

AD-A228 819

DTIC FILE COPY

Q9-Baseline Ada Library
Technical Report
Reusability Guidelines

DTIC
ELECTE
NOV 14 1990
S B D

STARS-QC-00340/001/01
5 May 1989

DISTRIBUTION LIMITED TO DOD AND DOD
CONTRACTORS ONLY Administrative (5 May 1989)

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE 5 May 1990	3. REPORT TYPE AND DATES COVERED Final	
4. TITLE AND SUBTITLE Baseline Ada Library Technical Report, Reusability Guidelines			5. FUNDING NUMBERS STARS Contract F19628-88-D-0031	
6. AUTHOR(S)				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Unisys Corporation 12010 Sunrise Valley Drive Reston, VA 22091			8. PERFORMING ORGANIZATION REPORT NUMBER GR-7670-1012(NP)	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Air Force Headquarters, Electronic Systems Division (AFSC) Hanscom AFB, MA 01731-5000			10. SPONSORING / MONITORING AGENCY REPORT NUMBER 00340	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This technical report proposes guidelines for the design and coding of reusable software parts. The proposed guidelines are intended for use by the STARS Prime Contractors and Subcontractors to foster the development of software components that offer common capabilities within well-defined application domains and that are suitable for installation in a library of reusable parts. The requirements presented by the guidelines are directed principally to software developers and, secondarily, to maintainers of a reusable parts library. Each proposed guideline or requirement is accompanied by a rationale, and where possible, illustrative examples. The guidelines are based upon STARS task Q9 experience and interpretations of the materials listed in the references section of the report. <i>Approved STARS (Software Technology for Ada Library)</i>				
14. SUBJECT TERMS Software parts Software characteristics			15. NUMBER OF PAGES 38	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR	

TECHNICAL REPORT

STARS Q9 BASELINE Ada LIBRARY

REUSABILITY GUIDELINES

CONTRACT NO. F19628-88-D-0031

CDRL 00340

5 MAY 1989

PUBLICATION NO. GR-7670-1012 (NP)

Prepared for:
Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:
Unisys Defense Systems
12010 Sunrise Valley Drive
Reston, VA 22091

PREFACE

This document was produced by Software Architecture and Engineering in support of the Unisys STARS Prime contract. This CDRL is for the Baseline Ada Library Task, Q9, of the Unisys STARS First Increment. It is CDRL type A006 number 00340 for the Reusability Guidelines.

Accession For	
NTIS GPA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	<i>per letter</i>
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



TABLE OF CONTENTS

1.0 Introduction	2
2.0 Assumptions	3
3.0 Approach	4
3.1 [AUSNIT] Characteristics	5
3.2 [IBM360] Characteristics	6
3.3 Modifiability and Understandability	6
4.0 Proposed Guidelines	8
4.1 Purpose	8
4.2 Part Definitions	8
4.2.1 Simple Part	8
4.2.2 Generic Part	9
4.2.3 Schematic Part	9
4.2.4 Complex Part	9
4.2.5 Interface Part	9
4.2.6 Implementation Part	9
4.2.7 Bundled Part	9
4.2.8 Rationale	9
4.3 Part Selection	10
4.3.1 Utility	10
4.3.2 Completeness	11
4.3.3 Uniqueness	12
4.4 Part Design/Coding	13
4.4.1 Reliability	14
4.4.2 Understandability	14
4.4.2.1 Uniformity	15
4.4.2.2 Simplicity	17
4.4.3 Loose Coupling	18
4.4.4 Cohesion	19
4.4.5 Isolated Error Handling	20
4.4.6 Independence and Portability	21
4.4.6.1 Compiler Independence	21
4.4.6.2 System Independence	23
4.5 Part Identification	24
5.0 Recommended Tools	26
6.0 Bibliography and References	28
Appendix A: ANNA	30
1.0 Advantages	30
2.0 Disadvantages	30
3.0 Examples	31
3.1 Parameter Constraints	32
3.2 Access Program Behavior	32
3.3 Relations Between Access Programs	33
4.0 ANNA Examples	34
4.1 Annotated List Package	34
4.2 Annotated Set Package	37
4.3 Recommendation	38

1. Introduction

This technical report proposes guidelines for the design and coding of reusable software parts. The proposed guidelines are intended for use by the STARS Prime contractors and subcontractors to foster the development of software components that offer common capabilities within well-defined application domains and that are suitable for installation in a library of reusable parts. The requirements presented by the guidelines are directed principally to software developers and, secondarily, to maintainers of a reusable software parts library. Each proposed guideline or requirement is accompanied by a rationale and, where possible, illustrative examples. The guidelines are based on Q9 task experience and interpretations of the material listed in the "References" section of this report.

This report is divided into six sections, the first being this introduction. The second section defines the assumptions that apply to the guidelines presented in Section 4. Section 3 explains the basic approach taken in the development of the guidelines. Section 5 recommends tools to assist in the achievement and enforcement of the guidelines discussed in Section 4. Section 6 lists the references to the literature used to develop this report. Appendix A is a report on an experiment performed using ANNA, a Ada annotation system [ANNA], to complete the functional specification and verify the implementation of a few selected reusable software components.

2. Assumptions

The context for the proposed guidelines is defined by the following assumptions.

- (1) Parts are restricted to compilable Ada software components and any accompanying documentation, test data, and test programs required by the guidelines themselves. This permits the guidelines to be more specific and makes it possible to recommend software tools to identify and validate reusable parts.
- (2) No particular overall design methodology is assumed. Whereas the application of certain software development methodologies and principles, e.g., object oriented, modular decomposition, and information hiding, are more likely to result in software components that are reusable, the guidelines concentrate on the attributes, discernible in a part itself, that make it reusable.
- (3) It is assumed that software developers are working in an environment that encourages their interest in reusing software components; that they have an interest in developing and accessing a library of reusable parts--whether maintained locally or nationally; that issues of security, proprietary rights, maintenance of part integrity, and the like, are resolved; that there is an active staff to maintain the library; and that the application domains of interest to library subscribers have been established and are supported by well-defined taxonomies.
- (4) The greatest productivity from reuse can be realized if parts are reused "as is". A reusable part is a software component that can be used in more than one system or application. Parts can be reused "as is" or "with modification". These guidelines emphasize the development of parts with attributes that promote their reuse "as is" or whose modification is well-defined by a set of parameters (e.g., generic parts).
- (5) All documents referenced by the proposed guidelines are available to all developers and librarians working in the STARS Software Engineering Environment (SEE). This assumes that the reusability guidelines should not be a standalone document, but may refer to other STARS standards and requirements that promote part reusability.

3. Approach

A comparison of [AUSNIT], [IBM360], [REUSE], and the draft report of these guidelines reveals different approaches to defining guidelines for the reuse of software components. [AUSNIT] divides its guidelines according to three areas of concern: design, Ada interface, and documentation. [IBM360] takes a structural approach by organizing its guidelines according to the sections of the Ada Language Reference Manual (LRM). [REUSE] attempts to discuss software component reuse from the perspective of the software life cycle model--starting with the procurement process, requirements definition, parts specification, and so on, all the way through to parts maintenance in a library of reusable parts.

These guidelines have taken a different approach by asking what are the characteristics of a software part that make it reusable, and for which of these characteristics can guidelines be formulated to ensure that a part exhibits these characteristics. This approach has advantages over those of the other guidelines in that it attempts to link directly reuse characteristics with design and coding practices. The expectation is that the more these guidelines are followed in the development and selection of parts, the more likely it is that they will exhibit the characteristics that help make them reusable, and, the greater the likelihood that they in fact will be reused. The direct link between a guideline and one or more reuse characteristics also provides a greater motivation for following a guideline--something lacking in the other reusability guidelines, and compels a precise definition of the characteristics of reuse.

Table 1 compares the reuse characteristics listed by [AUSNIT] and [IBM360] with the characteristics proposed for organizing the guidelines of this report. The table divides the reuse characteristics into three groups: one for minimal selection criteria; one for those characteristics most applicable to interfaces, and one for those most applicable to implementations. The characteristics of understandability and reliability are important for both interfaces and implementations, but obviously have a different meaning in each context. Simplicity and uniformity are grouped under understandability since together they support that characteristic. Definitions for the proposed characteristics are to be found in the appropriate subsection of the proposed guidelines. An explanation of the mapping of the [AUSNIT] and [IBM360] characteristics to the proposed reusability characteristics is provided here.

Table 1
Comparison of Reuse Characteristics

Proposed	[AUSNIT]	[IBM360]
Selection Criteria		
Utility	Application Independence	--
Completeness	Accessibility	--
Uniqueness	--	Efficiency
Design/Coding Characteristics		
Interface		
Understandability	Communicativeness	Understandability
Simplicity	Simplicity,	Balance,
	Conciseness	Information Hiding
Uniformity	Self-Descriptiveness	Uniformity
Reliability	Functional Scope	Completeness
Loose Coupling	--	Loose Coupling
Cohesion	Generality	High Cohesion,
	--	Abstractness,
	--	Primitiveness
Implementation		
Understandability	Modifiability	Modifiability
Simplicity	Simplicity	Localization
Uniformity	Self-Descriptiveness	--
Reliability	--	Reliability
Independence	Machine Independence	Environment Independence
Isolated Error Handling	--	Error Handling,
	--	Protection Against Error

3.1. [AUSNIT] Characteristics

[AUSNIT] does not define the reusability characteristics listed in Table 1 under the [AUSNIT] column, so that the mapping to the proposed characteristics on the left may not be totally accurate. In any case, in Table 1 application independence is assigned to utility because it is the principle reason for selecting a part for a reuse library. Accessibility is matched with completeness because completeness requires that all references in a part to other parts be resolvable in the context of the library. Without this the part may not be completely accessible. The mapping of communicativeness, simplicity,

conciseness, and self-descriptiveness for interfaces to understandability, simplicity, and uniformity is straightforward. Functional scope implies that the interface of a part leaves nothing to be desired in terms of the operations or data types provided, and this corresponds to the meaning of reliability for specifications as defined here, i.e., the specification can be trusted to provide all necessary operations. Generality can be achieved by a high level of cohesion. Code that is understandable is modifiable, so modifiability has been mapped to understandability. Machine independence is certainly one aspect of code independence. The remaining reusability characteristics listed under [AUSNIT] map directly to proposed characteristics.

3.2. [IBM360] Characteristics

[IBM360] takes its list of reusability characteristics from [REUSE]. In Table 1, efficiency is mapped to uniqueness because it provides a reason for having more than one implementation of a part in the reuse library. Balance between generality and specificity and information hiding are two characteristics that keep the interface simple. Completeness here refers to the adequacy of an interface which the proposed guidelines define as interface reliability. Abstractness and primitiveness are two aspects of highly cohesive interfaces. Again modifiability is a result of understandability, and localization is one aspect of the simplicity of an implementation. The remaining reusability characteristics listed under [IBM360] map directly to proposed characteristics.

3.3. Modifiability and Understandability

The mapping between understandability and modifiability requires further explanation. [AUSNIT] and [IBM360] both regard modifiability as a reuse characteristic. This report has a more limited view of the usefulness of modifiability.

Modifiability only has relevance for implementations (Ada "private parts" (7.2) and "bodies" (3.9)), as Table 1 implies. The modification of specifications can only be trivial (additions or deletions of operators or parameters) before it crosses over into a full redesign, and, as such, does not require any guidelines. (A modification of a specification implies an inadequacy of the original specification, and raises the issue of whether the new specification should supplant the original.) Generic specifications are certainly modifiable through generic instantiations (12.2), but modifiability in this case can only refer to a selection criteria (is a generic specification to be preferred to a non-generic specification), since by its nature a generic specification is modifiable. In order to transform a non-generic specification into a generic specification, one need only consider what elements are to be parameterized and follow

the Ada LRM for declaring a generic specification.

Having limited modifiability to implementations, one cannot formulate any real guidelines to ensure modifiability because the issue is too broad in scope, i.e., what is the realm of possibilities that the implementation must accommodate. Without reviewing a specific implementation and without knowing all contexts in which the part might be used, the issue becomes even more intractable. If the elements of possible change can be parameterized, then a generic specification will be the appropriate solution. Otherwise, everything is possible and so nothing can be anticipated or accommodated in the implementation. [IBM360] does provide a few guidelines that address "modifiability" in that, if followed, make it easier to change the ranges on type declarations and the value of constants. These same rules support ease of code maintenance and code clarity, and represent good Ada coding practice. As such, this report prefers to include these guidelines with understandability.

Understandability is itself a vague characteristic. What is clear to one coder may not be clear to another. But one may assume that if certain aspects of a part are attended to, it may have a greater chance of being understood. This report identifies simplicity and uniformity to be the two attributes supportive of understandability that can be defined and for which guidelines can be written.

4. Proposed Guidelines

This section describes the proposed guidelines for the design and coding of reusable Ada software parts.

4.1. Purpose

These guidelines provide criteria to assist

- (1) software developers to design and develop Ada specifications and code that have maximum potential for reuse, and
- (2) librarians of reusable parts libraries to determine the suitability of a software part for inclusion in their libraries.

It is assumed that the reader is familiar with ANSI/MIL-STD-1815A and the goals of the STARS Prime effort. ANSI/MIL-STD-1815A terms, when they first appear, are emphasized by the use of quotes and a reference in parentheses to the defining section of the standard. References are also made in the rationales to the experiences of the Q9 Task funded under the STARS Prime contract.

4.2. Part Definitions

For the purpose of these guidelines a software part is an Ada "compilation_unit" (10.1) or a set of Ada compilation_units that can be parsed and semantically analyzed in conformance with ANSI/MIL-STD-1815A. A part is distinguished from a tool in that it cannot execute by itself. A code generator is a tool used to create a part. Executable programs used to test parts are test programs.

These guidelines distinguish various types of parts, as defined in this section. The definitions included here are not necessarily exclusive. A schematic part may be complex. A bundled part may consist of no more than a part descriptor, package specification, and package body. Parts that are schematic or complex may bundle parts that are simple, generic, schematic, or complex. For example, a menu manager may rely on a virtual terminal interface and implementation to effect its required display functions.

4.2.1. Simple Part

A simple part is a single non-generic compilation_unit with an empty "context_clause" (10.1), e.g., a subprogram, a package specification, a package body.

4.2.2. Generic Part

A generic part is a "generic_declaration" (12.1) with its implementing "subprogram_body" (6.3) or "package_body" (7.1).

4.2.3. Schematic Part

A schematic part is a set of compilation_units created by a code generator and some input that defines the non-persistent elements of the generated set, e.g., in a generated parser the grammar rules define the non-persistent input set.

4.2.4. Complex Part

A complex part is a set of compilable compilation_units that collectively implement a single functional or data abstraction, e.g., virtual terminal, menu manager.

4.2.5. Interface Part

Interface part refers to the non-private part of an Ada package specification, or the specification of an Ada subprogram.

4.2.6. Implementation Part

Implementation part refers to the private part of an Ada package specification, or a package or subprogram body.

4.2.7. Bundled Part

A bundled part is the logical set of compilation_units needed to define and implement a complex part, including the part's interface and implementation, all dependent interfaces (each bundled with an implementation), and all supporting documentation (part descriptor, manuals), tools, test programs, and test data necessary to identify, understand, use, test, and validate the part. (Part descriptor is described in the "Part Identification" section of these guidelines. For an example of a supporting tool, consider the program that creates Ada packages from menu definitions that are accessed by a menu manager. In general, a bundled schematic part must include the code generator, as a supporting tool, and the input data set used by the generator to produce the schematic part.)

4.2.8. Rationale

A precise definition of part is required to establish the kinds of objects the reuse library must be prepared to handle and how best to serve those that access the library. The different definitions make it possible to fine-tune the guidelines and make them more specific.

Requiring that parts be compilable makes it easier to ensure a part's integrity and to identify its dependencies. Non-compilable Ada code fragments are not considered parts because they are difficult to describe, categorize, and verify. Tools are not considered parts because these guidelines wish to avoid prescribing user interface and runtime environment requirements, which a tool's design must address. (For example, some software tools in the SIMTEL20 use a command line interface to obtain user selected runtime options, while others obtain user options through Ada Text_Io programs. For tools both the method and format of user interface must be prescribed to ensure uniformity of interface and tool portability. In addition, the SEE may prescribe virtual interfaces for certain tools.)

The distinction between interface and implementation parts highlights the fact that Ada package specifications may include implementation details in the private part. This is important, because it means that one interface may not support multiple implementations. The distinction is also important because the guidelines for interfaces, which emphasize "reuse as is", are somewhat different from the guidelines for implementations, which emphasize "reuse with modification". However, as defined here an interface part may be less than a compilation unit and an implementation part more than a compilation unit so that one may not be able to identify a simple part strictly as an interface or implementation part.

The notion of bundled part is introduced to permit the reuse library to include and reference more than compilable code. (These guidelines make no assumption about the way a library retrieves or references parts.) As a practical matter, the more complex a part, the more it may require supporting documentation, tools, test programs, and test data. These guidelines require that supporting documentation, tools, test programs, and test data be bundled with a part, but, except for a part descriptor, they do not prescribe their format or content.

4.3. Part Selection

A software part is selected for inclusion in a library of reusable parts on the basis of whether it exhibits utility, completeness, and uniqueness. These characteristics are considered minimal, and do not necessarily preclude other criteria that reuse part librarians may adopt. This section also serves to refine the definition of part by excluding minimally useful, incomplete (inadequately defined), and duplicative parts.

4.3.1. Utility

A software part has utility if the function it provides or the data type it defines performs or abstracts an essential,

common, and persistent requirement or element of an application domain (vertical utility) or more than one application domain (horizontal utility).

Parts that provide esoteric functions or define types that are implementation bound have a low probability of reuse. Commonality studies and taxonomies for a particular application domain will indicate functions or data abstractions that persist within the application throughout various tools or systems. The software parts that realize these functions or abstractions are likely candidates for reuse.

- * The utility of a schematic part may be limited by the input data set used to define the part, while the input data set itself may have greater utility as far as reuse is concerned. (In this case "reuse with modification" becomes more valuable than "reuse as is".)

Rationale: In the Q9 task commonality of function across diverse tool sets endowed a part or potential part with utility. Even where a function was not realized in many tool sets, as long as its conceptual realization would have provided greater utility to a tool, e.g., a virtual terminal, the potential part was deemed to have utility. The perceived usefulness of a part has to extend beyond the current set of software tools or systems using the part.

4.3.2. Completeness

A software part has completeness if it provides all the necessary source code, external references, and documentation to compile and use it successfully.

- * Test data and/or test programs are essential for completeness if one cannot judge the adequacy of a part's operation without it.
- * Schematic parts lack completeness if they do not reference their code generator and defining input data set.
- * Implementations for the interfaces referenced in a bundled or complex part are not essential for completeness, if the interfaces provide sufficient information to code an alternate implementation.

Rationale: A part that does not adequately describe its external references or the interfaces it depends on may present insurmountable obstacles to reuse. The Q9 task found difficulty in reusing parts whose dependencies were not clearly described or isolated. For example, the virtual terminal, a complex part examined by Q9, was more easily ported because the bundled part included a test program that could be used to determine the

reliability of its performance in a Unix VT100 terminal/Sun console environment. Having the implementations for all referenced interfaces is certainly desirable, but is not essential for reusing a complex or bundled part. Inadequate interfaces make reuse impossible, while the absence of implementations may only raise the cost of reusing a part. A good interface should make that cost more predictable.

4.3.3. Uniqueness

An interface part is unique if it represents a standard specification, a non-standard specification, or an approved alternative to a standard specification. (It is assumed that the reuse library initially hosts Ada interfaces that represent the SEE's accepted standard specifications.)

An implementation part is unique for one or a combination of the following reasons:

- (1) environmental (hardware, operating system, ANSI/MIL-STD-1815A Chapter 13 constructs, special compiler features);
 - (2) algorithmic (e. g., for graphics B-spline versus cubic spline);
 - (3) Booch component forms (see section 3.3 of [BOOCH] and Table 2);
 - (4) tuning factors (size, speed, efficiency).
- * Because an Ada "package_declaration" (7.1) can be both an interface part and an implementation part, the definition of uniqueness allows for multiple package_declarations, provided that the difference between each

Table 2
Booch Component Forms

Concurrent	Space	Garbage Collection	Iterator
Sequential	Bounded	Managed	Noniterator
Guarded	Unbounded	Unmanaged	Iterator
Concurrent		Controlled	
Multiple			

package_declaration is only discernible in the private part of the package_declaration.

- * Where interface parts compete to implement a specification, and the differences between competing parts can be parameterized, a generic part should be used as the standard interface.
- * The unique features of a part that distinguish it from its competitors should be documented by the reuse library and made available as a part selection criterion.

Rationale: Uniqueness is a necessary condition to prevent cluttering the reuse library with duplicate parts. Care must be taken to prevent the inclusion of parts that differ only in nomenclature. Interface parts that implement standard specifications should not be duplicated. The reuse library also has the obligation to expand its collection of parts in a rational manner, which includes documenting the differences between parts that at least superficially provide the same functionality. (It is an open question whether one includes a highly useful part that is non-standard, but implemented, while a standard interface is defined, but unimplemented. For example, the Q9 baseline library includes a virtual terminal that does not follow CAIS. Should this part be included in the library? In general, as a matter of policy should the library concern itself with competing standard interfaces?)

4.4. Part Design/Coding

This section defines the characteristics that promote part reusability and prescribes design and code guidelines for each characteristic. The first column of Table 1 lists these characteristics, which are understandability (simplicity, uniformity), reliability, loose coupling, cohesion, independence, and isolated error handling. Reliability and understandability apply both to interfaces and implementations, while loose coupling and cohesion apply to interfaces, and independence and isolated error handling to implementations.

The guidelines presented here address the reuse characteristics of compilation units--the minimal part component. Distinctions are made in the definitions of characteristics and their guidelines for different kinds of parts, when required. For example, a complex part may not be able to demonstrate all the characteristics described here because it may have to "with" parts that are not independent or lack cohesion. This is not to be regarded a failing of the complex part as a whole, but a recognition of the fact that some complex parts may interface to other parts that are not reusable across systems or may be poorly designed.

4.4.1. Reliability

An interface part is reliable if it compiles and is adequate and complete, i.e., the types and operators defined by the interface provide either singly or in combination all avenues of access to the exported type that a user might need, or the part follows an accepted standard.

An implementation part is reliable if it performs as specified and required.

- * A reliable part exhibits consistency between documented behavior and actual performance.
- * A reliable part is upwardly compatible with all previous versions.
- * A reliable part documents all known "bugs".
- * A reliable implementation does not use the pragma SUPPRESS.
- * Confidence in reliability is achieved by unit testing, system testing, repeated usage without unexpected failure, and no occurrence of undocumented behavior.

Rationale: The modification of a part's interface for reuse may suggest a lack of generality in the part's design. Standard interfaces, such as CAIS, purport to be complete and do not warrant modification. Performance is the only measure of reliability for an implementation. The minimal information required here and in the documentation standards should provide sufficient information on a part's reliability. The rule on pragma SUPPRESS is intended to ensure parts in a reuse library do not use this pragma, but there is no objection to its use, especially in embedded systems, if the justification for its use is documented. The pragma may be commented out for an implementation to satisfy this rule. The more complex a part, the more difficult it may be to prove its reliability. Use of compilable annotations, such as described in [ANNA], offers a means of verifying the constraints and semantics of software parts.

4.4.2. Understandability

A part is understandable if it conforms to minimal standards of documentation, nomenclature, and format (uniformity) and is simple.

- * Understandability is essential for interfaces, and desirable, but not essential for implementations.

- * Understandability is not essential for an implementation, if "reuse as is" is assumed, provided that the implementation is reliable.
- * Understandability is essential for an implementation, if "reuse with modification" is assumed, since modification is impossible if the part cannot be comprehended.
- * Standard format and nomenclature and simple code are helpful, but not essential for understandability, whereas standard documentation is essential for understandability if it requires information not provided in the Ada code itself, but is necessary for correct use of the part.

4.4.2.1. Uniformity

A software part has uniformity if it has a standard format, uses standard nomenclature, and is described by standard documentation.

- * Standard format can be achieved and enforced through the use of formatters. (These guidelines do not recommend any particular format, since any standard format that is adopted by STARS will ensure uniformity. The standard format proposed [STANDARDS] is acceptable, but currently there is no formatter that implements [STANDARDS].)

In addition, the following guidelines, applicable to the format of identifiers, are useful but not enforceable through formatters:

- * Underscores must separate the words of an identifier (3.6.1.3 of [IBM360]).
- * Overly similar names are not used (3.6.1.4 of [IBM360]).
- * Standard nomenclature can be achieved by adherence to the taxonomy exported by the reuse library. This taxonomy should establish conventions for naming parts, functions, abstract data types and their operators. (Adherence to this guideline will help ensure that developers do not inadvertently redevelop parts already exported by the reuse library, and can help identify possible new parts for the library.)
- * Standard documentation can be enforced through the use of a program development language (PDL). A PDL provides the following advantages:
 - (1) A uniform means of expression for documenting and explaining parts.

- (2) Greater opportunities for automated support in managing parts.
- (3) The means to identify inconsistencies between comments and compilable code.
- (4) Enforcement of requirements for specific information essential to correct use and understanding of parts.

A PDL requires the use of labeled comments (directives) in order to ensure that specific information is provided with a part. These guidelines recommend the following directives be used for types and program units declared in interfaces and implementations. (These directives are based on [BYRON].)

Algorithm: for implementations; identifies the algorithm by name if standard or by reference if published, or provides a high level description.

Effects: describes conditions that cause exceptions to be raised; explains semantics of a program unit of a part;

Errors: lists any text the part associates with runtime errors to identify and explain their occurrence. (When an error occurs during the execution of a part, the text may be directed to a terminal or error log.)

Invariant: for types; describes assumptions about the type or object whose violation may result in an error.

Modifies: describes the global variables modified by the unit.

Overview: for interfaces describes usage; for implementations describes the implementation.

Notes: describes any design assumptions for a program unit that are important and must be satisfied for correct usage; describes obligations--actions whose effects propagate to the interface; lists host and target dependencies, built-in limitations.

N/A: lists the directives that have not been used.

Raises: lists exceptions raised by a program unit.

Recovery: for interfaces; indicates what actions may be taken to recover when an exception is raised;

Requires: for interfaces; describes preconditions imposed on the user of the unit of a part.

Tuning: describes performance and size constraints.

Rationale: Uniformity supports reuse by making it easier to identify and understand a part's purpose, use, and limitations. The directives proposed here provide additional information essential and useful for using a part correctly. The use of formatters and PDL tools makes it easier to enforce uniformity standards. For example, it is much simpler to submit a part to a formatter prior to installation into a parts library to ensure standard indentations or reserved word formats than to ask programmers to follow such conventions. Standards in nomenclature are important to ensure that similarities between parts are not masked by names, but they are much more difficult to develop and enforce. These guidelines merely indicate that there are advantages to establishing some standards in this area. For example, in the development of list and string parts for Q9, "&" was used to identify the operator for string concatenation and list appending because of the similar semantic connotation.

4.4.2.2. Simplicity

An interface is simple if it is loosely coupled and has model cohesion.

- * See sections on loose coupling and cohesion.
- * The identifiers of functions with the same input list but different result types should not be overloaded (3.8.1.2 of [IBM360]).
- * Interfaces must not expose implementation details.

An implementation is simple if it exhibits the following characteristics:

- * Numeric literals are used only in declaration blocks of the compilation_units of the part.
- * Nesting of packages or subprogram units is no deeper than one level.
- * There are no GOTO's in the part.
- * There are no double negatives in the implementing code of the part.
- * No boolean expression uses both AND's and OR's.
- * No USE clause appears in a context_clause in the part.
- * Loop statements use explicitly declared types for integer ranges (3.7.2 of [IBM360]).

- * Nested "if" statements are avoided by use of "elsif"'s (3.7.3.1 of [IBM360]).
- * Program units that open files close them before completion.
- * Types are used for integer discrete ranges (3.5.2.1 of [IBM360]).
- * Types or subtypes are used for array index designations. (3.5.2.2 of [IBM360]).
- * Named constants are used for parameter defaults (3.8.1.3 of [IBM360]).

Rationale: These features are not essential for reusability, but their violation warrants scrutiny because an unduly complex part is difficult to understand, maintain, and prone to error. Numeric literals require an explanation--why one value versus some other. Where a literal relates to an internally defined constraint, attributes can be used instead to recover the value. Where a literal relates to an exogenous variable, its declaration can be isolated to a separate part that characterizes all exogenous factors. Where a literal is arbitrary, a comment must note this. Packages and subprogram units that are not visibly coupled with their enclosing scope should not be nested. Nested units are difficult to access for reusability, and unless a nested unit is coupled with its enclosing unit, there is no justification for the nesting. Double negatives are unnecessary and confusing. Use clauses only hide the origin of identifiers and must be avoided.

4.4.3. Loose Coupling

(The following definitions are taken from [EMBLEY-FEB87]).

Compilation_unit A is visibly coupled with compilation_unit B if A accesses directly the data structures of B.

Compilation_unit A is surreptitiously coupled with compilation_unit B if it uses undocumented information about compilation_unit B's data structures.

Compilation_unit A and B are loosely coupled if they are neither visibly nor surreptitiously coupled.

- * An interface part has loose coupling if it prevents the parts that depend on it from being visibly and surreptitiously coupled with it.
- * Visible coupling is prevented by exporting data abstractions using "private" (7.4) and especially "limited private" (7.4) types.

- * Surreptitious coupling is prevented by full disclosure of a part's external effects in the part's interface.
- * Implementations, when accessing an interface, must rely only on the documented behavior described in a part's interface.
- * The values of exported constants should be deferred or access programs should be used to return the values of constants.
- * Interfaces must never export objects.
- * Always provide for the initialization of private and limited private types.

Rationale: Loosely coupled parts are easier to reuse because all dependencies are clearly specified and isolated.

4.4.4. Cohesion

[EMBLEY-FEB87] defines five strengths of cohesion with respect to abstract data types, with the highest cohesion strength being model. Model cohesion is defined negatively in terms of the four other strengths--separable, multifaceted, non-delegation, and concealed. As defined here cohesion applies only to parts that export abstract data types (ADT). Model cohesion is achieved in a part if it does not have:

separable strength

An ADT part has separable strength if the part exports an operator (function or procedure) that does not use a domain of the ADT it exports; or the part has a logically exported domain of the ADT that no operator of the part uses; or the part has two or more logically exported domains whose operators do not share any of the domains of the ADT.

multifaceted

An ADT part has multifaceted strength if it does not have separable strength, and it exports two or more domains of the ADT. Because it is not separable some operator must share two or more exported domains.

non-delegation

An ADT part has non-delegation strength if it has neither separable nor multifaceted strength, and it has an operator that can be delegated to a more primitive ADT.

concealed

An ADT part has concealed strength if it has neither separable, multifaceted, nor non-delegation strength

and it has a logically hidden ADT.

- * Interfaces exhibiting model cohesion are to be preferred to those that do not, all other features being equal.
- * There is no necessity to declare related program units in one package if they do not require access to a common abstract data type. (Grouping of related functions is a convenience.)

Rationale: Cohesion is not essential for reusability per se, but an interface that has model strength is more general, and, therefore, more likely to provide a common capability that can be reused. (See [EMBLEY-FEB87] for illustrative examples of the five cohesion strengths.)

4.4.5. Isolated Error Handling

- * Simple and generic parts must export all exceptions that they raise, unless the exception is handled within an implementation part.
- * If explanatory text is associated with an exception, the part must not use `Text_Io` to display the text when the exception is raised. Instead, either the part must provide on its interface access to the text, or it must pass the text to another `compilation_unit` that logs or displays the text associated with the exception.
- * No `compilation_unit` of a part may propagate a predefined exception without renaming it. (3.9.2.2 of [IBM360]).
- * For each declared exception on an interface, a formal comment must describe the conditions that raise the exception.
- * In general, exceptions should not be used as a substitute for normal methods of altering the control sequence of a part.

Rationale: Part reuse is hindered by idiosyncratic methods of handling errors. Requiring the isolation of the transmittal of error messages makes a part more reusable as it makes no assumptions about how a system is communicating with its user. By isolating the interface, the user of the part has greater freedom to alter the explanatory text associated with an exception. It is also important, especially for real time embedded systems, that they have the ability to intercept and handle exceptions when they occur. Because a part cannot know the complete context of its use, it is futile to raise exceptions it does not export.

4.4.6. Independence and Portability

A software part is independent if it is reliable and compiles without error or modification under at least two different compilers and executes to normal completion without modification on at least two different operating systems.

A schematic or complex part is independent if all of its simple and generic parts are independent.

A software part is portable if its interface and implementation parts are independent and all the interface parts it "with"'s are independent. (The implementation parts associated with each "with"'d interface part may or may not be independent.)

- * All independent parts are portable, but a portable part may not be independent.
- * The guidelines of Section 2 of [PAPPAS] are hereby incorporated by reference, with the following qualifications:
 - (1) isolation of dependent code is preferred to coding to the "weakest" system,
 - (2) coding should not be constrained by consideration of compiler features that violate the Ada LRM.

Further guidance here concentrates on compiler and system independence.

4.4.6.1. Compiler Independence

A software part is compiler independent if it is reliable and compiles under two or more Ada compilers without modification and error on one or more operating systems.

- * Compiler independence is achieved by avoiding standard numeric types, non-standard pragmas, and special utilities provided by a compiler that depend on peculiarities of an operating system.
- * Where avoidance of compiler dependencies is impossible the part should be made portable by isolation of compiler dependencies in a separate part.

Rationale: All standard numeric types are dependent on a compiler's choice of bit representation, which in turn depends on the underlying hardware. On the Q9 task segmentation errors were encountered by the fact that Alsys Standard.Integer is not equivalent to Verdix Standard.Integer. This problem can be

isolated in one of two ways:

- (1) Declare a separate package of numeric types as subtypes of the appropriate standard numeric types, and use these types instead of the standard numeric types. Create a separate package for each available compiler. For example, the Q9 task created a package called `Portable_Numeric_Types`--one for the Alsys compiler and one for the Verdix compiler. The Verdix version included the following type declarations (among others):

```
subtype SHORT_SHORT_INTEGER is Standard.Tiny_Integer;
subtype SHORT_INTEGER is Standard.Short_Integer;
subtype INTEGER is Standard.Integer;
subtype LONG_INTEGER is Standard.Integer;
```

The Alsys version included the following subtype declarations (among others):

```
subtype SHORT_SHORT_INTEGER is Standard.Short_Integer;
subtype SHORT_INTEGER is Standard.Integer;
subtype INTEGER is Standard.Long_Integer;
subtype LONG_INTEGER is Standard.Long_Integer;
```

(These declarations standardize INTEGER representation for the two compilers at 32 bits and prevent the need for different versions of parts that are dependent on the underlying representation. For example, the conversion of DURATION to STANDARD.INTEGER is acceptable under Verdix, but not possible under Alsys, while conversion of DURATION to PORTABLE_NUMERIC_TYPES.INTEGER is always possible. In this way parts that have the DURATION to INTEGER conversion but use Portable_Numeric_Types do not need more than one version for both compilers.)

- (2) Declare one package that defines the numeric types in terms of absolute universal ranges. In the this example Q9 sought an integer type that was the largest that either the Alsys or Verdix compilers could represent in a Sun Unix environment. This could have been achieved by the following declaration:

```
type INT is range System.Min_Int .. System.Max_Int;
```

The use of subtypes as in the first example offers the advantage of access to all supporting attributes and Text_IO functions of the Standard numeric types. The disadvantage lies in maintaining more than one copy of Portable_Numeric_Types. The special type declaration of the second example has a disadvantage in that Text_IO programs must be redefined for the special type since such programs depend on standard numeric types (e.g., STRING uses

POSITIVE and TEXT_IO.COUNT is implementation defined).

In the case of non-standard pragmas or special compiler features, the functions that depend on these should be isolated in separate packages. For example, the Q9 task declared a System_Environment package parallel to the Alsys provided package for use in a Verdix environment and then implemented a separate package body to support the interface with the Verdix compiler. (System_Environment provided functions such as ARG_COUNT, ARG_VALUE, and so on, to retrieve user entered command line arguments accessible through the underlying operating system.)

4.4.6.2. System Independence

A software part is system independent if it is reliable and executes to normal completion on two or more operating systems.

- * System independence is achieved by avoiding use of ANSI/MIL-STD-1815A Chapter 13 constructs (e.g., representation clauses), interface pragmas that access operating system utilities or machine coded programs, dependence on operating system defined entities such as file and directory naming conventions, and reliance on the values of System.System_Name, System.Storage_Unit, and System.Memory_Size, etc.
- * Where avoidance is impossible, the access to these special utilities must be isolated in separate compilation_units to facilitate ease of conversion for potentially reusable parts that may depend on these characteristics.
- * The interface and effects of system dependent utilities must be well documented and isolated in a bundled part.

Rationale: These guidelines have incorporated [PAPPAS] by reference because of the thoroughness of that document. [AUSNIT] also provides an excellent discussion of the difference between reusability and portability. These guidelines value portability in a part because it is easier to estimate the costs of reusing a part if system and compiler dependencies have been isolated.

The Q9 task found that the effort of rehosting compilation_units from one system to another was greatly reduced if system dependencies were isolated. For example, a package providing file access functions can hide operating system defined file naming and access conventions in the low level subprograms it accesses. An application system that uses the functions of the file package can avoid knowledge of the actual file conventions of the operating system on which it will run. When this is done correctly, the application system only needs to identify files by strings.

Portability is probably a more practical goal than part independence. Certainly, parts that are classified in [REHOST] as Software Abstractions should be independent by definition; those classified as Hardware Hiding are by definition compiler or system dependent; while those classified as Functional Abstractions should be at least portable.

4.5. Part Identification

- * A software part (generic, schematic, complex) shall be identified by a part descriptor.
- * The part descriptor shall be associated with a part by name and shall have as a minimum the following directives:

Part Name: a descriptive name identifying the part, consistent with the taxonomies for the part's domain.

Aliases: synonyms for the part name; consistent with the taxonomies for the part's domain.

Domain: the application area the part belongs to; if domain independent "ADT" or "Software Abstraction"; the reuse library's supported taxonomies shall define the range of possible domains.

Function: the purpose of the part; if a specification its purpose is to "define".

Objects: what the part declares or manipulates.

Environment: a description of the hardware, operating systems, and compilers under which the part has been compiled and used.

Keywords: possible indices for the part; consistent with the reuse library taxonomies.

Test Level: level the part has been tested; one or more than one of the following levels in Table 3 are possible.

Rationale: The descriptor directives listed here are thought minimal, and values for aliases and keywords optional. Additional directives are possible and should be drawn from those listed in Section 7.9.2.2 "Submission Information" of [REUSE] and the requirements of the reuse library itself. The aliases, domain, and keywords directives are based on the constructs appearing in the ADT descriptor of [EMBLEY-MAR87] and are intended to help the reuse library to classify the part. (The "description" directive used by [EMBLEY-MAR87] has been omitted

Table 3
Test Levels

Level	Description
Executed	Used in executable program
Walk Through	Tested by inspection
Global	Top level functionality tested
Unit	Outputs and effects of each program unit tested
Complete Unit	Unit tested and each decision point of each program unit tested

because the "overview" directive is required for each program unit of a part. The environment directive is important to establish the system environment with which the part is compatible "as is". The notion of function and object is based on [PRIETO-DIAZ] and is to be preferred to an elaborate description.

5. Recommended Tools

The Q9 task relied on a number of software tools to provide a basis for the requirements of the guidelines and to assist in the development of reusable parts. These tools are listed here along with a brief description of their function and usefulness for reuse.

Pager: This is an Ada program that permitted the physical bundling of software parts. Basically it takes a list of files--in this case Ada compilation units with their supporting documentation and a part descriptor--and places them in a single file. The tool is also used to unbundle a part. Such a tool would be useful if parts are to be submitted to and extracted from a reuse library in a bundled format.

Formatter: Both the Alslys and Verdex compilers offer formatters. The NOSC/Ada tools also included a "pretty-printer" that Q9 rehosted. Attention was given to the formatter that best approximated the STARS foundation format standards, which was the Alslys formatter. However, none of these formatters is able to check for special directives. Formatters should test for required directives to ensure that parts submitted to the library are properly documented. They also need to be more robust in the types of format options that they support. For example, none of the formatters listed here would permit a selected identifier to appear as "XXX.nnn". A DIANA based formatter could also detect violations of simplicity and transform the code to comply. For example, "uses" clauses could be removed automatically and the affected simple identifiers replaced with selected identifiers.

Standards Checker: The NOSC/Ada standards checker and style checker offer some capability to check for the use of standard numeric types, nesting, and the use of USE in a context clause. It is not hard to imagine development of standards checker to verify most of the criteria of the proposed guidelines. In particular, the attributes of independence, coupling (except for surreptitious), cohesion (in part), and simplicity could all be validated with a standards checker. (Uniformity can be maintained by formatters and automated PDL, while reliability is amenable to automated testing and ANNA.)

Byron: Q9 did not have Byron (a PDL) to use, but many of its directives were found useful for the reuse parts. Byron also supports a tool to analyze code dependencies and embedding which would have proved useful in a commonality study and to provide evidence where reuse principles were violated. However, Byron had three recognizable deficiencies: (1) the required comment identifiers could not be altered ("--|" conflicts with ANNA conventions); (2) some directives had to appear in two places--the specification and the body--which makes little sense if one realizes that the same interface may be used with different

5 May 1989

STARS-QC-00340/001/01

implementations, and, more importantly, the interface should be the defining document at all times; and (3) some required comments were thought excessive such as the "synopses" and comments on subprogram parameters. But the concept of standard comments and standard PDL regulated by a software tool is important.

ANNA: This tool offers an important capability to document and verify the constraints and semantics of Ada programs. Its value to reuse lies in a standard language to describe program constraints and semantics. At all times short and precise formulae are to be preferred to wordy and ambiguous comments that cannot be tested or verified. An experiment was performed using this tool to explore the approach offered by ANNA; the results of this experiment are reported in Appendix A.

Others: Additional tools could be devised to check the completeness of a bundled part. Such a tool would be able to enforce the standards for a bundled part and would be able to check all external references against the units in a library.

6. Bibliography and References

[ANNA]

Luckham, David C., Friedrich W. von Henke, et. al. ANNA, A Language for Annotating Ada Programs, Reference Manual, Springer-Verlag, Berlin, 1987.

[AUSNIT]

Ausnit, Christine, Christine Braun, et. al. Ada Reusability Guidelines, Softech, Inc., Waltham, Massachusetts, April 1985.

[BOOCH]

Booch, Grady. Software Components with Ada, The Benjamin/Cummings Publishing Company, Inc., 1987.

[BYRON]

The Byron User's Manual, Version 1.1, Intermetrics, Inc., Cambridge, Massachusetts, 1984.

[CAMP]

McNicholl, Daniel G., Constance Palmer, et al. Common Ada Missile Packages (CAMP), Defense Technical Information Center, May 1986.

[EMBLEY-FEB87]

Embley, David W. and Woodfield, Scott N. "Cohesion and Coupling for Abstract Data Types," Proceedings Sixth Phoenix Conference on Computers and Communications, Phoenix, Arizona, February 1987, pp. 229-234.

[EMBLEY-MAR87]

Embley, David W. and Woodfield, Scott N. "A Knowledge Structure for Reusing Abstract Data Types," Proceedings of the 9th International Conference on Software Engineering, Monterey, California, March-April 1987, pp. 360-368.

[FREEMAN]

Freeman, Peter. "Reusable Software Engineering: Concepts and Research Directions," ITT Proceedings of the Workshop on Reusability in Programming, 1983, pp. 129-137.

[IBM360]

Reusability Guidelines, STARS Program CDRL 360, IBM Systems Integration Division, Gaithersburg, Maryland, 17 December 1988.

[MEYER]

Meyer, Bertrand. "Reusability: The Case for Object-Oriented Design", IEEE Software, Vol. 4, No. 2, March 1987, pp. 50-64.

5 May 1989

STARS-QC-00340/001/01

[PAPPAS]

Pappas, Frank. Ada Portability Guidelines, Softech, Inc., Waltham, Massachusetts, March 1985.

[PARNAS]

Parnas, D. L., P. C. Clements, and D. M. Weiss. "Enhancing Reusability with Information Hiding," ITT Proceedings of the Workshop on Reusability in Programming, 1983, pp. 240-247.

[PRIETO-DIAZ]

Prieto-Diaz, Ruben; Freeman, Peter. "Classifying Software for Reusability," IEEE Software, January 1987, pp. 6-16.

[Q13-REPORT]

"Evaluation/SEE Inclusion Criteria". Document Number STARS-QS-21013/001/00. November 11, 1988.

[REHOST]

"Technical Report Tools Rehosed"; CDRL 00300, Document Number STARS-QC-00300/001/00. Software A&E, November, 1988.

[REUSE]

Reusability Guidebook, V4.4 (draft), STARS, September 1986.

[STANDARDS]

"Ada Code Standards" (Draft), STARS, 15 December 1987.

[TRACZ87]

Tracz, Will. "Automatic Parameterization: A Case Study," Computer Systems Laboratory ERL 402, Stanford University, 29 October 1987.

[TRACZ88]

Tracz, Will. "Modularization: Approaches to Reuse in Ada," IBM SID, 23 September 1988.

Appendix A: ANNA

As part of the STARS Q9.4.5 task (see [REHOST]), select reusable parts (sets and lists) were commented using ANNotated Ada (ANNA), Release 0.52, whose syntax and semantics is described in [ANNA], to explore its advantages and disadvantages. Two areas of interest were explored. These included using ANNA to

- (1) Describe formally the expected behavior of access programs, and
- (2) Describe the expected relations between access programs.

ANNA can also be used to describe conditions that raise exceptions and to prescribe additional constraints on subprogram parameters. Examples of ANNA commented code are provided in a below and in Section 4.

1. Advantages

The advantages of ANNA are:

- (1) Because ANNA comments are compilable under the ANNA tool, they offer a means to verify the correctness of Ada interfaces and implementations without affecting the object image of the Ada source code when compiled under normal Ada compilers.
- (2) ANNA allowed us to express formally the effects of access programs on an interface. This in turn made it possible to uncover design flaws or inconsistencies. This occurred in two ways--(a) either the formal representation of the effects of an access program made the inconsistent behavior of its implementation readily apparent, or (b) the formal representation exposed a flaw in the design itself because it focused attention on what was actually taking place in terms of a formal abstraction versus the procedural implementation.
- (3) ANNA makes it possible to prevent surreptitious coupling (defined in Section 4.4.3) on interfaces by providing a means to describe formally the expected behavior of access programs.

2. Disadvantages

The chief disadvantages of ANNA are implementational:

- (1) Many transformations to an executable object image are not implemented in the current version of ANNA. This meant that except for very simple constructs, we could

not produce an object module with the ANNA generated constraints. For example, although ANNA semantically analyzes most annotations, it does not yet produce the code for axioms. Thus, we could not demonstrate how the axioms would ensure the correctness of the underlying code.

- (2) ANNA exposes the difficulty of trying to describe the behavior of access programs on access types when the real interest is in the values they reference. This is apparent in the List example, where assertions are being made about returned lists as being identical but not equal to an input list.
- (3) ANNA itself is difficult to learn. A textbook, much like those used to describe Ada, is required to help understand the correct way to use ANNA. An understanding of propositional calculus is also necessary.
- (4) ANNA cannot be used to describe constraints on objects outside of Ada. For example, in an attempt to put ANNA comments in the package specification `System_Environment`, we could use ANNA to constrain the input to `ARG_VALUE` by formally requiring it to be greater than or equal to the value returned by `ARG_COUNT`, but input to `ENV_VALUE` could not be constrained because there was no way to describe the domain of acceptable Unix environmental variables. These entities could only be defined by the operating system.

3. Examples

The following examples show code commented without ANNA and with ANNA. They are only fragments of more complete compilation units. ANNA annotations are preceded by

"--|" or "--:"

where "--|" denotes an annotation (containing assertions about Ada program states very similar in form to propositional logic) and "--:" denotes virtual Ada text (auxiliary statements used to support the annotations, but written in pure Ada).

Although one does not need the ANNA tool to write comments in ANNA style, the fact that ANNA will parse and analyze the comments and produce executable code to enforce the behavior being described, gives ANNA its unique advantage.

3.1. Parameter Constraints

The functions in this example represent an Ada solution to the C-oriented interface between executable programs and the UNIX shell. In UNIX, the main entry of a program can be passed a variable number of arguments (from the UNIX command invoking the executable), which is presented to the main program as two parameters: a variable-length array of variable-length strings, and an integer representing the actual length of the array. In the Ada example below, this interface is hidden behind two separate, but logically related, functions.

Without ANNA:

```
function ARG_COUNT return POSITIVE;
-- returns the number of arguments on the command line

ILLEGAL_ARG_INDEX exception;

function ARG_VALUE (INDEX : in NATURAL) return STRING;
-- returns the INDEXth argument on the command line of
--   the program
-- if INDEX is >= ARG_COUNT, the function will raise
-- ILLEGAL_ARG_INDEX exception
```

With ANNA:

```
function ARG_COUNT return POSITIVE;
-- returns the number of arguments on the command line

ILLEGAL_ARG_INDEX exception;

function ARG_VALUE (INDEX : in NATURAL) return STRING;
--| where not (INDEX >= ARG_COUNT) => raise ILLEGAL_ARG_INDEX;
```

3.2. Access Program Behavior

Without ANNA:

```
type List is private;

function "&" (List1, List2 : in List) return List;
```

```
-- Effects:
-- This returns a list containing List1 and List2.
-- If List1 and List2 are the same list then a copy is made
-- of List1 and List2 is appended to the copy.
```

With ANNA:

```
type List is private;

--: function Identical (L1, L2 : in List) return Boolean;

function "&" (List1, List2 : in List) return List;
--| where return L : List =>
--|   if Identical (List1, List2) then
--|     not Identical (L, List1) and not Identical (L, List2)
--|   else
--|     Identical (L, List1) and not Identical (L, List2)
--|   end if;
```

The implementation of Identical is defined in the package body.
List is declared in the private part as an access type.

3.3. Relations Between Access Programs

Without ANNA:

```
generic
  type Universe is (<>);
package Set_Package is

  type Set is Private;
  Null_Set : constant Set;

  function Is_Empty (Set1 : Set) return Boolean;
  function Is_Member (Element : Universe; Of_Set : Set)
    return Boolean;
  subtype Number is Integer
    range 0 .. (Universe'Pos (Universe'Last) -
      Universe'Pos (Universe'First) + 1);
  function Number_In (Set1 : Set) return Number;

private
  type Set is Array (Universe) of Boolean;
  Null_Set : Constant Set := (Others => False);
end Set_Package;
```

With ANNA:

```

generic
  type Universe is (<>);
package Set_Package is

  type Set is Private;
  Null_Set : constant Set;

  --: function "=" (Set1, Set2 : Set) return Boolean;

  function Is_Empty (Set1 : Set) return Boolean;
  function Is_Member (Element : Universe; Of_Set : Set)
    return Boolean;
  subtype Number is Integer
    range 0 .. (Universe'Pos (Universe'Last) -
      Universe'Pos (Universe'First) + 1);
  function Number_In (Set1 : Set) return Number;

  --| axiom
  --| for all X : Set; E : Universe =>
  --|   Is_Empty (X) <=> Set_Package."=" (X, Null_Set),
  --|   (not (Is_Empty (X)) <=> Number_In (X) > 0,
  --|   (Number_In (X) = 0) <=> Is_Empty (X),
  --|   (Is_Member (E, X)) -> not (Is_Empty (X)),
  --|   not (Is_Empty (X)) ->
  --|     (exist F : Universe => Is_Member (F,X));

private
  type Set is Array (Universe) of Boolean;
  Null_Set : Constant Set := (Others => False);
end Set_Package;

```

4. ANNA Examples

The following subsections demonstrate the use of current ANNA capabilities in two actual package specifications from the designated set of reusable components, a list abstract data type package and a set abstract data type package.

4.1. Annotated List Package

```

generic

  type Item_Type is private;
  with function Equal (X, Y : in Item_Type) return Boolean;

```

package Lists is

type List is private;

Empty_List : exception;

--: function Identical (L1, L2 : in List) return Boolean;
 --: function List_Equal (L1, L2 : in List) return Boolean;

function "&" (List1, List2 : in List) return List;
 --| where return L : List =>
 --| if Identical (List1, List2) then
 --| not Identical (L, List1) and not Identical (L, List2)
 --| else
 --| Identical (L, List1) and not Identical (L, List2)
 --| end if;

function "&" (Element1, Element2 : in Item_Type) return List;

function "&" (Element : in Item_Type;
 List1 : in List) return List;
 --| where return L : List =>
 --| not Identical (L, List1) and not List_Equal (L, List1);

function "&" (List1 : in List;
 Element : in Item_Type) return List;
 --| where return L : List =>
 --| Identical (L, List1) and not List_Equal (L, List1);

function Length (L : in List) return Integer;

function Firstvalue (L : in List) return Item_Type;

function Tail (L : in List) return List;

function Is_In (L : in List;
 Element : in Item_Type) return Boolean;

function Isempy (L : in List) return Boolean;

--| axiom
 --| for all E1, E2 : Item_Type; L : List; N : Integer =>
 --|
 --| List_Equal (L, (E1 & E2)) ->
 --| Equal (Firstvalue (L), E1) and
 --| Equal (Firstvalue (Tail (L)), E2) and
 --| Length (L) = 2,
 --|
 --| Length (L) = N -> Length (E1 & L) = N + 1,
 --| (Length (L) > 0) <-> (exist E : Item_Type => Is_in (L, E)),
 --|
 --| Equal (Firstvalue (E1 & L), E1),
 --| List_Equal (Tail (E1 & L), L),

5 May 1989

STARS-QC-00340/001/01

```
--
--|      Is_in (E1 & L, E1),
--|      Is_in (L & E1, E1),
--|      Is_in (E1 & E2, E1),
--|      Is_in (E1 & E2, E2),
--
--|      not (Is_in (L, E1)) <-> Iempty (L);

private

type Cell;
type List is access Cell;

type Cell is
  record
    Info : Item_Type;
    Next : List;
  end record;

end Lists;
```


4.2. Annotated Set Package

```

generic
  type Universe is (<>);
package Set_Package is

  type Set is private;
  Null_Set : constant Set;

  function "*" (Set_1 : in Set;
               Set_2 : in Set) return Set;
  function "+" (Element : in Universe;
               Set_1 : in Set) return Set;
  function "+" (Set_1 : in Set;
               Set_2 : in Set) return Set;
  function "+" (Set_1 : in Set;
               Element : in Universe) return Set;
  function "-" (Set_1 : in Set;
               Set_2 : in Set) return Set;
  function "-" (Set_1 : in Set;
               Element : in Universe) return Set;
  function "<" (Set_1 : in Set;
               Set_2 : in Set) return Boolean;
  function "<=" (Set_1 : in Set;
               Set_2 : in Set) return Boolean;
  function Is_Member (Element : in Universe;
                    Of_Set : in Set) return Boolean;

  --: FUNCTION "=" (Set_1 : Set; Set_2 : Set) RETURN Boolean;

  function Is_Empty (Set_1 : in Set) return Boolean;

  subtype Number is
    Integer
      range 0 ..
        (Universe'POS (Universe'LAST) - Universe'POS (Universe'FIRST) + 1);

  function Number_In (Set_1 : in Set) return Number;

  --| axiom
  --|   for all X, Y : Set; E : Universe =>
  --|   Empty Set
  --|     Number_In (Null_Set) = 0,
  --|     not (Is_Member (E, Null_Set)),
  --|   Empty/Non-Empty Sets
  --|     Is_Empty (X) <=> Set_Package."=" (X, Null_Set),
  --|     (not (Is_Empty (X))) <=> Number_In (X) > 0,
  --|     (Number_In (X) = 0) <=> Is_Empty (X),
  --|     (Is_Member (E, X)) -> not (Is_Empty (X)),
  --|     not (Is_Empty (X)) ->

```

```

--|      (exist F : Universe => Is_Member (F, X)),
-- Set Intersection
--|      Is_Member (E, (X*Y)) ->
--|      (Is_Member (E, X) and Is_Member (E, Y)),
-- Set Union
--|      Is_Member (E, (X+Y)) ->
--|      (Is_Member (E, X) or Is_Member (E, Y)),
-- Set Difference
--|      Is_Member (E, (X-Y)) ->
--|      (Is_Member (E, X) and (not Is_Member (E, Y))),
-- Element Addition
--|      Set_Package."="
--|      (Set_Package."+" (E, X), Set_Package."+" (X, E)),
--|      Is_Member (E, (E + X)),
--|      Is_Member (E, (X + E)),
-- Element Substraction
--|      Is_Member (E, X) <-> not (Is_Member (E, (X - E))),
-- Subset
--|      Set_Package."<" (X, Y) <-> (for all F : Universe =>
--|      Is_Member (F, X) -> Is_Member (F, Y)),
-- Proper Subset
--|      Set_Package."<=" (X, Y) <->
--|      Set_Package."<" (X, Y) and not Set_Package."=" (X, Y);

```

private

```

type Set is array (Universe) of Boolean;
Null_Set : constant Set := (others => False);

```

end Set_Package;

4.3. Recommendation

For ANNA to be used fully in the development of the SEE, it itself needs to be fully implemented. It would be a worthy task for STARS to complete its development and insist on the use of ANNA in the code being developed. We cannot overestimate the precision and clarity that formal propositional annotations give to an interface, and highly recommend the further development of ANNA. Unfortunately, resource constraints for this task limited exploration of ANN use to relatively simple software parts.