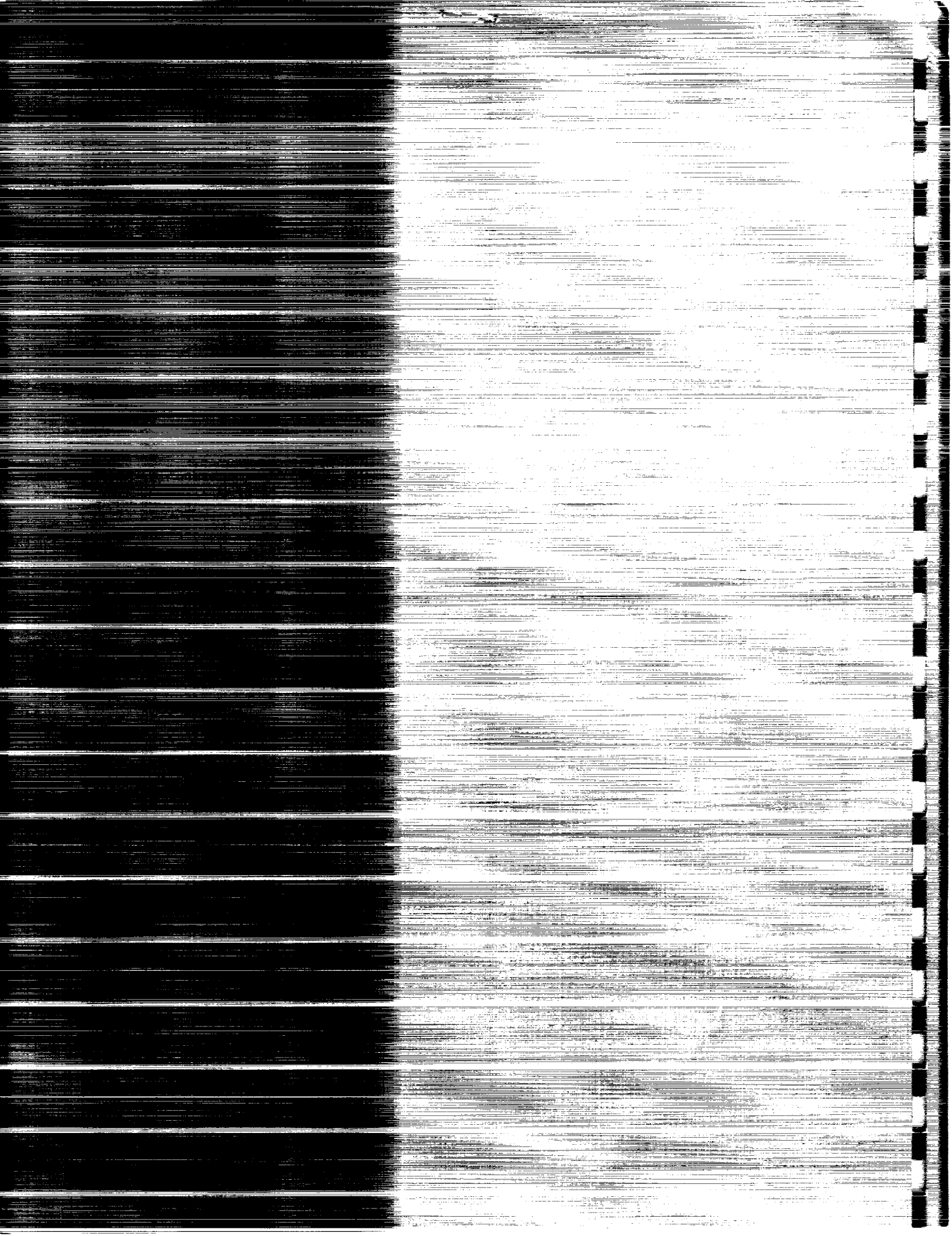
EVOLUTION OF ADA TECHNOLOGY IN THE FLIGHT DYNAMICS AREA: IMPLEMENTATION/TESTING PHASE ANALYSIS

NOVEMBER 1989

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771



SOFTWARE ENGINEERING LABORATORY SERIES SEL-89-004

EVOLUTION OF ADA TECHNOLOGY IN THE FLIGHT DYNAMICS AREA: IMPLEMENTATION/TESTING PHASE ANALYSIS

NOVEMBER 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

Kelvin Quimby	(Computer Sciences Corporation)
Linda Esker	(Computer Sciences Corporation)
John Miller	(Computer Sciences Corporation)
Laurie Smith	(Computer Sciences Corporation)
Mike Stark	(Goddard Space Flight Center)
Frank McGarry	(Goddard Space Flight Center)

Single copies of this document can be obtained by writing to

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

ABSTRACT

This report presents an analysis of the software engineering issues related to the use of Ada for the implementation and system testing phases of four Ada projects developed in the flight dynamics area. These projects reflect an evolving understanding of more effective use of Ada features. In addition, the testing methodology used on these projects has changed substantially from that used on previous FORTRAN projects.

TABLE OF CONTENTS

Executive Summary E-1

Section 1 - Introduction 1-1

1.1 Purpose. 1-1
1.2 Background 1-1
1.3 Scope. 1-2
1.4 Organization 1-2

Section 2 - Implementation in Ada 2-1

2.1 Impact of Design on Implementation 2-1

2.1.1 The GRODY Design. 2-1
2.1.2 The Second-Generation Ada Projects: GOADA and GOESIM. 2-6
2.1.3 The Third Generation: UARSTELS 2-11
2.1.4 Using a Compiled Design 2-18

2.2 Transition From Design to Implementation 2-19
2.3 Use of Ada Features in Implementation. 2-21

2.3.1 Data Types. 2-21
2.3.2 Exceptions. 2-25
2.3.3 Abstract Data Types 2-28
2.3.4 Generics. 2-29
2.3.5 Tasks 2-30

2.4 Resources. 2-32

2.4.1 Hardware Resources. 2-32
2.4.2 Software Development Tools. 2-33
2.4.3 Personnel Resources 2-34

Section 3 - Testing in Ada 3-1

3.1 Unit Testing 3-2
3.2 Incremental/Integration Testing. 3-9
3.3 Build/System Testing 3-14
3.4 Testing Tools. 3-16

Section 4 - Project Characteristics 4-1

4.1 Software Size Metrics. 4-1
4.2 Reuse. 4-4
4.3 Development Effort Through System Testing. 4-6
4.4 Productivity 4-9
4.5 Change Characteristics 4-10

TABLE OF CONTENTS (Cont'd)

Section 5 - Summary and Recommendations 5-1

Glossary

References

LIST OF ILLUSTRATIONS

Figure		
2-1	GRODY Top-Level Design	2-2
2-2	Simulation Parameter Database Package of GRO Simulator Subsystem	2-4
2-3	Parameter Database Package	2-8
2-4	Generic Ephemeris Model	2-12
2-5	UARSTELS Generic Hardware Package	2-13
2-6	UARSTELS Spacecraft Hardware Package	2-15
2-7	Generic Model Fixed-Head Star Tracker	2-16
2-8	Structure of UARSTELS Generic Package "Sensor Output"	2-17
2-9	Proportion of Total Declarations and Statements That Are Type Declarations	2-24
2-10	Profiles of Exception Declarations and Raise Statements for Ada Projects	2-26
2-11	Use of Generic Package Feature for Ada Projects	2-31
2-12	Effort Distribution for Ada Projects Through System Testing	2-37
3-1	UARSTELS Generic Star Catalog	3-6
3-2	GOADA Ada Library Structure	3-8
4-1	Component Reuse	4-5
4-2	Distribution of Effort by Life-Cycle Phase	4-8
4-3	Changes That Affect Different Numbers of Components	4-11
4-4	Comparison of Error and Change Rates by Phase	4-13
4-5	Error Source	4-14

LIST OF TABLES

Table		
2-1	Profiles of Ada Features Used.	2-22
2-2	Software Development Tools Used for Implementation and Testing.	2-35
2-3	Experience of Ada Developers at Beginning of Coding Phase.	2-35
4-1	Ada Software Size Measures at End of System Testing.	4-2
4-2	Ratio of SLOC to Total Instructions.	4-3
4-3	Line Count Profiles at End of System Testing.	4-4
4-4	Average Number of SLOC per Instruction	4-4
4-5	Reuse 1 Levels at the End of System Testing. .	4-6
4-6	Predicted Versus Actual Total Staff Hours Through System Testing	4-7
4-7	Productivity Measures Through System Testing.	4-9
4-8	Error and Change Rates Through System Testing.	4-12

EXECUTIVE SUMMARY

This report is a continuation of the study Evolution of Ada Technology in the Flight Dynamics Area--Design Phase Analysis (Quimby and Esker, 1988). It covers the software engineering issues related to the use of Ada and supporting development tools during the implementation and system-testing phases of the four simulation projects discussed in the previous document: the Gamma Ray Observatory (GRO) Dynamics Simulator (GRODY), the Geostationary Operational Environmental Satellite-I (GOES-I) Dynamics Simulator (GOADA), the GOES-I Telemetry Simulator (GOESIM), and the Upper Atmosphere Research Satellite (UARS) Telemetry Simulator (UARSTELS).

The following points summarize this analysis:

- The object diagram notation introduced by GRODY was helpful in implementing the design and in communicating changes in the design among project members during the remainder of system development. However, maintaining the design document proved to be a labor-intensive activity. Consideration should be given to automating some portion of this activity by using a commercially available object-oriented Computer-Aided Software Engineering (CASE) tool.

- This study raises the question of whether it is effective to develop a compiled design as a part of the life cycle. Further study will be required to find out what factors might be involved in determining the effect on overall project productivity and software quality of developing a compiled design.

- The integrated software development environment used to develop these simulation systems is highly rated by all Ada development personnel. However, the effectiveness

of such an environment might be strongly dependent on the availability of adequate computer resources to support it.

- Proper use of the generic feature of the Ada language has been shown to be a key factor in the development of software components that can be reused without modification on subsequent systems. More important, preliminary evidence suggests that high levels of reuse in the flight dynamics environment can be achieved effectively only through deliberate engineering of components for use on entire classes of systems.

- Ada development personnel found that bottom-up, incremental testing that uses an iterative approach to develop incremental builds of increasing functionality was easier and more efficient than the standard top-down approach to testing used in this environment.

- Component reuse was higher on the three simulation systems that reused GRODY code than on the typical FORTRAN simulation system. The lower error rates on the two telemetry simulators may be due to their smaller size and lower complexity. Productivity was higher on the UARSTELS telemetry simulator than on the previous Ada projects.

SECTION 1 - INTRODUCTION

1.1 PURPOSE

This report is one of a series on the development of Ada technology at Goddard Space Flight Center (GSFC) and the Systems Sciences Division of the Computer Sciences Corporation (CSC). A previous report (Quimby and Esker, 1988) analyzed the technical issues related to the use of Ada during the design phases of five Ada projects that have been developed in the flight dynamics environment over the last several years. This report is concerned primarily with the implementation and system-testing issues related to the use of Ada on the four simulation projects discussed in the previous report. In addition, material on design issues not covered in the previous document is covered here, especially design issues that have heavily affected implementation and testing. Implementation is defined here as including coding, unit-testing, and integration-testing activities. System testing is defined as the formal validation of the completely integrated system according to a system-test plan developed during the implementation phase (Wood, 1986).

1.2 BACKGROUND

The general background information related to this study is provided in Section 1.2 of the Design Phase Analysis report (Quimby and Esker, 1988). Each Ada project described in that section was categorized as a first-, second-, or third-generation Ada project on the basis of the technical innovations introduced. GRODY was the first major application written in Ada and thus is classified as a first-generation Ada system. GOADA and GOESIM were started after GRODY was nearly completed. Because these projects drew heavily on the lessons learned from the GRODY project, they can be viewed as second-generation Ada projects.

Finally, UARSTELS began shortly after the two GOES projects entered implementation. Because this project emphasized improving the designs developed on GOADA and GOESIM, it can be viewed as a third-generation Ada project.

1.3 SCOPE

This report covers the technical issues associated with coding and testing software systems in Ada as these issues have evolved over the history of all four of the Ada simulation projects: GRODY, GOADA, GOESIM, and UARSTELS. The three simulation systems that followed the GRODY project were required to reuse as much of the GRODY code as feasible and were thus greatly influenced by the design of GRODY. Therefore, Section 2.1 presents detailed information about the design of the GRODY system, which is necessary in order to understand the design, implementation, and testing of the three subsequent Ada simulator projects. Much of the impact of GRODY's design on these subsequent systems was not fully understood until well into implementation, and thus was not included in the previous report (Quimby and Esker, 1988). This material is included here.

1.4 ORGANIZATION

Section 1 of this report provides some background information on the projects studied. Section 2 discusses the relationship between the design of these projects and the Ada features used in their implementation and explains how the understanding and use of these features has changed from project to project. Section 3 discusses the methodology used in testing as it has evolved over the series of projects. Section 4 presents some of the measurable characteristics of each of the four Ada simulation systems. Section 5 presents a summary of the lessons learned concerning implementation and system testing of these Ada projects and includes a number of recommendations for future Ada projects.

SECTION 2 - IMPLEMENTATION IN ADA

2.1 IMPACT OF DESIGN ON IMPLEMENTATION

Although code reuse can be beneficial, an emphasis on reuse of a first-generation Ada system is likely to be premature in any environment.

GRODY was the first flight dynamics satellite simulation system written in Ada. This project significantly influenced the design and development of subsequent satellite simulation systems, which reused substantial portions of GRODY code. Reuse of major portions of GRODY code brought with it reuse of major portions of the GRODY design. Sections 2.1.1 through 2.1.3 trace the history of the major design issues and decisions that have been made on the projects GRODY, GOADA/GOESIM, and UARSTELS. The design issues discussed below are important not only because of the impact they have had on the implementation of the system from which they originated, but also because of their impact on the design and implementation of subsequent simulation projects written in Ada in the flight dynamics area.

2.1.1 THE GRODY DESIGN

The GRODY design used nesting as a mechanism to restrict visibility and used globally visible enumerated types to name simulation input and output parameters. These name decisions made it necessary to rework GRODY code extensively before it could be reused on subsequent systems.

One of the earliest decisions made by the GRODY design team was to partition the system into two subsystems, "User Interface" and "GRO Simulator" (Figure 2-1). The GRODY system description defined the "User Interface" as the subsystem that "provides all contact with the user through screen

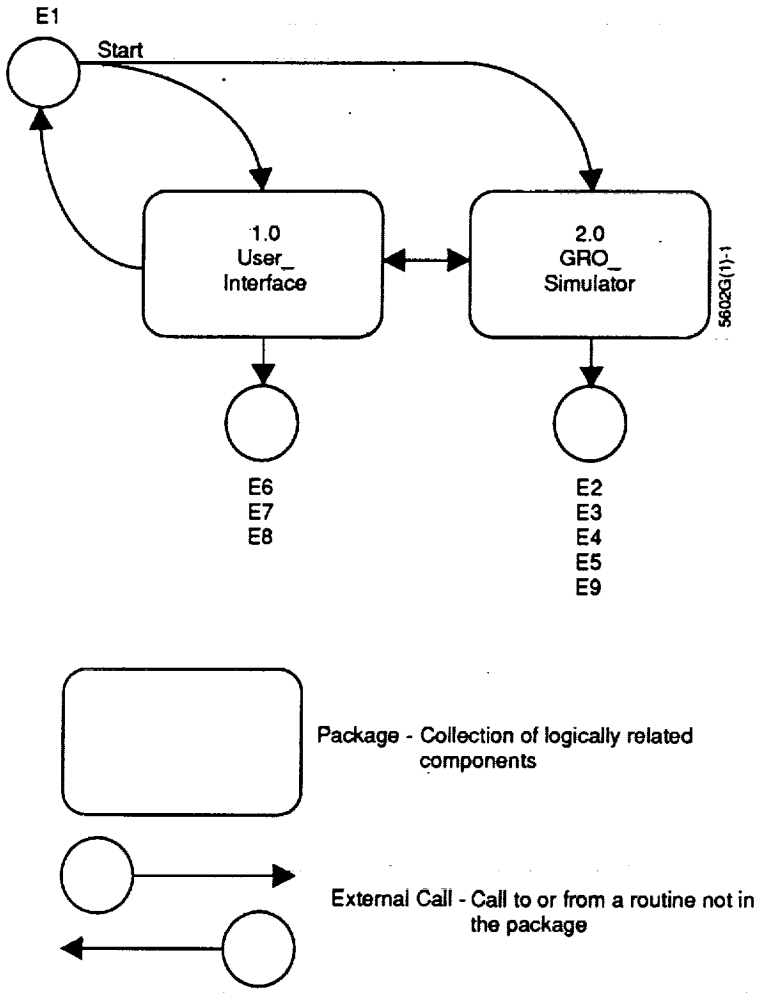


Figure 2-1. GRODY Top-Level Design

displays and generated output" (Lo et al., 1987). The "GRO Simulator" subsystem implemented the simulation of the spacecraft. These subsystems were designed to execute concurrently, except that "GRO Simulator" would be paused or stopped to allow the user to view or modify various simulation parameters within the system. Subsequent Ada dynamics simulators have retained this basic high-level system architecture.

These two high-level entities represent "objects" in object-oriented design. The "GRO Simulator" object stored the initial conditions for a simulation within the package "Simulation Parameter Database" (Figure 2-2). These values were read by the various "Truth Model" components whenever these data were needed for a computation. The computed simulation parameters were maintained as state data within the "Truth Model" components of the "GRO Simulator" subsystem. The GRODY design also stored the simulation results within the confines of the "User Interface" subsystem, using a file management subsystem implemented in a package named "Simulation Results." This package contained the declaration of a single file, "Result File," to which all simulation data was logged, including simulation parameter updates, error information, ground commands, and the simulation data itself (Lo et al., 1987). The "Parameter Database" and "Simulation Results" packages were intended to provide single objects for system input and system output, respectively. Each of these packages managed data for a single file object. The only practical way to do this was to restrict to a handful the number of different types of data objects to be stored in these files.

Given the restricted number of types that could be written to the files and the nearly 200 different simulation parameters and result outputs in the system, the design team used two approaches to minimize the number of types required

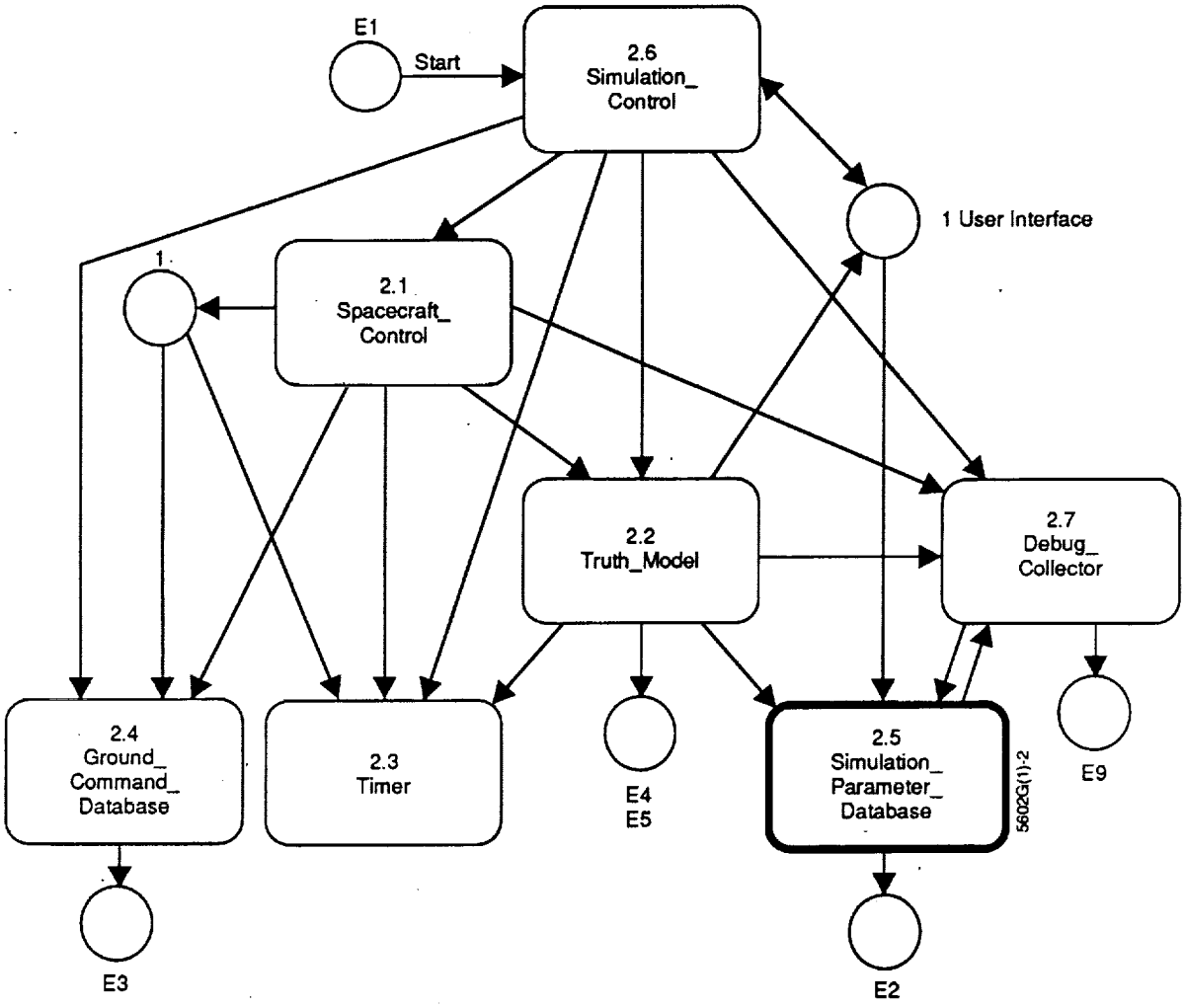


Figure 2-2. Simulation Parameter Database Package of GRO Simulator Subsystem

by the system. First, the GRODY team determined that 11 forms of data collectively define all of the different fundamental types of information needed by the system:

- Integers
- Real numbers
- Booleans
- Strings
- Arrays of Booleans
- Arrays of integers
- Arrays of real numbers
- Arrays of three-element vectors
- Arrays of 3x3 matrices of real numbers
- Two-dimensional arrays of real numbers (3x3 matrices)
- Time

Second, these 11 types could be further reduced to a single data type by using a variant record type (named PARAMETER VALUE), with 11 different component types in the variant portion of the record. Thus, each simulation parameter and result parameter was declared as an object of type PARAMETER VALUE, constrained to one of these 11 data types.

The choice of the representation of data affects the structure of individual statements, subprograms, and the overall system architecture. Conversely, the design of a particular system architecture can affect the structure of data manipulated by the system:

> ... decisions about structuring data cannot be made without knowledge of the algorithms applied to the data, and ..., vice versa, the structure and choice of algorithms often

> strongly depend on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably intertwined. (Niklaus Wirth, 1976)

In the case of GRODY, the way the system was designed determined the representation of data to be manipulated by the system. This data representation in turn determined the form in which individual declarations and assignment and control statements were implemented throughout the entire system. The following sections discuss how the design and implementation of GRODY has shaped subsequent Ada projects.

2.1.2 THE SECOND-GENERATION ADA PROJECTS: GOADA AND GOESIM

The first two follow-on projects from GRODY were the first simulation systems to use separately compilable entities to minimize recompilation overhead incurred during implementation and testing.

The next two Ada projects in the flight dynamics area were production satellite simulation systems to be used in support of GOES-I. GOESIM is a batch system used to generate telemetry data in support of the GOES-I attitude ground support system. GOADA is the same type of simulator system as GRODY. Both of these projects were required to reuse as much of the GRODY design and code as possible. Typically, components to be reused were extracted from GRODY by the GOADA team, modified, and then incorporated into both GOES-I simulators. Thus, a discussion of the GOADA project can provide an understanding of how the use of Ada in implementation has evolved in making the transition from a research-oriented project to production Ada systems.

The GOADA designers intended (Agre, 1989) that the GOADA design maximize reusability in two ways:

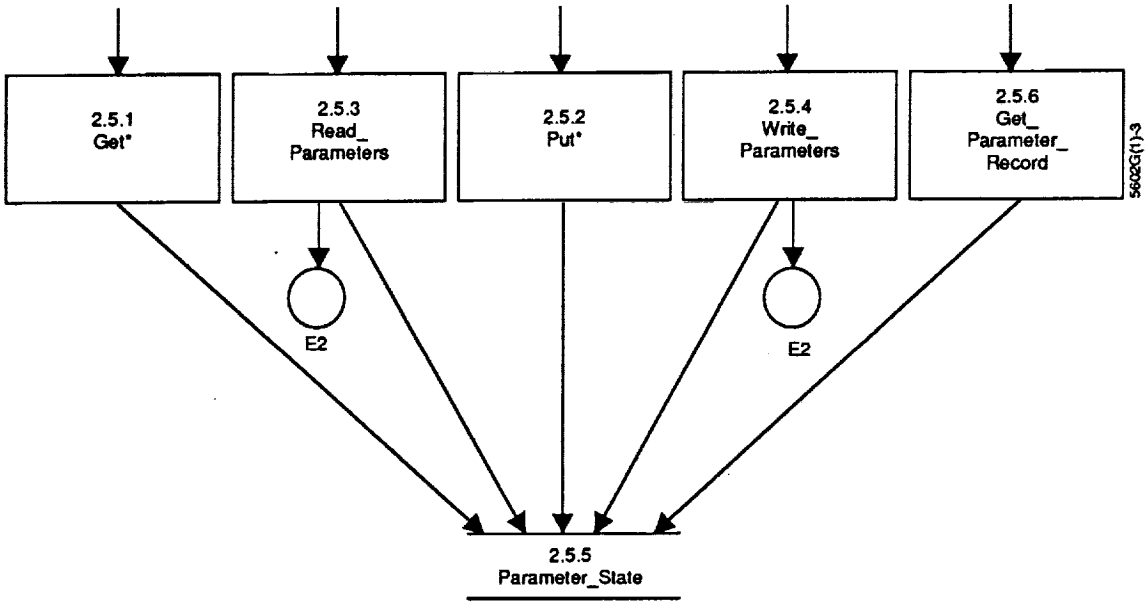
- "Reuse of GOADA for future simulator development efforts."

- "Reuse the design and code of the GRODY simulator where applicable in the GOADA design."

It became apparent during the preliminary design phase of GOADA that these two requirements were contradictory. GRODY had been implemented in a manner such that only a small fraction of the design and code could be reused verbatim. Moreover, even though GRODY code could be modified for reuse on GOADA, several team members concluded that reused code from a first-time Ada project was not a likely source of components that were deliberately designed for reuse on subsequent simulation systems.

Quimby and Esker (1988) discussed in detail the primary reason the GRODY system was not suitable as a source for reusable software to be used on future systems: the use of nesting in GRODY to restrict component visibility. As a consequence, the GOADA team was forced to undertake a major restructuring of the entire GRODY system to allow component reuse with modification of some of this code in GOADA.

Although nesting of GRODY components was a major contributor to textual and compilation dependencies among components within the system, a deeper problem facing the GOADA team was the GRODY system architecture. A single, very large data structure called "Parameter State" was used in the "GRO Simulator" subsystem to provide the initial values of all of the simulation parameters. Since 139 parameters were used in the simulation, an array of 139 different elements was used to store these initial values. The array itself was maintained within the package "Parameter Database," which exported the operations used to read and write to the state array "Parameter State" (Figure 2-3). The declaration of this single array required nearly 300 lines of code. Each time a parameter was added, modified, or deleted, this data structure had to be modified and the unit recompiled.



Subprogram - A procedure or function

Package State Memory - Local variables in a package

*Overloaded for all types of GRO_Simulator_Types. PARAMETER_VALUE.

Figure 2-3. Parameter Database Package

Just as all of the initial values of the simulation parameters were maintained in a single variable in GRODY, all 139 of the simulation parameter names (as well as an additional 31 simulation result names) were declared in the declaration of a single enumeration type, called DATUM NAME. This particular structure greatly increased the compilation dependencies among components in a system that already had extensive compilation dependencies due to its nested architecture. The dependency problem was particularly severe because these two type declarations were contained in a single type package called "GRO Simulator Types." This package was referenced by all components in the simulator subsystem and by components in the user interface subsystem that handle the retrieval, manipulation, and storage of simulation parameters. Just as for the data structure "Parameter State," each time during the development of GRODY a parameter was added, renamed, or deleted, this type declaration had to be modified and the unit recompiled. However, recompiling "GRO Simulator Types" meant that almost the entire system had to be recompiled. Such an overhead was clearly unacceptable for a system the size of GRODY and GOADA.

The GOADA team recognized that the GRODY code would have to be substantially redesigned and reworked to accommodate an approach suitable for programming a production system. They decomposed DATUM NAME into a number of enumeration types, each of which had a much smaller number of enumeration literals. Then they localized these type definitions in separate type packages associated with individual objects or subsystems. Thus, a package called "CSS Types" was developed for the "Generic Coarse Sun Sensor" package, a package "Solar Sail Types" was developed for the "Generic Solar Sail" package, and so forth. With this approach, an enumeration type such as CSS PARAMETER NAMES contained only those parameter names needed by the package modeling the coarse Sun

sensor, in this case about 29 names. The GOADA team also broke the "Parameter State" data structure into smaller data components and then distributed these components among the various packages that model particular hardware objects. As in the case of DATUM NAME, the simulation parameters related to the manipulation of the coarse Sun sensor (CSS) data were extracted from "Parameter State" and localized in the body of the package that models the actions of the CSS. These parameters collectively represented the state of the package "Generic Coarse Sun Sensor."

The GRODY architecture also had an adverse effect on the way in which the design of the system was implemented in code. For example, the sensors and actuators modeled in GRODY were to be abstract state machines according to the design. As abstract state machines, each of these hardware objects was supposed to maintain state information needed by the package that modeled the onboard computer (OBC). However, because much of the data (i.e., the initial simulation parameters) manipulated by each of these objects was not maintained as state information in the package bodies of the individual hardware components, these objects were not true abstract state machines as the design intended. By restructuring these hardware objects so that each maintained the state of all of the data that it used and manipulated, the GOADA team was able to realize to a greater degree the advantages of object-oriented design.

GOADA and GOESIM made significant contributions in the flight dynamics area to the implementation of an architecture more suited to the development of medium- to large-scale systems. Although other problems inherent in the design and implementation of code reused from GRODY remained in these systems, the greatly reduced compilation dependencies resulting from

the restructured components have substantially increased the ease with which these components can be reused on subsequent systems.

2.1.3 THE THIRD GENERATION: UARSTELS

UARSTELS is the most innovative system in the flight dynamics area to date in its use of the Ada generic facility to create verbatim-reusable components.

The UARSTELS project was characterized as a third-generation Ada project in Quimby and Esker (1988), primarily because of its contribution to the development of software components that can be reused verbatim on a subsequent simulation system. UARSTELS has clearly demonstrated that the development of reusable components is an achievable goal in the flight dynamics area. On the other hand, the project members stated that they felt severely constrained by the requirement to reuse GRODY code when trying to create components that could be reused on a much wider range of satellite mission projects.

While most of GRODY's code was difficult to reuse, subsequent Ada teams were able to reuse a few components that were implemented as generic packages. These components include a set of utilities, a numeric integrator, and a "Generic Ephemeris Model." In the opinion of the UARSTELS development team, the GRODY project's most important contribution to implementation was the manner in which it used generic packages in structuring the "Generic Ephemeris Model" component (Figure 2-4). This package provides three different options for generating time-tagged satellite position and velocity data that describe an orbit, and does so by instantiating three different generic packages within its body. UARSTELS used this approach of nesting generic instantiations in developing the "Generic Hardware" package (Figure 2-5).

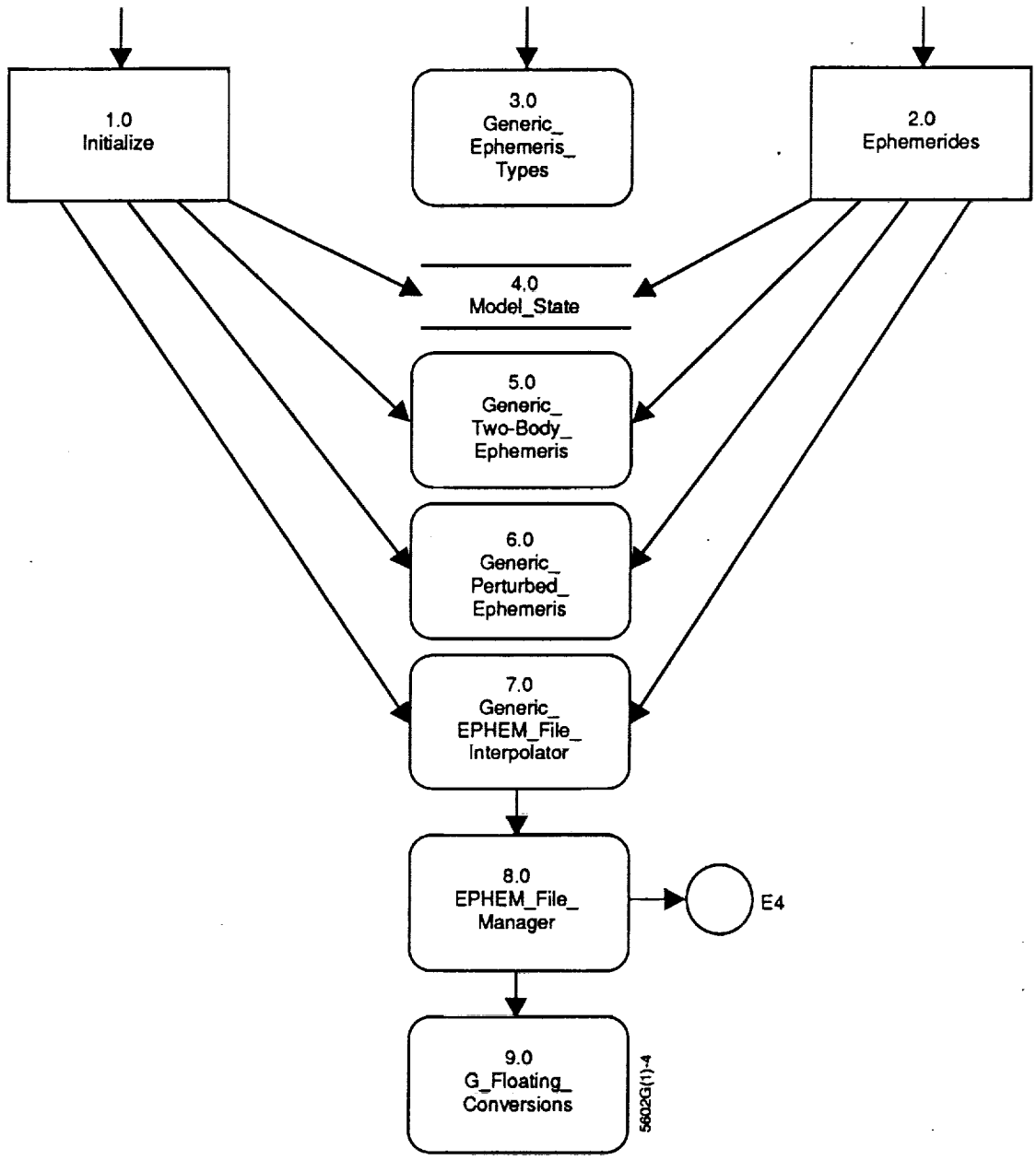


Figure 2-4. Generic Ephemeris Model

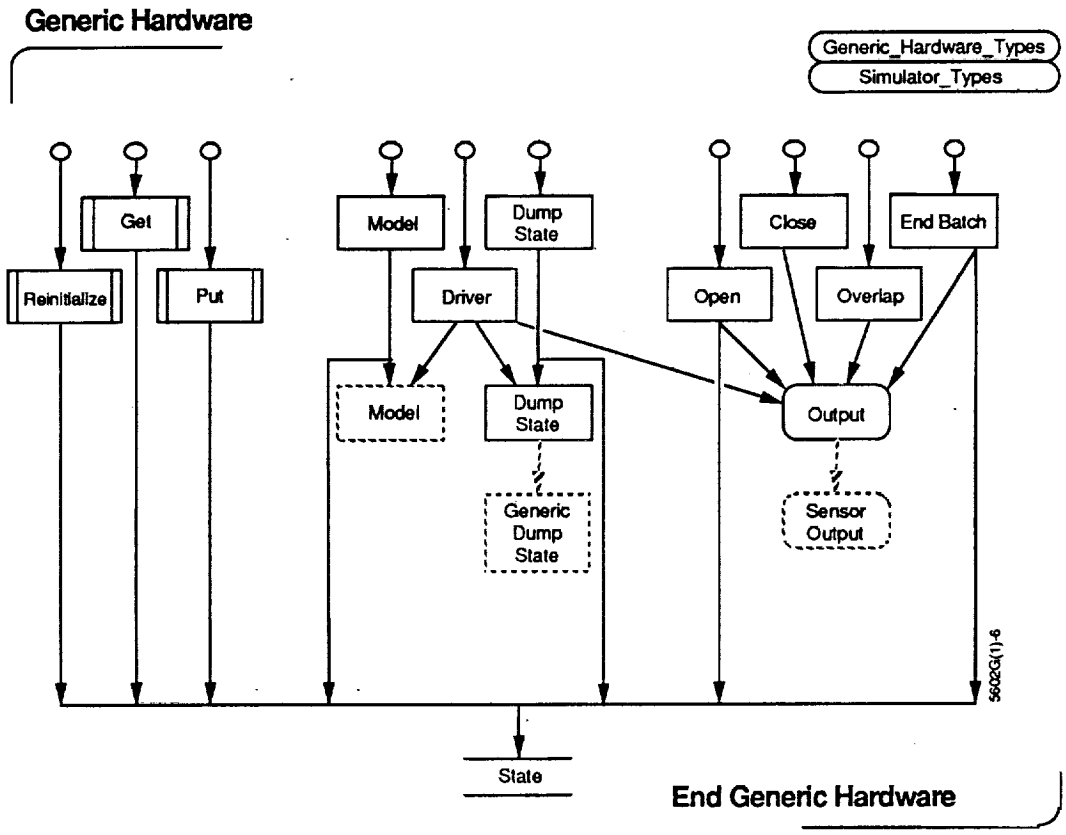


Figure 2-5. UARSTELS Generic Hardware Package

This package is a high-level abstraction used to instantiate 17 different software components that represent the various hardware objects residing on the UARS satellite (Figure 2-6). The hardware-specific details belonging to each hardware object have been extracted from previous hardware packages developed for GRODY and GOADA and replaced by generic parameters that are provided for each instantiation of a particular hardware component (Booth, 1989).

The package "Generic Hardware" illustrates the extent to which the UARSTELS project has increased the use of the generic features of the Ada language in building components out of a layered sequence of generic instantiations and generic parameters supplied to successively higher-level generic objects (Stark and Booth, 1989). Thus, the instantiation of a hardware object such as the fixed-head star tracker (FHST) represents a structure composed of several levels of instantiated generic components. The package "FHST" imports the generic packages "Generic Model FHST" and "Generic Hardware" and instantiates these in its specification as the function "Model" and the package "Hardware," respectively. The generic function "Generic Model FHST" has three generic subprogram parameters, "Corrupt," "External Model," and "Digitize" (Figure 2-7). The subprograms "Corrupt" and "Digitize" are exported by instantiations of the "Generic Sensor Corruption" and "Generic Sensor Digitization" packages, respectively. Finally, the body of the "Generic Hardware" package contains a nested package "Output" that is an instance of the generic package "Sensor Output." In the specification of "Sensor Output," there are three instances of generic file output packages: "Report Writer," "Data Set," and "Plot File" (Figure 2-8).

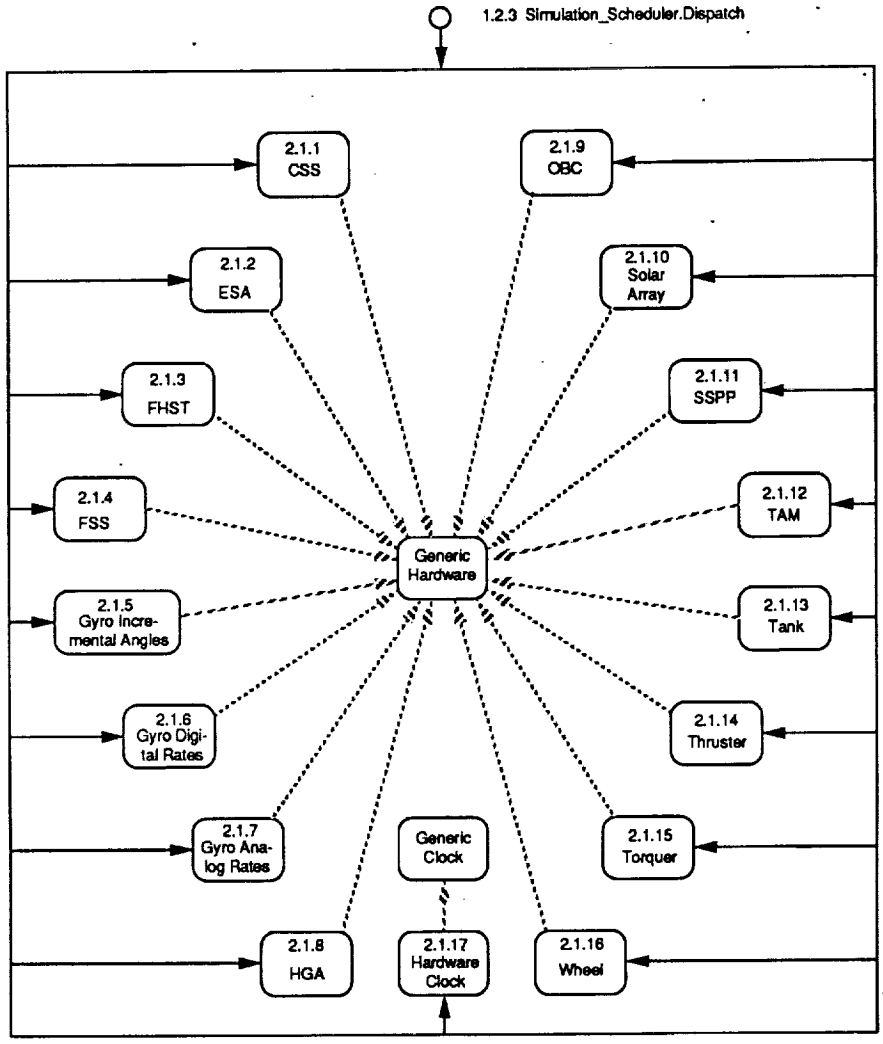


Figure 2-6. UARSTELS Spacecraft Hardware Package

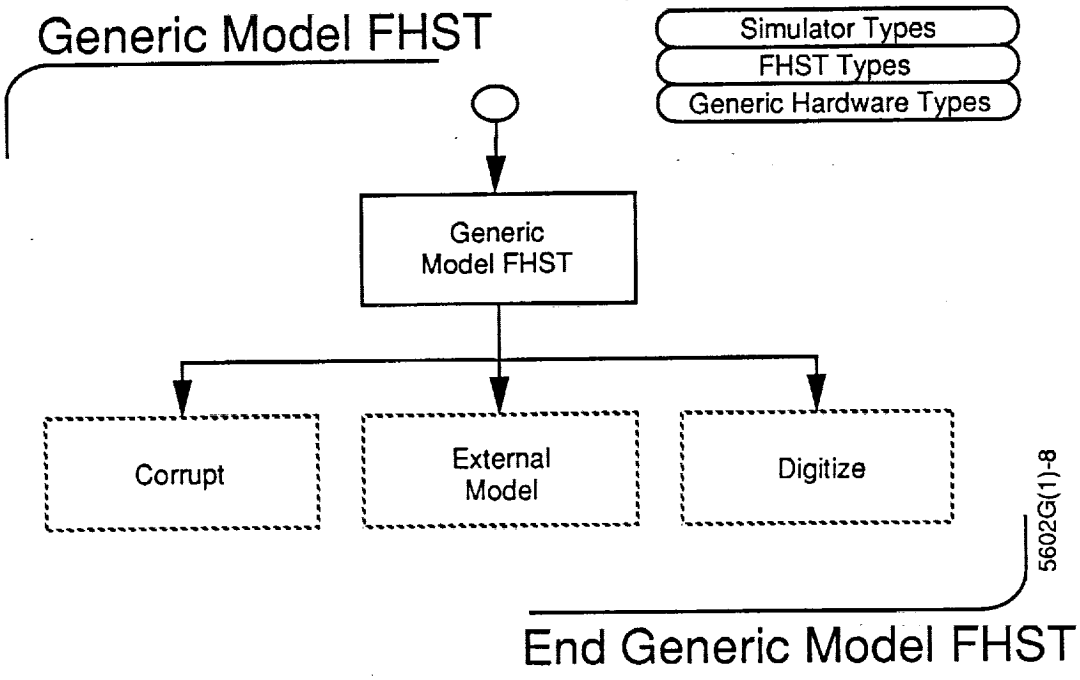


Figure 2-7. Generic Model Fixed-Head Star Tracker

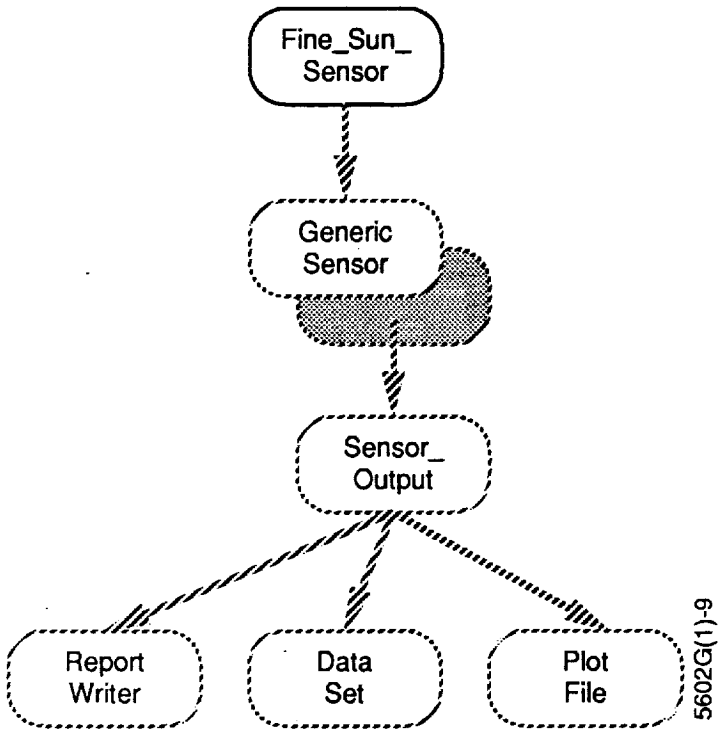


Figure 2-8. Structure of UARSTELS Generic Package "Sensor Output"

2.1.4 USING A COMPILED DESIGN

Further research will be required to determine when a compiled design is beneficial for simulation systems produced in this environment.

The Ada projects GOADA, GOESIM, and UARSTELS developed a compiled design during the design phase of the life cycle (Quimby and Esker, 1988). At the time of the critical design review (CDR), the designs of these systems were expressed in compiled program design language (PDL) that included compiled control statements and calls to subprograms in lower-level packages written as comments. At the time, most development personnel viewed this form of design as potentially beneficial to the development process.

At present, it is not clear what effect developing a compiled design has on the life cycle of an Ada simulation system in the flight dynamics environment. The detailed design for any simulation system in this environment undergoes substantial changes between the CDR and the end of acceptance test. Many of these changes are due to requirements changes, which often occur during development because the satellite has not been fully specified at the time development begins. Given that some portion of the requirements for a simulation system can be expected to undergo significant changes, it is probably advisable to invest an effort in developing detailed PDL only in those aspects of the system that are well defined and stable.

Another factor affecting the advisability of developing a compiled design in Ada is the availability of computer resources. This issue was not discussed in Quimby and Esker (1988), although that document did note that Ada compilers are complex systems that consume far more central processing unit (CPU) resources than do FORTRAN compilers. Developers interviewed during the coding phase felt that access to

hardware resources was not adequate during detailed design. A scarcity of computer resources thus may inhibit the rate of progress on development of the design by slowing down the process of getting a design compiled.

One of the section managers raised the concern that the development of a compiled design may inhibit the development of the project by turning the developer's effort from design issues to coding. The large investment in effort to develop and compile a detailed design may also reduce the developers' ability or willingness to modify the system when it is technically good to do so.

The life cycle model may also have an effect on the usefulness of developing a compiled design. A waterfall model was used in developing these systems. It is possible that developing a compiled design may be more suitable with alternative software development models that offer greater flexibility in response to requirements uncertainty or volatility, such as the spiral model, rapid prototyping, transformational analysis, or incremental development (Agresti, 1986). Further research will be required to determine if a compiled design is beneficial, the conditions under which it would be beneficial, and the level of detail to which it should be elaborated.

2.2 TRANSITION FROM DESIGN TO IMPLEMENTATION

The graphic notation introduced by GRODY and improved on by GOADA and UARSTELS was helpful to developers in implementing the design.

The transition from design to implementation on the GRODY project was difficult for a number of reasons. The major reason was simply that the design was not complete at the time of the CDR. Although some package specifications had been compiled at that time, much of the system was still undefined. In addition, even after the CDR had been held,

the team was holding many discussions about how to structure the system.

The transition from design to implementation on the three simulation projects that followed GRODY was smoother. During the design phase, developers on these projects were fully exposed to the mechanics of developing Ada code when they compiled package specifications and PDL. In addition, they had to become familiar with unit compilation dependencies, context clauses, and Ada library structures. This experience was a benefit of developing a compiled design.

The difficult transition for the GOADA design team was unnesting the GRODY code during preliminary and detailed design. By the time the formal coding phase had started, the GRODY code had been restructured. The projects GOESIM and UARSTELS were able to reuse most of the unnested packages that they needed from GRODY by taking them directly from the GOADA project.

The other issue in the transition from design to implementation was the effects of a graphic representation of the design on the coding and testing phases of the projects. Most developers felt that the graphic notation was helpful in communicating design changes among the developers on a given project. However, substantial overhead was required to maintain and update the design document. A commonly expressed opinion was that too much effort was required to update the design notebook manually, and that the resources for doing so were too scarce. The design documentation was maintained using a graphics software package (MacDraw®) on Apple MacIntosh® personal computers. Although this system is easy to use and produces professional-looking documents, the scarcity of these computers and the lack of CASE capabilities in the graphics package limited the usefulness of this

approach. A more recent Ada project is experimenting with the use of a commercially available object-oriented CASE tool that runs on more widely available IBM-compatible personal computers. Other CASE tools that support object-oriented design are being introduced in the market and should be evaluated for potential use in this environment as they become available.

2.3 USE OF ADA FEATURES IN IMPLEMENTATION

The requirement to reuse code from the first-generation Ada project has made limited progress in making optimal use of the features of the language.

The Software Engineering Laboratory (SEL) analyzed each of the four projects GRODY, GOADA, GOESIM, and UARSTELS after each project completed system testing, using the Ada Static Analysis Program (ASAP). This tool was developed at the University of Maryland for collecting software metric data on Ada projects (Doubleday, 1987). ASAP can be used to generate a profile of Ada features used on an analyzed system. Table 2-1 shows the profiles of some of these features for the four simulation systems. The use of these features is discussed in the remainder of this section.

2.3.1 DATA TYPES

The use of variables, constants, and formal parameters has changed slowly from the FORTRAN environment.

A previous study of the GRODY project (Godfrey and Brophy, 1989) described what the authors considered to be a counter-productive proliferation of derived types and subtypes in the system. However, team leaders from each of the subsequent projects GOADA, GOESIM, and UARSTELS stated that they were constrained by what they considered to be GRODY's under-use of Ada's powerful data typing capability. The data objects declared in the reused GRODY code reflect the strong

Table 2-1. Profiles of Ada Features Used

	GRODY[1]	GOADA	GOESIM	UARSTELS
Type Declarations	417	772	372	726
Package Specifications	53	109	71	78
Generic Package Specifications	9	30	23	41
Generic Package Instantiations	44	99	68	116
Exceptions Declared	103	162	87	70
Raise Statements	388	710	360	295
Tasks	8	3	0	0

[1]Estimate was extrapolated from the data successfully processed by ASAP. ASAP was unable to analyze portions of the GRODY code, apparently because of the heavily nested architecture of this system.

FORTRAN legacy in the environment. Most numeric variables, constants, and record, array, and matrix components are of type REAL, a user-defined floating-point type meant to mimic the REAL*8 type widely used throughout the FORTRAN code developed for flight dynamics applications. As an approximate measure of the extent to which user-defined data types have been used in these systems, Figure 2-9 shows the proportion of total Ada declarations and statements that are type declarations for each project. (Line counts are shown later, in Table 4-1.)

One of the reasons for Ada's strong typing mechanism is to allow the compiler to prevent an important class of programming errors, the accidental mixing of variables of different types in the same expression (Barnes, 1989). Furthermore, subtypes can be used to provide range checking on variables at runtime and thus obviate explicit range checks in source code. Although they currently disagree about the degree to which user-defined and derived types should be used in simulation systems, the team leaders of the three Ada projects that followed GRODY all stated that subtypes should be used much more extensively in future systems.

Developers from the GOADA project stated that the reused user-interface code from GRODY prevented them from employing user-defined enumeration data types. Instead, input and output to the user interface were restricted to the predefined type STRING, which is an unconstrained array of characters. Use of user-defined enumeration types and "Enumeration IO" causes the compiler to generate code that verifies the correctness of character strings during execution and input/output operations. Use of the predefined type STRING instead of enumeration literals requires that developers write their own runtime checks.

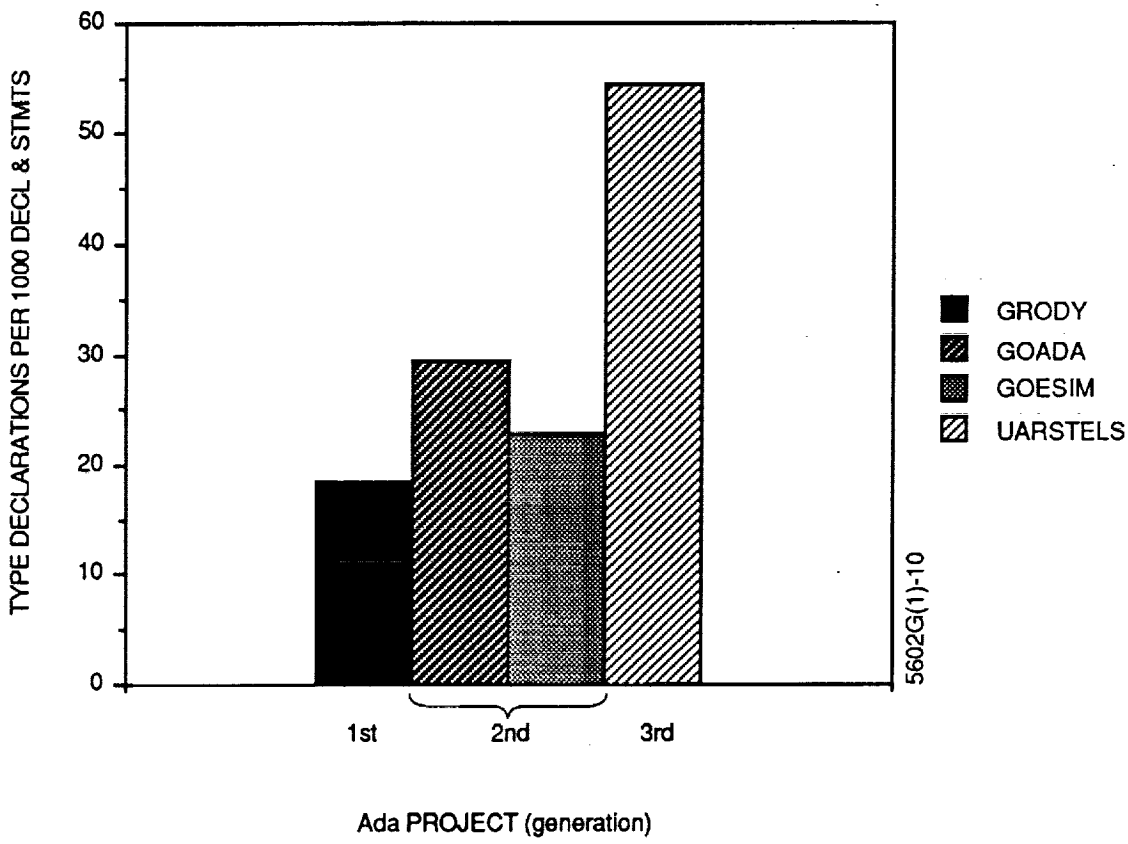


Figure 2-9. Proportion of Total Declarations and Statements That Are Type Declarations

Most developers expressed the opinion that user-defined enumeration types should be used to a greater degree on future Ada projects.

2.3.2 EXCEPTIONS

Appropriate use of the exception mechanisms in Ada is only apparent in the most recent Ada projects developed in this environment.

Appropriate use of the exception mechanisms provided by Ada has, in general, evolved very slowly in the flight dynamics environment. This has been primarily due to two factors: (1) an insufficient understanding of this feature of the language among design and development personnel, and (2) postponement, because of schedule pressure, of design decisions about raising and handling exceptions. As a result, exceptions have often been designed into a system during the later stages of implementation and even well into system testing. More rigorous and thorough unit and integration testing of software components during implementation will expose inadequate or missing exception-handling situations that were not discovered during design. In addition, with increasing reuse of well-engineered components that export user-defined exceptions, developers of packages that invoke these reused components will become more aware that their design and code needs to recognize and handle these exported exceptions. Figure 2-10 shows the proportions of exception declarations and raise statements for the four simulator projects.

The designers of the Ada language have been very explicit in proscribing the use of exceptions as regular condition flags (Barnes, 1989). However, there are several places in the GOADA project where exceptions are used as a part of the normal processing within a system. For example, the subprogram "Schedule Events" of the package "Simulation Scheduler" uses exceptions to control processing associated with pausing

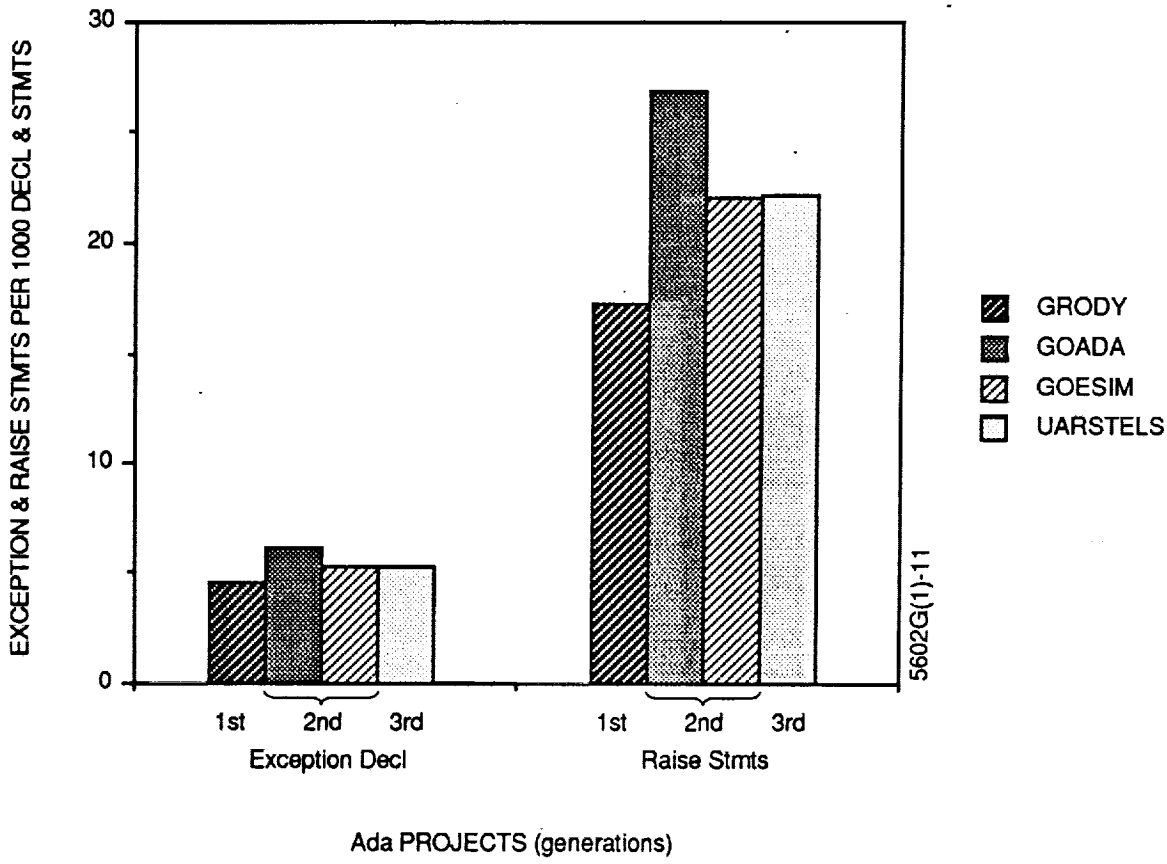


Figure 2-10. Profiles of Exception Declarations and Raise Statements for Ada Projects

with pausing and stopping the simulation and with detecting ground commands issued during a simulation run.

Another issue concerning the use of exceptions is related to the concept of levels of abstraction. Booch (1987, p. 53-54) advocates that software components be designed so that they export exceptions whose level of abstraction is consistent with the level of abstraction the component itself represents within the system. Specifically, components should not propagate predefined exceptions such as "Constraint Error," "Numeric Error," and "Data Error," beyond package boundaries, since these are predefined exceptions that are raised by the runtime system when an incorrect value is assigned to a variable. Although a number of packages within the GOADA system did propagate predefined exceptions, more recent Ada projects are only exporting user-defined exceptions as recommended by Booch (1987).

Another objection to raising predefined exceptions is that the specification of a package cannot explicitly export these exceptions. A developer who uses this kind of package would have to rely on documentation stating that one or more predefined exceptions might be raised. In addition, the developer could not write an exception handler that could distinguish between the predefined exception raised by this particular package and the same predefined exception raised by some other program unit in the system. GOADA violated this advice in that most of the hardware components captured and raised "Constraint Error" and "Numeric Error" exceptions to the next higher level in the system architecture. This practice has been discontinued on the subsequent simulation projects.

2.3.3 ABSTRACT DATA TYPES

Ada systems are evolving toward the use of abstract data types and abstract state machines for most problem domain objects in the flight dynamics area.

Abstract data types are one of the most important kinds of components that can be developed using the Ada package construct. The first use of abstract data types in the flight dynamics area occurred on the Flight Dynamics Analysis System (FDAS), a research project that was the first Ada project started in this environment. For this project, two generic packages were used to implement complex data structures: a generic balanced binary tree and a generic stack. Although abstract data types were not used at all on GRODY, they have been used on all subsequent simulator projects. A major reason for this adoption was that GSFC licensed the source code for a large number (501) of commercially available Ada software components by Grady Booch. This collection of components includes all of the standard complex data structures, such as stacks, linked lists, queues, dequeues, graphs, binary trees, maps, and sets. The GOADA project used an instantiation of a generic queue package to implement an event-driven scheduling algorithm. The GOESIM project employed a generic ring component by Booch.

The use of abstract data types can be extended beyond data structures to problem domain objects such as sensors. The simulators discussed in this document distribute some of the hardware object states to a parameter database component. However, it is possible to implement sensors as abstract data types and then build components around these types (Stark and Booth, 1989).

2.3.4 GENERICS

The greatest technical advance in implementing Ada systems in the flight dynamics area is occurring in the use of the generic feature of the Ada language.

GRODY was the first Ada project to write generic components, although only a handful of packages were generic. These packages included "Generic Utilities" and the generic packages that compose the generic ephemeris component. The use of generic packages was expanded on the GOADA and GOESIM projects, which used them to implement the hardware objects in the system.

The UARSTELS project was the first Ada system to use the generic feature of the language extensively. The UARSTELS development team was the first to recognize and advocate that the design and development of verbatim-reusable components in Ada requires the use of the generic feature of this language. The UARSTELS team also recognized that the consumption of a few minutes of CPU time for a recompilation to effect a change across a number of generic instantiations can simplify the development process. (However, scarcity of computer resources may detract from this benefit.) Using generics allows the developer to limit the scope of a change to the template that specifies the algorithm being used (the generic itself), or to a specific generic parameter that is supplied to the generic during instantiation. This approach partially automates the process of changing the system, in that the compiler propagates the change across all logically affected components. Without generics, developers would have to edit and compile n source files to effect this change over n components. This occurred several times as changes that affected all 17 hardware components were made by modifying the generic hardware package and then compiling the 17 instantiations with a single "recompile" command.

Figure 2-11 shows the use of generics on these four projects as measured by the ratio of generic package specifications to non-generic package specifications. There is a clear trend of increased use of generics, from GRODY (0.17) to the second-generation projects (0.28 and 0.32 for GOADA and GOESIM, respectively), to 0.53 for UARSTELS.

2.3.5 TASKS

Ada tasks will continue to be used where appropriate in the flight dynamics environment.

The task construct has been the most difficult Ada feature to use effectively on this series of simulator systems, particularly on the GRODY project. When design problems led to deadlock situations during the later stages of GRODY implementation and system testing, the developers attempted to address the problem (inappropriately) by adding more tasks until the user-interface subsystem contained seven different task objects. The original purpose of using tasks on GRODY was simply to toggle between the execution of the simulator and execution of the user interface so that the user could enter commands, view output from the simulator, and so forth. The GOADA project, which reused the GRODY user-interface subsystem, redesigned it so that the desired ability to toggle between executing the user interface and the executing the simulator could be accomplished using only three tasks. Tasks have not been used on the GOESIM and UARSTELS projects because they are not appropriate applications for this particular feature of the language, given the sequential nature of the processing requirements on these projects.

Since a satellite houses a number of sensors, actuators, and other hardware components that function in parallel, future satellite simulation systems could be designed with multiple tasks to take advantage of future multiprocessor

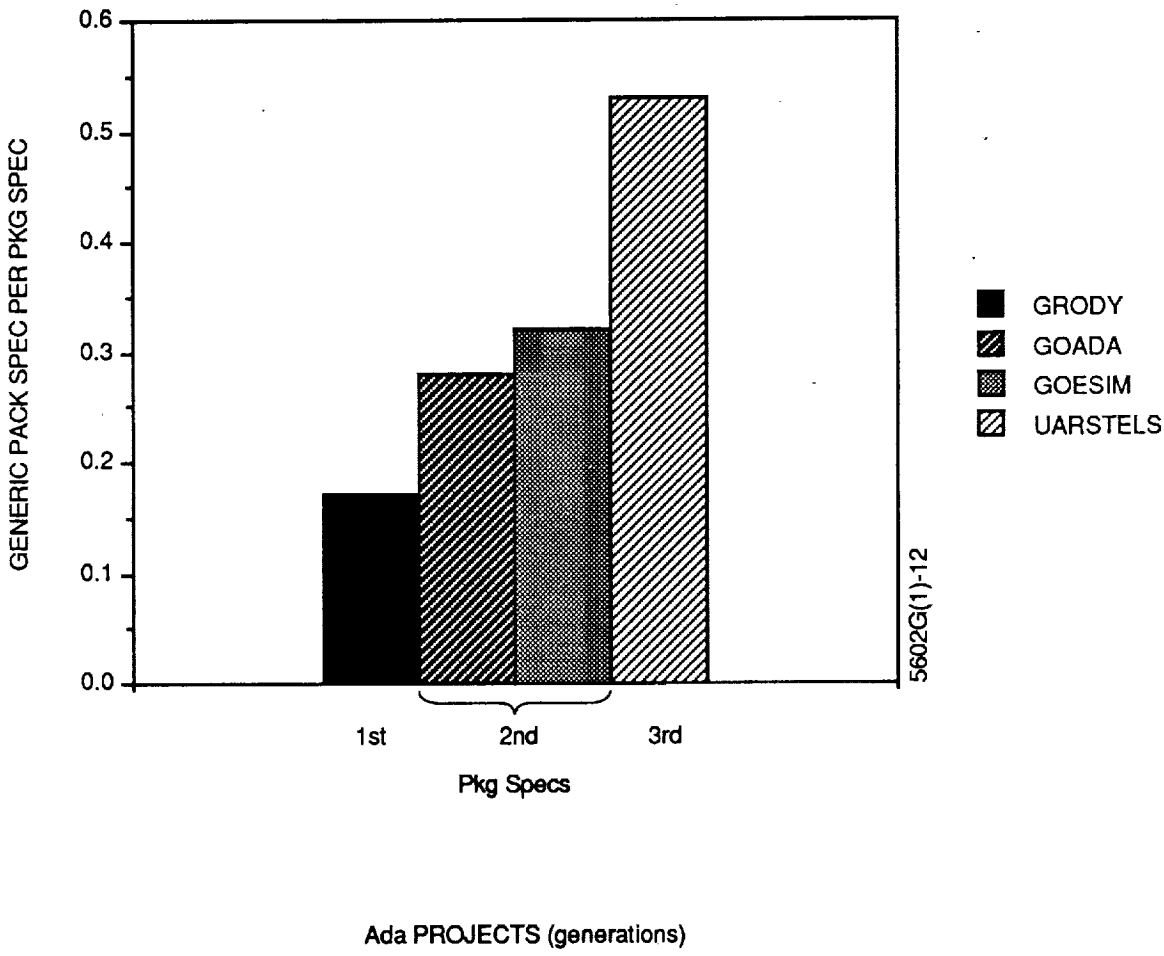


Figure 2-11. Use of Generic Package Feature for Ada Projects

or high-performance single-processor computer systems. At present, the overhead associated with the rendezvous mechanism of Ada tasks on the current version of the Ada compiler used in this environment is still too high to allow single-CPU systems to execute satellite simulation systems with a large number of tasks within required time constraints.

Ada is the first high-order language that supports parallel or concurrent program execution without operating system calls or low-level monitors or semaphores at the source code level. As such, it substantially simplifies the process of developing portable concurrent systems (Gehani, 1984). However, developing concurrent software systems is inherently more difficult than developing sequential systems. Lack of experience or knowledge in this area appears to be the primary reason for the difficulties developers in this environment have had in using the task construct of Ada.

2.4 RESOURCES

2.4.1 HARDWARE RESOURCES

Preliminary observations suggest that computer resources should be increased to handle CPU-intensive Ada compilers and the associated recompilation overhead incurred during development of Ada systems.

A common complaint among all Ada personnel was that access to hardware resources was not adequate during development. Developers estimated that on average, computer terminals were only available about 60 percent of the time they were needed. In addition, system throughput was often poor, especially for developing GOADA's compiled design on the VAX 11/780. Compiling a single GOADA subprogram sometimes took 20 to 30 minutes or more during peak hours. Finally, the

communication between the terminals and the development system was unreliable and subject to frequent failure.

During development, recompilation of the system often required many hours and further degraded the performance of the host system. This situation did not improve during coding and system testing, even with the introduction of the more powerful VAX 8810. The demand for computer resources for developing three Ada simulation systems in parallel increased beyond the effective capacity of the newer VAX.

2.4.2 SOFTWARE DEVELOPMENT TOOLS

The software development environment for Ada in the flight dynamics area is highly rated by all Ada software development personnel.

The coding and system testing of the GOESIM, GOADA, and UARSTELS projects was performed first on a Digital Equipment Corporation (DEC) VAX 8600 superminicomputer and later moved to a VAX 8810. The entire GRODY development took place on the VAX 8600. All four projects used the DEC Ada compiler and other tools in the Ada Compilation System (ACS). The other major DEC-supplied tools used on all projects included the symbolic debugger (SD) and the Code Management System (CMS). The first tool to be widely adopted on Ada projects outside of GRODY was the Language Sensitive Editor (LSE), introduced by developers on the FDAS project. The LSE is used primarily because of the ease with which a user can isolate and correct compilation errors without leaving the editor. The LSE is particularly important for development and testing because it is the focal point from which most other tools are invoked.

Package Helper and Lister, two other tools written by developers on the FDAS team, were also adapted by the projects that followed GRODY. Package Helper receives a user-specified

Ada package specification and then generates templates for the package body and the subunits to match the specification. Lister was used to generate listing files from the VAX. Table 2-2 lists the software development tools used in the implementation and testing phases.

2.4.3 PERSONNEL RESOURCES

Experienced Ada personnel continue to be a critically important but scarce resource in this environment.

A standard approach to development in this environment is to staff software projects with more junior personnel during the coding phase (Table 2-3). At this point in a project's life cycle, the more senior personnel have already generated a design and are available to assist the less-experienced personnel and provide guidance as they work on the system. This same approach is being used on the Ada projects, and it appears to be an effective way to handle an extremely scarce resource: software engineers with project experience in Ada.

The problem, then, is to staff projects with personnel having the right mix of software development expertise and application area expertise. Ada's greatly increased syntactic and semantic complexity, compared to FORTRAN, has exacerbated this problem. Managers have preferred to staff the Ada projects developed in this environment with software development personnel who have undergraduate or graduate degrees in computer science. Developing well-structured Ada software that uses the features of the language effectively requires a strong foundation in computer science or software engineering. On the other hand, satellite attitude determination and orbit propagation are skills that require a detailed knowledge of physics and mathematics. Few individuals with both sets of skills can be found; moreover, it is unlikely that developers already on staff with a background in one area can be trained to the same level of competence in the

Table 2-2. Software Development Tools Used for Implementation and Testing

	GRODY	GOADA	GOESIM	UARSTELS
ACS	yes	yes	yes	yes
CMS	yes	yes	yes	yes
LSE	no	yes	yes	yes
SD	yes	yes	yes	yes
Source Code Analyzer	no	no	no	no
DEC Test Manager	no	no	no	no
Package Helper	no	yes	yes	yes
Lister	no	yes	yes	yes

Table 2-3. Experience of Ada Developers at Beginning of Coding Phase

	GRODY	GOADA	GOESIM	UARSTELS
Number of personnel on project	7	7	5	3
Personnel with previous experience in application area	1	1	1	3
Personnel with previous project experience in Ada	0	1	1	1

other area. Thus, it remains an important part of task planning to attempt to retain a mix of personnel with differing areas of expertise and skill levels.

Figure 2-12 shows the distribution of total activity through system testing for the four simulation projects. The left graph in this figure shows this distribution for GRODY and GOESIM, and the right graph shows this distribution for GOADA and UARSTELS. The shapes for each pair are very similar. The more pronounced inverted-U shape in the left figure is probably due to the fact that the developers on these two projects were only available to work part-time but that during the implementation phase of these projects they were scheduled to work a larger percentage of their time on these projects. GOADA and UARSTELS, however, were predominantly staffed with full-time personnel, resulting in a flatter effort distribution.

Does staffing a project with part-time personnel affect either productivity or the quality of the system? Although there is not enough information available at this point to answer this question, developers on both GRODY and GOESIM reported that having to split their time between two or three different projects in different languages and even different hardware and operating system environments was overwhelming. In addition, the use of full-time personnel allows smaller teams, which limits the number of channels of communication between the team members, thus eliminating a well-known source of errors and misunderstandings in developing large systems (Brooks, 1975).

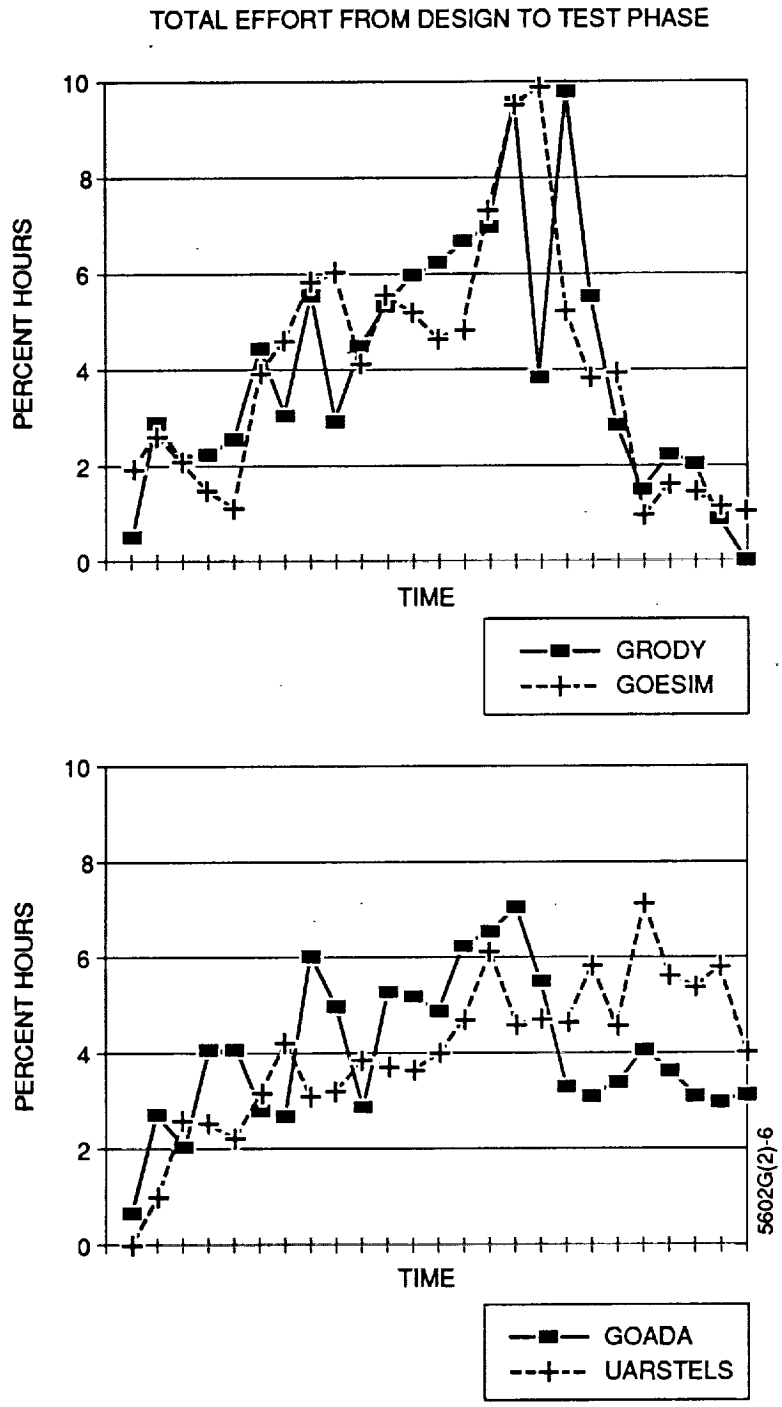


Figure 2-12. Effort Distribution for Ada Projects Through System Testing

SECTION 3 - TESTING IN ADA

The approaches to testing on the Ada projects developed in this environment have undergone major changes from those employed on FORTRAN systems.

In the life-cycle model used in the flight dynamics environment, software testing activities include unit testing, integration testing, build testing, system testing, and acceptance testing (Card et al., 1985). Unit, integration, and build testing are considered parts of the implementation phase of the life cycle, whereas system testing is considered part of the test phase (Wood and Edwards, 1986). Testing as a development process includes all testing activities performed by the project development team. It ranges from testing and debugging individual units (unit testing) through testing the behavior of the entire system as a "black box" (system testing). Postdevelopment acceptance testing is performed by the users to determine if the system satisfies the original requirements. Since this document is concerned with Ada software development as a process performed by the developers of these systems, the emphasis in this section will be on all testing activities up through system testing.

FORTRAN and Ada systems have fundamental structural and semantic differences that require that the entire issue of testing be reexamined in this environment. In general, the established testing procedures and methodologies used in the flight dynamics environment have been based on experience with FORTRAN. The approach to testing used on these Ada projects strongly suggests that the standard categories of testing recommended for software development in FORTRAN for this application area are not appropriate for Ada.

The recommended approach to testing for FORTRAN projects is to unit test individual components, and then integrate these tested components incrementally from the top down (Card et al., 1985). Section 3.2 will explain that our experience with these projects indicates that we are evolving towards replacing this technique with a bottom-up, incremental testing process that minimizes the distinction between unit and integration testing and instead concentrates on an iterative approach to developing incremental builds of increasing functionality.

3.1 UNIT TESTING

The Ada library package is now considered the basic unit for unit testing. Units are tested in combination with lower-level components invoked by the unit being tested.

Unit testing as defined in the flight dynamics environment owes its origins to system development with FORTRAN. For this language, unit testing is the testing of individual subroutines that are usually stored in separate source files. The term "unit" as used here is synonymous with the term "module," which is usually defined as a named and bounded contiguous sequence of statements (Yourdon and Constantine, 1978, p. 416). A module can be compiled independently from other modules (or units) and is callable from any other module within a system (Myers, 1979). This separation of source files has made it relatively easy in FORTRAN development environments to test individual modules separately from the rest of a system.

Although unit testing of FORTRAN systems in this environment is classified as a testing activity, it is considered part of the implementation phase of the life cycle, rather than part of either testing phase. As such, unit testing is carried out by the developer of the unit and is less formal than system or acceptance testing. It is usually performed

in the developer's working area and uses the developer's own test data, drivers, and stubs that are written for each unit as it is tested. Once the unit has been successfully tested, it is submitted to a controlled library (Card et al., 1985).

One of the discoveries made early during the first Ada projects was that the concept of testing is much less straightforward for systems developed in Ada. The first testing problem that the developers faced on FDAS and GRODY was determining what "unit testing" meant in Ada. This problem is complicated by several factors. First, Ada is a block-structured language that allows the textual nesting of program units within the declarative region of other program units. For example, procedures and functions that are nested in the declarative region of another subprogram, or are declared within the body of an enclosing package, cannot be unit tested using the traditional driver-and-stub approach of FORTRAN (Myers, 1979, p. 12). This is because scope rules of the language hide nested subprograms, which cannot be invoked directly from any driver outside of the component within which the procedure or function is nested. Although the subunit feature of the Ada language allows these nested units to be localized within a separate file (with a stub indicating the logical position of the separate unit within the declarative region of the encompassing subprogram), they are conceptually nested and still cannot be unit tested apart from the parent subprogram.

A second but related difficulty with the unit testing issue was the language's provision of additional program constructs, particularly the package and task. In a typical Ada system (including FDAS and GRODY), nearly every subprogram is declared within a generic or nongeneric package. Few, if any, are independent, compilable entities (called library subprograms). Although subprogram bodies are

usually implemented within separately compilable entities called subunits, they are still an integral part of the package within which their specifications are declared, and thus cannot be unit tested apart from the package to which they belong.

Unit testing was further complicated by Ada's scope and visibility rules, particularly as these are related to the package feature of the language. For example, the declarations of local variables or constants (objects) in a subprogram often refer to type declarations that are local to the package specification or package body within which the subprogram is declared. Other local object declarations in the subprogram may refer to types exported by one or more packages that are themselves imported by either the specification or the body of the package within which the subprogram is declared, or that are imported by the subprogram subunit itself. In addition, functions exported from a referenced package or the parent package may be used in initializing the variables or constants that are declared within the subprogram. Finally, executable code in the subprogram may reference or modify state information maintained within the body of the encompassing package.

The problem in unit testing subprograms in Ada, therefore, is that they are often dependent upon information specified in the body of the enclosing package, as well as information exported by any package imported by the enclosing package. Because of these considerations, all Ada projects since GRODY have adopted the convention that the Ada library package is the "unit" in unit testing. Although each subprogram exported by the package is individually tested, it is always tested as a component of the encompassing unit or package. This is especially true for packages that represent abstract state machines, which maintain state information within the body of the package. A subprogram that is exported by a

package (declared in a package specification) can be unit tested by developing a driver that invokes the subprogram name qualified with the name of the package that exports it, such as "Thrusters.Fire."

The UARSTELS package "Generic Star Catalog" provides an example of the kind of component typically developed in a satellite simulation system (Figure 3-1). This generic package is an abstract state machine with a state area that maintains a catalog of all stars above a specified brightness threshold. The state area also maintains several subsets of this catalog, each of which represents the collection of stars within the field of view (FOV) of each star tracker on the satellite. The package exports procedures "Initialize," "In FOV," and "Position of." To test the instance of the generic package as a unit, a driver was constructed that invoked each of the exported routines in sequence. The DEC symbolic debugger was used to examine changes to the state of the package body after each routine was invoked or to examine any parameter values by a call. Thus, the "Initialize" routine was called by the driver, and the symbolic debugger was used to examine the contents of the dynamic array created after "Initialize" read in the "Star Catalog" from a file. The function "Position Of" was tested by examining the right ascension and declination value returned for a specific star identification number.

The DEC symbolic debugger has been extensively used by all Ada projects in the flight dynamics environment. The importance of such a tool cannot be overemphasized for any program larger than a few hundred lines of code. The GRODY project was particularly dependent on the symbolic debugger because of its heavily nested system architecture (Godfrey and Brophy, 1989). The nesting of package specifications in package bodies precluded the use of driver routines that could invoke exported subprograms of these nested packages.

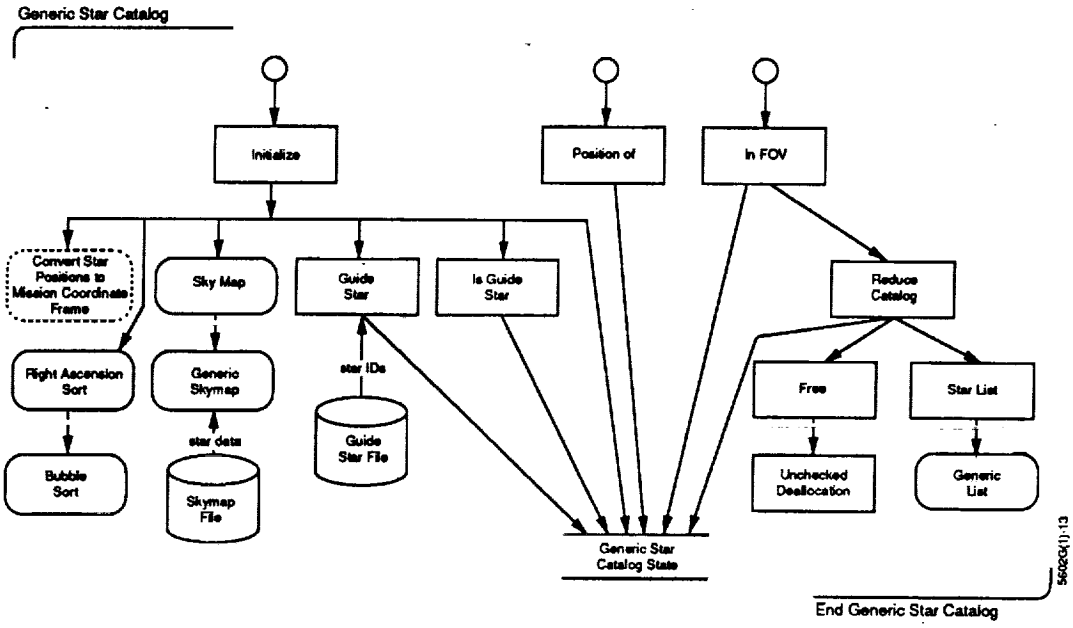


Figure 3-1. UARSTELS Generic Star Catalog

Furthermore, the specification of the encompassing package was itself often nested within the body of another package, and the specification of this encompassing package nested within the body of another package, and so on. Even though the subprogram bodies were subunits and thus were localized within individual files, from the tester's point of view the routine that was to be tested was nested up to seven levels below the driver routine that was developed to invoke it. The only way to test this lower-level routine was again to use the symbolic debugger. The debugger was also useful when executing a procedure or examining data hidden in a package body.

After Build 1 of FDAS was delivered, the team was joined by two developers who had project development experience in Ada. On the basis of their recommendation, all nested packages were placed into individual files. This heavy emphasis on library packages initiated by the FDAS project has been followed by all of the Ada projects after GRODY and has simplified testing on these projects. However, it became apparent on the GOADA task that having one Ada program unit per file is a necessary--but not sufficient--condition for effective unit development and testing. The GOADA team discovered this after they tried to use the parent Ada library of the GOADA system as the configured source library of the system. This library architecture resulted in frequent large-scale recompilation of much of the system, because low-level components were often modified during the early phases of implementation. This approach resulted in serious delays as developers waited for the system to be recompiled.

The GOADA team solved this problem by restructuring the Ada program libraries and organizing the individual program units to allow these units to be developed and tested with minimal recompilation overhead (Figure 3-2). The sublibraries were

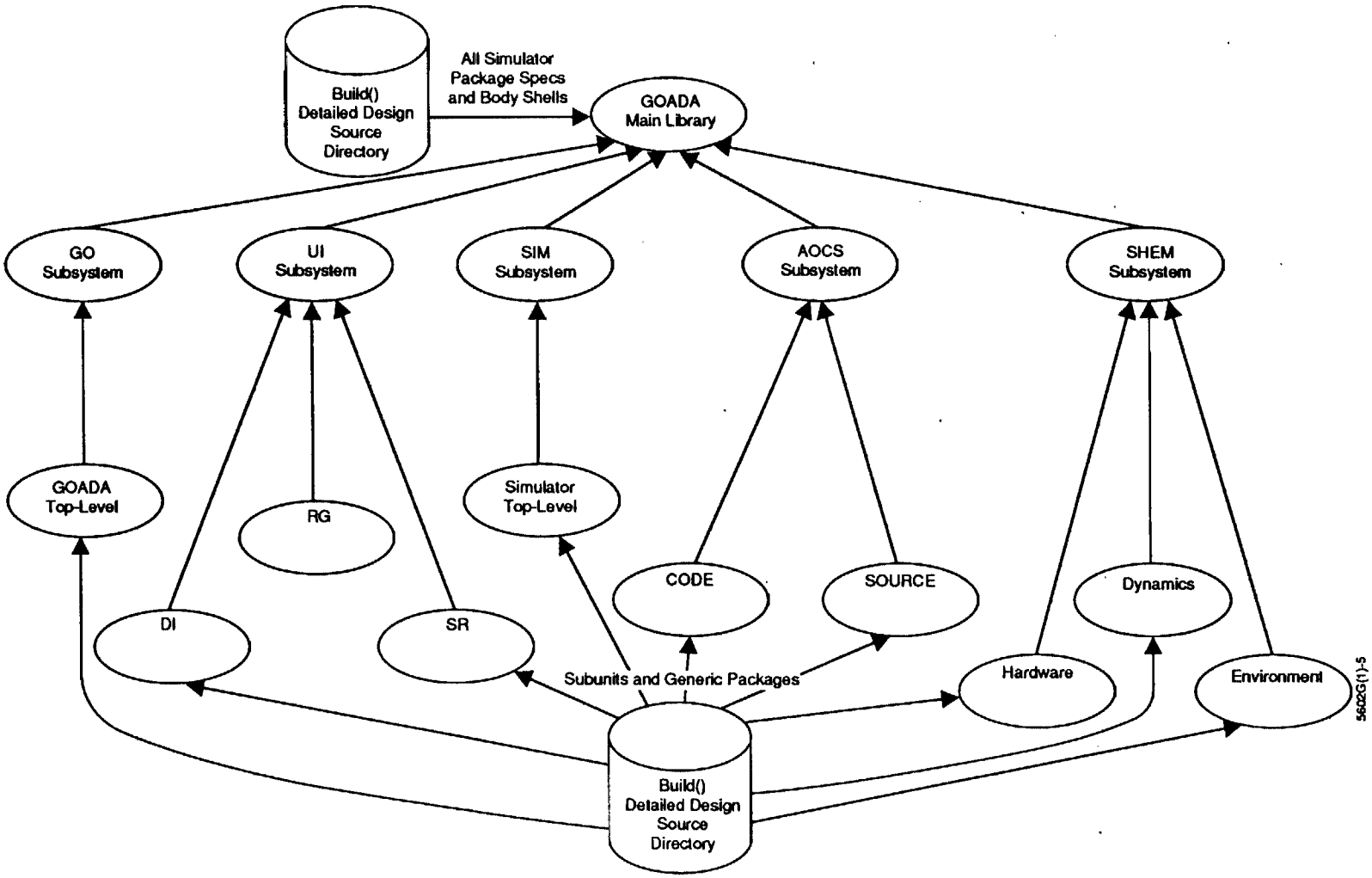


Figure 3-2. GOADA Ada Library Structure

used to control compilation dependencies at the subsystem level. Primarily, the team placed specifications for math and utility packages needed throughout the system within the parent library. They then set up several Ada sublibraries of this parent library to hold components associated with individual hardware subsystems. The individual developers' libraries were set up as sublibraries of these subsystem libraries. Package specifications that were common to two or more developers working on an individual subsystem were placed in that subsystem's library. Test versions of individual package specifications that were referenced by two or more subsystems were placed in the parent library. The development versions of these same package specifications were placed in a subsystem library, where they could be recompiled without affecting the other subsystems. All other packages (specifications or bodies) that were specific to individual subsystems were kept within the libraries of individual developers.

This type of Ada library structure was also used on FDAS, GOESIM, and UARSTELS. It is important to emphasize here that this mapping of a software system architecture to a program library architecture was not possible in the GRODY project, because of the nested architecture adopted for this system. These other projects, however, were able to utilize the separate compilation features of the Ada language in conjunction with the library management features of the DEC Ada software development environment to unit test components more effectively.

3.2 INCREMENTAL/INTEGRATION TESTING

For the Ada systems developed in this environment, bottom-up, incremental testing is replacing the top-down integration testing approach used on FORTRAN systems.

For FORTRAN systems developed in the flight dynamics environment over the last several years, integration testing is considered a part of the implementation phase of the life cycle (Wood and Edwards, 1986), rather than as a part of the test phase (McGarry et al., 1983). A primary reason for this is that a top-down incremental approach to integration testing is now being followed, with unit tested modules added to the system one or a few at a time and the combination of components tested after each of these additions to the system. This process may begin early in the implementation phase, with the first few unit-tested modules integrated well before most other modules within the system have even been coded.

Incremental testing has been used in one form or another on most projects developed in this environment. There are two major approaches to incremental testing: top-down and bottom-up. Top-down testing is the recommended approach to integration testing for the FORTRAN systems developed in the flight dynamics environment (Card et al., 1985). This approach starts with the top-level driver module, with stubs substituted for the first level of subordinate modules that are called by the driver. Once the driver is tested, the called stubs are replaced with the real modules that previously have been unit tested, with additional stubs constructed to substitute for the modules called by the newly added subordinate modules. This composite of modules is again tested, and any errors found can usually be attributed to the newly added subordinate modules, since the parent module or modules have already been tested. This process continues until the last stub has been replaced by an actual module.

One of the major changes these Ada projects have introduced into the software development process in this environment is to replace the top-down approach used in the FORTRAN projects

with a bottom-up, incremental approach to testing. Although the advantages of bottom-up testing as a methodology do not depend on the programming language used, the newness of the Ada language allowed the exploration of different approaches to software development in Ada, including testing. FDAS was the first project to introduce this technique. All subsequent projects have adopted this bottom-up approach to a greater or lesser degree.

One of the main problems with the top-down approach is the amount of effort that must be expended to develop the stubs. Writing stubs is far more difficult than many developers realize (Myers, 1979, p. 94). For the module being tested to perform properly, the modules being called must return a meaningful result. This may require the stub to simulate the actual module with such fidelity that the stub may be as complex as the module.

Although top-down incremental testing is not used on the more recent Ada projects in this environment, top-down design is. For the most part, the main program and the top-level package specifications that compose major subsystems are written and compiled first, followed by the next layer of package specifications that compose lower-level subsystems, and so forth. This coding of package specifications is one of the primary activities associated with developing the preliminary design, and has become a standard practice since it was introduced by the GOADA project. During detailed design, the Ada package bodies and the compilable PDL for the associated subunits that implement the specifications previously compiled are written, from the higher levels of the system architecture to the lower levels.

During implementation, the terminal or leaf modules of the system are coded and unit tested first. These are the

subunits associated with the lowest-level packages of the system, which typically are mathematical utilities and file input-output operations. Subunits associated with the next-higher level of the system are coded and "unit tested" next. However, here the term "unit testing" also includes integration testing, since the unit is tested in combination with the lower-level package or packages invoked by the unit, rather than with stubs. This essentially simultaneous approach to unit and integration testing suggests that these terms do not adequately describe what the testing process encompasses. The term "incremental testing" is more descriptive of this approach to testing.

The Ada compiler provides type checking of actual parameters against formal parameters. That is, the compiler rigorously verifies the interfaces between components during the early phases of implementation, a process that on a FORTRAN project can only be approximated with extensive manual testing, rigorous code inspections, or use of software tools that perform this type of checking for FORTRAN systems. Any changes that are made to subprogram specifications or package specifications are flagged by the link command, and any units that invoke the modified subprogram must be updated to reflect these changes.

Bottom-up incremental testing initiates the integration testing process at the bottom of the system hierarchy--that is, at the terminal or leaf modules of the system. In this case, drivers rather than stubs are constructed. Drivers are easier to write than stubs, because they only have to provide values for the actual parameters that match the formal parameters of the invoked module and, perhaps, print out the values returned from the module (Myers, 1979, p. 98). Unlike stubs, drivers do not have to compute or return any values.

Once the terminal modules have been tested, the driver used to test them can be replaced by the module that actually invokes them. Another driver is constructed that invokes the new module that has been added to the system, and the new collection of components is linked and executed. Any errors found at this point are probably due to the added module, since the lower-level modules have already been tested. It is important to emphasize here that the lower-level modules have been tested already as a coherent collection of components. Furthermore, as each higher-level module in the calling hierarchy is integrated into the system, the lower-level modules are being retested, providing further verification of the correctness of the system at that point in the testing.

The GOADA project team decided early in implementation to use bottom-up, incremental testing. However, it made this decision after developing a plan that called for all subsystems to be developed in parallel, as is typical for flight dynamics FORTRAN systems. GOADA did not change this plan to accommodate an incremental approach to testing. As a result, numerous problems arose during integration and system testing that would have been resolved earlier in development if a full bottom-up, incremental approach to development had been used. One of the lessons of this experience is that an incremental development effort requires careful planning to ensure that components are developed in the sequence in which they will be needed by higher-level components within the system.

One other major factor argues in favor of the bottom-up incremental approach to development: reuse. In the flight dynamics environment, the components most likely to be used from one system to the next lie at the bottom of the system architecture. These are the terminal units of a program that perform the bulk of the computational and data

manipulation capabilities of the system. In FORTRAN, these reusable software routines are those that perform numerical analysis operations. In Ada, they include not only numerical analysis components, but also those that implement complex data structures, sort and search utilities, and numerous other tools (Booch, 1987). The "Generic Utilities" subsystem developed on GRODY was used with little modification on all subsequent Ada projects within this area. Similarly, the computationally intensive "Generic Ephemeris Model" developed on GRODY has also been used on all subsequent systems, as mentioned in Section 2.1.3. Finally, commercially licensed components that implement complex data structures such as queues, rings, and linked lists, as well as various character, string, and numeric utilities, have been incorporated easily into all of these Ada projects.

With bottom-up incremental development, these lower-level reused components can be incorporated into a system early in the implementation phase of the life cycle. Computationally intensive simulation models can be written and tested well before other parts of the system have been fully defined, since the reused mathematical and utility components have already been rigorously tested through prior use on several other systems. This would not be possible with a top-down approach to development, since the mathematical and other utilities would not be incorporated into the system for testing until late in implementation.

3.3 BUILD/SYSTEM TESTING

Build and system testing is a language-independent activity. However, the greater modularity incorporated in the more recent Ada projects may be a major factor in the decreased effort required for system testing on these systems.

Build tests and system tests are similar. The major difference is that build tests are performed during the implementation phase of the life cycle, and system tests are performed during the test phase (Card et. al., 1985). The development team performs both kinds of tests and evaluates their results. The planning for both is formal, with separate build- and system-test plans developed by the project team. The tests are based on the design and the requirements of the system or build, and are meant to test the functionality of the system or the completed build as an integrated unit.

A previous study of the system-test phase of the GRODY project reported that the system-test plan developed for GRODY was essentially the same as the system-test plan for GROSS, the FORTRAN version of the Ada project (Seigle et al., 1988). This is not surprising, since during system test the developers are running functional tests on an executable binary image, rather than working with Ada source code. Developers on each of the subsequent dynamics and telemetry simulators have also reported that the types of tests used in system and build testing were essentially language independent.

One of the major problems that the GRODY project faced during the system-test phase was the lack of personnel with a sufficiently detailed understanding of the spacecraft to analyze test results (Seigle et al., 1988). A similar situation existed on the GOADA project. The task leader was the only task member with flight dynamics experience at the beginning of the system-test phase. This situation improved when a developer with expertise in the application joined the project and when a member of the original GOES-I Dynamics Simulator in FORTRAN (GOFOR) project also assisted part-time in system testing.

The GOESIM team also had no development team members who had experience in the application area when system testing commenced. However, they had planned and were able to obtain the assistance of some analysts to examine the output from the system tests they ran. Two of the four task members on the UARSTELS project had experience in either attitude ground support systems or telemetry simulation systems.

Just as the architecture of the system affects all the other phases of the life cycle, it can affect the ease or difficulty with which an error can be isolated and corrected. The GRODY team reported that the heavily nested architecture used in their system required a great deal of recompilation each time a change was made to almost any part of the system (Seigle et al., 1988). Depending on how heavily loaded the computer was at the time the change was made, each recompilation could take many hours to complete, effectively preventing other testers from making any other corrections to the system.

3.4 TESTING TOOLS

Testing tools other than the symbolic debugger were introduced late in this environment. Future research will be needed to determine their effectiveness.

The symbolic debugger, which is heavily used during coding, is also used during build- and system-testing to locate errors. The source code analyzer and the performance and coverage analyzer are two of the major modification and testing tools used for the more recent projects, especially the Extreme Ultraviolet Explorer Satellite (EUVE) Dynamics Simulator (EUVEDSIM) project. Some developers are also looking at the DEC Test Manager, which will be used first by EUVEDSIM. Future Ada projects developed on this type of hardware and operating system environment will probably use all of these tools as the experience base grows.

SECTION 4 - PROJECT CHARACTERISTICS

4.1 SOFTWARE SIZE METRICS

Productivity and size measures of projects will need to take into account the degree to which Ada generics are used in a particular system and the complexity of that system.

The ASAP software metric tool mentioned in Section 2.2 produces a detailed breakdown of line counts by component and by project. ASAP was run on each project at the end of the system testing phase, and the results are shown in Table 4-1. Of the two dynamics simulators, GRODY and GOADA, GOADA is larger (17.4 percent in source lines of code, or SLOC) because it has greater complexity. For example, GOADA provided a batch operation mode in addition to GRODY'S interactive mode. GOADA has more failure modes, three times as many input screen displays, over twice as many ground commands, a thruster history report generation capability, and so forth.

In general, there has been a correlation between the size of the source code of flight dynamic systems in thousands of SLOC (KSLOC) and the complexity of these systems. In an effort to achieve a quantitative measure of complexity, Boland et al. (1989) assigned relative values to spacecraft components or capabilities, and showed that, for attitude ground support systems at GSFC written in FORTRAN, there is a strong correlation between spacecraft complexity and size of the source code in KSLOC. However, with the use of Ada, this correlation holds only if the features of the language are used in a similar way or to a similar degree among the systems being compared. For example, GOESIM is considerably larger (35.3 percent in SLOC) than UARSTELS, and yet the UARS satellite has about 10 percent greater complexity than GOES-I (Boland et. al., 1989). Hence the software system

Table 4-1. Ada Software Size Measures at End of System Testing

	GRODY	GOADA	GOESIM	UARSTELS
SLOC	125,991	147,876	87,535	64,720
Lines of code plus comments (LOC&C)	93,328	114,445	67,872	52,834
Lines of code (LOC)	57,661	73,229	42,326	36,001
Comments	35,667	41,216	25,546	16,833
Blank lines	32,664	33,431	19,663	11,836
Instructions[1]	22,586	26,352	16,343	13,313
Declarations and context clauses	5,959	10,090	6,509	7,169
Statements	15,109	16,262	9,834	6,144
Number of program units	482	671	526	431
Average SLOC/unit	261	220	166	150
Average LOC/unit	120	109	80	84
Average number of instructions/unit	47	39	31	30

[1]The Ada Language Reference manual defines "statements" as an action to be performed. The authors here have chosen to use the term "instruction" to include statements, declarations, clauses, and programs.

would be required to support more capabilities. The difference in source code size between these two projects is due primarily to the degree to which the UARSTELS project used generic packages in place of multiple copies of similar components. Thus, both productivity measures and size measures of projects will need to take into account the degree to which Ada generics are used in a particular system.

The source line counts in Table 4-1 also should be viewed with caution for other reasons. The definition of line count being used results in large differences of the particular counts. As shown in Table 4-2, for the Ada simulation systems developed in this environment, there are about five times as many SLOC as there are instructions.

Table 4-2. Ratio of SLOC to Total Instructions

	GRODY	GOADA	GOESIM	UARSTELS
$\frac{\text{SLOC}}{\text{total instructions}}$	5.6	5.6	5.4	4.9

The line counts are greatly affected by the particular coding style adopted by the project or organization that is building the system. The major factor contributing to the high ratio of SLOC to instructions in the flight dynamics environment is the style of code recommended by the Ada Style Guide (Seidewitz et al., 1987). Developers are encouraged to use readable English names and phrases in naming objects and subprograms, and a liberal use of white space is recommended to highlight logically related blocks of code and to separate type, variable, and constant declarations for readability (Table 4-3). As a result, there is a consensus among the Ada development personnel that the Ada code on these projects is much easier to read and understand than any of the FORTRAN code that was reused from previous FORTRAN simulation systems.

The free-format capability of the language, the recommended code indentation and code layout from the Ada Style Guide, and the use of long names result in Ada instructions that on the average span several lines of code (Table 4-4).

Table 4-3. Line Count Profiles at End of System Testing

		GRODY	GOADA	GOESIM	UARSTELS
Blank lines	(%)	26.0	22.6	22.5	18.2
Comment lines	(%)	27.8	27.9	29.2	25.9
LOC	(%)	46.2	49.5	48.3	55.9
		100.0	100.0	100.0	100.0

Table 4-4. Average Number of SLOC per Instruction

GRODY	GOADA	GOESIM	UARSTELS
2.58	2.78	2.59	2.72

4.2 REUSE

Software reuse was substantial on the second Ada project. Software reuse has remained steady on the projects under study; however, software is now being designed for greater reuse on other projects.

Figure 4-1 shows the percentages of components for the four projects that are reused verbatim, reused with slight modification (no more than 25 percent of the original component changed), reused with extensive modifications (greater than 25 percent changed), and combined with new components. Reuse on GRODY was limited to imported FORTRAN procedures obtained from previous dynamics simulators. The estimated amount of reuse at the time of the CDR for the three follow-on projects to GRODY ranged from 30 percent for GOESIM to 50 percent for UARSTELS. The actual amount of reuse proved to be lower than estimated during implementation and system testing, with the three projects leveling out so that the amount of reuse is approximately the same across all three projects (Table 4-5).

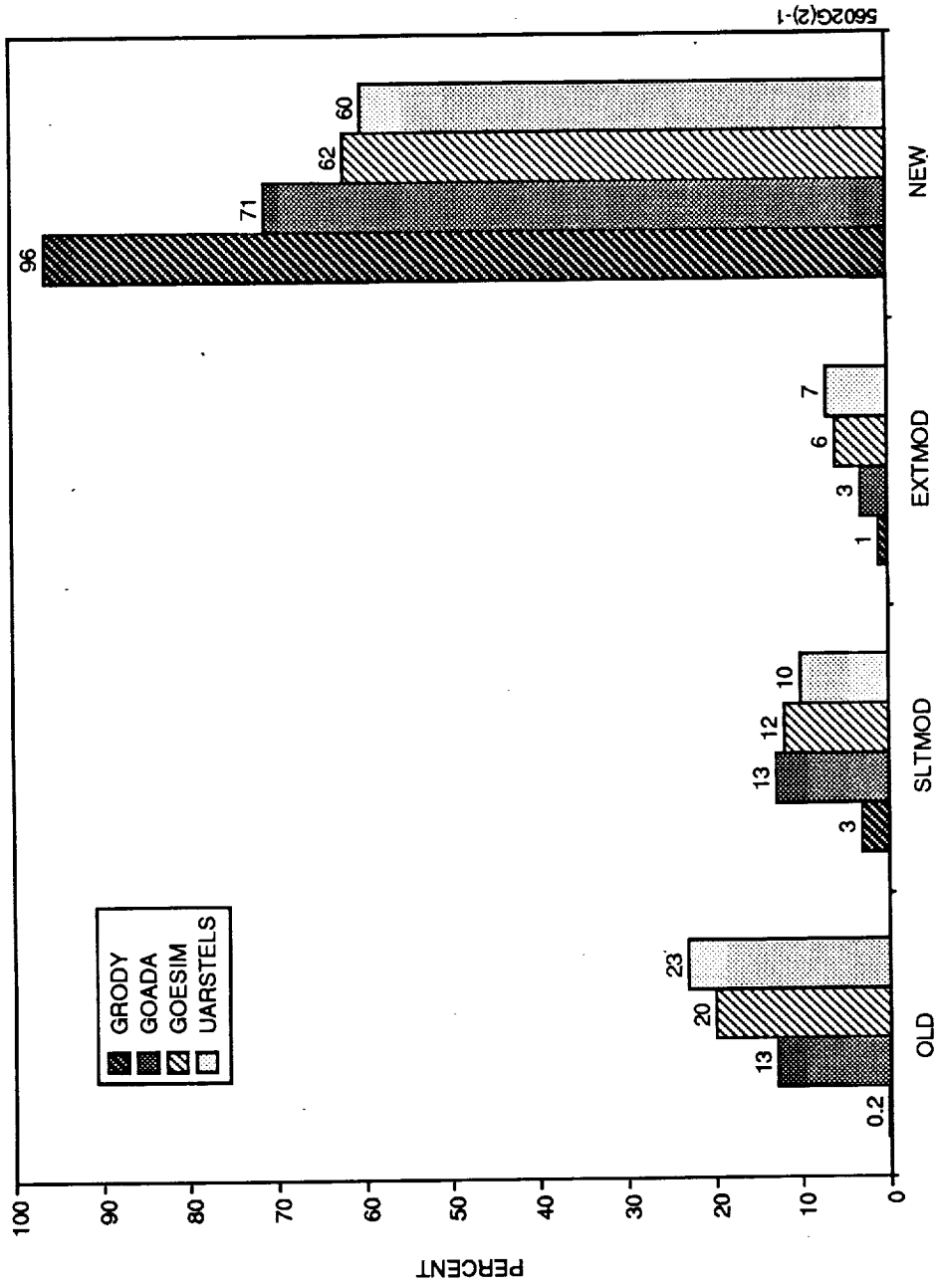


Figure 4-1. Component Reuse

Table 4-5. Reuse 1 Levels at the End of System Testing

GRODY	GOADA	GOESIM	UARSTELS
3 %	26 %	32 %	33 %

1 reuse = (no. of verbatim components + no. of slightly modified components)/total)

Reuse on these projects has for the most part been limited to reuse of GRODY components that were of general use in these types of simulators, such as the mathematical utilities and the ephemeris packages. Since the three subsequent projects were developed on overlapping schedules, one project had little opportunity to provide components that could be reused by the other projects. The major exception to this was the UARSTELS project: its components were designed and developed so that the subsequent telemetry simulator project for the EUVE (EUVETELS) required the minimum amount of new code.

4.3 DEVELOPMENT EFFORT THROUGH SYSTEM TESTING

There is a noticeable trend indicating a change in the life cycle: slight increases in the design and implementation phases, and a decrease in the system test phase.

Table 4-6 compares the predicted effort for development through system testing to the actual effort. Managers planned the projects and predicted effort based primarily on the recommended approach outlined in the Managers Handbook (Agresti et al., 1984). However, because this approach is based on FORTRAN, the managers said they subtracted approximately 5 to 10 percent of the estimated effort recommended for implementation and system testing and shifted it into predicted design effort. The actual total effort (through system test) for all four projects is within 10 percent of the predicted effort.

Table 4-6. Predicted Versus Actual Total Staff Hours Through System Testing

	GRODY	GOADA	GOESIM	UARSTELS
Predicted Effort Hours	22,700	22,750	12,070	10,450
Actual Effort Hours	21,993	24,096	11,690	9,521
Difference	-3 %	+6 %	-3 %	-9 %

Figure 4-2 shows the distribution of effort by life-cycle phase. There was no immediate significant change to the effort expended during the life-cycle phases; however, the Ada life cycle is changing slightly with each project and is now slightly different than that expected for a FORTRAN project. The GRODY projects shows no additional time needed for design than that needed for its FORTRAN counterpart; however, all subsequent Ada projects required additional time during the design phase. This appears to support the developer's statements that the design of GRODY was incomplete at the time of the CDR.

Figure 4-2 also shows a decreasing trend in the percentage of effort required during the system test phase when compared to FORTRAN and a greater percentage of effort required during the implementation phase. This may mean that the Ada compiler is capturing interface errors that are normally not found until system testing in FORTRAN. Since these Ada interface errors are found in implementation, they are also being corrected in implementation instead of during system testing. Thus, the lower percentage of effort expended during the test phase may simply reflect a shift of this effort to the previous phase (implementation).

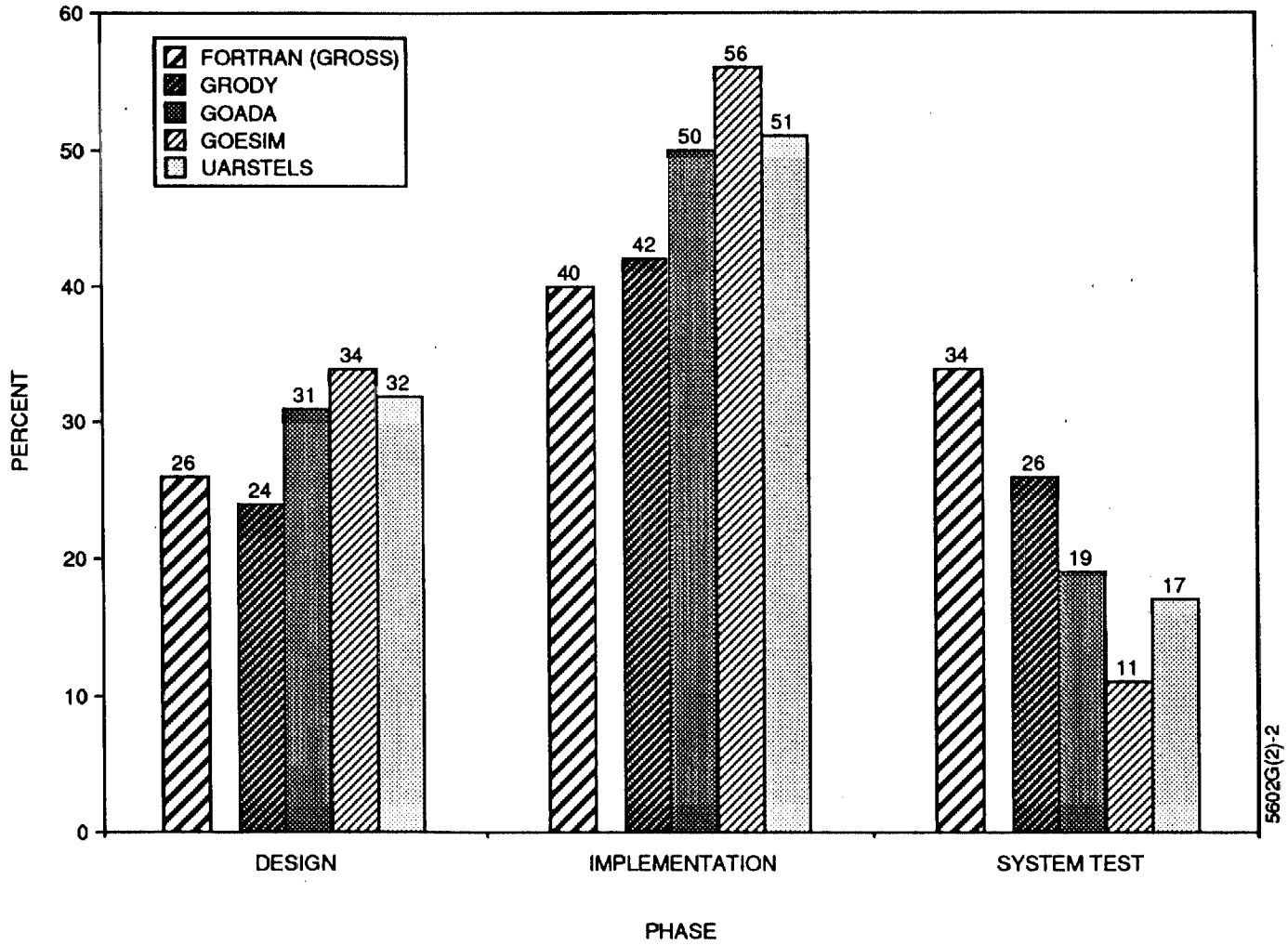


Figure 4-2. Distribution of Effort by Life-Cycle Phase

4.4 PRODUCTIVITY

Productivity in terms of effort/LOC has improved over time. However, caution must be exercised, since LOC does not always represent the true size of the system in terms of functionality.

The productivity of all four projects in terms of line counts per staff day is shown in Table 4-7. Productivity has improved slightly; however, caution must be used in using line counts as a measure of productivity. For example, GOESIM shows a slightly higher productivity than UARSTELS in terms of SLOC/staff day. On the other hand, as pointed out in Section 4.1, although GOESIM was 35.3 percent greater in size than UARSTELS, the UARS satellite has about 10 percent greater complexity than the GOES satellite. Since UARSTELS cost nearly 20 percent less to develop than GOESIM, and it had more capabilities supporting the more complex satellite, real productivity would actually be higher on UARSTELS. Further research would be required to determine a measure of productivity that takes into account other factors, such as software system functionality, and the extent to which the generic feature of Ada is used in a system.

Table 4-7. Productivity Measures Through System Testing

	GRODY	GOADA	GOESIM	UARSTELS
SLOC/staff day	55.1	49.1	59.9	54.4
LOC&C/staff day	40.8	38.0	46.4	44.4
LOC/staff day	25.2	24.3	30.0	30.2
Instructions per staff day	9.9	8.7	11.2	11.2
Declarative	2.6	3.3	4.5	6.0
Executable	6.6	5.4	6.7	5.2
Components/staff day	0.2	0.2	0.4	0.36

4.5 CHANGE CHARACTERISTICS

Error rates have decreased from first- to third-generation Ada projects. However, the cause of this decrease is uncertain and requires further study.

On the average, most changes to any of the projects were isolated and completed within 1 day. Ninety-two percent of the changes took less than 1 day to isolate, and 86.4 percent of the changes took less than 1 day to complete. When the effort to isolate change is contrasted with the effort to complete change, no significant difference is found between the two distributions in terms of projects or phases. Yet, even though the distributions are similar, the effort to isolate change does not appear to predetermine the effort to complete change.

Figure 4-3 shows the profiles of changes per thousand instructions, classified according to whether one, two to four, or five or more components were affected by the change. The most obvious difference among these projects here is that GRODY had nearly twice as many errors that affected only one component than did the other three Ada simulation systems. Since the heavily nested architecture of GRODY resulted in fewer and larger components per thousand instructions than on the later three Ada projects, there may have been a tendency in GRODY for errors to affect only one component. A more likely explanation could be that errors tend to cross component boundaries and affect more than one component with the smaller, more numerous components in the more recent systems.

Table 4-8 shows the error density for the four Ada projects. The error density for the two dynamics simulators is nearly twice that of the two telemetry simulators; however, dynamics simulators are approximately twice the size of telemetry simulators and are considerably more complex systems. The

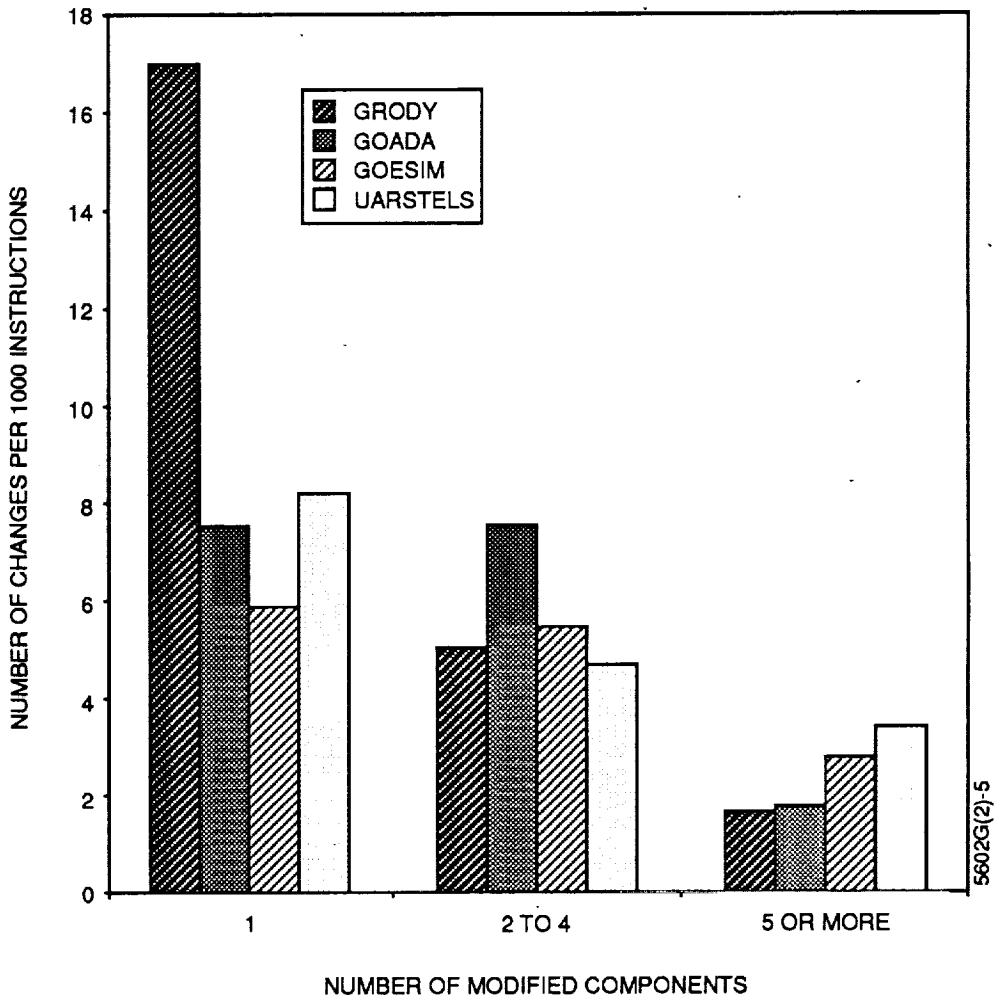


Figure 4-3. Changes That Affect Different Numbers of Components

Table 4-8. Error and Change Rates Through System Testing

	GRODY	GOADA	GOESIM	UARSTELS
No. errors per 1000 instructions	10.1	10.9	5.4	6.9
No. changes per 1000 instructions	23.9	27.9	12.5	16.5

greater error rate thus may be primarily due to the greater size and complexity of these systems. It is also possible that the lower error rate on the two telemetry simulator systems is at least partly due to an increased experience with the language. As seen in Figure 4-4, the error and change rates were not substantially different from implementation to system test phase.

Sources of error are classified by the SEL as originating from requirements, from functional specifications, from design, from code, or from previous changes (Figure 4-5). Errors are primarily coding errors, and the proportion of errors due to coding is increasing across these projects, whereas the number of errors due to design is decreasing. Since design errors are generally more expensive to fix than coding errors, this probably indicates an improvement in the development process.

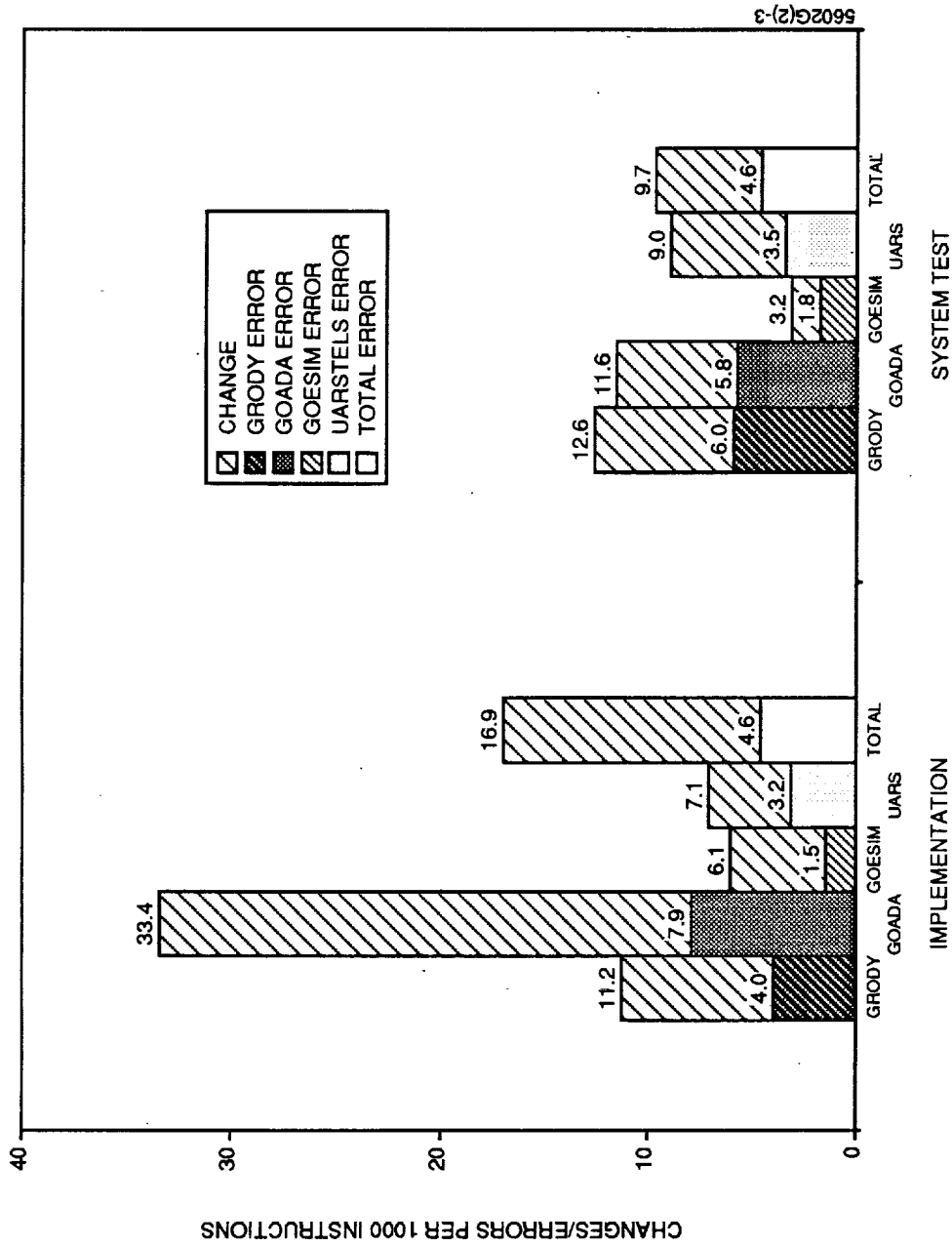


Figure 4-4. Comparison of Error and Change Rates by Phase

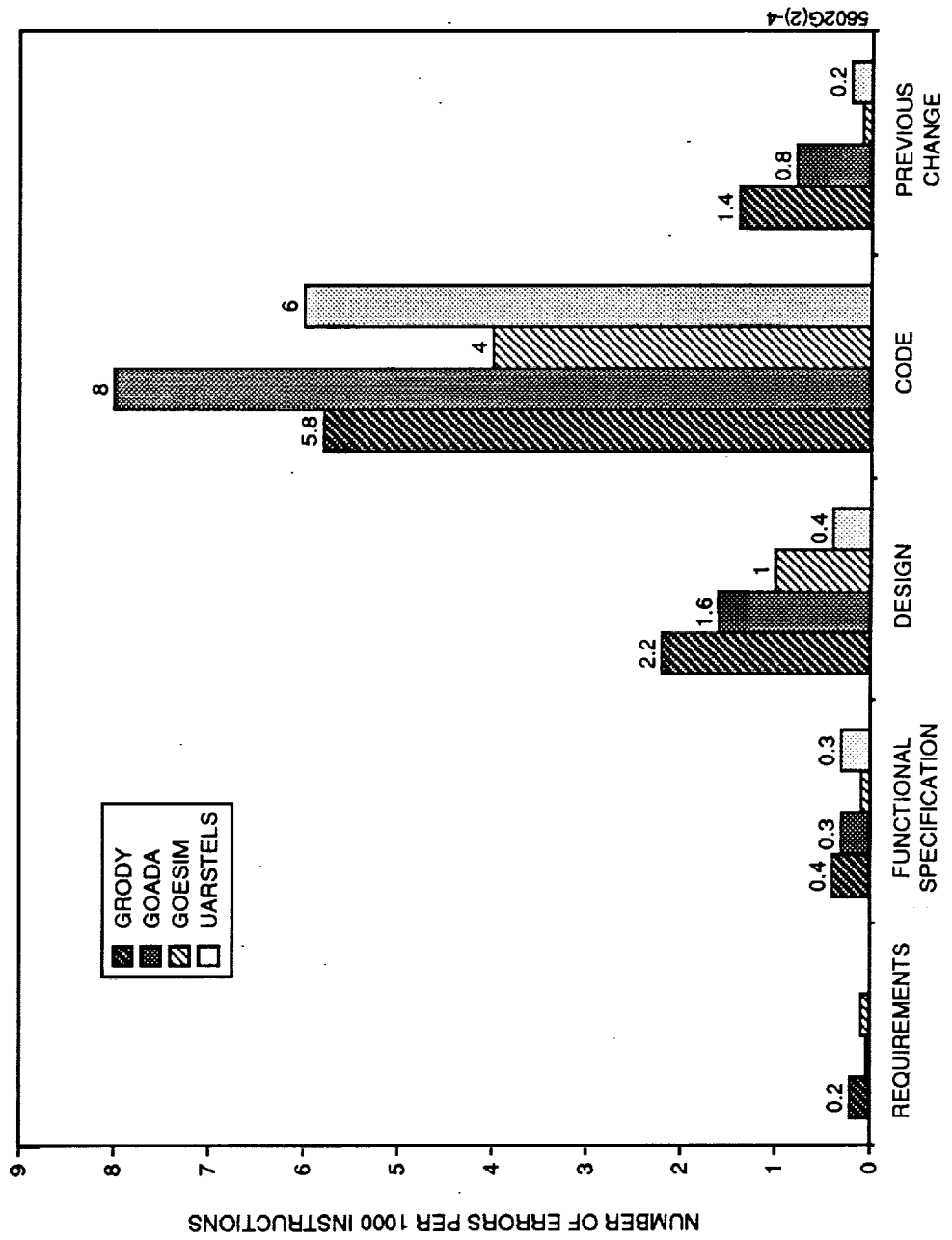


Figure 4-5. Error Source

SECTION 5 - SUMMARY AND RECOMMENDATIONS

As Quimby and Esker (1988) pointed out in their analysis of the design phase, the transition from developing software systems in FORTRAN to developing systems in Ada is an evolutionary process. This learning process is more limited when these projects are being developed in parallel, since the opportunity to pass lessons learned from one project to another arises primarily through the movement of personnel. Since only three of the developers of these Ada simulation systems had worked on a previous Ada project, the level of Ada expertise developed in this environment is less than was expected at the beginning of this study. The development of several projects entirely in a time-ordered sequence, with some significant fraction of the development personnel moving from one completed project to the next, is not likely to occur.

The design diagram notation introduced by GRODY has been refined and improved on subsequent projects, and is viewed as helpful for implementing the design and for documenting the design in the form of a system description after a project is completed. However, developing and maintaining these diagrams is a labor-intensive activity, and is made more so by the limited availability of graphics-based personal computers. **CASE tools that support object-oriented design methodologies would help automate this process, and should be evaluated for use in this environment.**

An implicit assumption of Quimby and Esker (1988) was that the use of a compiled design would be beneficial to the overall development of an Ada project. However, at present there is insufficient evidence to support or refute this assumption. The contribution, if any, of a compiled design to the development process may depend upon a complex interaction of other variables, including the availability of

design resources (CASE tools), the availability of computer resources for development (hardware and software), the life-cycle model used on the project, and so on. Further experience and research will be required to determine the effectiveness of a compiled design, or the circumstances in which the use of a compiled design might prove effective.

Generally, personnel rated the DEC interactive development environment very highly for automating a major portion of their development activities that would otherwise have to be handled by time-consuming desk work. The usefulness of these tools was hampered by an overloading of the host computer and limited availability of terminals. Adding more terminals should be considered only if additional CPU and disk storage capability is made available to support the increased load more terminals would impose on the system.

Given that the GRODY project was the first Ada simulation system developed in this environment, greater progress in understanding how to engineer components to be reusable might have occurred if GRODY code was not reused. The schedule pressures associated with the development of production software systems do not result in an environment conducive to the design and development of high-quality, verbatim-reusable software. Consideration should be given to the idea of developing verbatim-reusable software independent of any particular mission, with the idea that all future missions in which Ada will be used as the application language could draw on this pool of reusable components.

Although the EUVETELS project was not covered in this study, preliminary evidence from both this project and UARSTELS strongly suggests that a combination of factors could greatly increase the level of verbatim reuse in this

environment. These factors include the proper use of Ada generics, the development of reusable requirements specifications, and the deliberate engineering of software components to be reusable on multiple missions.

The manner in which the Ada units and subsystems have been tested on these projects has undergone considerable change when compared to FORTRAN projects. On the basis of this experience, serious consideration should be given to replacing the standard approach to testing in this environment with a bottom-up, incremental testing process that eliminates the distinction between unit and integration testing and instead concentrates on an iterative approach to developing incremental builds of increasing functionality.

GLOSSARY

ACS	Ada Compilation System
AGSS	Attitude Ground Support System
ASAP	Ada Static Analysis Program
ATR	assistant technical representative
blank lines	Lines that contain only a carriage return (<CR>)
CASE	Computer-Aided Software Engineering
CDR	critical design review
CMS	Code Management System
comments	Lines that begin with comment token, "--"
CPU	central processing unit
CSC	Computer Sciences Corporation
CSS	coarse Sun sensor
DEC	Digital Equipment Corporation
declaration	Ada instruction that declares an identifier, establishes a scope, or places the unit in some visibility context
EMS	Electronic Mail System
ephemeris	time-tagged sequence of positions that represents the orbit of a satellite
EUVE	Extreme Ultraviolet Explorer Satellite
EUVEDSIM	EUVE Dynamics Simulator
EUVETELS	EUVE Telemetry Simulator
FDAS	Flight Dynamics Analysis System
FHST	fixed-head star tracker
FOV	field of view
GOADA	GOES-I Dynamics Simulator
GOES-I	Geostationary Operational Environmental Satellite-I
GOESIM	GOES-I Telemetry Simulator
GRO	Gamma Ray Observatory
GRODY	GRO Dynamics Simulator
GSFC	Goddard Space Flight Center

instructions	sum of Ada declarations and Ada statements
KSLOC	thousands of SLOC
LOC	lines of code
LOC&C	lines of code plus comments
LSE	Language Sensitive Editor
NASA	National Aeronautics and Space Administration
OBC	onboard computer
P&CA	performance and coverage analyzer
PDL	program design language
SCA	source code analyzer
SD	symbolic debugger
SEL	Software Engineering Laboratory
SLOC	source lines of code
statement	An Ada instruction that defines an action to be performed. Includes abort statement, block statement, accept, array statement, assignment, case statement, code statement, delay, entry call, exit, goto, if statement, loop statement, procedure call, raise, return and select statement.
UARS	Upper Atmosphere Research Satellite
UARSTELS	UARS Telemetry Simulator

REFERENCES

Agre, A., et al., Geostationary Operational Environmental Satellite-I (GOES-I) Attitude Dynamics Simulator in Ada (GOADA) System Description, Computer Sciences Corporation, February 1989

Agresti, W. W., New Paradigms for Software Development. Washington, DC: IEEE Computer Society Press, 1986

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

Barnes, J. G. P., Programming in Ada, Third Edition. Wokingham, England: Addison-Wesley, 1989

Boland, D., et al., CSC/TM-89/6031, A Study On Size and Reuse Trends in Attitude Ground Support Systems (AGSSs) Developed for the Flight Dynamics Division (FDD) 1976 - 1988, Computer Sciences Corporation, February 1989

Booch, G., Software Components with Ada. Menlo Park, CA: Benjamin/Cummings Publishing Company, 1987

Booth, E., CSC/SD-89/6025, Upper Atmosphere Research Satellite (UARS) Telemetry Simulator (UARSTELS) System Description, Computer Sciences Corporation, January 1989

Brooks, F. P., The Mythical Man-Month--Essays on Software Engineering. Reading, Massachusetts: Addison-Wesley, 1975

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

DeRemer, F., and H. H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Transactions on Software Engineering, June 1976, Vol. SE-2, No. 2

Doubleday, D. L., "ASAP: An Ada Static Source Code Analyzer Program," Master's Thesis, Department of Computer Science, University of Maryland, College Park, MD, August 1987

Gehani, N., Ada: Concurrent Programming. Englewood Cliffs, NJ: Prentice-Hall, 1984

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

Lo, P., et al., CSC/SD-87/6034, Gamma Ray Observatory (GRO) Dynamics Simulator in Ada (GRODY) System Description, Computer Sciences Corporation, December 1987

McGarry, F. E., et al., SEL-81-205, Recommended Approach to Software Development, Software Engineering Laboratory, April 1983

Myers, G. J., The Art of Software Testing. New York: Wiley-Interscience, 1979

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

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

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

Stark, M., and E. Booth, "Using Ada to Maximize Verbatim Software Reuse," Proceedings of the TRI Ada '89 Conference, October 1989

Wirth, N., Algorithms + Data Structures = Programs. Englewood Cliffs, NJ: Prentice Hall, 1976

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

Yourdon, E., and L. Constantine, Structured Design. New York: Yourdon Press, 1978

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-82-806, Annotated Bibliography of Software Engineering Laboratory Literature, M. Buhler and J. Valett, November 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, L. Esker, 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-88-004, Proceeding of the Thirteenth Annual Software Engineering Workshop, November 1988

SEL-89-001, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, M. So et al., May 1989

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

SEL-89-003, Software Management Environment (SME) Concepts and Architecture, W. Decker and J. Valett, August 1989

SEL-89-004, Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, Lessons Learned in the Transition to Ada from FORTRAN at NASA/Goddard, C. Brophy, November 1989

SEL-89-006, Collected Software Engineering Papers: Volume VII, November 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

[7]Basili, V. R., Maintenance = Reuse-Oriented Software Development, University of Maryland, Technical Report TR-2244, May 1989

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

[7]Basili, V. R., Software Development: A Paradigm for the Future, University of Maryland, Technical Report TR-2263, June 1989

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

[7]Basili, V. R., and H. D. Rombach, Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment, University of Maryland, Technical Report TR-2158, December 1988

[2]Basili, V. R., R. W. Selby, Jr., 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, Jr., "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, Jr., "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

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," Proceedings of the Sixth Washington Ada Symposium (WADAS), June 1989

[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

[7]Rombach, H. D., and B. T. Ulery, Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL, University of Maryland, Technical Report TR-2252, May 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

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," Proceedings of TRI-Ada 1989, October 1989

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

[7]Sunazuka. T., and V. R. Basili, Integrating Automated Support for a Software Management Cycle Into the TAME System, University of Maryland, Technical Report TR-2289, July 1989

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.

[7]This article also appears in SEL-89-006, Collected Software Engineering Papers: Volume VII, November 1989.