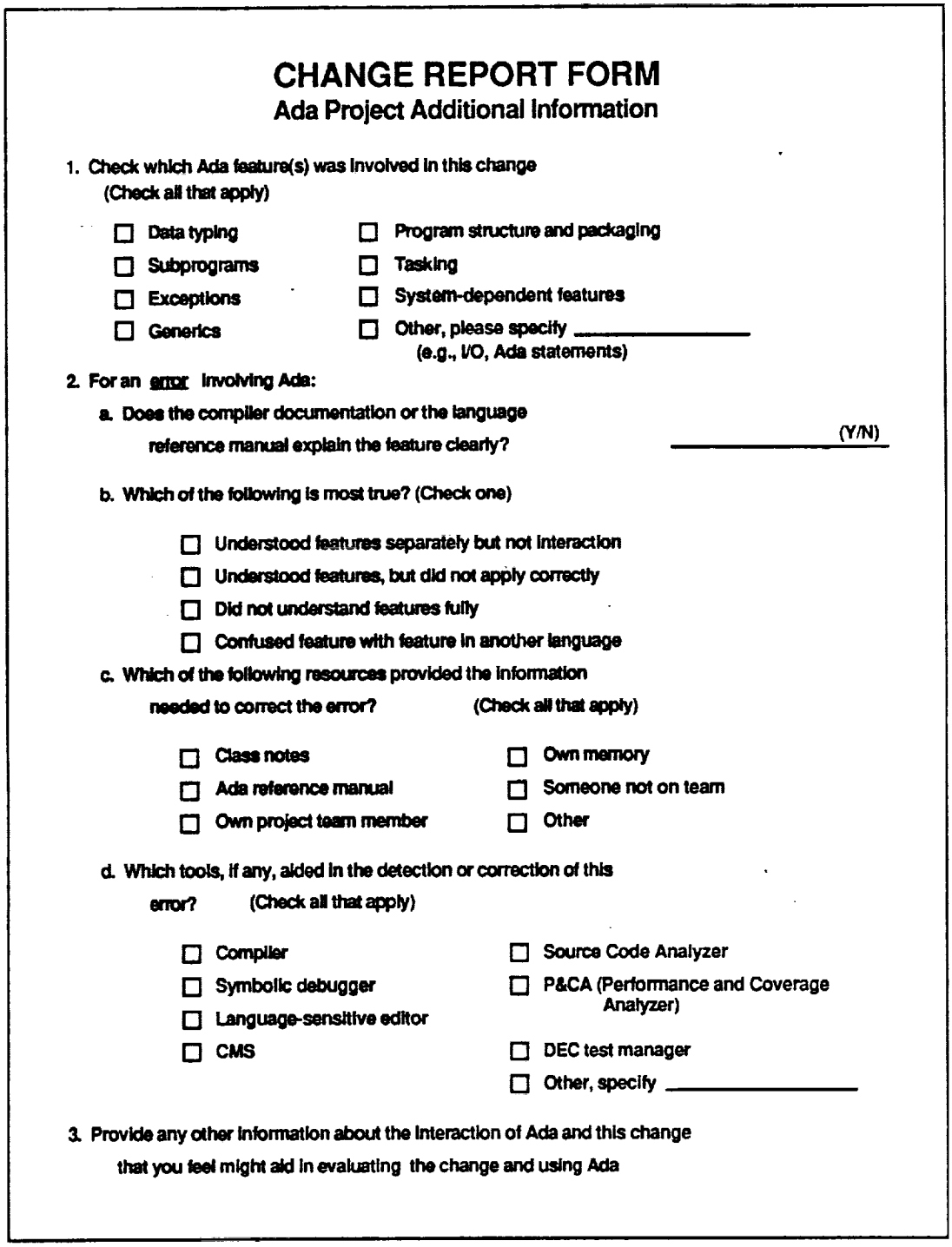
For Librarian's Use Only

Number:_____
Date:_____
Entered by:_____
Checked by:_____

5150G(1).1

JULY 1987

Figure 1-5. Change Report Form



CHANGE REPORT FORM
Ada Project Additional Information

1. Check which Ada feature(s) was involved in this change
   (Check all that apply)

   - ☐ Data typing
   - ☐ Subprograms
   - ☐ Exceptions
   - ☐ Generics
   - ☐ Program structure and packaging
   - ☐ Tasking
   - ☐ System-dependent features
   - ☐ Other, please specify _____
     (e.g., I/O, Ada statements)

2. For an error involving Ada:

   a. Does the compiler documentation or the language reference manual explain the feature clearly? _____ (Y/N)

   b. Which of the following is most true? (Check one)

   - ☐ Understood features separately but not interaction
   - ☐ Understood features, but did not apply correctly
   - ☐ Did not understand features fully
   - ☐ Confused feature with feature in another language

   c. Which of the following resources provided the information needed to correct the error? (Check all that apply)

   - ☐ Class notes
   - ☐ Ada reference manual
   - ☐ Own project team member
   - ☐ Own memory
   - ☐ Someone not on team
   - ☐ Other

   d. Which tools, if any, aided in the detection or correction of this error? (Check all that apply)

   - ☐ Compiler
   - ☐ Symbolic debugger
   - ☐ Language-sensitive editor
   - ☐ CMS
   - ☐ Source Code Analyzer
   - ☐ P&CA (Performance and Coverage Analyzer)
   - ☐ DEC test manager
   - ☐ Other, specify _____

3. Provide any other information about the interaction of Ada and this change that you feel might aid in evaluating the change and using Ada

5150G(1)-2

JULY 1988

Figure 1-6. Change Report Form (Ada Project Additional Information)

1.4.2 TYPES OF DATA COLLECTED AND USED

Several of the above data collection methods resulted in subjective information, but many of the methods produced quantitative data. Among the types of quantitative data collected are the following:

- Effort data by component level, weekly, from each programmer, manager, and support staff member
- Monthly estimates of size and schedule
- Component characteristics data for each new component
- Project characteristics--final size, number of components, phase completion dates, etc.
- Change and error information
- Amount of computer time used and number of runs made
- Number of executable versus nonexecutable versus commentary lines of code
- Number of each type of statement
- Number and types of modules

SECTION 2 - IMPLEMENTATION ISSUES

2.1 INTRODUCTION

An object-oriented approach was used during the design phase of GRODY. The design notation used object diagrams primarily, and PDL was developed for some of the modules during the design phase. Compilable Ada code was used for some of this PDL, but most of the PDL was an Ada-like type of pseudocode similar to that used in FORTRAN PDL. At the time of the critical design review for GRODY, many team members felt that the design was still not complete. Thus, a considerable amount of time early in implementation was spent on what the team considered design activities. More information on the design phase of GRODY can found in Reference 12.

2.2 THE ADA APPROACH

On the surface, it appears that the Ada team's approach to implementation was not significantly different from that used by the FORTRAN team. The implementation work was divided into builds, and portions of the work for each build were divided among the team members. Implementation activities are described below.

2.2.1 CODING PROCESS

Completion of the PDL involved writing prologs for each module. These included information on inputs and outputs, type information, and the purpose of the module.

The first build of GRODY was a complete implementation of the specifications for as many modules as possible in the system. (It was not possible to include the specifications for some of the nested modules until the bodies of their parent modules were included.) This early definition of the specifications was considered very useful because it defined all the interfaces very early in implementation. Since the

specifications are compilable, they were compiled to find any interface errors.

The initial build of GRODY also included the utilities necessary for GRODY that were not provided by Ada. These utilities were implemented in a generic library unit package.

GRODY had a global type package that contained many types used throughout the system. Although this package was considered necessary for GRODY's design, it caused many problems. It was difficult to define the appropriate type for all variables in the system so early in implementation, but these types were needed to define the other high-level interfaces. Many were enumeration types containing variable names, and changes to these were necessary during the course of implementation. Because this package was used in so many places, these changes caused considerable recompilation of large parts of the system.

Most of the developers felt that the original design was easy to follow and to translate into code. Some developers preferred to code directly from the design as documented by the object diagrams rather than to take the time to write the PDL. Most of the developers felt that they could develop Ada code more quickly than FORTRAN code.

The method of translating the design into Ada was fairly language-independent and depended more on the individual programmer. One programmer described his method as a process of examining all the inputs listed in the design, such as input parameters, necessary data bases, and information to be maintained by the package. Then he examined the function of the module as stated by the math specifications. After determining that all the necessary information was available, he decided on the best method for coding the module.

The developers used an Ada style guide (Reference 9) as a reference. This style guide was developed by some of the team to document recommended standards for Ada coding to be used throughout GSFC. It deals with the formatting of the Ada code and provides direction on the use of Ada features in order to promote good programming practices. For example, it gives guidelines on the use of such features as computed constants, infix operators, and derived or private types.

Most of the GRODY development team felt that the style guide was very useful, though they had some suggestions for its improvement. For example, according to the style guide, declarations should be spread out over multiple lines so that there is one line for the variable name, one for the type, and one for initialization. The team thought fewer lines could be used for declarations. Some developers felt that adapting to the style recommended by the guide required a period of adjustment, as they were accustomed to the style of programming used in some Ada textbooks.

One of the style difficulties encountered by the developers was determining the amount of documentation necessary within the code. That is, what should be commented and where? Team members felt that much of the explanation of the Ada code was obvious due to Ada's readability. The style guide recommended more commenting than the team felt was necessary.

2.2.2 DESIGN ISSUES

As mentioned in the previous section, it was necessary to continue some design activities during the early implementation phase of GRODY development. Some entire functions, such as the report generator and the plot generator, were designed after the CDR. Most developers felt that the completed design was fairly comprehensive and easy to follow.

Most changes in the design after the very early implementation stage were additions, rather than changes, to the original design. Other reasons for design changes include the following:

- Additions to the system
- Poor understanding of some features (for example, tasking)
- Errors in design
- Improvement of design
- Performance improvements
- Inadequate function of standard package

Certain design problems and deficiencies became obvious during implementation. One surprising aspect of the design is that it produced a highly interconnected system: the modules were dependent on other code and could not be executed independently. This made testing more difficult (see Section 3). Most developers had expected the object-oriented design approach to lead automatically to a modular system. FDAS also developed into a interconnected system using an object-oriented design methodology. So, it appears that while object-oriented design is not incompatible with independent partitioning of subsystems, it does not automatically lead to it.

One problem with the design that emerged during implementation was that the functionalities designed into procedures were not communicated well. During the design process, functionality was designed as a part of the package and was not explicit on the procedure level. Prologs that were written for the Ada code described the purpose of the procedure, but were not specific about the actual algorithms to be used. FORTRAN prologs usually contain the explicit

algorithm descriptions. Thus, there was more confusion about exactly what a particular piece of code should do. Many more meetings were necessary to clarify the functions of various modules.

Another design problem that became obvious during the design was the representation of tasking in the design. The design notation showed task dependencies, but was not explicit about control interactions (i.e., when tasks need to interact and when they do not). Thus, it was not clear to the developers where "accepts" should be coded or what sort of action should be taken if an expected rendezvous does not occur within a specified time interval. It was also hard to determine when a task should terminate. In some instances, it was possible for a task to indirectly call itself--that is, to produce a cycle. As a result, it was difficult to guarantee that no deadlocks would occur. (See Section 3.3.1 on tasking.)

Tasking was new to the developers and a successful implementation using the design description was difficult. In the future, it would be helpful to develop a better way to describe tasking in the design, including an overall description of task interactions. This is especially important since the most experienced members of a development team usually develop the design; those likely to perform implementation may be much less familiar with the particular details necessary to implement tasking so that it performs the function intended by the designers.

Some redesign of GRODY was necessary because features did not work exactly as expected or because standard Ada packages did not perform a required function. An example of this is the standard Ada calendar package. Initially, this package was implemented, but during integration testing it was discovered that this package was not producing the

correct numbers. This was because the resolution on the clock time was only 10 milliseconds and a higher resolution was necessary. The team rewrote the package using a private "long float" type to improve the resolution.

In a few instances, errors were discovered in the original design while the details of the procedures were being implemented. Some of these design errors were probably due to the designers' relative unfamiliarity with the use of Ada and with this type of application. Other design errors were similar to those that might occur in a FORTRAN project.

Some design changes occurred because the team found a better way to design a particular function. An example is the redesign and recoding of the report generator and simulator results sections of the system. The redesign enabled these sections to take advantage of the overloading features that had been coded. In a more time-constrained development, this type of redesign probably would not have occurred. Since GRODY's schedule was fairly unrestricted, there was a tendency to make changes to improve the quality of the system. Many improvements were made as the team's experience increased during implementation.

An example of a design addition is the "debug collector" package. The purpose of this package is to collect all the system's debug information into one area. It also provides an option to allow different amounts of debug information to be collected. Such additions as this were intended primarily to improve the quality of the system, but were not part of the basic system requirements. Normally, several levels of debug output are built into a FORTRAN simulator.

A final set of design changes was made very late in implementation, during the integration testing phases of the system. These changes were necessary to improve overall runtime performance of the system. For example, the team

discovered that calls to direct-access mixed input/output (I/O) were taking approximately 0.1 second of central processing unit (CPU) time for every call, and the design required many calls during each run. (A typical case was 5,000 calls for a 30-second data span.) The design was changed to buffer information and reduce the number of calls; this in turn reduced the amount of CPU time necessary for a run. This I/O problem was further complicated by the fact that the original design was intended to reflect the physical operation of the system that it was modeling--in this case, the OBC. However, this representation did not always result in the most efficient software design. In one case, it resulted in the computation and recording of ephemeris data four times each computing cycle, when once would be sufficient.

In other cases, inefficiencies were discovered in coordinating the screen management and the tasking (see Section 3.3.1). The original interaction between the Ada tasks and the screen management allowed the user interface to get control of the CPU whenever the computations paused for I/O. The user interface would then hold the CPU for a full time-slice even if no requests needed to be processed. This problem resulted from the method used by the DEC Ada runtime system to schedule tasks waiting for I/O when the screen management system was also in use.

2.2.3 TOOLS

The tools used during the implementation included the Ada Compilation System (ACS), the Ada compiler, and the Baron Templates. The VAX Configuration Management System (CMS) was used to provide configuration control of the source code. The debugger was used extensively during unit testing and integration and will be discussed in Section 3.2.2.

The Baron Templates are fill-in-the-blank templates for the primary Ada constructs, making it easier for the implementer to code the constructs properly. Those team members who were not as familiar with Ada found the templates useful, but the team members who had used Ada on other projects preferred to use the VAX Digital Standard Editor (EDT).

During implementation, each developer had a sublibrary in which the newly developed code was placed and unit tested. As soon as the units for a particular release were unit tested, integration began, and the source code was placed in a controlled library under CMS. All changes to this library were made by one person who would notify the team of any changes. For several reasons, the developers using GRODY were somewhat slow about placing their code in the controlled library. First, once the code is in the controlled library, a change form must be filled out to change the code and all changes must be carefully recorded. The developers tended to keep the code in their own libraries so they could correct as many errors as possible before a careful accounting of errors was necessary. In addition, once the code was in the configuration library, changes made to other modules were more likely to force developers to recompile large amounts of code: the Ada compiler forced a recompilation of any unit that was not current, even if the source code change was something as simple as a comment. Since the compiler was very slow, this was an annoying problem to the developers. In general, these large compilations were done overnight to optimize machine response time during the daytime, but many times a developer would be unaware that a change had been made that would affect him, and it would be necessary for him to recompile before he could proceed.

Some of the developers alleviated this problem by pulling routines out of the controlled libraries and into their own

libraries before changes were made. With their own copy of the routine, they would not be forced to recompile those routines when changes were made to the copies in the controlled library. This worked because the Ada compiler would accept the version of the routine in the developer's library and not look for the more recent version in the controlled library. Some developers used this method often to avoid some of the problems with unexpected recompilations caused by other developers' changes. This was a fairly effective way of delaying recompilation until a more convenient time--for instance overnight, when lengthy compilations could be accomplished more easily. This method is not recommended as a complete solution to the recompilation problem, because it causes a developer to use versions of modules that are not completely up to date, but it did provide the developers with a short-term solution that allowed them to make better use of their time.

All of the developers commented that extra recompilation was a problem, especially if it had to be done during the daytime, because GRODY's priority during the daytime was very low and recompiles received very little CPU time per elapsed time.

Another procedure that complicated the library structure and the configuration control was the fact that there was a separate library for each release. This meant that corrections put into one release library must also be placed into the other release library if they both contained that module. This was initially intended to save time by allowing parallel development, but some of the benefits were lost due to the overhead of maintaining two release libraries. See Section 3.2.2 for more information on library structure.

The teams felt that a tool to provide a graphic representation of the library structure would be useful. Since Ada

has a much more complex library structure than FORTRAN, a thorough understanding of that structure is important. Such a tool would be very helpful in configuration control.

A language-sensitive editor (LSE) was not available to the GRODY team, but many team members felt that one would be beneficial. One team member had used an LSE during FDAS and missed it on the GRODY project. Other team members felt that an LSE would be useful if it were fast and did not force the developer to make changes immediately after the need was discovered.

The team felt that some type of pretty printer would be useful in performing the tedious and time-consuming task of formatting the code. They also determined that an automatic stub generator would be helpful, to generate a dummy module that would satisfy the compiler and linker during testing without requiring that the actual routine be present. The team wrote a tool to do this automatic stubbing that will also create a skeleton for a package body and provide the appropriate documentary boxes for the names of the bodies and specifications. This tool also aided in documentation.

2.2.4 TEAM CONSIDERATIONS

During implementation, the work was partitioned out to the team members according to objects in the design. In general, these objects corresponded to packages that gave each team member a convenient, unified section of the project to work on. This method worked fairly well and team members preferred it to being assigned work that had been divided up by procedures.

As mentioned earlier, many meetings were necessary during implementation to clarify the actual function of specific procedures in the design. Close communication was also necessary to keep team members informed of changes in other portions of the system that would affect them.

2.3 COMPARISON OF ADA VERSUS FORTRAN

One of the most obvious differences between the Ada and the FORTRAN projects noted during implementation was the size of the systems being generated. The Ada language is responsible for some of this code increase, since about one-third of the code consists of specifications. Ada statements are longer than FORTRAN statements and Ada variable names are longer. More blank lines are used in Ada and certain types of statements, such as declarations, are spread over multiple lines for readability. The usual FORTRAN style does not spread out similar statements. The team felt that the Ada code was much more readable than the FORTRAN code. The Ada code also contained more comments, because features such as tasking were more complicated and needed more explanation. Ada also provides more capabilities, such as exception handling, that had to be coded and commented. Basic utilities that are provided with FORTRAN were not available with Ada and had to be implemented by the team. Other reasons for the size difference are described in Section 2.3.1.2 on "call-throughs" and Section 3.3.4 on size.

2.3.1 ADA-SPECIFIC FEATURES

The team attempted to use all the Ada features that seemed applicable to the problem. However, they wanted to be conservative and not jeopardize the project by relying on a feature that did not work properly or was too troublesome to implement. Overall, the team was very successful in their use of the features, but some features were found to be more difficult to implement than others.

2.3.1.1 Language Features

Generics, which were fairly easy to implement, were used in the utilities package to perform similar types of mathematical calculations for different types of objects. They were

found to be quite effective and reduced the amount of code necessary to perform certain functions. One early version of the compiler had problems with generics, but these problems were corrected in a later version.

Another feature that was easy to implement and quite beneficial was the capability of separating the bodies and specifications of procedures and compiling them individually. This capability was exercised in most cases, and wherever possible the specifications for the system were developed before the bodies were coded. (This was not possible for the nested units, where the body of the calling module must be coded before the specification of the nested unit.) The early development of the specifications helped clarify the interfaces, and the separate compilation of bodies and specifications reduced the amount of recompilation necessary when changes were made. The team did not save as much recompilation time as was expected, because there were many changes to the interfaces. These changes resulted from an inability to form an accurate, detailed, high-level view early in the implementation. They were also the result of problems with the types initially chosen for interface parameters. Another advantage of the separation of specifications and bodies is that it encouraged parallel code development. Another team member could easily code the body of a procedure, using the specification already coded. Parallel development is also used for FORTRAN, but there is no similar method for clearly defining the interfaces in advance.

The strong typing of Ada was new to most of the GRODY developers, and most felt it took time to become accustomed to using it. The tendency was to create too many types. A type would be created with a strict range for a particular portion of the application. Then other areas of the application would need a similar type, but the original one would

be too restrictive. So another type was created, along with a corresponding set of operations. Some of the difficulty with this method of typing began to emerge as early as critical design, when interface problems developed due to typing differences. The extent of these problems was not fully recognized until far into implementation.

The GRODY team considered tasking the most difficult feature to implement and test. First, tasking was a new idea and some aspects were not clearly understood by the team. This led to errors in both the design and coding of the tasks. The original design of GRODY called for two concurrent functions; thus, two tasks initially were designed. One task was in the user interface and could interrupt the processing at any point to write status information. The other task was in the simulator, which performed computations. The simulator and the user interface were intended to operate asynchronously and independently of each other. Either of these tasks could call the other. One by one, additional tasks were added in the user interface until there were eight or nine tasks. Some tasks were added to control various problems discovered during testing and will be discussed in Section 3.3.1. Other tasks were added because the developer viewed them as a good method of implementing a particular function.

Locally, from the point of view of a particular developer's section of the project, the additional task may have been a good idea. However, the problem arose when this new task had to interact with all the other tasks. Some of the new tasks could not operate correctly in a concurrent manner and ultimately had to perform their functions sequentially. That is, the functions to be performed by some of the new tasks were more appropriate for sequential processing than for parallel processing since the output of one function was required for the next function. The developers found the

originally designed tasks more difficult to code from the design than other types of units. The dynamic relationships of tasking could not be represented in the design, leaving the developers with questions about termination, rendezvous, and multiple threads of control.

Information hiding, a software engineering principle that is emphasized in Ada, was new to some developers with extensive FORTRAN experience. Two developers commented that they felt very uncomfortable with information hiding because they couldn't see what was going on in other portions of the code. They felt insecure about the function of private units and wondered whether they were really doing what was expected.

Portability is another desirable feature considered when the Ada language was developed. The team kept portability in mind when they designed and implemented GRODY, but they found good reasons for choosing features that were not portable. For example, originally the software-simulated, portable, floating-point number representation was chosen, but it was discarded in favor of the hardware-dependent representation for efficiency when the software representation was found to be too slow. Other nonportable features were used because no equivalent feature was available in Ada. These features included DEC utilities to interface with the standard math libraries and the DEC screen management software. In order to retain as much portability as possible, these features remained localized, even though some of them were fairly large modules.

Table 2-1 shows a subjective assessment of the team's attitudes concerning the various Ada features they used. These ratings were obtained by compiling the team's opinions when asked, "How easy was it to implement or use this particular

feature?" and, "When the feature was successfully implemented, were you pleased with the results? (Did you find the advantages of using this feature worth the trouble of implementation?)".

Table 2-1. Subjective Assessment of Ada Features

ADA FEATURES	IMPLEMENTATION EASE	BENEFIT
Tasking	–	+
Generics	+	++
Strong typing	0	0
Exception handling	0	+
Nesting	+	–
Separate specs/bodies	++	++

5399G(25)-46

Ratings represent a subjective assessment based on team member interviews.

2.3.1.2 Library Units and Nesting

A library unit is defined as the outermost level specification in a file for a package or procedure. In general, multiple library units compose the outermost lexical level of a program. A task cannot be a library unit. Nesting refers to the practice of encapsulating package, task, or subprogram specifications inside another package, subprogram, or task body.

The choice of using nesting or library units during implementation and the degree to which each is used greatly affects the final product. This choice seems to be an implementation question, but actually the representation of the design and the teams' view of that representation may influence the decision and cause a strong inclination toward one or the other. For GRODY, the design document showed dependencies, but did not actually call for a nested approach. The team felt that the design could have been implemented successfully using either, but it seemed to them

that nesting was a natural manifestation of the object-oriented design.

For GRODY, the library units went down about three or four levels, while nesting went eight to ten levels below that. During implementation, most team members felt an appropriate balance had been reached between nesting levels and the number of library units. One view of GRODY shows that it has 124 packages, of which 55 are library units. One team member who had worked on FDAS, which used library units almost exclusively, felt that heavy use of library units would have been more appropriate for GRODY also. In retrospect, most GRODY team members felt that nesting had been overdone, and provided some suggestions on future use of nesting and library units.

Experience with unit testing seems to indicate that library units should be used at least down to the subsystem level to make testing easier (see Sections 3.2.1.1 and 3.2.2). Below this level, the benefits of nesting sometimes dictate its use and thus determine the transition point from using library units to nesting.

An additional way to determine when the change from library units to nesting should occur is to examine the degree of interaction between modules. Modules that interact heavily should be library units. In GRODY, the Truth Model is a library unit, as are the four subsystems within the Truth Model, since these all interact heavily with one another. At the point where the interaction diminishes, it is preferable to switch to the use of nested units. In the example above, all four Truth Model subsystems that are library units contain nested components--for example, the sensors and actuators, which do not interact to the degree that the high-level modules do.

The final important consideration when choosing between nesting and library units is the issue of future reuse. Modules that have a high probability of future reuse should probably be library units. Library units can be reused much more easily than nested units since they are already separate and are not embedded in a larger module that may not be applicable for the future system. A nested unit embedded in a module not desired for reuse must be "unnested" to be reused. Experience gained during FDAS has shown that this can be a very labor-intensive process.

The design of GRODY suggested a highly nested implementation with many objects within objects. The degree of nesting during implementation was increased through the use of many "call-throughs," which are procedures whose only function is to call another routine. This was faithful to the design structure, so that a physical piece of code existed for every object in the design. The philosophy behind the use of the call-throughs was to group appropriate modules together, and to exploit information hiding. Nesting in general and call-throughs in particular help account for some of the additional SLOC in GRODY when compared to the FORTRAN version. It is estimated that of the 128,000 SLOC in GRODY, about 22,000 (including specifications and bodies) are the result of call-throughs. In retrospect, the team feels that some objects in the design should remain "virtual" objects: that is, they would exist in the design to clarify the logical structure of the system, but for efficiency they would be excluded from the implementation. While call-throughs provide a good way to collect functions into subsystems, their use should be limited to only two or three levels of nesting in the future.

Nesting has both good and bad effects on the resulting product. The primary advantage of nesting is that it structurally enforces the principle of information hiding, due to

the Ada visibility rules. With library units, the only way to avoid violation of information hiding is through discipline. Type declarations can also be located in one place with nesting.

One disadvantage of nesting is that it increases the amount of recompilation required when changes are made, since Ada assumes dependencies between even sibling nested objects or procedures, whether the dependency is real or not. Thus, with nested units more parts of the system need to be recompiled than with library units. It is also harder to trace problems back through nested levels than through levels of library units. There is no easy way to identify who called a module when it is nested, but that information is provided by the "with" clauses of library units. When nesting is used, a debugger becomes essential to see what is happening at the deeper levels (see Section 3.2.2). Due to the difficulties in debugging the deeper levels with nesting, it is now believed that the extensive use of nesting instead of library units will make the maintenance harder. This is contrary to the team's earlier expectations based on their experience with a small training project in which nesting worked well (see Reference 10).

Library units seem to have a lot of advantages. Besides requiring fewer recompilations when changes are made and providing easier unit testing, every library unit can be made visible to any other library unit merely by the use of the "with" clause. Library units allow smaller components, smaller files, and smaller compilation units. The resulting system is expected to be more maintainable, since it is easier to find the unit desired and since there is no excess code. Reuse is also easier with library units since the parts of the system are smaller and each small function of the system is more likely to be contained in a separate unit that could be plugged in to another application. Configura-

tion control is also easier with library units, since more pieces are separate; that is, the ratio of changes per piece is closer to 1:1. The major disadvantage seems to be that a complicated library structure develops, which can lead to errors by the developers (see Section 3.3.5).

The advantages and disadvantages of nesting versus library units are summarized in Table 2-2.

2.3.2 TEAM COMMUNICATION

During implementation, much more team communication was necessary for Ada than for FORTRAN. Since Ada was new to the developers, the usage of the language was often discussed at these meetings. In addition, developers shared their experiences so that other developers could benefit from better ways of implementing Ada or avoid pitfalls.

Another topic of discussion during these meetings was the function of some of the lower-level routines, including location of necessary conversions and initializations, since the design described the functionality on a package level and the specific function of a procedure was not always clear.

Communication was also very important during unit testing and integration because the developer needed to know whenever something that he depended upon in the controlled library had been recompiled so that he could plan to recompile his dependent code as Ada requires. Since recompilation was very time consuming, developers preferred to do this necessary recompilation overnight. All of the developers commented that it was very discouraging to try to run something in the morning only to discover that they needed to recompile due to a change in the controlled library.

Table 2-2. Nesting Versus Library Units

NESTING	
ADVANTAGES	DISADVANTAGES
• Information hiding • Visibility control • Type declarations in one place	• Enlarged code • More recompilations • Harder to trace problems through nested levels • Can't easily identify source of call for a unit of code • Type declarations in one place makes reuse more difficult • Maintenance more difficult • Debugger required • Larger unit sizes inhibit code reading • Harder to reuse units

LIBRARY UNITS	
ADVANTAGES	DISADVANTAGES
• Fewer recompilations • Easier unit testing • Smaller components • Smaller files • Smaller compilation units • Less code duplication • Easier maintenance • "With" clauses show source of other code units used • Easier reuse • Easier configuration control	• No information hiding • Complex library structure

5399G(25)-47

2.3.3 FDAS DIFFERENCES

The different implementation problem faced by the FDAS team caused some implementation differences. The goal of FDAS was to have interchangeable modules within the system that could easily be replaced with other modules to form a new system. Thus, the team wanted all the modules to be easy to locate, and they preferred to have smaller modules. Because of this philosophy, they used very little nesting and very few call-throughs. Most of the units in FDAS are developed as library units. The team found that it was much easier to have units that would plug in and pull out with Ada modules than with FORTRAN modules.

The FDAS team also found it extremely useful to separate the specifications and bodies of units and compile them separately. In fact, they even discovered a way to use two different bodies for the same generic specification.

2.4 RECOMMENDATIONS

Nearly all of the GRODY team recommended that all the PDL should be developed using compilable Ada during the design phase. Applying this recommendation would have lengthened the design phase and postponed the CDR until later in the development. This technique would also allow GRODY's Build 0, which consisted of the majority of the system's specifications, to be generated during the design phase. This initial build of the specifications was considered a good way to begin Ada implementation. Further, the practice of separating specifications and bodies was considered beneficial and was recommended for future developments. The specifications should be under configuration control at the start of implementation. It was felt that more time spent on the design during the design phase would have led to a smoother implementation phase.

One recommendation for future Ada developments is to incorporate an abstract data type analysis into the design process to control the generation of types. A general new type would be defined, and then many subtypes of that type could be used in various sections of the application. This type analysis would provide the following advantages: operations would be reused, there would be fewer main types to manage, and families of types would be developed that would inherit properties from each other. Careful consideration of types during the design process could prevent uncontrolled proliferation of new types during implementation.

Another implementation recommendation concerning types involves the global type package and the use of enumerated types. One suggestion was to break the global type package into smaller packages with fewer types in each to localize the impact of changes in the packages as much as possible. Another suggestion was to have the capability of more flexible enumeration types. Certain enumeration types could get their values from an external file with the values listed as string literals. Then the list of values could change and no recompilation of the type package containing the enumeration type would be necessary. This method would take more planning, but could prevent much of the recompilation caused by changes in the global type package.

A frequent recommendation heard from team members is that library units should be chosen instead of nested units beyond the first few hierarchical levels and whenever there is a high degree of interaction among the modules.

Another recommendation for translating the design into code is that the number of call-throughs be kept to a minimum. It was felt that it would be better to leave some of the design objects as abstract objects with no direct correspondence to an object or module in the code. This would

simplify the structure of the code and reduce the amount of unnecessary code.

One programmer commented that his code consisted solely of procedures and that he had not used functions at all. In retrospect, he felt he should have used functions for modules used to compute a single value since they can then be used in declarations. Another programmer recommended that similar functions be collected in one package instead of spread out in several different packages.

Several team members felt that certain design and implementation changes would have increased GRODY's potential for reuse. As previously mentioned, the heavy use of nesting affects potential reuse. Another factor affecting potential reuse is that the basic utilities written for the system are grouped into one large package. Some of these utilities are general and could be reused, while others are GRO-specific and would not be suitable for another system. Reuse potential would be increased by grouping the GRO-specific ones in a package separate from the more general ones. One suggestion was to develop a hierarchy of reusable packages.

Also, more consideration for operational efficiency might have produced a design that could be reused more easily. The design was developed to simulate the sequences of events as they occur in attitude control systems, but this caused many computations to be done repeatedly, instead of saving computed results for future needs. Future simulators might need to operate more efficiently and thus would not choose to reuse the more realistic design. Potential reuse of a system needs to be considered carefully during the design phase of a project.

The use of some standard, like the Ada Style Guide used by the team, is recommended to provide guidance on the usage of language features and to establish a consistent standard of

coding for teams with several members. Team members suggested changing the style guide so that fewer lines of code are used for declarations and less commenting is specified. Another related recommendation was to use some sort of tool like a pretty printer to provide the consistent style desired without requiring an extensive investment of time on the part of the implementation team.

Ada makes the concurrency constructs of tasking readily available and there is a tendency to use tasking because it seems a convenient way to perform functions. Tasking was very difficult to control and test and it should be used only when really appropriate to the particular situation. If tasking is appropriate for an application, an overall view of the actions and interactions of the various tasks planned should be examined during design. Using tasking also affects future reuse since small changes in tasks can cause changes in their interactions with other tasks, making them more difficult to reuse in a different system. See Section 3.4 for more description of the design of tasks.

SECTION 3 - UNIT TESTING, INTEGRATION, AND INTEGRATION TESTING ISSUES

3.1 INTRODUCTION

One of the early expectations of the Ada team was that Ada would make unit testing, integration, and integration testing easier. The interfaces were defined earlier and thus could be verified sooner, the compiler was expected to catch more of the errors in Ada than in FORTRAN, and, finally, the English-like nature of Ada should make the discovery of errors easier, since it would help describe what a module was to do. Actually, several factors made the unit testing and integration unexpectedly difficult, as described in this section.

3.2 THE ADA APPROACH

The Ada team began the unit testing, integration, and integration testing phases using the same approaches generally used by FORTRAN teams. Initially, they tried to unit test each module individually and planned to use code reading. As the testing progressed, some changes were made in the testing approach.

3.2.1 PROCESS

3.2.1.1 Unit Testing and Code Reading

Typically, as soon as the code for a module is developed and a "clean compile" (a compilation with no errors) is obtained, the code is ready for unit testing and code reading. These are the first steps in verifying that the code actually performs the required functions. During unit testing, a developer, usually the one who developed the code, executes a module independently to check its function.

Code reading is the visual examination of the compiled code to verify that the code will satisfy the function assigned to the module in the design. The code is also examined for any errors that might not be discovered during compilation, such as logic and style errors. Code reading is usually done by another developer who is familiar with the design but not the particular module. Code reading and unit testing are usually done concurrently (see Sections 3.3.2 and 3.3.5).

The introduction of Ada as the implementation language drastically changed the unit testing methods. Traditionally, when FORTRAN modules are being unit tested, numerous debug "write" statements are added to the code. This method was not used for the Ada code because the additional compilation required would be extremely time consuming. Instead, the Ada code was tested with a debugger, which was considered essential. (More information on the debugger is included in Section 3.2.2.) A debugger is available for FORTRAN in the flight dynamics development environment, but it is not used often by the developers. Thus, the difference in unit testing is that the FORTRAN code is more altered than the Ada code.

Unit testing was found to be harder with Ada than with FORTRAN. In addition, the unit testing of GRODY was much harder than the team expected. FORTRAN modules are already relatively isolated and can be tested simply by adding the necessary global COMMONs. This makes the unit testing of these modules easy. On the other hand, the Ada modules are much more interdependent and require large amounts of "with'd in" code before testing can be accomplished. The FDAS project probably had even more interdependence between modules than GRODY. The FDAS team also felt that this interdependence increased unit testing difficulty. In addition, the embedded specifications in GRODY made the unit

testing more difficult since it increased amount of stubbing required.

As a result of these problems, a different approach was used during Ada unit testing. Since it was difficult to test a unit in isolation when it depended on many others, Ada units were integrated up to the subsystem level and then unit testing was done. Subsystem integration means integration of all units from the lowest level up to the lowest-level library unit. This integrated subsystem was tested in pieces by choosing only a subset of the possible paths or units at a time. The debugger is used to examine a specific unit since the test drivers cannot "see" the nested ones. In the usual FORTRAN development, no integration occurs until after the unit testing is complete.

Thus, the biggest difference between the way FORTRAN and Ada projects are handled up to this point is the incremental integration. This represents a change in the development life cycle of an Ada product, since integration and unit testing are done alternately rather than sequentially.

The library unit/nesting level issue directly affects the difficulty of unit testing. The greater the nesting level, the more difficult unit testing is, since the lower-level units in the subsystem are not in the scope of the test driver. This makes a debugger a required testing aid with Ada projects. It was felt that the use of more library units and less nesting would have increased the ease of testing.

Two other methods of handling nesting during unit testing were tried, but neither was very satisfactory. In one method, an inner package is extracted from the outer package and tested separately by including the types and "with'd in" modules used by the outer package. The other method involves modifying the specifications of the outer package

so that the nested packages are in the scope of the test driver (or can be "seen" by the test driver). Both of these methods required time-consuming recompilation. The team found that the best method of testing nested units was to rely on the debugger and test them in conjunction with their encapsulating packages.

The importance of unit testing may be related more to the application area than to the implementation language. Regardless of whether the language was FORTRAN or Ada, a more important consideration might be the application. If it contains a heavy concentration of calculations ("number crunching"), then unit testing might be much more valuable since calculation errors become evident when only a small amount of localized code is executed. On the other hand, if the application contains a complex pattern of data manipulations like those found in FDAS, then execution of larger portions of the system is necessary to isolate errors.

Unit testing of a mathematical unit can be a problem when the tester is not sufficiently experienced in the particular application field. A tester without an appropriate math or physics background may have difficulty determining the correctness of answers produced by highly mathematical routines.

The math specifications provide the algorithm to be used, but often fail to provide the range of "reasonable" input and output values. In general, these mathematical units were not interconnected units, so they could be coded and tested easily as long as the I/O was known. Nesting was not a problem in testing this type of unit.

3.2.1.2 Integration

GRODY development included a period of integration after unit testing was completed, during which all the units developed for a particular release were combined into one load module and tested as a unit. This integration was not

as extensive as it usually is in a FORTRAN project because some integration had already been done during the unit testing phase (see Section 3.2.1.1). A separate library was maintained for each release so that integration testing could be conducted while coding and unit testing for the next release continued (see Section 3.2.2 for more information on library structure).

The first release of GRODY to be integrated included the complex user interface and was more difficult to integrate than the team expected. The user interface includes practically all of the tasks in the completed system and requires the use of modules from the simulator subsystem in addition to its own modules. Many of the modules from different subsystems were in later releases of the system and had not been coded at the time of the first release integration. Thus, many stubs had to be created in order to link the user interface.

Typically, types of problems that are discovered during integration include performance problems (such as length of time to allocate files, time to do I/O, and tasking interaction), space problems, and errors in flow of control and in interfaces between modules. In addition, user interface screen messages were improved during integration. Tasking interaction was one of the major problems encountered during GRODY integration testing, since tasking was new to the developers and there were some misunderstandings about the way it worked. Some redesign of the tasking was necessary and some new tasks were added during the integration period. A more detailed discussion of these problems is found in Section 3.3.1.

One interface error that was not discovered until integration testing was an array that was being passed by value rather than reference. This error caused a space problem;

multiple copies of the array were being stored and the error was difficult to locate since it occurred in a task, resulting in the deadlock of five tasks. One of the difficulties was isolating the particular task causing the problem.

The Ada team expected to have fewer interface problems during integration than would normally be encountered in a FORTRAN development, for two reasons: first, the early development of specifications allowed the interfaces to be clarified much earlier than is possible in FORTRAN; and second, there were fewer parameters involved in interfaces due to the use of records. Contrary to expectation, many procedures had parameters that needed to be added or deleted during integration testing, and in some cases whole new procedures were needed. One reason for this was the poor understanding of a module's function. For example, two individual developers each may have thought the other was handling the initialization of a particular parameter, when, in fact, neither was. Another very common misunderstanding concerned the units of particular parameters. These problems emphasize the importance of communication among developers during implementation. In fact, the Ada developers did spend more time in meetings, but most of the communication involved problems that were new to Ada and not the types of problems that are typical of any development effort.

Integration was done on the subsystem level by the lead developer of that subsystem. Integration testing for a release was then done by an individual assigned to that task. Testing was done on a functional level so the types of tests performed were similar to those performed on the FORTRAN system. Some of the difficulties encountered during integration testing seemed to be related to the fact that the tester was unfamiliar with both the application and the

design, and hence had difficulty isolating the source of problems. Thus, a problem would be reported to the developer of the module where it seemed to occur, but often the problem was actually in another module. In some cases, the improper handling of an exception would cause a problem to appear to be in another module.

Tasking further complicated the problem of isolating errors, especially where exceptions were concerned. First, it was difficult to determine if a task was running and which portion of the task was being executed. Then, if an exception occurred in the task, the task would terminate and the exception would not be propagated. Finally, exception handlers in tasks obscured system error messages so they were not displayed when they occurred. (See other problems with tasking and testing in Section 3.3.1.)

3.2.2 TOOLS AND LIBRARY STRUCTURE

The tools used during unit testing and integration included the compiler, DEC's ACS, DEC's CMS, and the debugger. The compiler was found to be very useful for pinpointing the types of errors that are often found during unit testing. Some early problems with the compiler included incorrect compilation of generics, incorrect code generation, and incorrectly optimized code. For example, in the code generated, type word would be used instead of type longword. In other cases, some necessary values were deleted from the optimized code so that the resulting code would not execute properly. With the generics, code was not compiling even when syntactically correct; later, the code generated was not algorithmically correct. During the course of this project, the compiler matured and most of the problems were corrected in later versions. However, the compilation remained very slow, which caused the developers to look for methods of avoiding excess compilation.

The debugger was found to be an indispensable tool during unit testing, integration, and integration testing, since it allowed the developers to violate the normal Ada scope rules and "see" into modules which were nested. It also helped to avoid the edit-compile-link-run cycle, since variable values can be changed during execution when problems occur. For instance, if a variable had not been initialized, it could be initialized with the debugger and execution could continue to see if the remainder of the module worked properly. One developer commented that the debugger made integration in Ada much easier than in FORTRAN, but integration without a debugger was much more difficult than in FORTRAN. For example, the debugger discovered the generation of an incorrect value when records were being skipped; code reading did not find this error because the logic leading to the error spanned several modules.

In the early phases of development, the debugger interface to the rest of the DEC system did not work properly and the GRODY team had to get a prerelease version of the debugger to continue using it. Even as the debugger matured, there were problems with it, and many of the developers commented that they wished it were improved. The debugger's limitations included its inability to enter some nested routines due to difficulty locating the source, its tendency to "get lost" and be unable to identify where it was, and its limited ability to provide information during tasking. The developers also commented that sometimes errors would occur when using the debugger that did not occur otherwise.

Code was managed using both the DEC ACS and the DEC CMS. Figure 3-1 shows the library structure used in GRODY. It shows that there was only one CMS library, which was the controlled library and contained all of the source code as it was placed under configuration control. The ACS

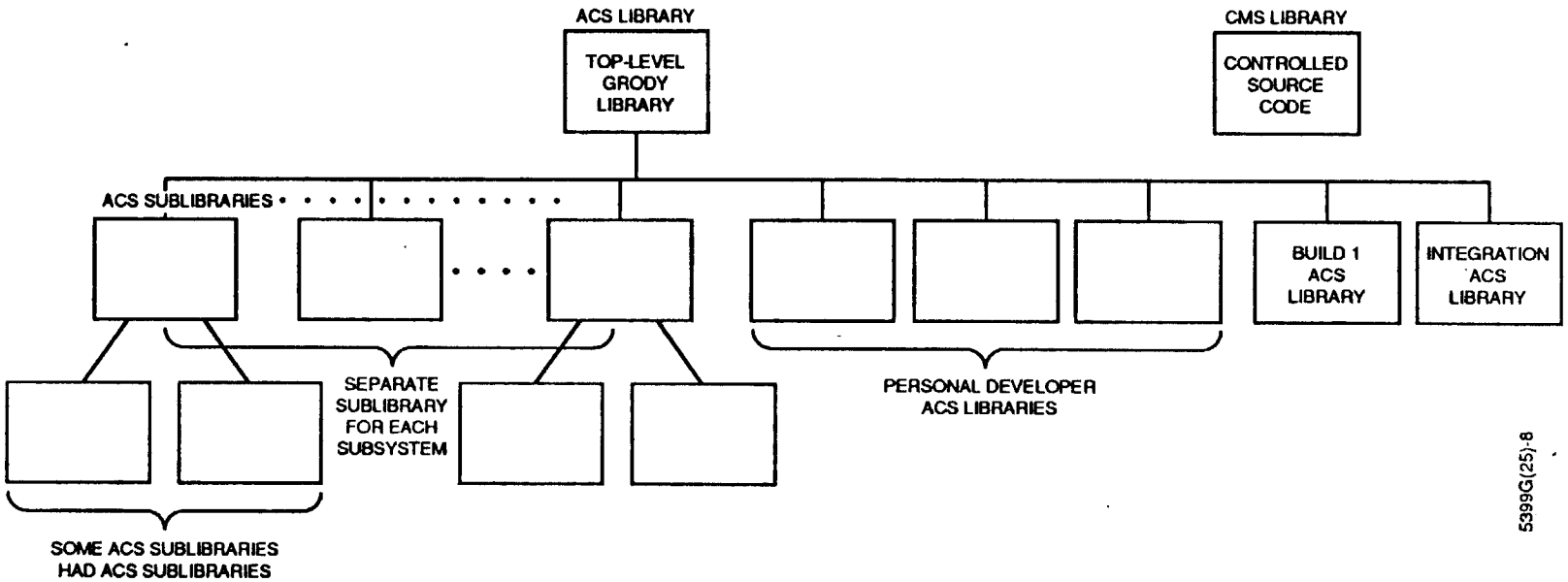


Figure 3-1. Ada Library Structure for GRODY

libraries are used by the Ada compiler and contain the object code, the files necessary to track module dependencies and other information necessary for automating Ada's complex library functions, and the source code necessary for automatic recompilations.

The top-level ACS library contained all the global code, such as utility packages and global type packages, used by all the subsystems. The next level of ACS libraries contained several types of ACS sublibraries. First, there was a library for each subsystem which contained the code for that subsystem. Some of these subsystem sublibraries had a level of sublibraries below them for portions of the subsystem. The library structure was designed in this manner because the ACS begins searching in the lowest-level library to find necessary components, and when they are not found, examines the next-higher parent library. On the level with the ACS subsystem libraries were individual developer libraries which the developer controlled. There were also two integration libraries on the level with the ACS subsystem libraries--one for integrating Build 0 and Build 1, and one for integrating Builds 2 and Build 3. (This library also included the code in Build 0 and Build 1.) This parallelism in the integration library structure was intended to increase the development progress, but there were some configuration problems with keeping two libraries current. When changes were made in the Build 0/Build 1 library, extra care had to be taken that the same changes were included in the other integration library. At the end of integration testing, all the code was moved into the top-level ACS library.

The existence of the CMS library helped to reduce the number of recompilations through the control placed on the library units. Only one developer was allowed to make changes to this library, and it was his responsibility to notify other

team members whenever changes had been made. With this library structure, the developer could use the ACS subsystem libraries or pull modules from the subsystem library into his own library to avoid recompilation of these modules (see Section 2.2.3).

The Ada project library structure was quite different from that usually used with FORTRAN and some developers had difficulty learning to use it properly. One developer commented that many of his compilation errors were caused by a poor understanding of the complicated Ada library structure and the dependencies of the code.

Although the tools available to the GRODY team were found to be useful, many other tools that were not available were placed on a wish list. Probably at the top of the list is a stubber that would take a specification and automatically generate a stub to satisfy the linker. This tool was written by members of both GRODY and FDAS, independently, to aid the integration process. A library of stubs was eventually built up using these tools. Another item on the list is a tool that could tell the compiler to ignore something, such as a module that was not present, for the same result as a stubber. Another desirable tool could examine a specification and identify any calling sources and external references.

Some additional capabilities were desired in the domain of the ACS. For example, a tool was mentioned that would make compilation more automatic and track the compilation order. At present, if a unit is changed such that the compilation order required for dependent units is altered, some of the recompilation must be done manually. These changes occurred frequently enough to warrant generation of a tool automating the process. Another desirable ACS-related tool is one that could automatically recompile out-of-date units. In the

current system, whenever an out-of-date unit is identified, the linker notifies the user and aborts the link. It would be much more convenient to have the option of automatically recompiling the unit and continuing with the link.

Other items on the wish list included tools to aid in documentation. For example, a tool might pull out all the specifications for documentation or even use them to create object diagrams or structure charts of the system. Some extraction tools were developed by the team to aid in documentation.

3.3 COMPARISON OF ADA VERSUS FORTRAN

3.3.1 ADA-SPECIFIC FEATURES

Just as some Ada features were easier to code than others, they also presented varying amounts of difficulty during testing and integration. The features that caused the most difficulty during testing were tasking, strong typing, the misuse of exception handling, and the extensive use of nesting.

The strong typing of Ada was found to be a mixed blessing during testing, especially for a team not accustomed to it. The rigor of the typing forced careful attention to that detail and prevented many of the typing errors usually seen early in FORTRAN implementation. Other errors were discovered during the compilation stage and were corrected even before unit testing and integration. However, the strong typing presented some new problems for the testers. It was more difficult to write test drivers to handle units with multiple types. One solution was to code the drivers as large "case" statements to test each type. The strong typing also increased the complexity of the I/O, which dealt with each type differently. One developer described the I/O as being "annoyingly different in the different levels of abstraction," due to the strong typing. The developer tended

to view many of the different types in the system as "merely numbers" when they were actually separate types. Multiple types also meant more operations and more code that required testing.

During testing, it became obvious that exception handlers must be coded very carefully to provide the maximum benefit to the Ada developer and user. In certain cases, improperly coded exception handlers tended to obscure errors instead of helping to pinpoint them. Suppose, for example, a module that is called to get the current Sun angle in turn calls the timer to get time necessary for calculation. If a serious error occurs in the timer routine, a FORTRAN program would terminate at that point. However, the Ada program will continue to execute, following the corrective measures specified in the exception handler. Suppose a null value is returned to the Sun angle module, which might use that and arrive at some incorrect calculation. This might cause a new exception at some point further on in the execution. By this point, it is hard to trace the error back to the timer routine. This problem can be avoided by ensuring that appropriate corrective measures are specified by the exception handler, instead of a "temporary fix" that just allows execution to continue. Properly coded exception handlers were found to be very useful in locating errors.

Tasking was, by far, the most difficult feature to test. Functional testing was used, but due to the concurrency, it was much more difficult to ensure the correctness of the tasks. It was necessary to show that the task was actually invoked and that it functioned properly. Traditional testing methods and the tools available did not provide as much of this information as the team would have liked. The misuse of exceptions in tasks further obscured the detection of errors. Error detection was also complicated by the fact that exceptions in tasks do not propagate except during

rendezvous, so tracebacks are not provided. Even when the exception occurs during rendezvous and propagates, the traceback begins at the rendezvous point and does not provide any previous information. Also, if the task calls a package not within its static scope and an error occurs, an exception cannot be propagated by an error in that package. In order to get a useful traceback during integration testing, the developers often needed to comment out the exception handlers in tasks. Then the higher-level exception handlers in the user interface or the operating system would catch the error and provide a traceback. Another problem was that tasking would prevent operating system error messages from being displayed on user interface screens that were created by the DEC screen management system.

One approach used to isolate tasking problems was to change each task to a sequential unit, one by one, until the problem was located. This was very time consuming, but was necessary since appropriate diagnostic information was not otherwise available. This is the method used to locate the error described in Section 3.2.1.2 concerning the incorrect passing of an array.

During integration testing, although the developers knew that lateral calls could occur in sibling level tasks so that some tasks could indirectly call themselves, they did not realize that this could cause cycles to occur. Thus, when two or more tasks needed to get a response from each other before proceeding, they formed a cycle and deadlocks could occur. This problem was aggravated by errors in the tasks and exception handling. These potential deadlocks were eliminated by creating a parent task to call any of the sibling tasks and control the exchange of information between them. The disadvantage of this approach is that it produced more tasks.

Another problem discovered during testing was caused by a deficiency in the DEC runtime system and the task scheduling. The runtime system makes a table entry to record the status of a task when it is elaborated (when its declaration appears). Possible status entries for a task include

- Ready--when a task has all the resources except CPU
- Suspended--waiting for a rendezvous
- Waiting on I/O
- Terminated
- Executing

The runtime environment uses these status entries for scheduling so that a task is scheduled when it is "ready," and execution continues until the time-slice expires or the status changes. "Ready" tasks are scheduled according to priority. The problem between the runtime system and the screen manager occurred when the screen management system was waiting for input from the keyboard. In this case, the status should have been marked "waiting on I/O," but instead it was marked "ready." The result was a very inefficient program, since the task spent a whole time-slice waiting for I/O. Thus, the tasks that were doing useful work got a very small percentage of the CPU. The development team was informed that this could be corrected by changing priorities, but they were unable to use this method of correction. Instead, another task was added to change the entry in the status table to reflect the true state of the task in question.

3.3.2 USEFULNESS AND IMPORTANCE OF CODE READING

Since the Ada compiler locates many more types of errors than the FORTRAN compiler, the types of errors found by code reading differ in the two languages. For example, common errors found during FORTRAN coding reading include wrong data types, calling sequence errors, and variable errors

(either the variable is declared and never used, or it is used without being declared). These types of errors were pinpointed by the Ada compiler and not found during code reading. Several Ada developers commented that code reading Ada would not be as interesting since many of the "interesting" errors had already been corrected. Most developers also felt that Ada code reading was easier due to its English-like nature. This difference was magnified by the fact that some of the FORTRAN code was reused code of an older variety without structured constructs to aid readability.

Several Ada coding features increased the difficulty of code reading. First, it was difficult to follow a path through parts of the system since so many call-throughs had been used. Second, the SEPARATE facility of Ada, which allowed pieces of the system to be compiled separately, resulted in the generation of many separate units that had to be examined to determine the correctness of a function.

The most common errors in Ada were style errors such as incorrect comments, format inconsistencies, or incorrect debug code. Other types of errors found during code reading included initialization errors and problems with design/code incompatibilities. Such incompatibilities might be caused by either design or implementation errors.

Some types of errors were difficult to find by code reading in either FORTRAN or Ada. Logic errors are difficult to locate in this application domain, and so very few were discovered, though enough were found to make code reading worthwhile. Errors that spanned multiple units were also overlooked in both development languages. One error not located during Ada code reading was an output problem in which records were being skipped. Code reading of the highly algorithmic routines appeared to be very important in

both application languages. Developers of both systems commented that code reading was the best way to locate such errors as using "less than" instead of "less than or equal."

The Ada team also felt that code reading had an additional benefit as a training tool. As one developer read another's code, he would discover different or improved ways of coding the problem.

3.3.3 UNIT TESTING AND INTEGRATION

The unit testing and integration of GRODY was more difficult than was expected and more difficult than in the FORTRAN system. Several factors contributed to the differences in difficulty between the two languages. The first factor is the experience of the two teams with the languages they were using. The FORTRAN team was quite experienced in FORTRAN and had well-established procedures for this testing and integration. Library structure and configuration control methods were familiar to all the FORTRAN team members. Library structure of the Ada system was new to the GRODY development team, and configuration control methods needed to be established.

The other major causes of difficulty in unit testing and integration of the GRODY system were the degree of nesting, the tasking in the system, and the exception handling. Nesting can be used in a FORTRAN application, but usually is not as extensive as it was in the GRODY system. Testing of nested units is more difficult in Ada due to problems with visibility of parameters not in the scope of the driver. Tasking and concurrent processing applications are inherently difficult to test. This capability did not exist for the FORTRAN team, so problems with concurrency did not arise. Similarly, Ada's exception-handling capability provides much more flexibility and power than FORTRAN. With Ada, many more errors can be trapped and handled, but that

also means that all possibilities must be considered and appropriate actions provided so execution can continue in a satisfactory manner. Coding these expanded capabilities requires new skills. Error handling in FORTRAN is limited and controlled by the system, whereas Ada error handling depends on action specified by the error handler and rules of propagation. Error handling is developer-controlled and there is no uniform outcome. Thus, some of the difficulty encountered by the Ada team was due to the expanded capabilities available in Ada, some was due to inexperience with the language, and some was due to overuse of nesting.

3.3.4 SYSTEM GROWTH

In general, the GRODY system growth curve was smooth during implementation, in contrast to the FORTRAN development, which tends to grow by chunks and fluctuates as large amounts of potentially reusable code are added to the library and unsuitable modules are deleted. However, the GRODY development growth history did show a notable difference in the amount of code added to the controlled library after the beginning of system testing. GRODY system testing began in in July 1987. By then, implementation had officially ended, and the controlled library contained about 90K SLOC. However, there were still portions of the systems that had not been unit tested or placed in the controlled library. Most of the remainder of the code was gradually unit tested during the fall through about mid-December 1987, and placed in the controlled library. By mid-December, the controlled library contained 119K SLOC. The delay was caused by a lack of analyst support to aid the developers in the final verification of these modules, as well as a shortage of testing time on the part of the developers. By this point in the GRODY development, most of the developers had been placed on other projects with more critical schedules

and they found it difficult to find time to complete the GRODY testing.

The final chunk of 7 to 8K SLOC was added in April 1988. This portion of the code contained the Kalman filter, which had been difficult to unit test. The remainder of the code added during system testing seems to be intended to correct errors. The final system size of GRODY is 128,064 SLOC. While it is not unusual for a FORTRAN project to begin system testing before all of the unit testing is completed, it is unusual to find as much phase overlap as we found in this project. It appears that some of this problem can be attributed to the lack of a firm schedule, rather than specific problems due to Ada.

3.3.5 ERRORS

A detailed discussion of the error statistics can be found in Section 4.3.7. This section will discuss some observations made by team members concerning errors. Ada team members felt there were about the same number of errors in Ada as in FORTRAN, but early data from the SEL change report forms indicated that more errors were discovered in the Ada code. Later statistics show that the rate of errors per line of code is about the same for the projects, although the actual number of errors is higher for the Ada project. Some team members felt that there were actually fewer errors in the Ada code, and that more of the errors were being isolated, at an earlier stage, with less testing to isolate them. There was also more reworking of the Ada code to correct errors and to make improvements, probably in part because of inexperience with Ada and in part because the team was not constrained by schedule pressure. This resulted in increased implementation time. This may be offset by increased reliability. Team members felt that future Ada

projects might be able to achieve lower error rates than FORTRAN projects.

The object-oriented methodology had an effect on the types of errors made (see Section 2.2.2 for more information on design issues). A poorer understanding of the function of the units resulted from the nonalgorithmic descriptions of these units below the package level. As a result, the Ada team required more meetings than FORTRAN teams usually require, in order to clarify the functions of the units. A number of errors occurred when a team member assumed, incorrectly, that a certain module performed a certain function.

Additionally, the strong typing of Ada required frequent type changes, especially for enumeration types used in parameters. This led to many recompilations (see Section 2.2.3) of large portions of the system, because the simulator types were located together in one high-level package. Except for the many changes necessary in parameter lists, interface problems were infrequent in contrast to FORTRAN, where these errors make up a significant percentage of the total usually found (see Reference 5).

As expected, producing a "clean" compilation (one without any errors detectable by the compiler) was more difficult in Ada than in FORTRAN. The team estimated that this usually takes about an hour. One reason for this is that an Ada compiler is much more stringent than a FORTRAN compiler and checks for many more types of errors. Secondly, the newness of Ada caused some problems in compilation, since some of the more obscure language features were not completely understood by the team. For example, "delay" and "terminate" cannot both be in a given task, and exceptions in a task body terminate the task rather than propagating the error, except at the rendezvous.

Once the code compiled successfully, the team felt more confident about the reliability of the code: they felt that syntactic, semantic, and logic errors were less likely to be present and that the longer time spent achieving correct compilation would be offset by a decrease in the amount of time necessary to unit test. In fact, this savings was not realized, because unit testing was much more difficult than the team had expected (see Section 3.2.1.1).

As previously described, error detection in tasks was much more difficult than in other types of units. A great deal of integration time was spent in debugging tasks and coordinating their execution. This problem did not occur in the FORTRAN development since there was no concurrent capability and no type of module corresponding to a task.

It was very hard to isolate the errors in some types of data structures, such as the complex tree structure with pointers that is found in FDAS. In this case, the debugger was no help, so code reading and hand tracing were required. This type of problem is language independent.

Exception handling features of Ada were very useful for error detection, but, as discussed earlier, they could obscure the errors when improperly coded. In FORTRAN the error would cause the unit to fail and a traceback would be provided, but in Ada the error is detected and the action taken depends on the code in the exception handler.

Another difficulty experienced in discovering errors in the Ada code is a tendency for programmers to be lazy. Overconfidence in the error-detection capabilities of the compiler and the runtime system causes errors to be overlooked. Programmers found they could look at the same error repeatedly and still fail to recognize the problem.

Team members also found that they had a certain intuitive understanding of FORTRAN that helped them recognize the source of an error which manifested itself in a particular way. This intuition did not seem to transfer to Ada. One developer very experienced in FORTRAN commented that he located FORTRAN errors by knowing what the compiler does internally. The actions of the Ada compiler are hidden, so this method was not useful in Ada.

Error correction was generally not a problem in either Ada or FORTRAN. One difference was noted in dealing with Ada nonlocal errors (those involving more than one routine). Here, recompilations were sometimes a problem due to the slowness of the compiler and general unfamiliarity with the library structure and dependencies of Ada. These factors sometimes caused other errors.

Style errors were the most common errors discovered during code reading. Other types of errors were initialization and problems with inconsistencies between the design and code. In general, logic errors are hard to isolate in this application domain, but enough were found to make code reading worth while.

Unit testing found some obscure errors. Among these were endless recursion, switching matrix element positions, and typographical errors.

3.4 RECOMMENDATIONS

As noted in Section 3.2.2, Ada's library structure is much more complicated than that usually used with FORTRAN. A thorough understanding of this structure is very helpful when developing a large Ada system and it was suggested that some training on library structure be included in any Ada training.

Code reading was considered as useful in Ada as in FORTRAN, but it might have a slightly different emphasis in Ada. The team felt that the greatest value of code reading occurred in highly algorithmic routines, where complicated equations needed verification, and in determining that the code actually conformed to the design. An additional suggestion was that code reading should be used as a training tool to help new developers improve their use of Ada.

The team had several recommendations concerning unit testing. First, they felt that unit testing should be redefined so that unit testing is actually done on the package level instead of the module level. Modules up to the package level would actually be integrated before unit testing is attempted. This simplifies the testing problems by allowing any embedded module (or nested module) to be tested in conjunction with its parent.

Second, they recommended less nesting to improve testing ease. They suggested that library units should be used down to a much lower level in the system. This would enable many more units to be tested separately and would reduce the amount of recompilation that the team found necessary during unit testing. Both methods used to isolate nested units and their encapsulating packages--extracting the nested unit, or modifying the specifications of the outer package, as described in Section 3.2.1.1--significantly increased the amount of recompilation necessary.

The number of stubs required for testing was a problem for the Ada team. One suggestion for future developments is to create a library of specifications at the beginning of implementation and keep it in the configuration library. Then the necessary modules for linking would be available for integration at any point.

One non-Ada-related suggestion for integration testing is to have at least one tester on the test team who is thoroughly familiar with the application. Then, when results are obtained, the tester will be able to determine whether they are mathematically correct. If they are not, the tester should be able to determine the source of the problem by examining the reasonableness of other intermediate results.

Several team members recommended incorporating exception handling features into the design, rather than as an implementation detail. Exceptions would be part of the abstractions developed during design. The design should specify what exceptions should be raised, where they will be handled, and what action should be taken.

The team recommended that tasking be used very conservatively. Ada makes syntactic constructs for tasks so readily available that it is easy to overuse this feature. The control of tasking should be considered during the design phase, where an overview of all the tasks should be developed that would contain a picture of each task, where it is located, and how it interacts with all the other tasks. An overall view during the design phase would probably have reduced the number of extra tasks added during testing by providing more control over task interaction. The team also felt that the design notation should show control interaction (i.e., when the tasks need to interact and when they do not). The design notation used shows dependencies of the tasks, but does not specify information such as how to handle accepts, when tasks should terminate, and what happens if a task terminates abnormally. It would be helpful to include this information in future design notation describing tasks.

SECTION 4 - MANAGEMENT ISSUES

4.1 INTRODUCTION

Since GRODY was one of the first Ada projects in the flight dynamics environment, there was little experience with managing Ada projects. Initially, a management plan similar to the type used for FORTRAN projects was used (References 5 and 6). The development was divided into the same life-cycle phases and the same review pattern was used. Similar mechanisms for evalulating progress were established. One of the project objectives was to determine the usefulness of these methods during an Ada development. The Ada team had expected some differences in the development cycles of Ada and FORTRAN. For instance, they expected design to take much longer in Ada than in FORTRAN. They also expected implementation and testing to be much easier in Ada and to require much less effort. As described in this section, these expectations were not met during GRODY.

4.2 THE ADA APPROACH

4.2.1 ACCOUNTING METHODS FOR RECORDING PROGRESS

During the implementation of GRODY, managers needed to establish accounting methods that would enable them to track progress carefully. GRODY's implementation schedule was originally divided into four builds. (A build is an increment producing a partial working version of the system.) The first, Build 0, included the specifications for the system and the necessary utilities that were unavailable in Ada. The second build consisted mainly of the user interface, but also included some of the skeleton of the simulator itself. The final two builds were combined to complete the "meat" of the simulator. The plan for the build structure was formulated during the detailed design. The team felt that the Build 0 should have been part of the design phase, but on this project it was not.

To track progress during a build, a system was used that assigned a certain number of points to the completion of a specific unit of code, and portions of the total points were earned when certain milestones were reached. During the design process, all the units of code that needed to be produced were identified, named, and marked with a number in the design. These units of code were divided into two types, A and B. Type A units were units with no procedural code, such as a call-through unit or a specification for a package, and type B units were algorithmic units. A-type units were assigned a total of four points: two were earned after a clean compile was achieved, and the remaining two were earned after completion of code reading. B-type units were assigned a total of 10 points. Three points were earned for coding the prolog and PDL, three points were earned after a clean compile, one point was earned after code reading, and the final three points were earned after unit testing. After the number of points was computed based on the units that had been identified during design, a growth factor of 20 percent of the total number was added to get the point total used by the managers for tracking progress. The ratio of the points earned to the possible points (including the 20 percent) gave the implementation status of the project.

The manager's report included the following information:

- The unit's name and number in the design
- The unit's point status
- A flag indicating a unit that had been added after CDR
- The unit's subsystem
- The file name of the unit
- A flag indicating whether the unit was a specification or a body

- The unit's developer
- A flag indicating a unit that was a library unit
- The unit's type (A or B)

In addition, notation on the report indicated which units were nested and which were tasks. The team found this collection of information so useful that they plan to include it in the system description document.

Figure 4-1 shows an example of the GRODY implementation status report. It is interesting to note that near the end of implementation, on June 26, 1987, the report listed a total of 731 units that had been defined. Of these, 62 units, or 8.5 percent, had been deleted from the design at some time during implementation, and 230 units, or 31.5 percent, had been designed after the CDR. At this point the implementation status was recorded as 89 percent complete. Figure 4-2 shows the accompanying implementation status summary.

4.2.2 TRANSITION FROM DESIGN TO IMPLEMENTATION

The Ada team found that the transition from the design phase to the code/unit-test phase was very easy. In general, the team members commented that the object-oriented design was easy to follow and that the code developed more quickly from this type of design than from the design methods usually used with FORTRAN. Some of the team members even felt that it was easier to code directly from the object diagrams than it was to use the PDL. The PDL for this project was not written in Ada and was not entered into the machine during the design process. PDL was entered as one of the first steps during implementation and was found to be very time consuming. Most of the team felt that the transition from design into coding would have been even easier if the PDL had been written in compilable Ada and entered into the machine during the design phase.

*** GRODY ***

IMPLEMENTATION STATUS

COMPONENT TYPE:									MAX PTS.
A. TASK SPEC/ PACKAGE SPEC OR BODY							2 + 2		= 4
B. TASK BODY OR PROCEDURE BODY							3 + 3 + 1 + 3		=10

OBJECT NUMBER	SINCE CDR	REPS PERSON	BUILDS	* = LIB UNIT COMP & FILE NAME	COMP TYPE A	COMP TYPE B	PROLOG PDL	COMPILE	CODE READ	UNIT TEST	CONF CNTL	
USER I/F												
1		DS	0	* UI S	X			2	2		C	
1		DS	0	UI B	X			2	2		C	
1	+	DS	0	* UI TYPES	X			2	2		C	
11		EB	0	* UR S	X			2	2		C	
11		EB	0	UR B	X			2	2		C	
113		EB	0	USERIS	X			2	2		C	
111		EB	1	UR TOPLVL		X	3	3	1	3	C	
112		EB	1	UR STDISP		X	3	3	1	3	C	
UI - REQUEST I/F												
113		EB	0	UR USERIB	X			2	2		C	
1131		EB	0	PASURS	X			2	2		C	
1132		EB	0	MODES	X			2	2		C	
1133		EB	0	USERHS	X			2	2		C	
1134		EB	0	ERLGS	X			2	2		C	
1131		EB	1	UR PASURB		X	3	3	1	3	C	
1132		EB	0	UR MODEB	X			2	2		C	
11321		EB	0	ACTNS	X			2	2		C	
11321		EB	1	UR ACTNB		X	3	3	1	3	C	
11322		EB	1	UR ROUTER		X	3	3	1	3	C	
11323		EB	1	UR EXITSM		X	3	3	1	3	C	

5399G(25)-48

Figure 4-1. GRODY Implementation Status Report

DATE: 6/27/87

*** GRODY ***

IMPLEMENTATION STATUS

SUMMARY

BUILD	TOTAL POINTS DEFINED	DONE	
0	1576	1490	
1	2196	2190	
2	970	734	
3	586	352	
TOTAL	5328	4766	(89%)

5599G(25)-49

Figure 4-2. GRODY Implementation Status Summary

Most of the team felt that Ada documentation was clearer than the usual FORTRAN documentation and provided a good method for describing functions. They also felt that the design notebook was easy to understand. The Ada packages, which corresponded to objects in the design, seemed to provide a natural way to view objects such as sensors or actuators in the context of the whole system. This packaging of objects seemed to make the development easier.

The one area where the object-oriented design was not easy to translate into code was tasking. Several team members felt that the design was not specific enough concerning the interaction of the tasks which were specified. (More information on these problems is included in Sections 2.2.2 and 2.3.1.)

One developer commented that he felt Ada could be used equally well without using object-oriented design methods. He felt these design methodologies had actually hurt the reuse potential of the system because they led to numerous links between the subsystems. Since the GRODY development was done in a top-down fashion, he felt that GRO-specific assumptions were incorporated into the system at a higher level than necessary. This approach resulted in easy adaptability to the changes in the GRO requirements, but not to future simulator projects. He suggested that some packages should have a bottom-up development so they would be more independent.

4.2.3 STAFFING CONSIDERATIONS

Work during the code/unit-test phase was allocated by assigning different team members certain packages to code. This seemed to work very well and also facilitated the addition of staff once the project was underway.

Parallel code development was also easy and effective in Ada. Contributing factors here were the early definition of interfaces and the fact that the interfaces were much better defined than they usually are in a FORTRAN development. In addition, the package specifications, which were done early, enforce consistency among modules.

Developers who were on the project from the beginning felt the design was very easy to understand and that it should be easy to add staff after design or during implementation, since interfaces were well defined and work was neatly divided into packages. New team members could begin implementing package bodies using existing specifications, without being too concerned about affecting other team members as long as they maintained the specified interface. However, two of the team members who were not on the project from the beginning had a slightly different view. One who joined the project after the design phase felt that it was difficult to get oriented on the project, even though he already knew Ada, because he did not understand the design notation or the philosophy behind it. Another developer who joined the project during the implementation felt that the nested structure of the system increased the difficulty of becoming familiar with work that had already been done, because it was more difficult to locate the procedures needed. (The team also felt that this problem would make maintenance more difficult.) However, the managers felt that staff could be phased in and become productive more quickly than in a FORTRAN project.

Another team member felt maintenance would be easier because she had found it much easier to reacquaint herself with portions of the code that had not been changed or examined for some time. During the Ada implementation, she needed to do some maintenance on a FORTRAN program she had previously

written and found that remembering what she had done before was much more difficult than when returning to the Ada implementation.

Much more time was spent in meetings during implementation than is usual during FORTRAN projects. During most FORTRAN projects, the implementation of COMMON blocks and the interfaces between modules seem to be the usual subject of meetings. During the Ada project, many meetings were necessary to discuss the function of procedures, because the functionality was designed at the package level and not at the procedure level. Developers were unclear as to the particular details that needed to be implemented in their modules and the details that were being implemented in other modules. For example, there seemed to be misunderstandings concerning details such as which module should perform certain initializations, or which module should handle necessary conversions. Integration was another point where more meetings seemed to be necessary to clarify details.

Another possible reason for the increased number of meetings during this project is newness of Ada to the team. Subjects discussed at many of the meetings included the use of some Ada features that did not work quite as the team expected or that had been implemented incorrectly. These meetings were also general information exchange sessions at which teams members could benefit from each others' knowledge. For instance, the team discussed better methods of implementing certain types of functions, minimizing the amount of necessary recompilation, and improving the use of the library structure. Also, tools developed by team members were discussed for the benefit of the whole team. Various methods of improving performance were also discussed at these meetings. In some cases, several methods were suggested and one team member would try to determine which method would

actually produce the best performance and what the implementation impacts of the method would be.

The team was very enthusiastic about using Ada over the course of the whole project. It has become a frequent occurrence to hear someone on the Ada team say, "Why don't you do that project in Ada?" or "That problem would be so easy to solve if you were coding in Ada!" It is also interesting to note that over the 2-1/2 years of the project, not one member has left the team.

4.2.4 ASSESSMENT OF ORIGINAL ADA PROJECT ESTIMATES

The SEL collects estimates at the beginning of each project, including the amount and distribution of effort to complete the project, the size of the system, the amount of reuse, and the initial schedule. Figure 4-3 shows the estimates made very early for both GRODY and GROSS and the actual figures collected near the end of the projects.

The estimates made for the FORTRAN project, GROSS, were based on historical data collected by the SEL on other dynamics simulator projects. The estimates on size, reuse, and schedule were generally close to the actual figures, only the effort estimates were significantly different. Effort totals for this project were affected by the unusually long acceptance-test period during which many enhancements were incorporated into the system.

Since GRODY was a first-time Ada project there was no previous experience to provide a basis for estimation, so the early estimates were based on FORTRAN experience with some guesses about what effect Ada might have. Thus, the initial estimates allocated a higher percentage of the effort for design and a lower percentage for code and unit testing. System testing was also estimated at a lower percentage. These adjustments in effort required by phase were based

EFFORTS ESTIMATES:	GRODY					SEL
	EST.	ACTUAL	ACT.**	EST.	ACTUAL	EST.
TOTAL MAN MONTHS EFFORT	175	132.5*	132.5	58	97.2	97.2
% EFFORT-REQUIRE. ANAL.	15	7.4	7.4	10	5.5	6
% EFFORT-DESIGN	40	14.4	22.7	15	18.6	24
% EFFORT-CODE	25	54.1	45.8	40	35.6	45
% EFFORT-SYSTEM INTEGRATION	15	24.0	24.0	25	15.3	20
% EFFORT-OTHER	5	N.A.	N.A.	5	24.9	5

* DOES NOT INCLUDE TRAINING
** BUILD 0 INCLUDED IN DESIGN EFFORT, NOT IMPLEMENTATIO!

SIZE ESTIMATES:	GRODY		GROSS	
	EST.	ACTUAL	EST.	ACTUAL
TOTAL LINES CODE (INCL. COMMENTS)	45,000	128,000	41,000	44,600
TOTAL LINES CODE (NO COMMENTS OR BLANKS	30,000	59,130	32,000	25,600
TOTAL COMPONENTS	550	776	NO EST.	
% REUSED CODE	0	2	42	36

SCHEDULE ESTIMATES:	GRODY		GROSS	
	EST. COMPLETION DATE	ACTUAL COMPLETION DATE	EST. COMPLETION DATE	ACTUAL COMPLETION DATE
REQUIREMENTS ANALYSIS	06/29/85	09/07/85	03/15/85	02/10/85
DESIGN	02/01/86	03/15/86	06/07/85	06/08/85
CODE/UNIT TEST	07/05/86	06/27/87	12/31/85	12/28/85
SYSTEM INTEG./TEST	10/04/86	06/01/88	04/31/86	05/03/86
PROJECT COMPLETE	12/06/86	NA	09/30/86	05/31/87

5399G(25)-50

Figure 4-3. Initial Project Estimates Versus Actual Figures

on general expectations in the Ada community at the time of estimation. The actual percentages of effort by phase were closer to the usual percentages of effort for FORTRAN projects. One reason for this is that much of the design work actually was done in the implementation phase, so the recorded figures for the design phase do not accurately represent the design work done on the project. The team felt that the original estimate of 40 percent of effort for design would probably have been reasonable if the design effort had really been completed during this phase, but effort numbers including the design done in Build 0 still were lower than expected. The team was probably most surprised at the percentage of effort required to complete system testing. They had expected to use much less effort to complete system testing than on the FORTRAN project; instead, they expended more. Total effort to complete the Ada project was expected to be almost three times that expended for a FORTRAN project. This high estimate was intended to allow for the need to learn Ada during development and to compensate for the effort that the Ada team planned to devote to experimentation with methodologies and so forth. In fact, the estimate was only 20 percent higher than the effort actually required to complete the project.

At the beginning of the project, the size of the Ada system was estimated to be the same as a FORTRAN system, but as we have already seen, it turned out to be about 2-1/2 times larger. (See Section 4.3.3 for some of the reasons why the Ada system was so large.)

Schedule estimates had the largest deviations between estimates and reality. For example, code and unit testing was expected to take about 5 months and actually took 15 months. Similarly, system integration and testing was estimated to take 3 months and took 13 months. Probably the two most

important factors affecting the schedule are the learning curve involved with a new language and the experimental nature of the project. Unexpected problems arose during both implementation and testing and it took time for the team to learn how to solve these problems. The experimental nature of the project allowed more time for the developers to make enhancements or rework code to improve its general quality. In addition, the developers were not devoted full time to this project and higher priority projects made heavy demands on their time.

4.3 COMPARISON OF ADA VERSUS FORTRAN

4.3.1 GROWTH HISTORY

During the course of the two projects, information was collected weekly on the size of the source code in the development libraries. Figure 4-4 shows the weekly history of the source code growth for both projects and demonstrates that there were significant differences in the projects. The Ada project shows a smooth increase in the amount of source code from the beginning of coding, first in steps up through about week 30, which corresponds to the final release, and then as a gradual increase until coding was completed, around week 60.

The FORTRAN growth pattern is much more unpredictable, for several reasons. First, a considerable amount of reusable code was added to the source library in a very short period at the beginning of the project, resulting in a rapid early growth of the library. The Ada library did not show this early rapid growth because there was no significant amount of reusable code to be added. Between weeks 20 and 23, the FORTRAN library actually decreased by more than 15K SLOC. This appears to have been caused by the combining of two major functions that had been developed separately up to this point, and that contained some common source code that needed to be deleted.

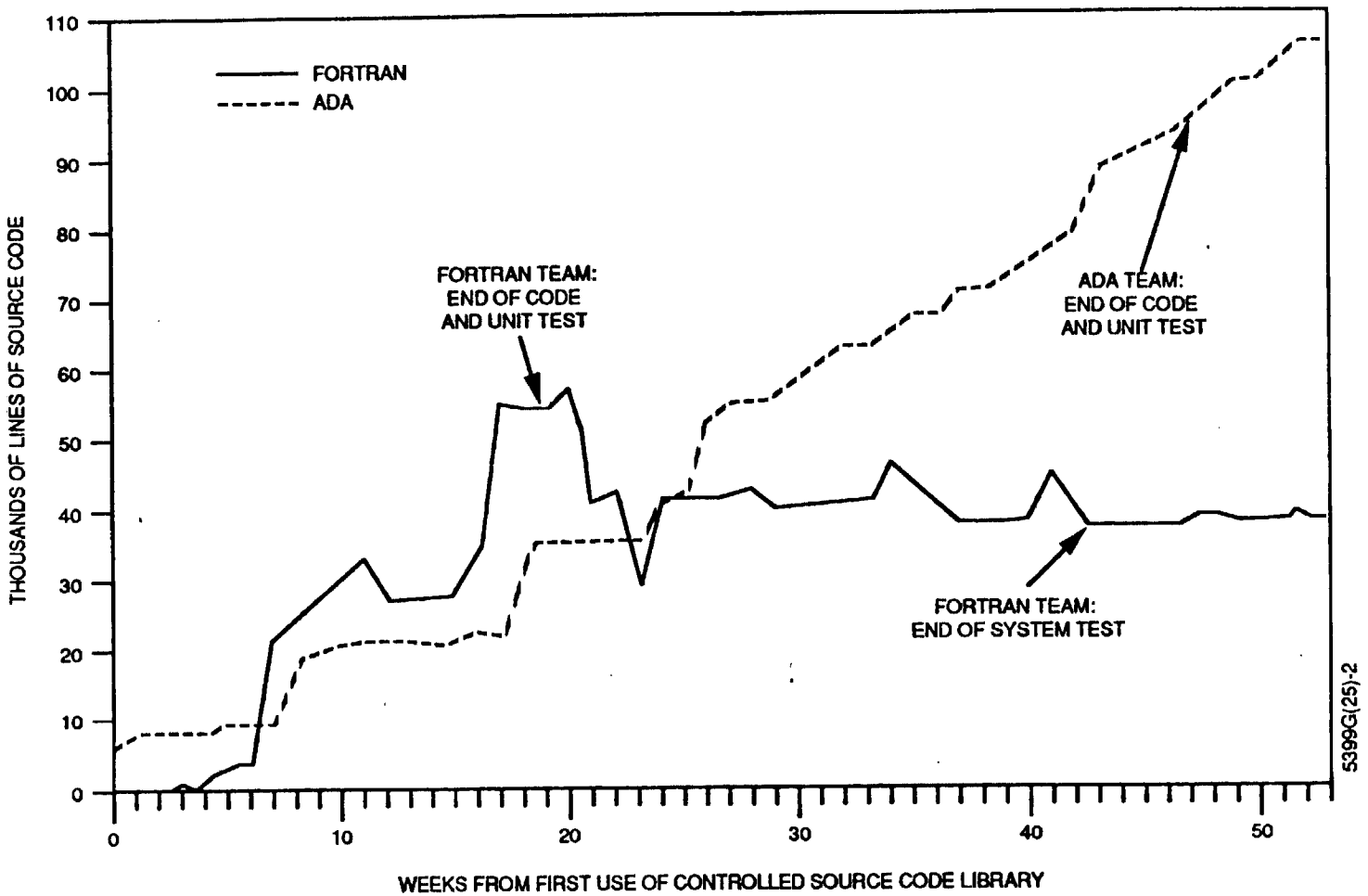


Figure 4-4. Growth in Source Code

4.3.2 CHANGE HISTORY

Another interesting metric that has been recorded weekly for both projects is the change history, or the number of changes made to each source library per week. For this purpose changes have been defined as any additions, deletions, or modifications made to a component. Figure 4-5 shows a relatively smooth curve for Ada with no erratic changes from week to week. Figures 4-6 and 4-7 show normalized growth history using two different normalization factors: number of components and SLOC. These figures emphasize the difference between the FORTRAN and Ada growth patterns. The Ada growth appears to be much smoother and much more predictable than the FORTRAN.

4.3.3 SIZE

At the beginning of this project, it was estimated that the Ada system might be about the same size as the FORTRAN system. Actually, the team was not sure what to expect for the completed size. There had been some reports of redeveloped Ada systems (from FORTRAN or COBOL) that were significantly smaller in number of lines of code (see Reference 1), but it was uncertain whether this would be true for this project. Actually, the Ada project is significantly larger than the FORTRAN project. As Table 4-1 shows, the Ada project yielded 128,000 SLOC, compared with 44,662 SLOC for the FORTRAN project. These figures are a little misleading, since the Ada line count includes 33,250 blank lines inserted for readability. The Ada line count also includes 35,620 lines of comments, compared with 19,000 lines of comments in the FORTRAN count. The Ada project had 59,130 lines of executable code compared with 25,100 for the FORTRAN project. Another way of viewing the Ada executable code is to count multiple lines that contain one Ada construct as one line (or to count by semicolons). To compare

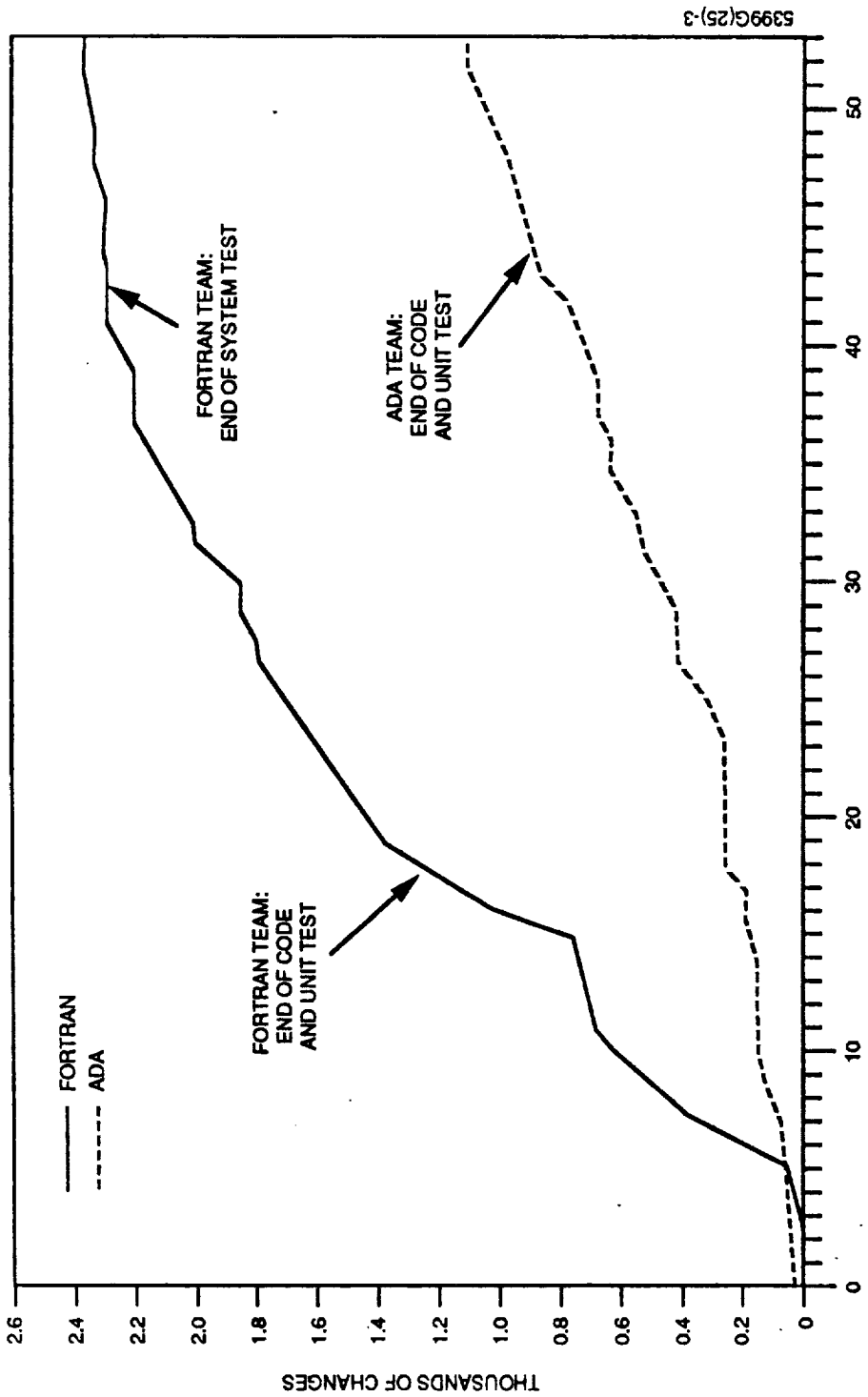


Figure 4-5. Growth in Changes to Source Code

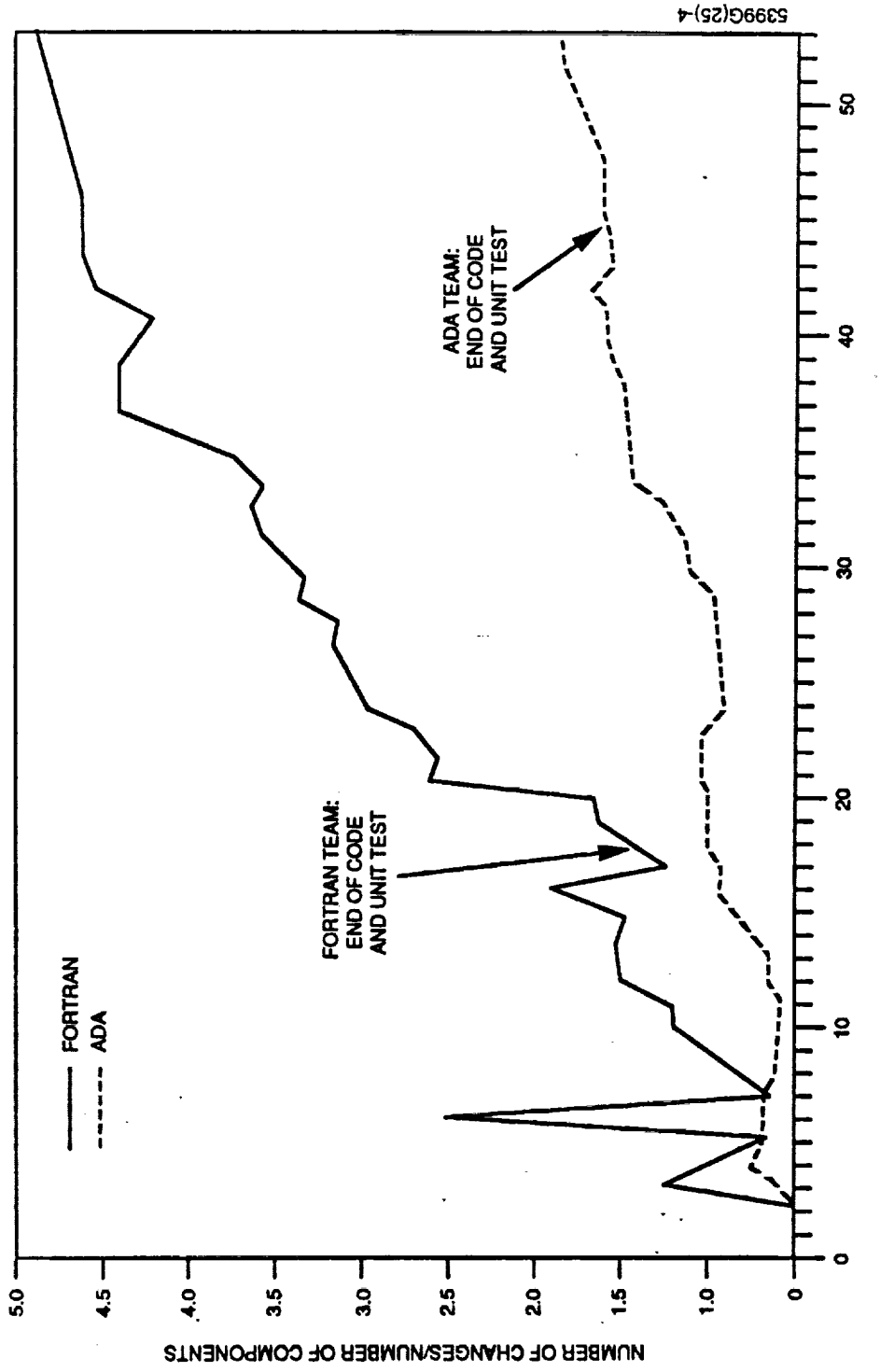


Figure 4-6. Growth in Changes Normalized by Number of Components

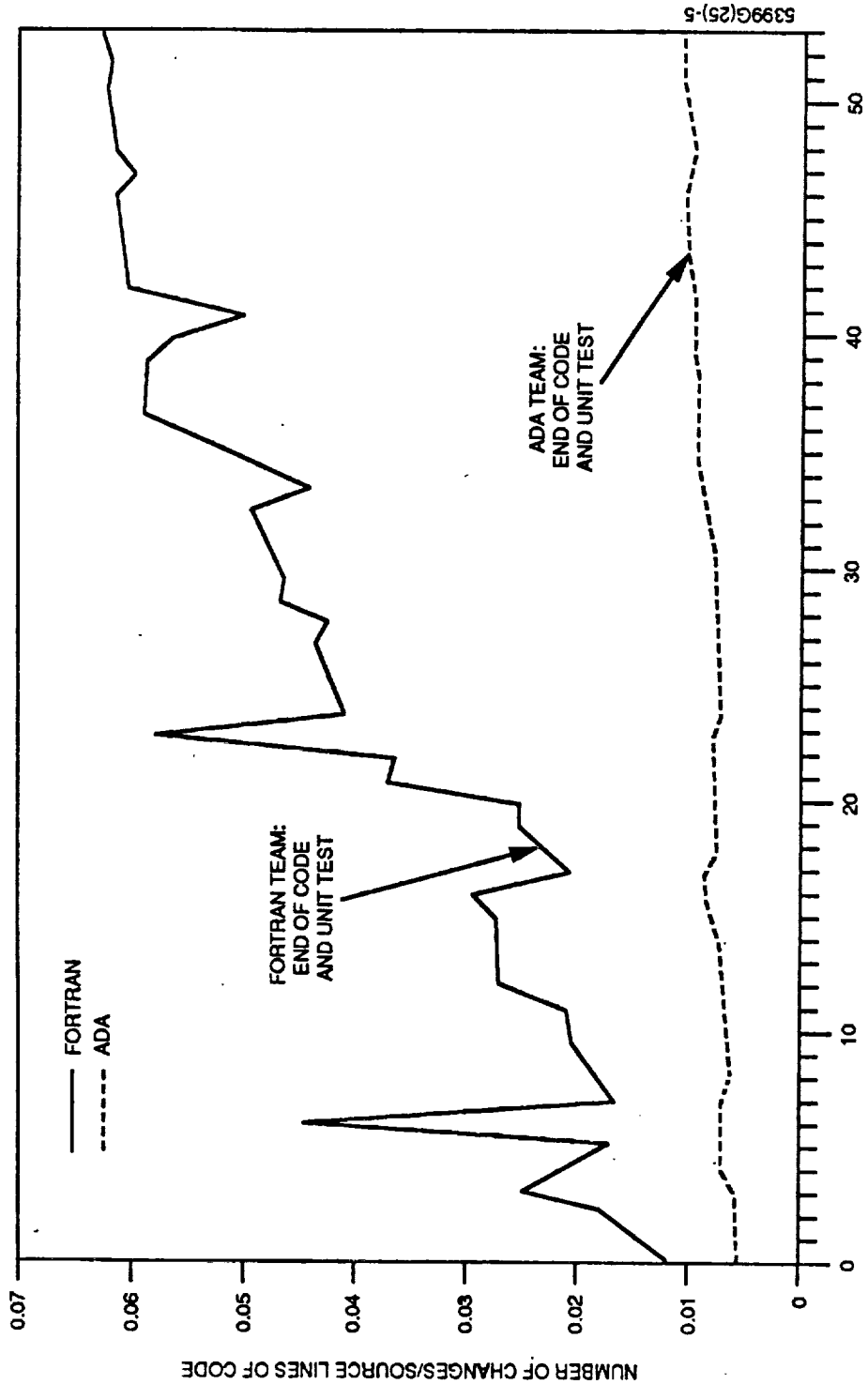


Figure 4-7. Growth in Changes Normalized by Source Lines of Code

Table 4-1. Project Size Comparisons

	FORTRAN	ADA
TOTAL LINES OF CODE	44,000 SLOC	128,000 SLOC
TOTAL COMMENTS	19,000	35,620
BLANK LINES	MINIMAL	33,250
EXECUTABLE LINE (INCLUDING TYPE STATEMENTS)	25,100	59,130
EXECUTABLE LINES (NOT INCLUDING TYPE STATEMENTS)	22,500	42,150
REUSED LINES	16,000 (36%)	2,500 (2%)
EXECUTABLE STATEMENTS	22,300	22,840
AVERAGE COMPONENT SIZE	135	208
LOAD MODULE	953 512-BYTE BLOCKS	2300 512-BYTE BLOCKS

5399G(25)-9

C-2

this figure with the FORTRAN, we would also need to count FORTRAN statements continued over more than one line as just one line. This yields a count of 22,840 semicolons (statements and declarations) for the Ada project and 22,300 statements for the FORTRAN.

The team felt there were several reasons why the Ada project was larger than the FORTRAN project. First, since they were not constrained by schedule pressure as the FORTRAN team was, they took the time to develop a system with increased functionality--more of the "nice to have, but not required" features. This resulted in a much more sophisticated user interface of approximately 40K SLOC, nearly four times the size of the FORTRAN interface. Second, the Ada language itself requires more code to write such constructs as package specifications, declarations, and so forth. Third, the Ada Style Guide required that certain constructs be spread over several lines for readability. Blank lines of code were inserted for formatting, and more extensive prologs were entered for the Ada code.

Since Ada is still a relatively new language, many of the supporting packages and utilities available in FORTRAN are not available in Ada. Utilities considered standard system routines in FORTRAN were not available for Ada and had to be written by the team. These included such procedures and functions as trigonometric functions, matrix multiply routines, minimum or maximum values, and array definition and manipulation.

A final factor that contributed to the larger size of this particular Ada system is the use of call-throughs, which were discussed earlier in the document (see Section 2.3.1.1). These call-throughs were implemented as a method of strictly representing the object-oriented design directly in the code so that every object in the design had a corresponding

object in the code. The team decided that this type of one-to-one correspondence was really not necessary and, in fact, caused additional code that in turn necessitated more compilation and increased testing complexity. The team felt that when making an estimate for future Ada systems compared with FORTRAN systems, they would expect the Ada system to be about twice as large as the comparable FORTRAN system.

Another interesting way to compare the size of the two projects is to examine the size of the load modules for each. This also shows the Ada system to be larger--occupying 2,300 512-byte blocks, compared to 953 512-byte blocks for the FORTRAN load module.

4.3.4 EFFORT

Table 4-2 shows the staff-hours of effort by phase on both projects. The figures in this table were collected by adding the total number of hours recorded during each phase using the phase dates listed in Table 4-3. Thus, the number of hours spent on design looks low since some of the activity during the implementation phase was actually design. Similarly, there is some overlap in activities during the end of unit testing and the beginning of system testing. Probably a better idea of the effort distribution can be obtained by looking at Tables 4-4 and 4-5, which show the staff hours spent on each phase activity, regardless of the phase in which the hours actually occurred. Hours spent on design are recorded as design hours whether they occurred during the design phase, implementation phase, or testing phase. The difference between Tables 4-4 and 4-5 is in the hours recorded as "other," which include activities such as documentation, attending meetings, and management tasks. Table 4-5 includes these "other" hours as part of the phase activity. The team had initially expected that the effort required during the Ada design phase would be much greater

Table 4-2. Project Effort Comparisons by Phase Dates

PHASE	STAFF-HOURS†		DURATION IN MONTHS	
	FORTRAN	ADA	FORTRAN	ADA
TRAINING	0	2345	0	6.0
REQUIREMENTS ANALYSIS	849	1541**	1.5	2.2
DESIGN	2830	2987	4.0	6.0
CODE/TEST	5397	11174	6.5	15.0*
SYSTEM TEST	2315	4968	4.0	11.0
ACCEPTANCE TEST	3775	NA	13.0	NA
TOTAL	15166	23015	29.0	40.2

5399G(25)-10

* OFFICIAL END OF CODE/UNIT TEST USED, BUT ACTUAL UNIT TESTING OVERLAPPED SYSTEM TESTING AND CONTINUTED FOR ANOTHER 4 MONTHS.

† DOES NOT INCLUDE SUPPORT HOURS.

** APPROXIMATELY 1000 HOURS OF THIS REQUIREMENTS ANALYSIS EFFORT OCCURRED DURING THE TRAINING PERIOD.

Table 4-3. Phase Dates

	FORTRAN		ADA	
	START	END	START	END
TRAINING	NA	NA	01/01/85	06/29/85
REQUIREMENTS ANALYSIS	01/01/85	02/10/85	06/29/85	09/07/85
DESIGN	02/10/85	06/08/85	09/07/85	03/15/86
CODE/UNIT TEST	06/08/85	12/28/85	03/15/86	06/27/87 *
SYSTEM TEST	12/28/85	05/03/86	06/27/87	06/01/88
ACCEPTANCE TEST	05/03/86	05/31/87	NA	NA

5399G(25)-11

* OFFICIAL END OF CODE/UNIT TEST PHASE, BUT SOME UNITS WERE NOT UNIT TESTED UNTIL 10/31/87.

END OF BUILD 0 WAS 10/12/86

Table 4-4. Project Effort Comparisons by Activity, Excluding Hours Recorded as "Other"

PHASE	STAFF-HOURS		PERCENT OF TOTAL ACTIVITY		PERCENT ACTIVITY REQUIREMENTS ANALYSIS → SYSTEM TEST	
	FORTRAN	ADA	FORTRAN	ADA	FORTRAN	ADA
TRAINING	0	2436	0	13.5	NA	NA
REQUIREMENTS ANALYSIS	1320	498	11.1	2.7	14.1	3.2
DESIGN	2223	5678	18.6	31.6	23.8	36.5
CODE/TEST	4252	6645	35.7	37.0	45.4	42.8
SYSTEM TEST	1562	2724	13.1	15.2	16.7	17.5
ACCEPTANCE TEST	2557	NA	21.5	NA	NA	NA
TOTAL	11914*	17981*	100.0	100.0		
TOTAL (REQUIREMENTS ANALYSIS → SYSTEM TEST	9357	15545			100.0	100.0

5399G(25)-12

* THESE TOTALS ONLY REFLECT THE HOURS RECORDED IN THE SEL DATA BASE WHERE TRAINING HOURS FOR GRODY WERE DEFINED AS THOSE HOURS RECORDED AS "OTHER" DURING THE PRE-PROJECT PHASE. THE REQUIREMENTS ANALYSIS HOURS WERE THOSE RECORDED AS PREDESIGN; THE DESIGN HOURS WERE THOSE RECORDED AS CREATE DESIGN OR READ AND REVIEW DESIGN; THE CODE AND UNIT TEST HOURS WERE THOSE RECORDED AS WRITE CODE, READ AND REVIEW CODE, TEST CODE, AND DEBUG CODE; THE SYSTEM TEST HOURS WERE THOSE RECORDED AS INTEGRATION TEST HOURS; AND THE ACCEPTANCE TEST HOURS FOR GROSS WERE THOSE HOURS RECORDED AS "OTHER" DURING THE ACCEPTANCE TEST PHASE. HOURS RECORDED AS "OTHER," SUCH AS MANAGEMENT, DOCUMENTATION, ATTENDING MEETINGS, ETC., WERE NOT INCLUDED.

Table 4-5. Project Effort Comparisons by Activity, Including Hours Recorded as "Other"

PHASE	STAFF-HOURS		PERCENT OF TOTAL ACTIVITY		PERCENT ACTIVITY REQUIREMENTS ANALYSIS → SYSTEM TEST	
	FORTRAN	ADA	FORTRAN	ADA	FORTRAN	ADA
TRAINING	0	2436	0	10.6	NA	NA
REQUIREMENTS ANALYSIS	1841	680	12.1	3.0	14.6	3.3
DESIGN	3361	6505	22.2	28.3	26.7	31.7
CODE/TEST	5443	9671	35.9	42.0	43.2	47.0
SYSTEM TEST	1962	3704	12.9	16.1	15.5	18.0
ACCEPTANCE TEST	2557	NA	16.9	NA	NA	NA
TOTAL	15164	22996	100.0	100.0		
TOTAL (REQUIREMENTS ANALYSIS THROUGH SYSTEM TEST)	12607	20560			100.0	100.0

5399G(25)-13

than that required in the FORTRAN project, and that the implementation and testing phases would then be much shorter. The table shows that the design effort of the Ada team was about the same as that of the FORTRAN team, and that the implementation effort was considerably greater than expected--closer to the effort of the FORTRAN team.

Several factors could be responsible for this difference. First, as mentioned in the lessons-learned design document (Reference 2), the CDR marking the formal end of the design phase occurred at a somewhat arbitrary point in the design, and many of the team members felt that the actual design work was not complete. As noted in Section 2.2.2, several entire functions were either designed or redesigned after the CDR. PDL was entered into the machine during a 3-month period in early implementation; normally, in a FORTRAN development, this would be accomplished during the design phase. Obviously, there was not a clear dividing line between the design phase and the implementation phase for the Ada project. This raises the question of the appropriate points for development milestones, such as the CDR, in an Ada development. A better point for the CDR may be the completion of package specifications and type declarations, allowing the use of the Ada compiler to check the design consistency.

Another factor in the effort difference is the size and functionality of the Ada system. It would be expected that it would take longer to write more lines of code and to produce modules that have more capability. In addition, the FORTRAN team reused about 36 percent of their code from previous FORTRAN systems, while the Ada team had no reusable Ada code, and the reused FORTRAN routines made up only about 2 percent of their code.

4.3.5 PRODUCTIVITY/COST

There are many ways to compute productivity. In the SEL, the calculation is usually made by dividing the total number of SLOC developed by the number of hours spent on the project. The number of hours is carefully recorded on weekly forms and includes the hours spent on all phases of the project, beginning with the requirements analysis and ending with the completion of acceptance testing. In order to compare the FORTRAN and Ada projects, the calculations were made using the number of hours spent on each project from requirements analysis to the completion of system testing, since acceptance testing has not yet been completed for the Ada project. As shown in Table 4-6, using the total number of SLOC for each project, productivity was 3.9 SLOC/hour for the FORTRAN project and 6.2 SLOC/hour for the Ada project. Because the Ada code included many blank lines of code that were not included in the FORTRAN line count, the Ada was recomputed excluding the blank lines, resulting in a productivity of 4.6 SLOC/hour. When the calculation considers the effort required to develop only the new lines of code and not the reusable code, the figures are 2.5 SLOC/hour for the FORTRAN, and 6.1 SLOC/hour for the Ada with blanks and 4.5 SLOC/hour without blanks. This would seem to imply that the Ada is more productive, but it took many more lines of code to develop the Ada system and the style guide caused many constructs to be spread over many lines.

Using only the number of executable lines of code, the productivity figure was 1.97 SLOC/hour for the FORTRAN project and 2.0 SLOC/hour for the Ada project. Because many of the Ada constructs use more than one line, productivity was recomputed for the number of executable statements (or semicolons) in the Ada project. Similarly, for the FORTRAN project, statements and their continuations were counted as one executable statement. This resulted in a productivity

Table 4-6. Productivity Comparisons

	FORTRAN	ADA
LINES OF CODE USED FOR COMPUTATION	PRODUCTIVITY SLOC/HR	PRODUCTIVITY SLOC/HR
TOTAL LINES OF CODE	3.9	6.2
TOTAL LINES OF CODE EXCLUDING BLANKS	3.9	4.6
NEW LINES OF CODE	2.5	6.1
NEW LINES OF CODE EXCLUDING BLANKS	2.5	4.5
EXECUTABLE LINES OF CODE	1.97	2.0
EXECUTABLE STATEMENTS	1.95	1.10
EXECUTABLE NEW STATEMENTS	1.25	1.08

5399G(25)-14

of 1.95 SLOC/hour for the FORTRAN project and 1.10 SLOC/hour for the Ada project. Looking at the number of executable new statements in the FORTRAN project yields a figure of 1.25 SLOC/hour compared to 1.08 SLOC/hour for the Ada project. These calculations would make the FORTRAN look more productive.

Perhaps a better way of viewing the productivity problem is to examine it from the standpoint of cost to produce the product. The total cost of the FORTRAN project from requirements analysis through acceptance testing was about 8.1 man-years of effort. The Ada project cost, using actual figures from requirements analysis through system testing and estimating the acceptance testing cost, is around 12 man-years of effort. Taking into consideration the percentage of reused code in the FORTRAN project and assuming that all the code generated was new code, it would have taken about 11.5 man-years of effort to develop the FORTRAN system. This makes the cost of developing the two systems roughly the same, especially when we consider that the Ada project was a "first-time" project and that it had slightly more functionality than the FORTRAN project.

4.3.6 SCHEDULE

Several team members commented that because the Ada project was "just an experiment and not the real, operational version," there was no sense of urgency to finish the project in any particular time frame and the deadlines kept slipping. It is difficult to determine how much of this schedule problem is really Ada-related and how much is caused by the lack of firm deadlines. The one firm deadline that was established--the June 30 deadline to finish coding--was met. Another team member commented that many of the team members were dividing their time among other tasks and it appeared that management tended to assign higher priority to other tasks with more urgent deadlines.

The learning curve was another factor that seems to have affected the Ada schedule. Even though the team had extensive training before the project started, in many areas a lack of experience with Ada caused delays or problems in implementation and integration testing. As previously discussed, some of these problems included misunderstandings about how to use Ada features, immaturity of tools, a lack of intuition when debugging Ada code, and a general lack of experience with tasking.

4.3.7 ERROR/CHANGE STATISTICS

Table 4-7 shows statistics on the number of errors and changes that occurred during the implementation and testing phases of GROSS and GRODY. (For the tables in this section, a date of October 31, 1987, was used for end of GRODY implementation, because unit testing for a significant portion of the system was done during this period.) Several interesting observations can be made from these statistics. First, it is significant to note that both the error rate and the change rate are almost identical for the two projects. The change rate for GROSS, computed by using the total number of changes divided by the source lines of code, is 58 changes/10,000 SLOC, while the change rate of GRODY (excluding blanks from the number of source ilnes of code used in the computation) is 56 changes/10,000 SLOC. Similarly, the error rate for GROSS is 23.3 errors/10,000 SLOC, while GRODY has an error rate of 24 errors/10,000 SLOC. Second, the percentage of total errors compared to the number of total changes is roughly the same for the two projects. Finally, a much higher percentage of both changes and errors occurred in the implementation phase of GRODY (that is, earlier in the process) than in GROSS. A higher percentage of the error corrections and changes occurred during the testing phase in GROSS. This seems to support the expectation that errors in Ada will be discovered earlier.

Table 4-7. Comparison of Errors and Changes in FORTRAN and Ada During Implementation and Testing

	FORTRAN		ADA	
	NUMBER OF CHANGES	PERCENT	NUMBER OF CHANGES	PERCENT
IMPLEMENTATION	148	56%	426	79%
TESTING	113	44%	112	21%
TOTAL	261	100%	538	100%
	NUMBER OF ERRORS	PERCENT	NUMBER OF ERRORS	PERCENT
IMPLEMENTATION	57	55%	165	72%
TESTING	47	45%	63	28%
TOTAL	104	100%	228	100%

5399G(25)-15

Figures 4-8, 4-9, and 4-10 show the types of changes recorded for both the FORTRAN and the Ada projects during the implementation phase and the testing phase, and the totals for both phases, respectively. It is interesting to note that roughly the same percentage of changes in each project, both overall and during implementation, were due to error corrections. Percentages of some of the other types of errors differ considerably between the two projects, but the differences may be caused by individual project differences and not the influence of the implementation language. For example, a much higher percentage of the changes in the Ada project were implemented to improve the user services or the clarity of the code, or to enhance the future ease of maintenance. A very low percentage of the FORTRAN changes belonged to these types. One obvious reason for this is that the Ada project had more time to enhance and refine the user interface. Some of the changes made to improve clarity or ease future maintenance took advantage of the Ada team's increasing knowledge as the project progressed. A much higher percentage of the changes in the FORTRAN project were made to implement requirements changes. This is an expected result, considering that the FORTRAN project was developed slightly earlier when requirements were less stable.

Similarly, we notice that more effort was spent on planned enhancements in the FORTRAN project, especially during the implementation phase. Several factors may have influenced this. First, the projects were on different schedules, with the FORTRAN development occurring first. Hence, the requirements team's view of the desired FORTRAN system may not have been fully developed at the beginning of implementation, but evolved as the system developed. Another possibility is that the more extensive design efforts during the Ada project may have forced an earlier view of the desired system so

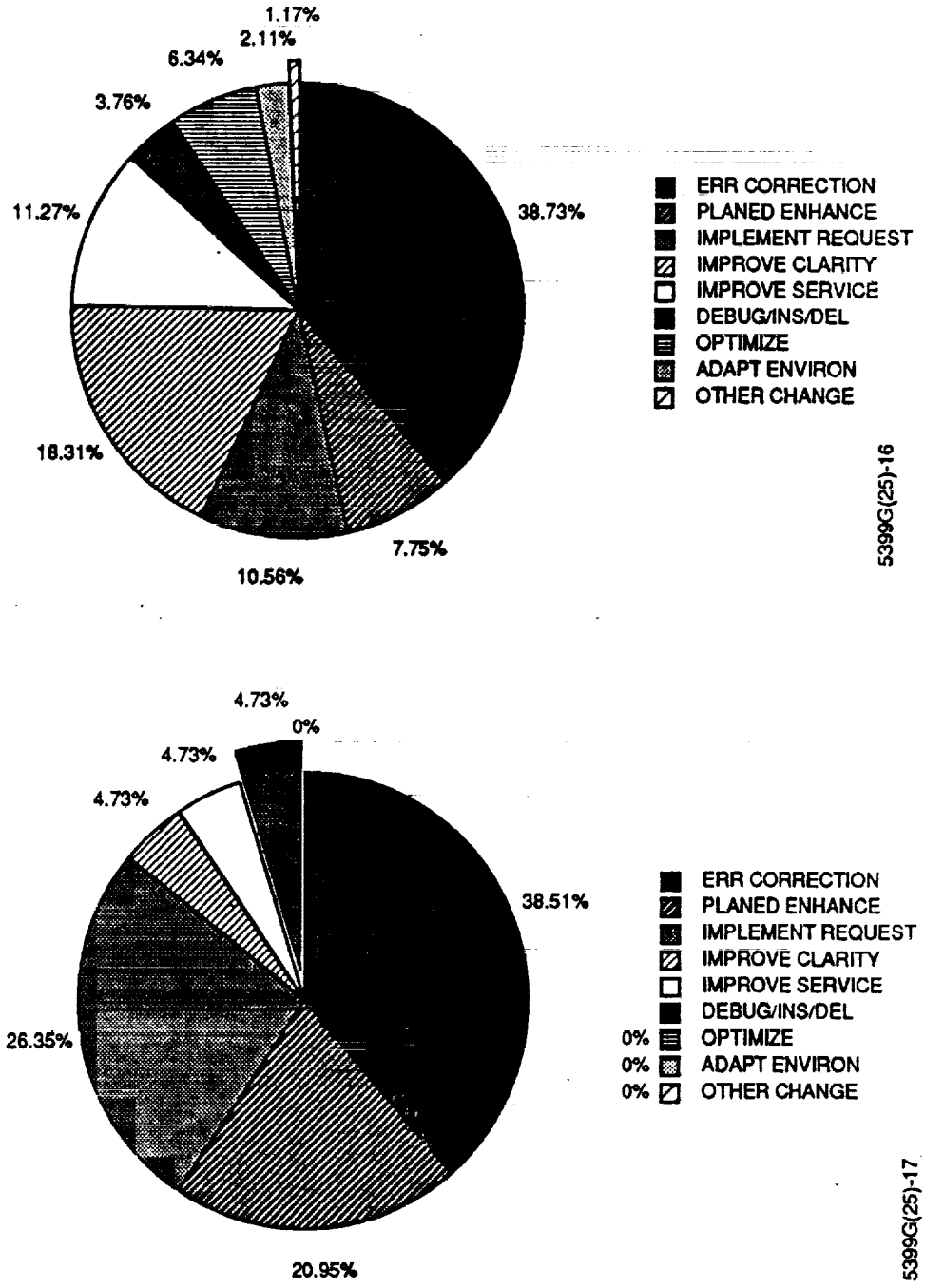


Figure 4-8. GRODY/GROSS Change Type--Implementation

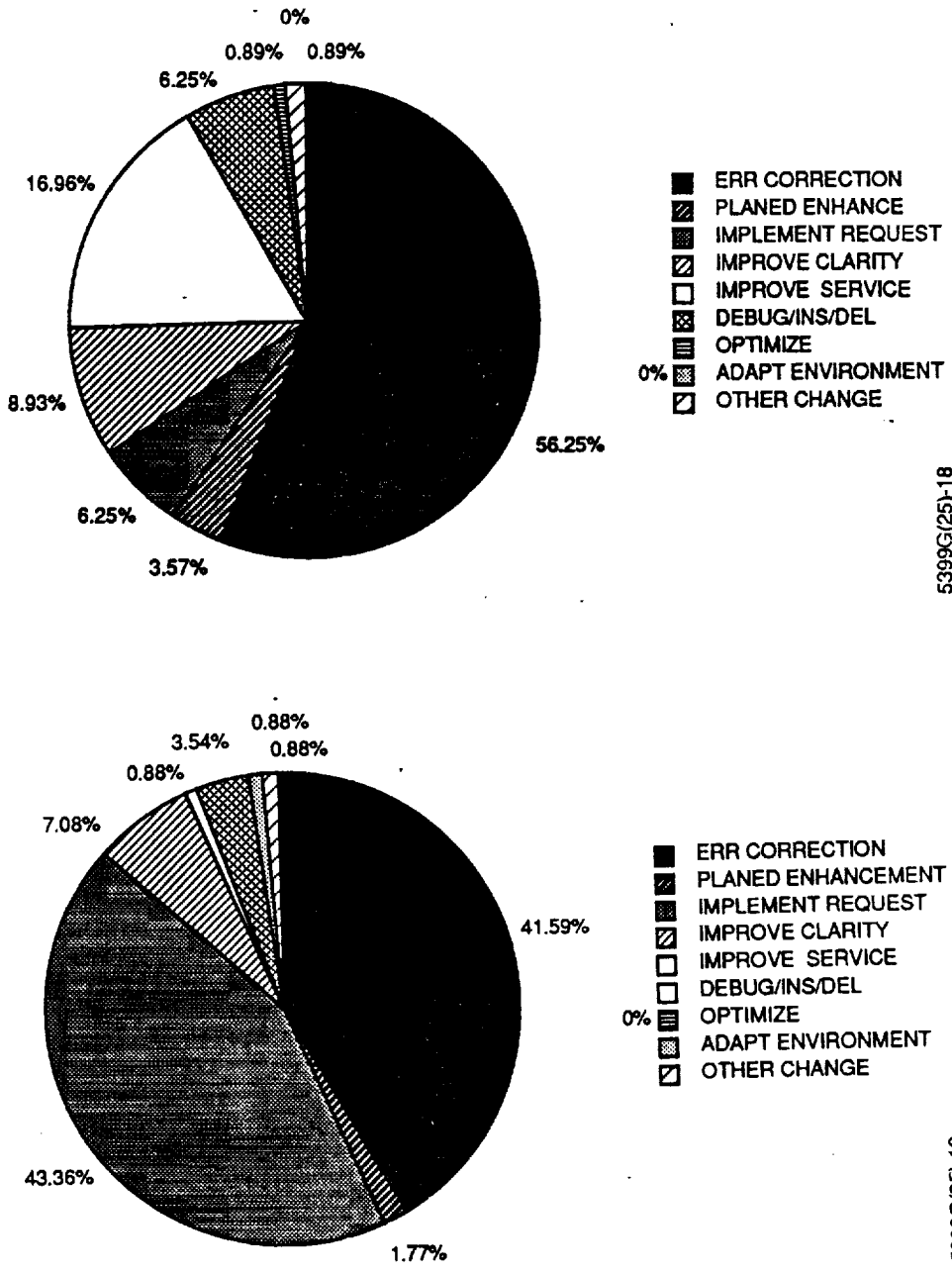
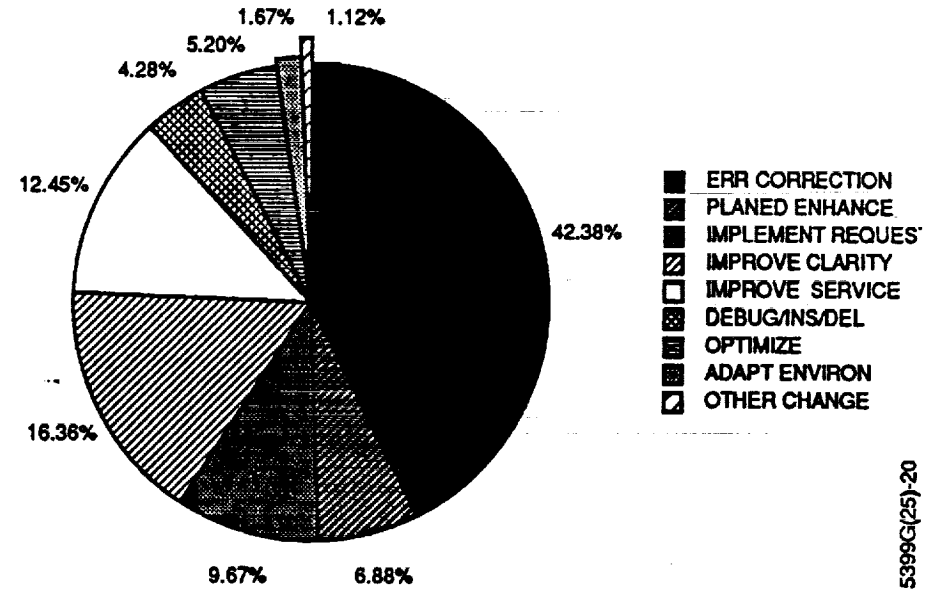


Figure 4-9. GRODY/GROSS Change Type--Test



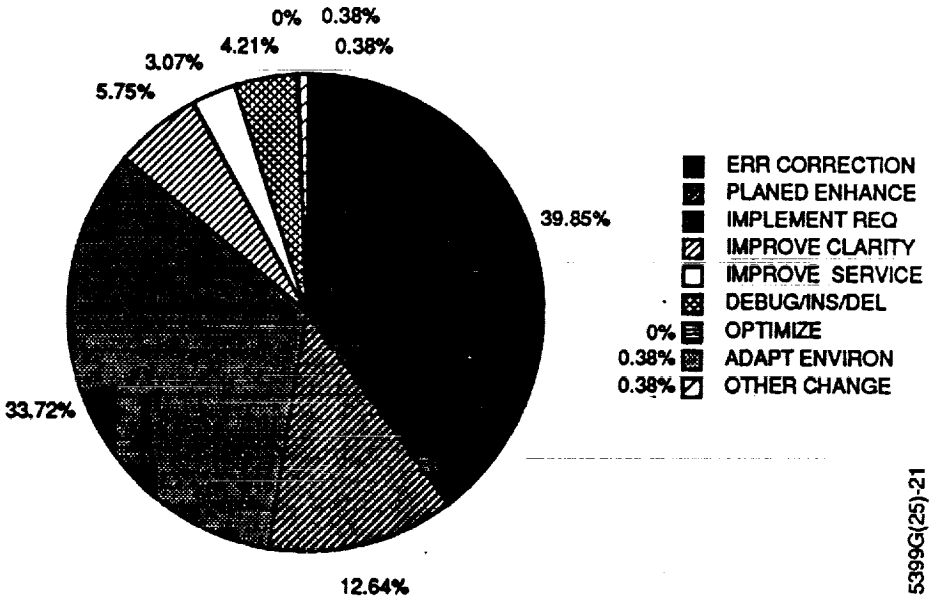


Figure 4-10. GROSS/GRODY Change Type--Total Project

that features that were added to the FORTRAN system as enhancements may have been designed into the Ada system. Finally, we can see that more of the changes in the Ada system were made for the purposes of optimization. This can probably be attributed to the newness of Ada and the lack of experience with it. The Ada developers were not yet familiar with the optimal methods for implementing certain features in Ada and found that they needed to make changes to improve performance.

Figures 4-11, 4-12, and 4-13 show the sources of errors found in both the FORTRAN and the Ada projects during the implementation phase and the testing phase, and the totals, respectively. Coding errors make up the highest percentage of errors in both the FORTRAN and the Ada systems. A high percentage of errors is attributed to the design of the Ada system. This is not suprising, since the Ada design was totally new while the FORTRAN design was a proven, reused design developed and refined over many similar projects. A significant number of errors in the Ada system were caused by previous changes. These errors can probably be attributed to the team's inexperience with Ada.

Figures 4-14, 4-15, and 4-16 show the classes of errors recorded for the two projects during the implementation phase and the testing phase, and the combined totals, respectively. Note that the percentages of the different classes of errors found in both the Ada and the FORTRAN systems were very similar. This would imply that the distribution of errors among the classes of errors is fairly language independent. The majority of errors in all classes for the Ada project were discovered at an earlier phase than those for the FORTRAN project. This is especially noticeable for initialization and external interface errors, which are virtually nonexistent in the Ada testing phase. One result in the error statistics that was surprising to the Ada team is

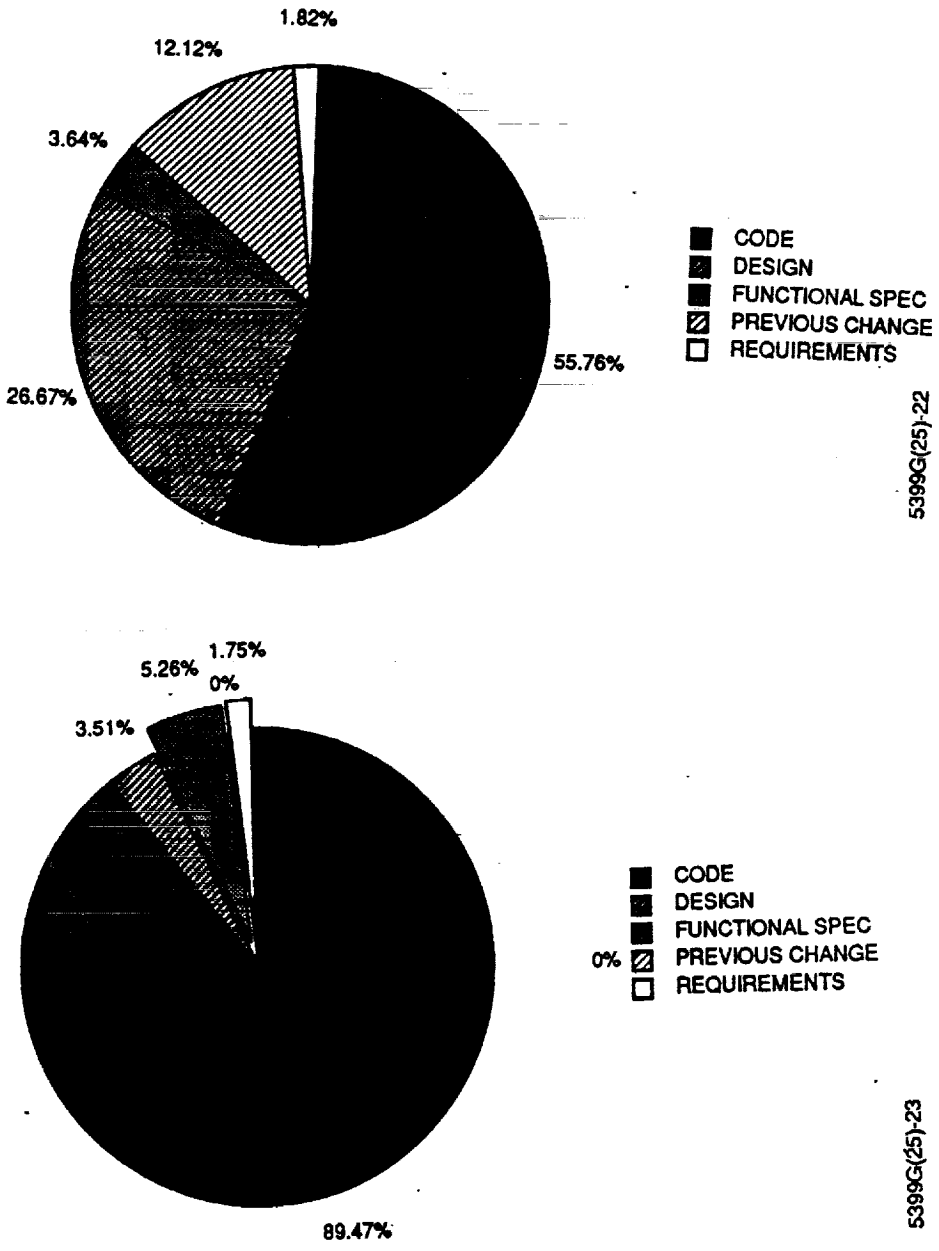
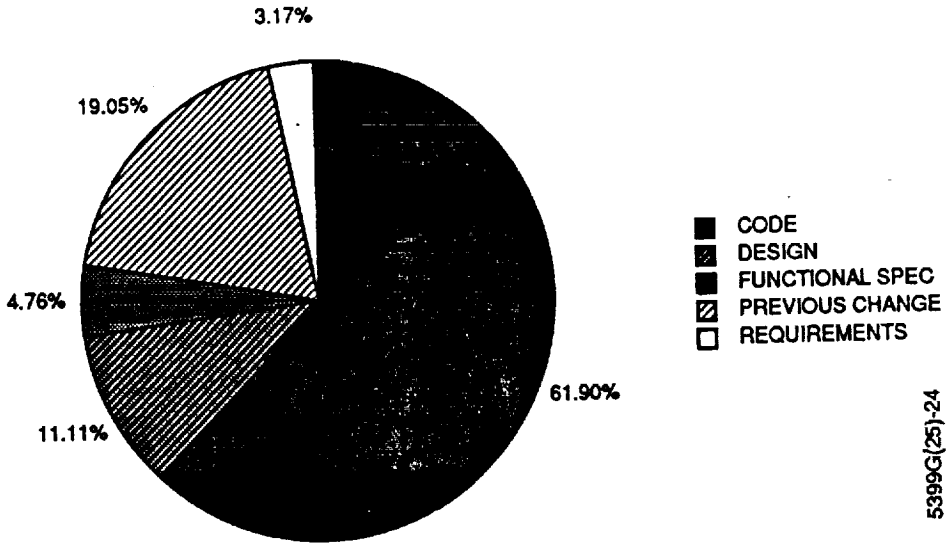


Figure 4-11. GRODY/GROSS Error Source--Implementation



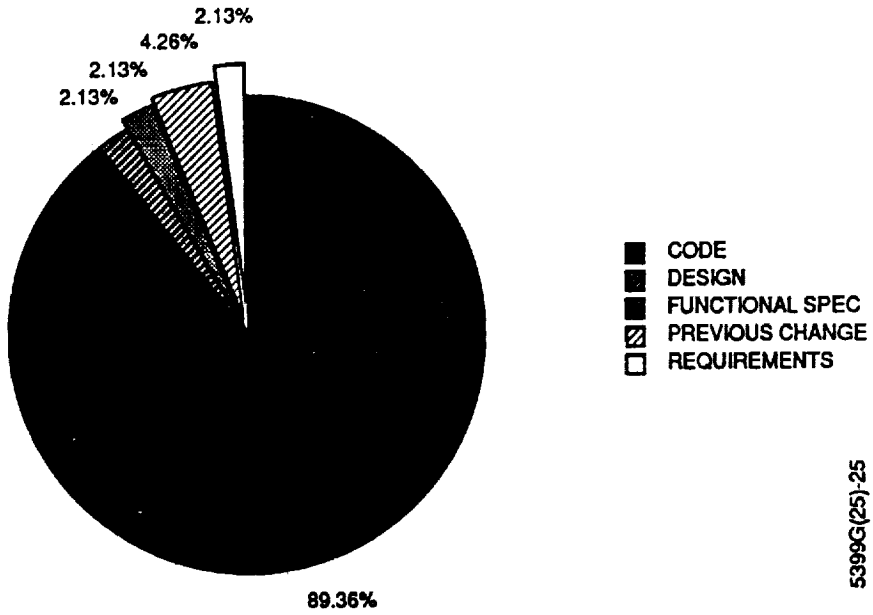
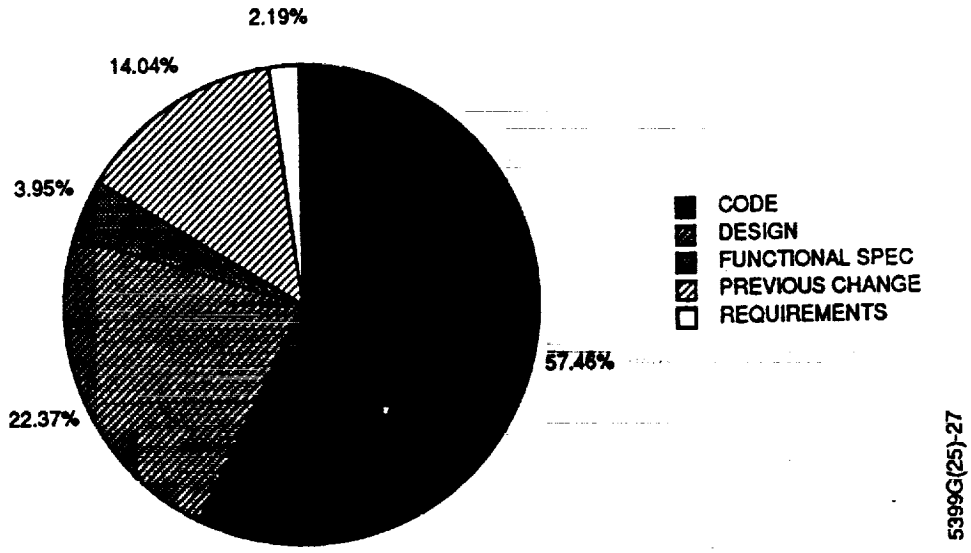


Figure 4-12. GRODY/GROSS Error Source--Test



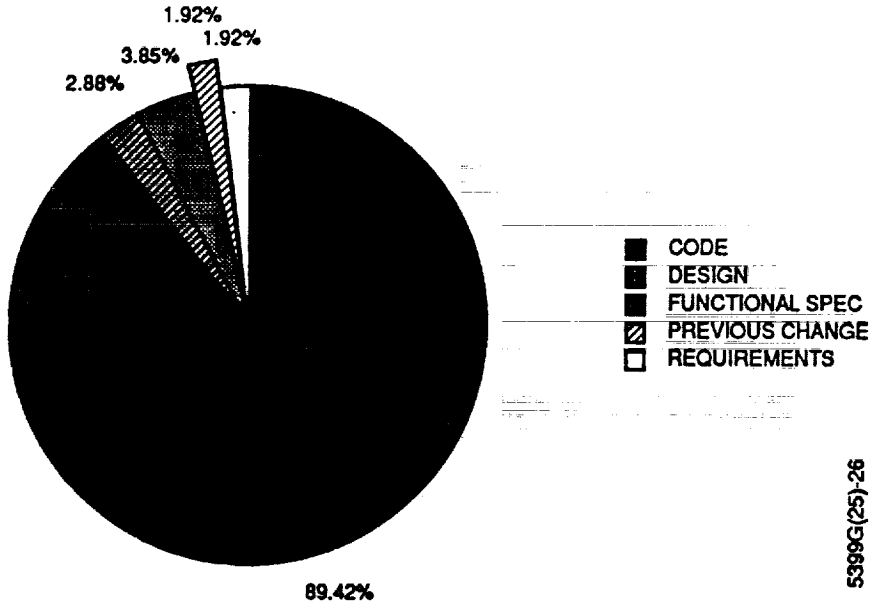


Figure 4-13. GRODY/GROSS Error Source--Total Project

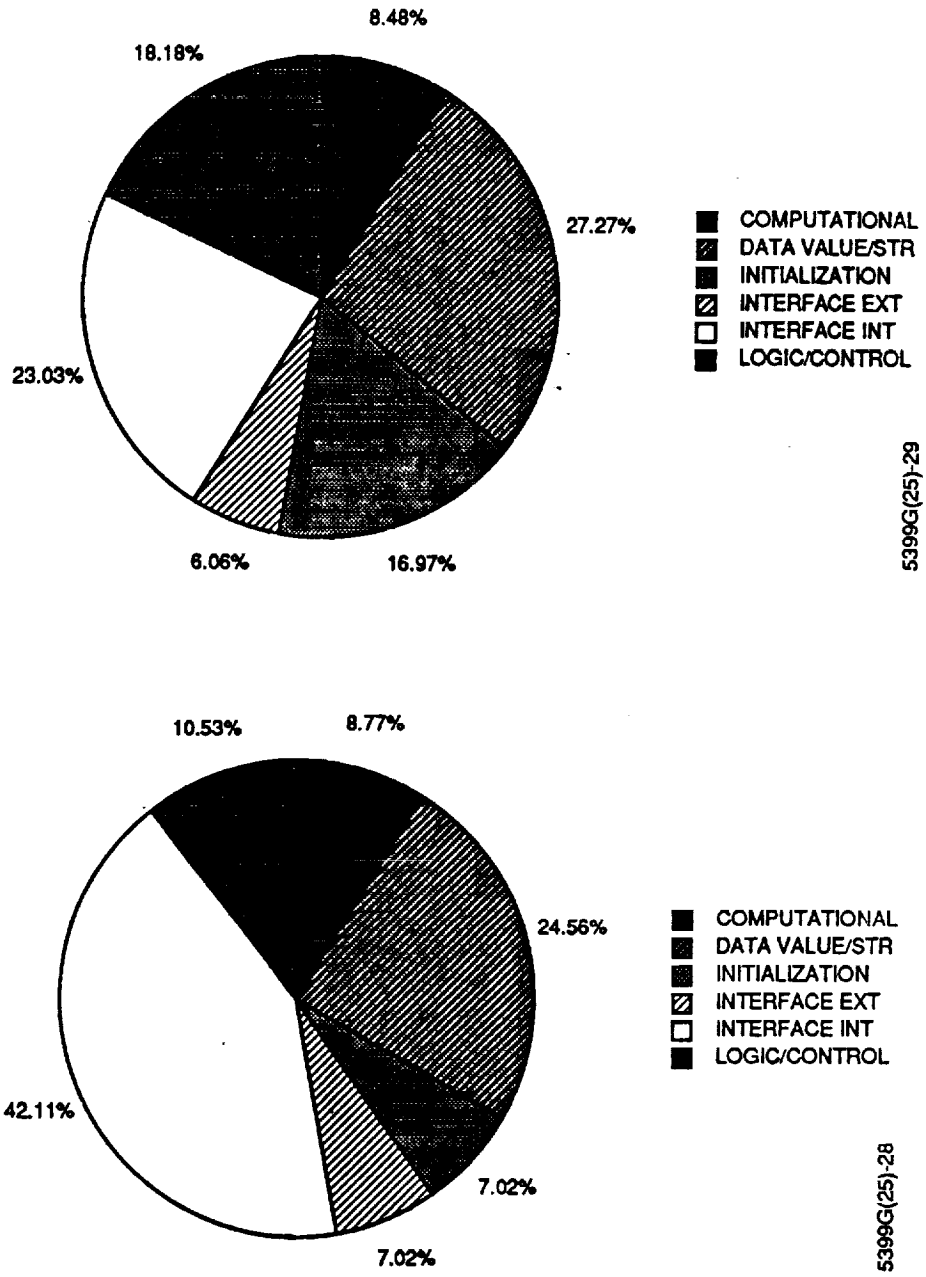


Figure 4-14. GRODY/GROSS Error Class--Implementation

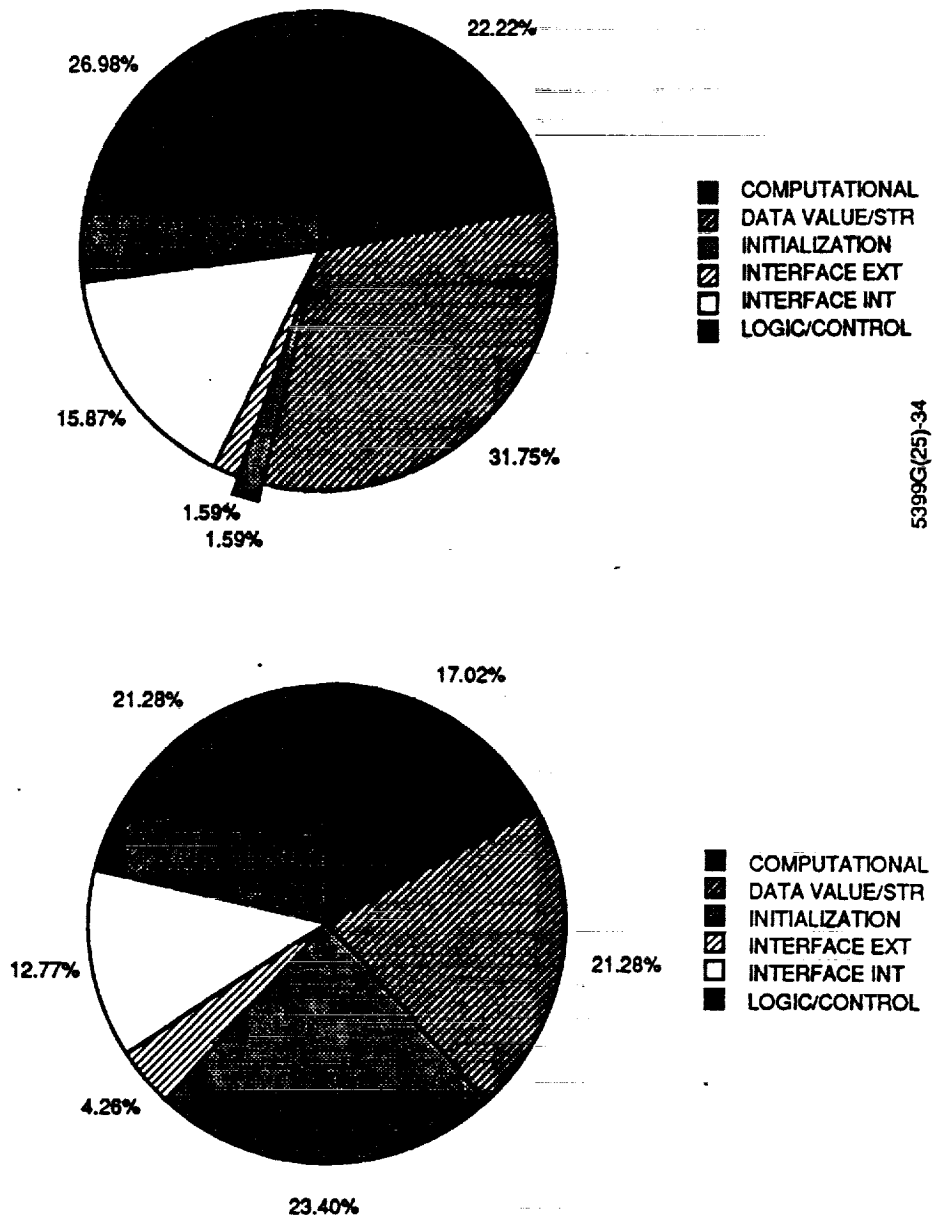
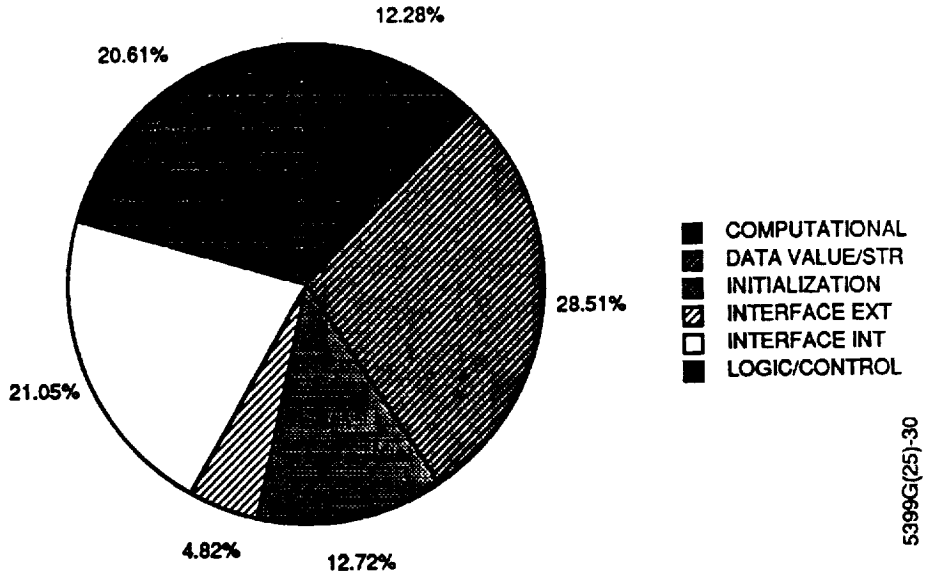


Figure 4-15. GRODY/GROSS Error Class--Test



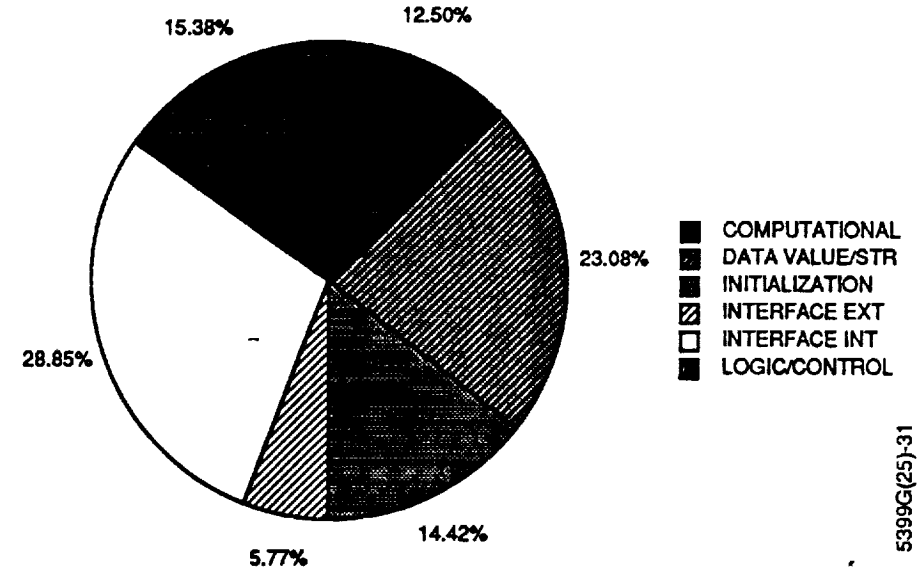


Figure 4-16. GRODY/GROSS Error Class--Total Project

the number of interface errors recorded. One possible reason for this is that in Ada the interfaces are defined when the specifications are developed, which occurs very early in the implementation. In GRODY, these specifications were compiled separately and then put under configuration control--even before the body of the unit was coded, in some cases. This meant that any subsequent changes had to be recorded. In the case of the FORTRAN development, where there are no specifications, nothing would be placed in the configuration library until the entire routine was unit tested, so no interface problems found or changes made up to this point were recorded. There were also more initialization errors in the Ada system than the team expected. Many of these initialization errors can probably be attributed to the teams' confusion over the exact function of procedure units and misunderstandings concerning the responsibility for initialization.

Figures 4-17, 4-18, and 4-19 show the length of time it took to isolate errors in both the FORTRAN and the Ada projects, and Figures 4-20, 4-21, and 4-22 show the effort required to complete the changes to correct those errors. It took longer to isolate errors in the Ada system, especially during implementation. Ninety-three percent of the FORTRAN errors were isolated in less than an hour, compared to 59 percent of the Ada errors. Several possible reasons have been suggested for this. First, the FORTRAN developers were very experienced in that language and had a great deal of intuition to aid them in locating errors. Another possibility is that the Ada compiler may have already found most of the "very easy" errors in the Ada code, leaving only the more difficult ones for the developers to correct. Overall, the vast majority of errors in both languages (94 to 96 percent) were discovered quickly with less than 1 day's effort. The effort to correct the errors was slightly less

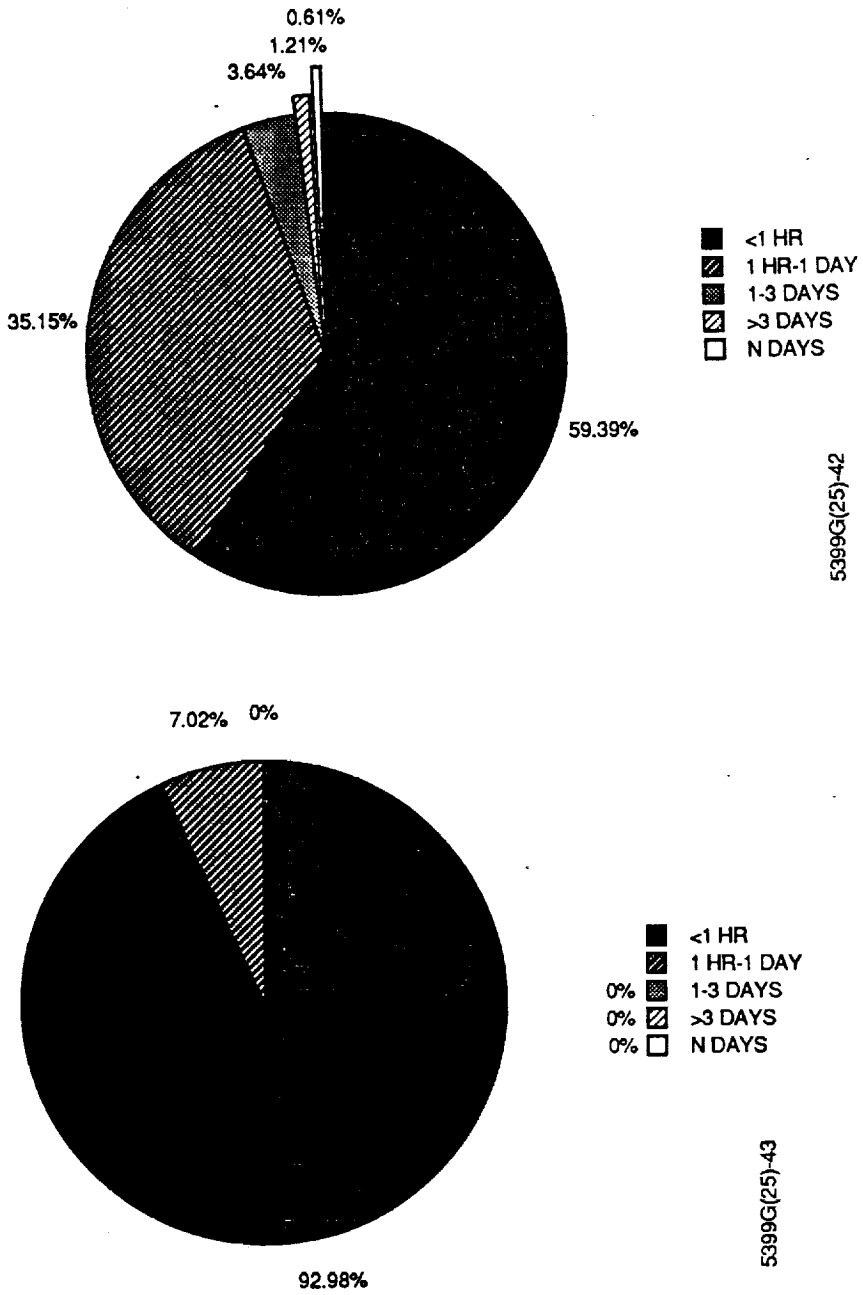


Figure 4-17. GRODY/GROSS Effort To Isolate (Errors Only)--Implementation

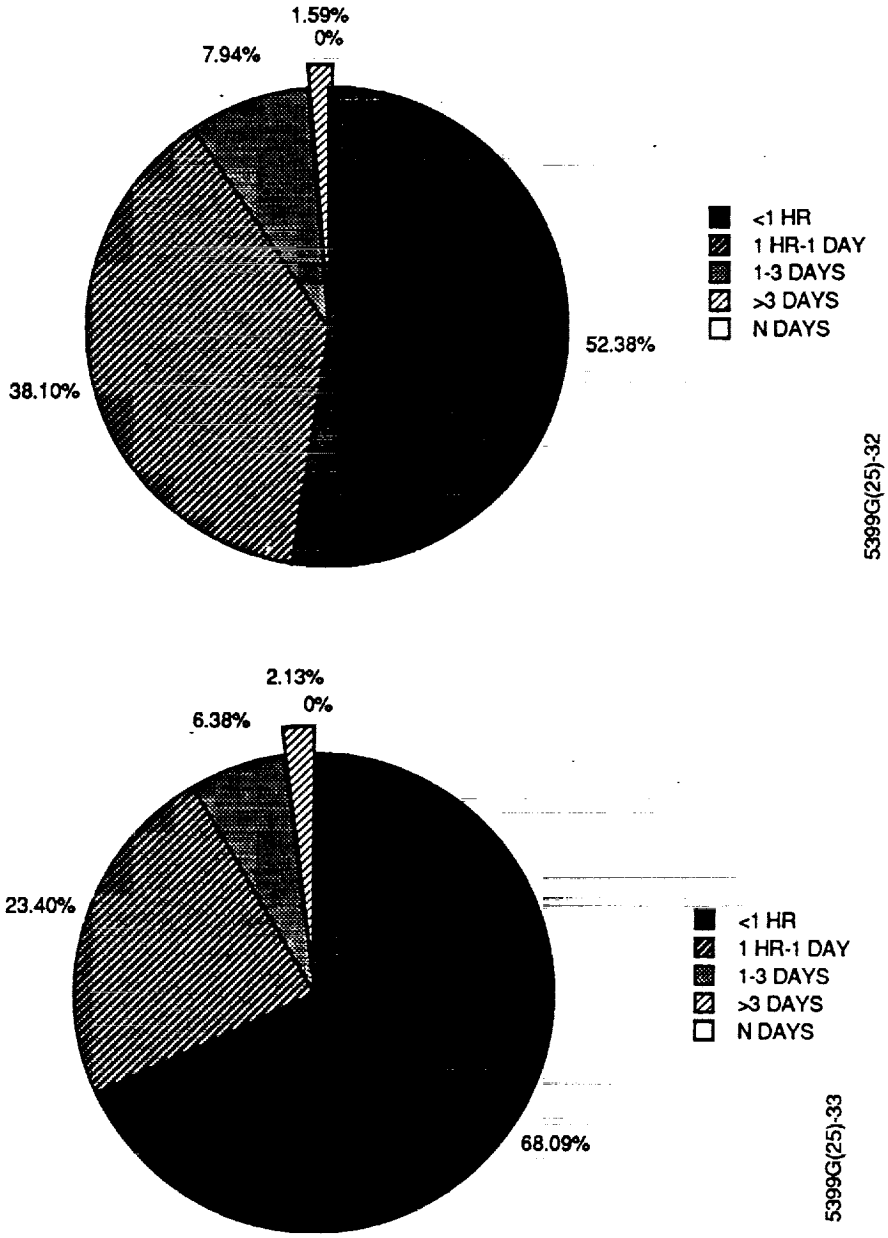
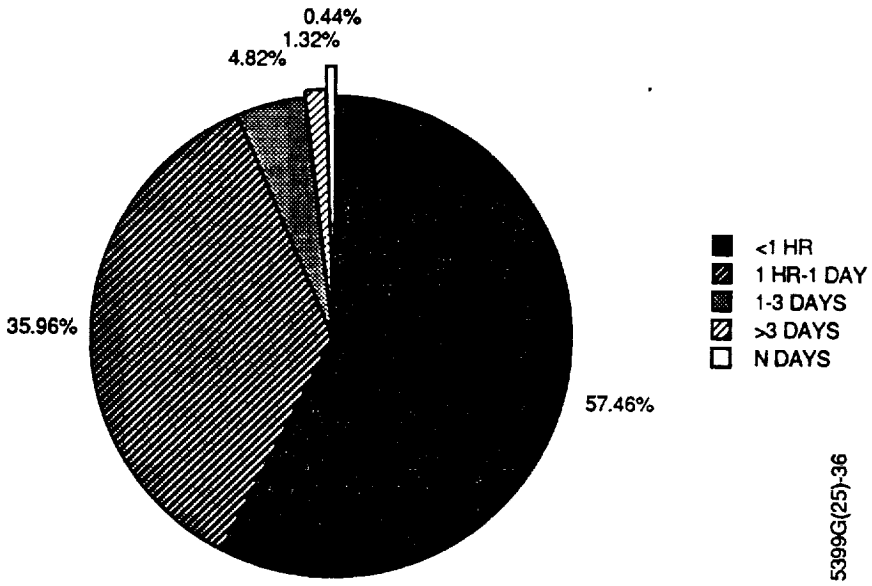


Figure 4-18. GRODY/GROSS Effort To Isolate (Errors Only)--Test



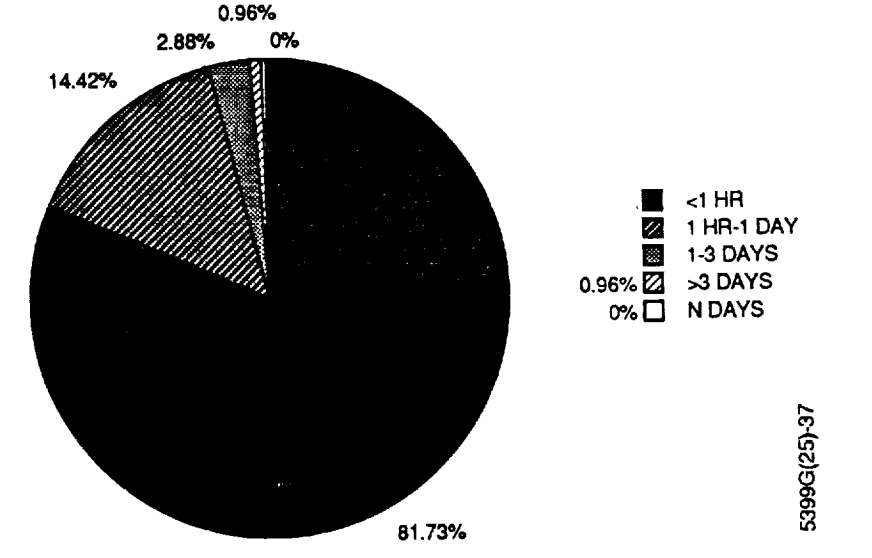


Figure 4-19. GRODY/GROSS Effort To Isolate (Errors Only)--Total Project

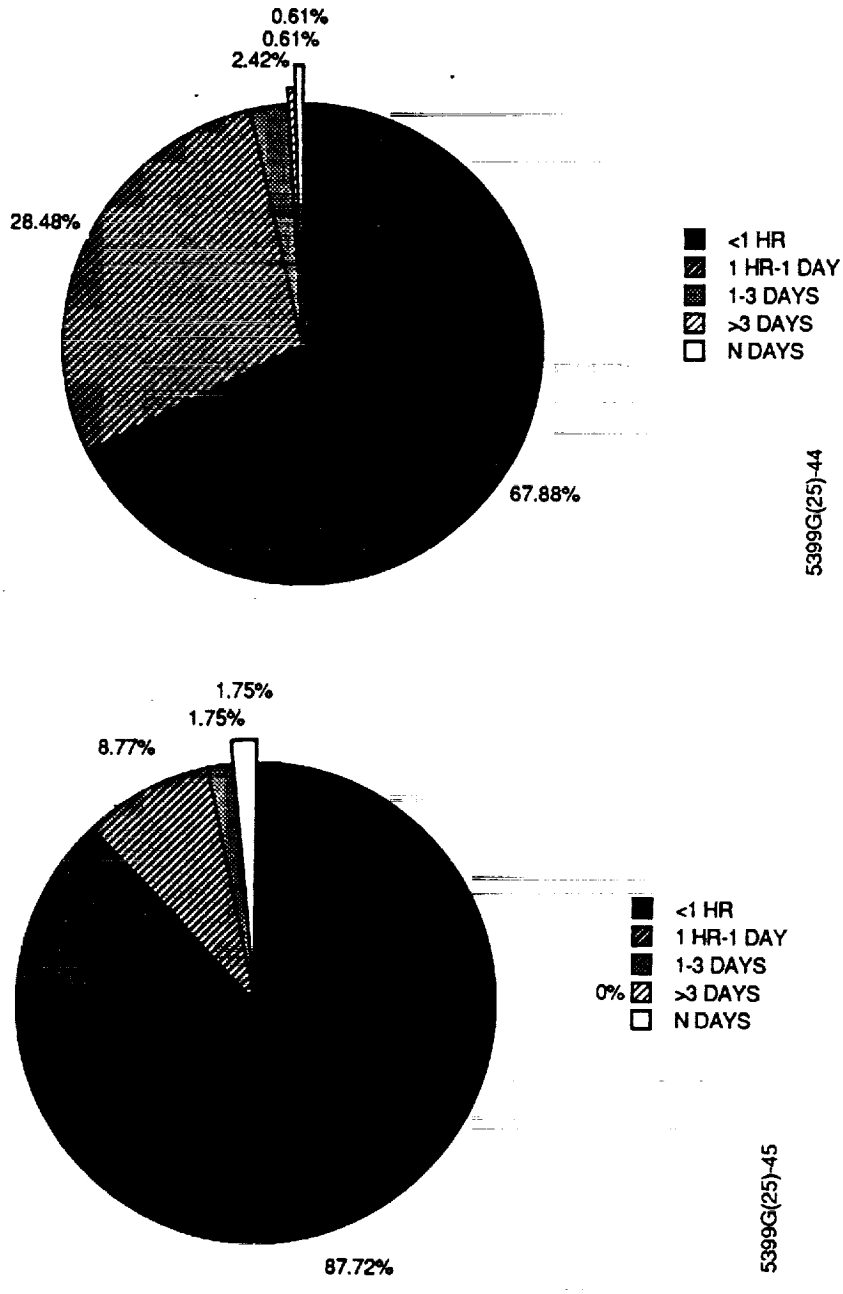


Figure 4-20. GRODY/GROSS Effort To Complete (Errors Only)--Implementation

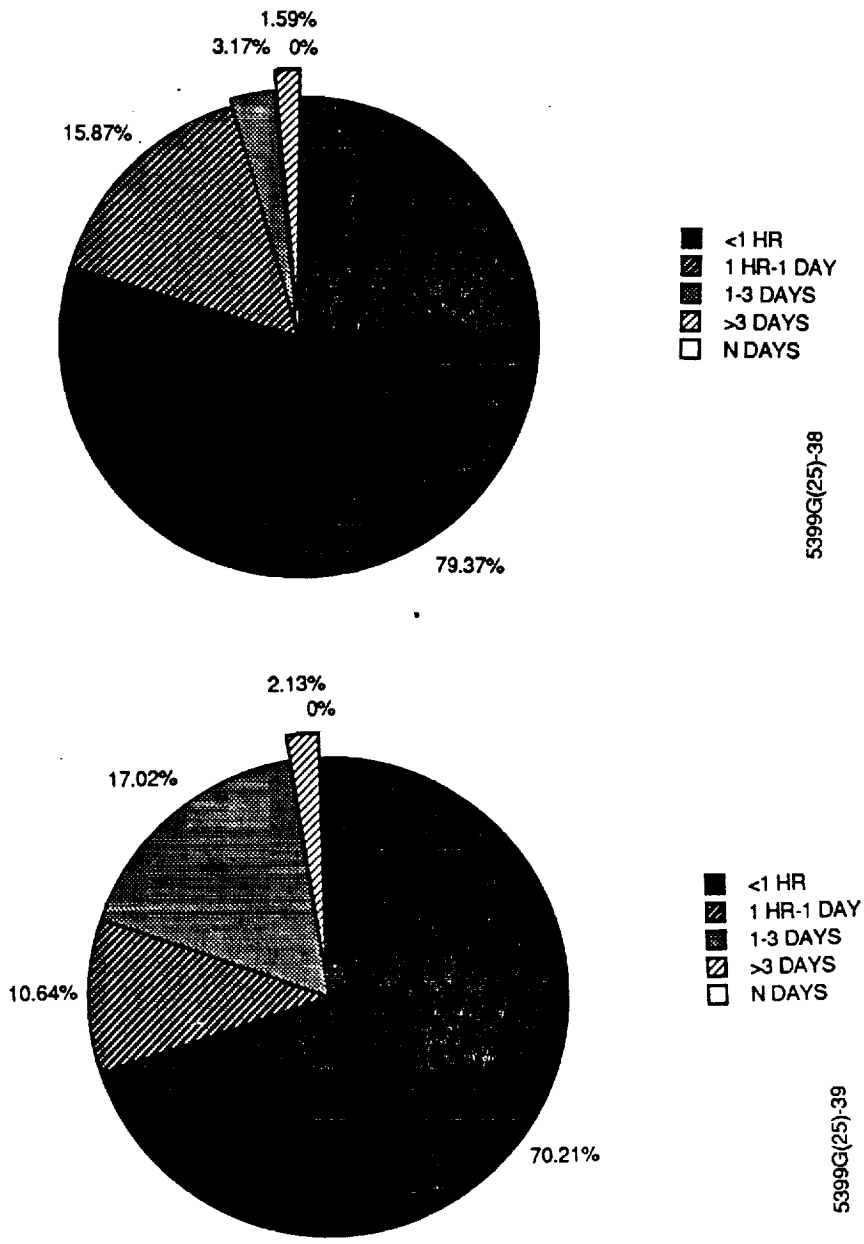


Figure 4-21. GRODY/GROSS Effort To Complete (Errors Only)--Test

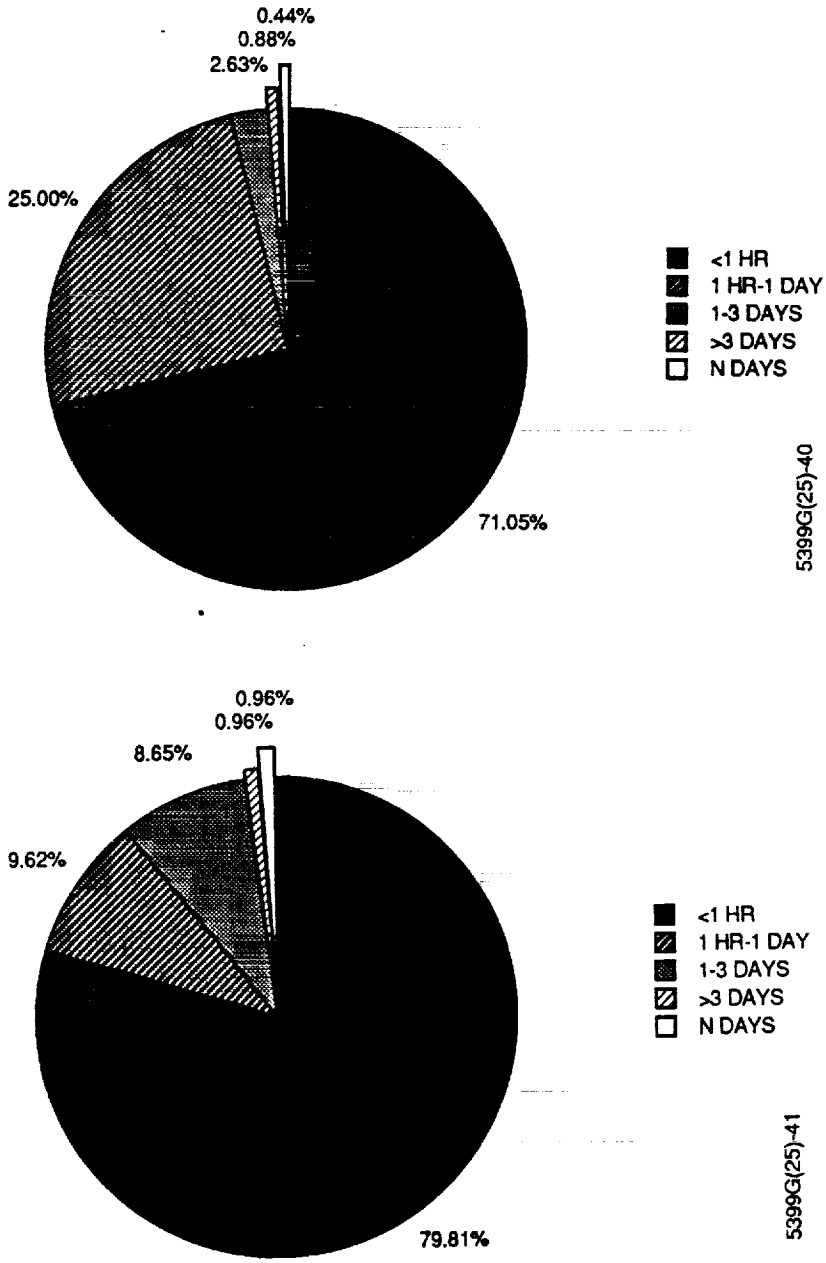


Figure 4-22. GRODY/GROSS Effort To Complete (Errors Only)--Total Project

for the Ada system than for the FORTRAN system. Ninety-six percent of the Ada errors were corrected in less than 1 day, compared to 89 percent of the FORTRAN errors.

4.4 RECOMMENDATIONS

In general, management found that the point system was very useful for evaluating progress on the Ada project, and they would recommend its use in the future. One recommendation is that units probably should not be considered to be of different complexities (and have a different point value) if specifications are developed during the actual design phase. The coding and entry of specifications accounted for most of the units classified type A and the team spent about three months developing these after CDR. The team felt this work should actually have occurred before CDR as part of the design phase. A separate point scheme could be used to evaluate the progress during the design phase while these specifications are being developed.

Distributing the implementation work by allocating packages to each programmer seemed to work well and is recommended for future projects. Parallel code development was effective with this type of work distribution.

To avoid confusion in the future concerning the precise function of a module, the design should specify the function of modules down through the procedure level, and not just at the package level. If this design approach is not taken, then extra time in the schedule should be allowed for meetings to resolve the ambiguities, or an alternate method of identifying the function of a procedure should be established.

SECTION 5 - SUMMARY AND RECOMMENDATIONS

This section summarizes some of the experience gained from the GRODY project. It looks at design issues that became evident during implementation, lessons learned in implementation, unit testing, and integration testing, and discusses management lessons and observations. Finally, it makes recommendations for future Ada implementations.

5.1 DESIGN OBSERVATIONS AND RECOMMENDATIONS

1. The team felt that the design phase for an Ada development project should include the implementation of the PDL in compilable Ada and that it should also include the completion of the specifications.

2. The design, documented using GOOD, was easy to understand and easy to translate into code. Some training in object-oriented methodology was necessary for both developers and managers.

3. During design, functionality was designed as a part of a package, resulting in some confusion concerning the functions of the procedures within the packages. It is recommended that the explicit function of each procedure be included in the design.

4. The representation of tasking in the design did not sufficiently describe task interactions. The team felt that the design should describe the location of "accepts" within tasks and the error handling necessary for the tasks. A detailed overview of the interaction of all tasks in a system is recommended.

5. To avoid generating too many types, an abstract type analysis could be incorporated into the design process.

6. Exception-handling procedures should be specified in the design and not left as an implementation feature. This will ensure consistent exception handling throughout the system and will ensure that appropriate actions are taken to solve the problem causing an exception.

7. Operational efficiency should be considered during design.

5.2 IMPLEMENTATION OBSERVATIONS AND RECOMMENDATIONS

1. The use of a style guide was very helpful in providing a uniform format for the code and guidance on use of Ada features.

2. The compiler and the debugger were the two most useful tools during the coding and unit testing of the Ada project. All the tools used were considered helpful, but in general they were considered immature and many desirable tools were not available.

3. Generics were easy to implement and found to be an effective method of reducing the amount of code necessary to perform similar functions.

4. The feature of separating specifications and bodies reduced the amount of necessary compilation and allowed early definition of the system interfaces. This feature also enabled the coding work to be distributed among the team members more easily.

5. Proper use of the strong typing in Ada requires some training. Careful planning should determine the number of types to be included in a system.

6. Tasking was difficult to implement and test. It is recommended that tasking be limited to only those problems that really require its use.

7. When portability is an implementation goal, the use of Ada machine-dependent implementation features (such as the hardware-dependent implementation of floating point numbers) should be avoided or isolated for future replacement.

8. Excessive use of nesting increases the amount of compilation necessary and complicates unit testing. The use of library units instead of nested units is recommended down to the subsystem level and for any units that interact heavily.

9. Care should be taken to minimize the size of global type packages so that changes that occur in these packages during implementation will not cause excessive recompilations.

10. The number of call-throughs should be minimized to reduce amount of compilation necessary and to reduce the total number of lines of source code.

11. Potential reuse could probably be increased by using fewer nested units and more library units and by grouping mission-specific functions in packages separate from the more general functions.

5.3 UNIT TESTING AND INTEGRATION OBSERVATIONS AND RECOMMENDATIONS

1. Code reading is still beneficial for Ada, but it needs a different emphasis. Types of errors found were style errors, initialization errors, design/code incompatibilities, and mathematical errors such as using a "less than" sign instead of a "greater than" sign. Code reading is also beneficial as a training tool.

2. Unit testing was unexpectedly difficult with Ada. It is recommended that unit testing be conducted at the package level instead of the procedure level.

3. The developer needs to have a good understanding of the ACS library structure to perform unit testing and integration effectively. The initial Ada training should include some Ada library structure training.

4. Ada's strong typing made testing more difficult by increasing the complexity of the I/O, and because multiple types resulted in larger amounts of code to be tested. However, the strong typing prevented many kinds of errors usually found in FORTRAN implementations.

5. Exception handlers must be coded carefully to ensure that the specified corrective measure really does correct the problem and not just postpone or avoid its detection.

6. The number of stubs required for testing was a problem. It is recommended that a library of specifications be developed early in implementation to be kept in the configuration library where they will be available for linking.

7. Recompilation speed was a problem during integration and testing. Most recompilation had to be done overnight due to the speed of the compiler and the size of GRODY. Good communication between the developers and the configuration manager was necessary to inform the developers of changes in the configuration library that would cause them to recompile.

8. The debugger was considered an essential tool for unit testing in Ada. It provided the only method for determining what was occurring in some of the nested units.

5.4 MANAGEMENT OBSERVATIONS AND RECOMMENDATIONS

1. It requires more Ada code to produce the same functionality provided by FORTRAN code. Based on this experience, the managers felt it was realistic to expect the Ada product to be 2-1/2 times the size of the FORTRAN

product. The Ada load module was over twice the size of the comparable FORTRAN load module.

2. The point system used to evaluate progress during the Ada project was useful.

3. Distribution of the implementation workload by allocating particular packages to each programmer worked well. Parallel code development was effective with Ada.

4. There was more reworking of the Ada code than of the FORTRAN code. Two possible reasons are the inexperience of the Ada team and the lack of schedule pressure.

5. New skills are required in order to gain the benefits that can be obtained from Ada features such as tasking and exception handling. These skills do not seem to be fully developed after training, but seem to require some on-the-job training before the power of the new features is realized.

6. The percentage of effort expended on implementation during the whole life cycle is similar to the percentage of effort normally expended in a FORTRAN implementation.

7. The change history and the growth history for the Ada project produced smoother curves than the FORTRAN project.

8. The change and the error rates per line of code were similar for the FORTRAN and Ada projects. A much higher percentage of both the errors and changes occurred in an earlier phase of the Ada project (i.e., during implementation rather than during system testing).

9. The highest percentages of errors in both FORTRAN and Ada were attributed to coding errors. In the Ada project, a significant number of errors were attributed to design, which is not unexpected, since the Ada design was

completely new while the FORTRAN design was reused and proven to be satisfactory. A significant number of errors in the Ada project were also attributed to previous changes, but this is probably due to inexperience with the language.

10. Very similar classes of errors were found in the two projects. The majority of errors in all classes were discovered at an earlier phase in the Ada project than in the FORTRAN project.

Many of the problems encountered during the implementation of the GRODY project can be attributed to the newness of Ada and the inexperience of the developers with the language. Further study is needed to determine if the added cost of development for Ada will decrease over time as experience increases and will produce benefits in increased reusability and decreased maintenance costs.

GLOSSARY

ACS	Ada Compilation System
CDR	critical design review
CMS	Configuration Management System
COSMIC	NASA Computer Software Management and Information Center
CPU	central processing unit
CSC	Computer Sciences Corporation
DEC	Digital Equipment Company
EDT	VAX Digital Standard Editor
FDAS	Flight Dynamics Analysis System
GRO	Gamma Ray Observatory
GRODY	GRO Dynamics Simulator in Ada
GROSS	GRO Dynamics Simulator in FORTRAN
GSFC	Goddard Space Flight Center
I/O	input/output
LSE	language-sensitive editor
NASA	National Aeronautics and Space Administration
OBC	onboard computer
PDL	program design language
PDR	Preliminary Design Review
SEL	Software Engineering Laboratory
SLOC	source lines of code

REFERENCES

1. A. Hoare, "The Emperor's New Clothes," Communications of the ACM, February 1981

2. T. Courtwright, "Ada Tools Update," SIGAda Meeting, Washington, D.C., September 18, 1985

3. W. Myers, "Ada: First Users--Pleased; Prospective Users--Still Hesitant," Computer, March 1987

4. Goddard Space Flight Center, "An Experiment with Ada--The GRO Dynamics Simulator Project Plan," F. McGarry and R. Nelson, April 1985

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

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

7. Computer Sciences Corporation, PCA/IM-85/055(455), The Ada Experiment--An Interim Report, W. Agresti, December 1985

8. R. C. Lingen, H. D. Mills, and B. I. Witt, Structured Programming Theory and Practice. Reading, MA: Addison-Wesley Publishing Co., 1979

9. Ada User's Group, Goddard Space Flight Center, "Ada Style Guide," September 1986

10. Software Engineering Laboratory, SEL-85-002, Ada Training Evaluation and Recommendations, R. Murphy and M. Stark, October 1985

11. Computer Sciences Corporation, Geostationary Operational Environmental Satellite-I (GOES-I) Attitude Dynamics Simulator in Ada (GOADA) Software Development Management Plan, PCA/IM-87/053, E. Booth et al., September 1987

12. Software Engineering Laboratory, SEL-87-004, Assessing the Ada Design Process and Its Implications: A Case Study, S. Godfrey and C. Brophy, July 1987

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

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

SEL-ORIGINATED DOCUMENTS

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

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

SEL-77-004, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SEL-82-706, Annotated Bibliography of Software Engineering Laboratory Literature, G. Heller, January 1989

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SEL-86-006, Proceedings From the Eleventh Annual Software Engineering Workshop, December 1986

SEL-87-001, Product Assurance Policies and Procedures for Flight Dynamics Software Development, S. Perry et al., March 1987

SEL-87-002, Ada Style Guide (Version 1.1), E. Seidewitz et al., May 1987

SEL-87-003, Guidelines for Applying the Composite Specification Model (CSM), W. W. Agresti, June 1987

SEL-87-004, Assessing the Ada Design Process and Its Implications: A Case Study, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-008, Data Collection Procedures for the Rehosted SEL Database, G. Heller, October 1987

SEL-87-009, Collected Software Engineering Papers: Volume V, S. DeLong, November 1987

SEL-87-010, Proceedings From the Twelfth Annual Software Engineering Workshop, December 1987

SEL-88-001, System Testing of a Production Ada Project: The GRODY Study, J. Seigle and Y. Shi, November 1988

SEL-88-002, Collected Software Engineering Papers: Volume VI, November 1988

SEL-88-003, Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis, K. Quimby and L. Esker, December 1988

SEL-89-001, Software Engineering Laboratory (SEL) Database Organization and User's Guide, M. So, G. Heller, S. Steinberg, and D. Spiegel, May 1989

SEL-89-002, Implementation of a Production Ada Project: The GRODY Study, S. Godfrey and C. Brophy, September 1989

SEL-RELATED LITERATURE

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

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

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

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

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

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

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

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

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

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

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

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

Basili, V. R., and J. Ramsey, Structural Coverage of Functional Testing, University of Maryland, Technical Report TR-1442, September 1984

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

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

[5]Basili, V. and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," Proceedings of the 9th International Conference on Software Engineering, March 1987

[5]Basili, V. and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," Proceedings of the Joint Ada Conference, March 1987

[5]Basili, V. and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, June 1988

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

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

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

[3]Basili, V. R. and R. W. Selby "Four Applications of a Software Data Collection and Analysis Methodology," Proceedings of the NATO Advanced Study Institute, August 1985

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

[5]Basili, V. and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," IEEE Transactions on Software Engineering, December 1987

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

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

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

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

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

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

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

[5]Brophy, C., W. Agresti, and V. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," Proceedings of the Joint Ada Conference, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," Proceedings of the Washington Ada Technical Conference, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

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

[5]Card, D. and W. Agresti, "Resolving the Software Science Anomaly," The Journal of Systems and Software, 1987

[6]Card, D. N., and W. Agresti, "Measuring Software Design Complexity," The Journal of Systems and Software, June 1988

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

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

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D., F. McGarry, and G. Page, "Evaluating Software Engineering Technologies," IEEE Transactions on Software Engineering, July 1987

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

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

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

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

[5]Doubleday, D., "ASAP: An Ada Static Source Code Analyzer Program," University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S. and C. Brophy, "Experiences in the Implementation of a Large Ada Project," Proceedings of the 1988 Washington Ada Symposium, June 1988

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

Jeffery, D. R., and V. Basili, "Characterizing Resource Data: A Model for Logical Association of Software Data," University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," Proceedings of the Tenth International Conference on Software Engineering, April 1988

[5]Mark, L. and H. D. Rombach, "A Meta Information Base for Software Engineering," University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L. and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, January 1989

[5]McGarry, F. and W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988

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

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (Proceedings), March 1980

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[5]Ramsey, C. and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," University of Maryland, Technical Report TR-1708, September 1986

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," IEEE Transactions on Software Engineering, March 1987

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," Proceedings From the Conference on Software Maintenance, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, January 1989

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," Proceedings of the 21st Hawaii International Conference on System Sciences, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," Proceedings of the CASE Technology Conference, April 1988

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1987

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," Proceedings of the Joint Ada Conference, March 1987

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[5]Valett, J. and F. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

[5]Wu, L., V. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," Proceedings of the Joint Ada Conference, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," Journal of Systems and Software, 1988

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

NOTES:

[1]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[4]This article also appears in SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986.

[5]This article also appears in SEL-87-009, Collected Software Engineering Papers: Volume V, November 1987.

[6]This article also appears in SEL-88-002, Collected Software Engineering Papers: Volume VI, November 1988.