

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



REPORT ON ADA* PROGRAM LIBRARIES WORKSHOP

**Naval Postgraduate School
Monterey, California
November 1-3, 1983**

By: Joseph A. Goguen and Karl N. Levitt
Computer Science Laboratory
Computer Science and Technology Division

Prepared for:
Brian Scharr
Ada Joint Program Office

Contract No. N00014-83-M-0088

SRI Project 6186

DTIC
ELECTE
S JAN 2 1985 D
D

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

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025-3493
Telephone: (415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-2046
Telex: 334 486

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

84 12 20 018

AD-A149 570

DTIC FILE COPY



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER N00014-83-M-0088	12. GOVT ACCESSION NO. AD-A149570	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Report on Ada Program Library Workshop	5. TYPE OF REPORT & PERIOD COVERED	
	6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) Joseph A. Goguen Karl N. Levitt	8. CONTRACT OR GRANT NUMBER(s) N00014-83-M-0088	
9. PERFORMING ORGANIZATION NAME AND ADDRESS SRI International 333 Ravenswood Avenue Menlo Park, CA 94025	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ECU6186	
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research 800 N. Quincy Street Code: 614A:DAW Arlington, VA 22217	12. REPORT DATE Nov 1-3, 1983	
	13. NUMBER OF PAGES	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Ada Joint Program Office Room 3D-139 The Pentagon Washington, DC 20301-3081	15. SECURITY CLASS. (of this report) Unclassified	
	15a. DECLASSIFICATION, DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Unclassified		
18. SUPPLEMENTARY NOTES The workshop was organized by SRI International for the AJPO at facilities made available by the Naval Postgraduate School at Monterey, California.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Program Libraries, Reusable Software, Ada Libraries, Program design for Reusability, Programming Environments		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The goal of the Ada Program Libraries Workshop was to explore concepts, problems and approaches relevant to an on-line library system for creating, documenting and maintaining Ada systems. The term "library" was interpreted in a broad sense, as potentially including document, specifications, designs, in addition to compiled Ada code. The Workshop included both prepared presentations (of which there were twelve) and group discussion among all workshop participants; these activities		

DD FORM 1473 1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

(over)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (con't)

dominated the first day and a half. Each working group was proposed by a workshop participant, who would also serve as chairman and later prepare a brief report of the working group results for inclusion in this document.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A/1	



SRI International



REPORT ON ADA* PROGRAM LIBRARIES WORKSHOP

**Naval Postgraduate School
Monterey, California
November 1-3, 1983**

**By: Joseph A. Goguen and Karl N. Levitt
Computer Science Laboratory
Computer Science and Technology Division**

**Prepared for:
Brian Scharr
Ada Joint Program Office**

Contract No. N00014-83-M-0088

SRI Project 6186

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

Approved:

**Karl Levitt, Acting Director
Computer Science Laboratory**

**Donald L. Nielson, Acting Director
Computer Science and Technology Division**

Table of Contents

0 Forward	1
1 Introduction	2
1.1 Goals and Organization of the Workshop	2
1.2 Executive Summary	3
1.2.1 Why Libraries?	3
1.2.2 What Next?	4
1.2.3 Research Recommendations	5
1.2.4 Policy and Non-Technical Issues	5
1.3 Summary of Working Group Conclusions	6
1.3.1 Library Documentation	6
1.3.2 Methodology	8
1.3.3 Library Searching	9
1.3.4 Applications	10
2 LIL: A Library Interconnection Language for Ada	12
2.1 Introduction	12
2.2 Issues and Approaches	14
2.3 Library Content	16
2.4 Program Composition	18
2.4.1 Packages and the Using Hierarchy	22
2.4.2 Theories	24
2.4.3 Generic Entities	25
2.4.4 Views	27
2.4.5 Instantiation	29
2.4.6 Package Stubs	31
2.4.7 Environments	35
2.4.8 Transformations	36
2.4.9 Control Abstractions	40
2.5 Library Organization	41
2.5.1 Truth Management	41
2.5.2 Organization by Semantics	42
2.5.3 System Families	43
2.5.4 Cataloguing	44
2.6 User Interface and Management Issues	44
2.7 Acknowledgements	46
2.8 References	46
3 Prepared Lectures	52
3.1 Why DoD Needs Software Environments	52
3.1.1 Discussion	53
3.2 Conceptual Architecture for a Software Engineering Environment	53
3.2.1 Discussion	53
3.3 An Overview of Ada Libraries	54
3.3.1 Discussion	54
3.4 LIL: A Library Interconnection Language for Ada Programs	55
3.4.1 Discussion	55

3.5 DCP Approach to Ada Libraries	58
3.5.1 Discussion	58
3.6 Flexibility vs. Efficiency for Reusable Components	59
3.6.1 Discussion	59
3.7 Mapping Clear Specifications to Ada Packages	60
3.7.1 Discussion	60
3.8 General Requirements for an Elementary Math Functions Library	60
3.8.1 Discussion	60
3.9 Knowledge Based Tools for Data Type Implementation	61
3.9.1 Discussion	61
3.10 Library Organization and User Interfaces	62
3.10.1 Discussion	62
3.11 Version Control in Program Libraries	63
3.11.1 Discussion	63
3.12 Using ANNA for Specifying and Documenting Ada Packages	64
3.12.1 Discussion	64
4 Reports of the Working Groups	66
4.1 Library Documentation	66
4.1.1 Participants	66
4.1.2 Initial Questions	66
4.1.3 Initial Working Group Discussions	67
4.1.4 Assumptions	68
4.1.5 Scenarios	69
4.1.6 Documentation	70
4.1.7 Policy and Non-Technical Issues	71
4.2 Methodology	72
4.2.1 Introduction	72
4.2.2 Issues	72
4.2.3 Initial Questions	73
4.2.4 Preliminary Report	73
4.2.5 Final Report	74
4.3 Library Searching	75
4.3.1 Introduction	75
4.3.2 Initial Questions	76
4.3.3 Library Objects	76
4.3.4 Basis for Searching	77
4.3.5 Catalogue Information Structuring	78
4.3.6 Characterization of Ada Library Users	78
4.3.7 Tools Supporting Searching	79
4.3.8 Impact on Software Engineering Methodology	79
4.4 Applications	80
4.4.1 Initial Questions	80
4.4.2 The Issue of Incentives	81
4.4.3 Impediments and Potential Disadvantages to Reusability	81
4.4.4 General Approaches to Application-Oriented Reusability	81
4.4.5 Initial Candidate Applications	83

4.4.6 Discussion	84
I. Schedule of Workshop	85
II. Names and Addresses of Participants	86
III. An Example of LIL	91
IV. Slides from Prepared Lectures	94
IV.1 Conceptual Architecture for a Software Engineering Environment	95
IV.2 An Overview of Ada Libraries	96
IV.3 LIL: A Library Interconnection Language for Ada Programs	97
IV.4 DCP Approach to Ada Libraries	98
IV.5 Flexibility vs. Efficiency for Reusable Components	99
IV.6 Mapping Clear Specifications to Ada Packages	100
IV.7 General Requirements for an Elementary Math Function Library	101
IV.8 Knowledge Based Tools for Data Type Implementation	102
IV.9 Library Organization and User Interfaces	103
IV.10 Version Control in Program Libraries	104
IV.11 Using ANNA for Specifying and Documenting Ada Packages	105

List of Figures

Figure 1:	Taxonomy of Library Entities	18
Figure 2:	The View NATD :: POSET => NATURAL	28
Figure 3:	Some Software Components	32
Figure 4:	A Vertical Composition	33
Figure 5:	A Horizontal Composition	34
Figure 6:	A Realization of SORT[NATURAL] with LIST[NATURAL]	34
Figure 7:	Theories Involved in a Generic Package	35
Figure 8:	A LIL Environment	36
Figure 9:	Hyperprogramming Taxonomy	36
Figure 10:	Organization of Library Entities for a Package	43

0 Forward

by Brian Schaar and Jack Kramer

The Ada program approaches two critical aspects of the ever increasing costs of fielding and maintaining Department of Defense (DoD) Mission Critical Computer Systems (MCCS). The first aspect, of course, is the language standardization effort itself. The second aspect is the improvement, availability, and use of Ada Programming Support Environments (APSEs) throughout the MCCS life cycle.

Software reusability has been an important aspect of the program from the start. There was an explicit requirement in the Ada language requirements document, STEELMAN, for an "easily accessible library of generic definitions and separately translated units". In the section on design goals, the ANSI/MIL-STD-1815A-1983, Military Standard Ada Programming Language, recognizes that "like many human activities, the development of programs is becoming even more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components has been a central idea in this design."

There needs to be both a short term approach to providing a library of reusable components as well as a long term approach which is integrated into the DoD APSEs of the future. The long term approach must become central to the concept of reusable software and the way we construct software in the future. The future library system must be an integral part of the way we do business. It must not only be a place for storage of software components but must assist in the capture of information about components, assist the potential user in locating the proper software components, and help the user to integrate and test them as a part of the new system.

In order to better understand the idea of a library of software components and the re-utilization of these components, a workshop on Ada Program Libraries was held at the Naval Postgraduate School, Monterey, California, November 1-3, 1983. The scope of the workshop included concepts, problems and approaches relevant to an on-line library system for creating, documenting and maintaining Ada systems.

1 Introduction

This first section begins with a brief description of the goals and organization of the workshop, followed by an executive summary of the results of the workshop, and more detailed summaries of the conclusions of the four working groups.

1.1 Goals and Organization of the Workshop

The goal of the Ada Program Libraries Workshop was to explore concepts, problems and approaches relevant to an on-line library system for creating, documenting and maintaining Ada systems. The term "library" was interpreted in a broad sense, as potentially including documentation, specifications, designs, requirements, historical information, uncompiled source code, test cases, etc., in addition to compiled Ada code. This effort resulted from an unsolicited proposal to the Ada Joint Program Office by SRI International, entitled "Workshop on Ada Library Principles." There were two major deliverables associated with the contract. The first was for SRI to prepare a "strawman" approach to Ada libraries and present it to the workshop, to provide a context for discussions. The talks by Levitt, Goguen and Meseguer, together with the paper "LIL: A Library Interconnection Language for Ada" by Goguen, fulfilled this requirement; the talks of Levitt, Goguen and Meseguer are summarized in Sections 3.3, 3.4 and 3.10, respectively, and the paper by Goguen appears as Section 2. The second deliverable was this report.

The workshop was organized by Joseph Goguen of SRI International for AJPO at facilities made available by the Naval Postgraduate School at Monterey California. Brian Schaar and Jack Kramer made numerous valuable suggestions about the organization of the workshop; and David Hsiao and Gordon Bradley of NPGS provided help with local arrangements. We very much wish to thank these, and numerous others, especially the workshop participants (who are listed in Appendix II) for their aid in making the workshop so interesting.

The workshop included both prepared presentations (of which there were twelve) and group discussions among all workshop participants; these activities dominated the first day and a half. In addition, seven working groups were proposed, to meet in parallel sessions and later present their conclusions to the workshop as a whole. Each working group was proposed by a workshop participant, who would also serve as chairman and later prepare a brief report of the working group results for inclusion in this document. The working groups proposed were on:

Library Documentation; Methodology; Library Searching; Applications; Tasking; Expert Systems; and Short Term Solutions. Only the first four of these attracted a sufficient number of participants; their reports are given in Section 4, and summarized in Sections 1.3.1-1.3.4 below. The workshop schedule is given in Appendix I.

1.2 Executive Summary

This subsection presents a brief summary of the most important conclusions and recommendations of the workshop. These conclusions are largely extracted from the deliberations of the four working groups (see Section 4 for their full reports, and Section 1.3 for summaries). Like the working groups themselves, these conclusions were also influenced by the twelve formal lectures that were presented (see Section 3 for summaries) and the SRI "strawman" approach (see Section 2).

1.2.1 Why Libraries?

The basic purpose of an Ada Program Library is to reduce the cost and to increase the reliability of Ada system programming by *reusing* existing Ada code to the maximum possible extent. This is in contrast to the currently visionary goal of automatic programming, and to the currently usual practice of starting each new project from scratch. There was essentially universal agreement in the workshop that it should be possible to construct an Ada program library systems that would have significant economic benefit, and that this avenue should therefore be explored.

The program library associated with an Ada compiler contains only fully compiled program units, and is therefore of limited usefulness for program development. In order to obtain a more significant degree of reusability, a much more comprehensive library system is needed. Such a system should store source code, documentation, test cases, etc., as well as compiled code; moreover, it should also provide facilities for cataloguing and retrieving Ada programs, and for assisting the user as he combines library units and newly created programs. Some kind of on-line card catalog will be necessary, as well as a librarian.

1.2.2 What Next?

We first address the short term situation. It was felt that existing database technology could be harnessed to yield largely "passive" but still very useful systems in which components are catalogued after their documentation has been captured. It would be important to instrument such systems so that we can learn how to do better, as well as to have (as mentioned above) an on-line card catalog and a librarian. The quality of the documentation also would be a critical factor.

Long-term solutions should be more active, supporting the creation of documentation along with the creation of library components; much of the documentation (such as writer, date, version, size, and results of executing test data) could be generated automatically. Special support for combining and modifying library components should be provided, including version and configuration management. More innovative forms of documentation, such as graphic displays, data type animations, and computer generated audio explanations, might be provided for library components. Knowledge-based systems might provide advice on what components are best for a particular situation, and on how to construct members of a particular system family. (The last two features will require that some form of *semantic* specification is stored with library components, e.g., ANNA or LIL.) The goal is to make reusability as easy and natural as possible. These systems should also aim at integration into APSE's.

It would be very interesting to undertake some experiments in Ada programming methodology, to see what works and what doesn't. Some application areas suggested for such experiments are operating systems, database systems, protocols, compilers, and message systems. Because these application areas are well-understood, it should be possible to compare the quality of a system produced with library components with one produced with newly created components. Methodological issues that could be tested in connection with libraries include the systematic use of parameterization, of design and specification languages, of knowledge-based search (i.e., semantic indexing), of animations and displays. Other approaches supporting the reuse of software could also be tried, either separately or in combination with libraries and/or with each other. These include Very High Level Languages (VHLLs), application generators, knowledge-based systems, and system families.

1.2.3 Research Recommendations

For the short term, passive library systems should be constructed and instrumented. The information collected from early sites should be carefully analyzed and evaluated. Experiments should be carried out on these systems, exploring the value of various methodological approaches. Networking issues must also be considered; because the Ada community is already distributed, Ada libraries will clearly also be distributed.

Longer term research goals should center on developing a methodology especially suited to the reuse of Ada software components, and tools to support that methodology. Possibilities include the use of: specification and design languages; system families; knowledge-based systems; parameterization, vertical and horizontal structure, theories and views (as described in the LIL paper); the reuse of designs and requirements; and integration with configuration and version management (see Tichy's talk).

It should be noted that the Ada program library associated with a compiler does not permit reuse of packages that have been developed in a top-down manner, because of limitations to Ada's **separate** statement. In order to overcome such limitations, one must have a level of description above Ada in which to express interconnections of Ada program units. Such an approach might also be useful for configuration and version management, component documentation, and library searching.

1.2.4 Policy and Non-Technical Issues

It seem likely that issues such as incentive, personnel, user confidence, quality control and ownership will have at least as much impact on the success of the library concept as technical issues. Particular suggestions included rewarding DoD contractors for adding to the library and for using components in it, as well as for use of their components by others. Strong policies will be needed regarding the quality of documentation, and tools might be provided to help enforce them. Methods will be needed for disseminating documentation ("advertising components"), for validating the quality of components, and for educating personnel in the effective use of library Ada systems.

1.3 Summary of Working Group Conclusions

This subsection presents summaries of the conclusions reached by each of the four working groups formed by the Ada Program Libraries Workshop.

1.3.1 Library Documentation

The deliberations of the "Library Documentation" working group were in fact somewhat broader than its title might suggest. Its conclusions are summarized here under three main headings: Scenarios, Documentation, and Non-technical Policy Issues.

1. Scenarios. The working group developed short term and long term scenarios for how a software component library might be constructed and operated. Both scenarios must provide cataloguing, updating and retrieving capabilities; and careful attention must be paid to the transition from the short term to the long term solution library system.

a. Short Term.

- The short term will be "passive" with emphasis on retrieving and searching. Components will be registered with the system "after the fact" of development. Components will be catalogued after their documentation has been captured.
- For a component to be useful it must be well documented. *Quality* is much more important than quantity.
- A "card catalog" will be critical. It was felt that existing database technology could be used. A hierarchical schema could provide easy ability to add, delete and update the catalog. Most importantly, the retrieval language could be simple and oriented toward its use.
- We must instrument the system now in order to learn how to build future systems.
- A librarian would be required. It is critical that a follow up debriefing system be implemented with the initial system.

b. Long Term.

- The long term system must be an active part of the user's everyday work environment. Where possible, the system should automatically construct the necessary documentation and appropriate cataloging information when a component is registered. It should take minimal user effort to add a component to the library.
- Reusability must become an integral part of our future system development methodologies and also must become central to our software engineering environments. Reusability must be natural, not just something forced by management.

- Information and presentation mechanisms "higher" than code will be mandatory for quick user understanding of a component and how it might fit into his system.
 - The system should know something about the user, the available components, and the application area in order to help the user find the best component for his needs. There probably should be some sort of working set kept for each user and application area.
 - A wide physical and organizational dispersion of potential users will require some form of automatic feedback mechanism. The feedback mechanism should be part of the user's environment and be capable of automatically forwarding information to the branch and central libraries when appropriate.
2. Documentation. The working group spent some time trying to understand what documentation should be captured and how to capture it.
- a. There will be differences between what can be expected from documentation captured as part of a software engineering environment and documentation that must be captured off-line after the component is developed.
 - b. We must allow for unconventional documentation facilities (e.g., video or sound) but there must always be some minimal documentation available on all types of output devices. There may also be multiple representations of the properties of a component (e.g., Ada, ANNA, LIL, English), but these must be kept consistent.
 - c. In general, components will be part of a larger whole. Therefore we will need proper configuration management of the context information as well as of components and their associated documentation. There may be many different bodies associated with a particular Ada package specification.
 - d. There is a strong possibility that the system and the user will be subject to information overload. Therefore, the system must provide only what is necessary at each step of use. The system should also be able to generate information to reduce redundancies, and to address the consistency problem that redundant information causes over time.
 - e. Feedback is critical to the system. We must find out what is useful documentation for the many varied uses to which the system will be put, how to best present the information, and where critical information was missing. There are differences between local and global information feedback requirements.
3. Policy and Non-Technical Issues. The working group spent part of the last day talking about some of the issues which must be addressed if a software component library is to be successful. These issues were of a nontechnical nature, but the group felt they may have at least as much impact on the success of such a system as the technical issues.

- a. Pump priming will be necessary. We might require use of the library as part of DoD contracts. Contractors could then be rated on their reuse of components from the library and their contribution to the library as the contract proceeds; contractors could be rewarded for their reuse and contributions. These rewards need not cease at delivery of the user system, but could continue for some period afterwards.
- b. User confidence in the product is critical. Several mechanisms should be available such as software acceptance tools, user experience ratings, a "Good Housekeeping" seal of approval, and software reviews. Degrees of validation of a component along with statistics on critical path and flow analysis should be available insofar as they apply to components. Standard tools should be available to apply to components when they are registered with the library system.
- c. Proprietary issues must also be addressed. An appropriate and effective mechanism for providing economic incentives and royalties would be the best way to encourage library use and insertion. Solutions to this problem must address issues such as the levels of documentation to be provided and the various products to be provided for different fees.
- d. The issue of warranty should be addressed.
- e. The question of who can and who should operate a library and its various branches must be addressed.

1.3.2 Methodology

The Methodology Working Group reached the following main conclusions:

1. We should experiment with standard programming paradigms using the Ada language, including:
 - a. Generic components with a fixed set of data type classes. This will be a low-risk domain-specific approach to reusability.
 - b. Inheritance with limited subclassing. A generic package may implement a parameterized object (as in object-oriented programming). Another package may inherit its capabilities using the `with` clause to override or add data types or operations.
 - c. Nested generics. Outer levels of generic packages may be useful for transforming information between the interior of a component and its outer local environment. Some such mechanism is necessary to incorporate generally reusable components with Ada's linear elaboration. (See also the suggestions for LIL in Section 2.)
 - d. Domain independent components.
2. We should develop standard formalisms for describing components.

- a. This is necessary for cataloging and retrieving.
 - b. The package specification + formalized comments may contain sufficient information (e.g., ANNA).
 - c. Formal (mathematical) specification of components may be used if automated tools can be provided.
3. We should provide user support via automated tools:
- a. Check compliance to standard form. Check for sufficiency of information. Such tools would perform many quality assurance functions as well as aid in document preparation.
 - b. Process information for ease of retrieval (storage, search, display). The structure of the library will depend upon the knowledge representation techniques chosen and upon the ability to acquire and codify that knowledge.
 - c. Knowledge-based retrieval capability. An intelligent library management system will be supported by knowledge-based techniques. In fact, until a formal specification of library components is developed, such techniques will be the only ones available to provide automated assistance.

1.3.3 Library Searching

The Library Searching Working Group concentrated on what kinds of information would be needed to support searching and how this searching might be accomplished. The model adopted consisted of a catalogue or encyclopedia that contains descriptions of the program objects stored in the library. This catalogue could be implemented as a database against which queries can be made to retrieve descriptions of objects. Most of the discussion centered on searching for Ada packages. Although many existing libraries contain Ada subprograms, these were not considered as appropriate units for reuse.

Information in the catalogue must be structured to reflect the way the catalogue is used, and may use many different hierarchies of information to search for objects. For example, in searching for packages one might define a theory (in the sense of LIL) and then search for a package specification (and thus, a package body) that matches the theory. Within the library, there can be many package specs for each theory and there can also be many package bodies for each package spec.

The Library Search Working Group also considered the impact of software component reusability on software engineering methodology. It was recognized that building systems from

reusable components placed a greater emphasis on being able to specify the function and design of a system, so that its parts could be obtained from the library and assembled into programs; the need to support the reuse of designs that are similar to what we want but must be modified before they can be used was also identified. It was felt that the concepts needed to support programming for reusability are new concepts not embedded in most current programming practice, so that considerable training might be needed to effectively use Ada libraries.

The working group felt that each site should have a librarian whose job would be to maintain the Ada library, add new items to the library, and to support library searching. The librarian position would be a senior position requiring substantial expertise; perhaps a strong background in reading formal specifications would be an asset.

1.3.4 Applications

The Applications Working Group reached the following main conclusions:

- There are a number of promising techniques to reusability, each of which will have its role. Among these techniques are:
 1. Component libraries, which will be useful at low levels in a system design or as an interface in connecting larger subsystems.
 2. System Families, which will be useful when the application is well-understood, and it is possible to design a very general system and instantiate it for particular applications.
 3. Very-High Level Languages (VHLL), which should be useful in developing prototypes or in developing systems in specialized areas (e.g., theorem proving, report generation).
 4. Application Generators, which can be viewed as manipulating large components that are user-programmable.
 5. Knowledge-Based Systems, which can be used to develop systems describable in terms of rules operating on large databases. It is likely that the product will have suboptimal performance and be useful primarily as a prototype. Also an expert in the application is essential in order to make effective use of a knowledge-based system.
- The issue of incentives is vital to making reusability a viable, cost-effective technique. DoD will have to "prime the pump" by establishing the organizational structure that will allow reusable products to be developed, to be advertised, to be maintained, to be evaluated, and to be used.

- It will be necessary to conduct several studies in order to evaluate the feasibility of reusability. We declared several candidate applications as most likely to be successful: operating systems, communication protocols, database systems, navigation systems, message systems, and C3 systems. In addition, a study is recommended to define a large collection of lower-level reusable components, each of which is as general as possible.

2 LIL: A Library Interconnection Language for Ada

by Joseph A. Goguen, SRI International

Abstract

This paper discusses problems, concepts and approaches relevant to an on-line library system supporting the creation, documentation and maintenance of Ada software systems. The ultimate goal of research in this area is to make Ada programming significantly easier, more reliable, and more cost effective by using previously written Ada code and previously accumulated programming experience to the maximum possible extent. The main suggestions made in this paper are as follows: systematic (but limited) use of semantics, by explicitly attaching theories (which may be informal) to program units by means of views (a new concept defined in this paper); use of library entities and a library interconnection language (called LIL) to assemble programs out of existing code; maximal use of generic library entities, to make them as reusable as possible; support for different levels of formality in both documentation and validation; and finally, facilitation of program understanding by animating abstract data types and module interfaces.

2.1 Introduction

We envision a library system to be used while building and modifying Ada programs. Because it is unrealistic to expect that all of a user's needs will be met exactly by existing library entities, it is necessary to provide help in retrieving and utilizing the entities that best fit current needs, and in modifying an entity to meet a user's needs. This implies that powerful cataloging and retrieval services should be provided. To insure effective use of the library, it is also necessary to provide help in combining library entities. This will require powerful techniques for appropriately instantiating, enriching, restricting and combining entities, as well as methodological guidelines to ensure the proper use of such techniques. Generic (also called "parameterized") entities are one of the most important ingredients in these techniques. Basic methodological issues include the need for enforcing consistency of data representation and of control flow, when library entities are combined.

An Ada library system should be part of the Ada Program Support Environment (APSE) [DoD 80], [Buxton & Druffel 81]. Ada facilities that support abstraction, including packages, subprograms, generics and controls on exporting types, provide a good basis for both software

composition and for library organization. We believe that recent work on program specification and design, provides a good notation and set of techniques for structuring the use of such a library system. In particular, making (limited) use of semantics seems both valuable and feasible at this time. This would be useful for ensuring that interfaces really match (since Ada itself provides only syntactic information about interfaces). Another semantic issue is utilizing program design information and development history. Techniques from artificial intelligence and information science² might also be useful, for example automatic indexing and cataloging schemes, methods for fast search and retrieval, voice technology, and expert systems to advise users on the selection and application of library components.

It should be noted that we interpret the library concept broadly, so as not to exclude any *information that usefully might be stored and retrieved in the context of a large Ada development project*; some would no doubt prefer a term like "Ada programming environment database" for this concept, since "Ada program library" has been defined as the current collection of compiled units. Possible library entities (in our broad sense) include program units, documentation, specifications, requirements, transformations, design histories, and project status information such as summaries and projections of cost. Although Ada has features that make it particularly suitable for such a library system, the ideas presented here are applicable to the design of programming environments for other languages, and also for language independent environments.

To integrate a diverse collection of library entities, it is important to have precise descriptions for each kind of entity, expressed in a common formalism; otherwise there can be no assurance that the representations and assumptions used in the various components and tools will be compatible. For example, [Cohen & Jackson 83] argue strongly that the European Esprit project should be firmly based upon a formal ground. [Goguen 83] gives a more formal account of the language design principles that we use; see also [Goguen & Meseguer 84a, Goguen & Meseguer 84b].

²This is the area that applies techniques like Shannon's theory of information to determine optimal ways of organizing information. Some results are briefly discussed in Section 2.5.4.

2.2 Issues and Approaches

We believe that a better understanding of certain basic issues is needed in order to provide an adequate basis for an Ada library system. Without such a basis, there is serious danger of building library systems that are inconsistent, hard to use, and hard to modify³. These general issues include the following:

1. What should be in a library? Possibilities beyond just compiled Ada program units include the corresponding uncompiled Ada texts (especially the interface information provided by the specification parts of generics), version and configuration information, requirements, specifications, documentation, transformations, histories and management information.
2. What techniques for program composition take maximum advantage of the features of Ada and of the library concept? Candidates include instantiating, enriching and restricting entities.
3. How to construct families of related programs? (Would program transformations and expert systems be useful in this regard?)
4. What documentation and specification techniques produce intuitively clear and mechanizable descriptions of a program's functional behavior and external interface?
5. How to best identify the library entities that are most relevant to a user's needs? What cataloging services (e.g., taxonomies) and reference services (e.g., search strategies) should be provided?
6. How to integrate libraries into an APSE (e.g., with module test, linkage and interpretation facilities)?
7. How to best present information to users? Possibilities include multi-media support (e.g., graphics and natural language) for various modes of system use, including program composition, retrieval (e.g., clever use of menus and icons), documentation and modification.
8. What about management issues, such as policies for investment, quality control, and distributing and encouraging documentation?
9. What experiments could be performed to test the viability of various approaches to these problems?

These issues divide into four major categories and are treated in the four following sections:

³These dangers are much less acute for libraries in semantic domains that are already very well understood, such as numerical algorithms.

Section 2.3 addresses the basic issue of what to put into a library; Section 2.4 considers programming methodology, to optimize the use of Ada and of the library; Section 2.5 considers library organization; Section 2.6 discusses user interface and management issues.

This paper suggests some ideas for dealing with the problems listed above. The main ones are enumerated below, with descriptions that are not intended to be self-contained; further details are given later.

1. Systematic (but limited) use of **semantics**; in particular, explicitly providing **theories** (which are just sets of axioms) attached to program units via **views** (see Section 2.4.4 for this term).
2. A variety of different methods for program construction, so that the process of programming will consist, as much as possible, in the application of these methods, rather than in just writing code; we call this **hyperprogramming**⁴.
3. Maximal use of **generic** (i.e., parameterized) library entities. This is intended to make them as reusable as possible.
4. Support different levels of formality in axioms, and degrees and kinds of **validation** (such as informal arguments, testing, and formal proofs); this should support a practical user interface and also aid in pinpointing weakspots during debugging (see Section 2.5.1 for an explanation of this).
5. Facilitation of program understanding by **animating** abstract data types, and otherwise illustrating and explaining behavior at module interfaces (see Section 2.6 for further explanation).

It should be observed that because this paper focusses on libraries, the main issue that it addresses is reusability. This means that while broad issues of programming methodology are discussed, many more detailed issues are ignored. Issues having some relevance to reusability but ignored in this paper include some difficult ones, such as tasking and exceptions, as well as *some relatively simple ones, like the Ada modes in, out and in out.*

⁴Programming-in-the-large refers to manipulations at the module level rather than the code level. The term **hyperprogramming** is intended to reflect an integration of programming-in-the-large with programming-in-the-small.

2.3 Library Content

What should be stored in a library? For example, it seems clear that complete compiled operating systems should *not* be stored; but it would be useful to store elements out of which a variety of different (but related) operating systems could be constructed. And it would be useful to store design (e.g., configuration) information for constructing particular operating systems. At the other extreme, a library should not bother to store individual program statements. However, it does make sense to store Ada packages, both as Ada source code and as compiled machine code.

Much recent research (some of it is cited below) supports the position that it isn't enough to store code: some of the knowledge that went into constructing the code should also be stored, particularly for large systems. For example, one should have not just the components of some operating system, but also explicit information about how to put those component together to construct the system. Information on which versions of those components to use is also needed in large system development efforts. Moreover, the same components (or different versions of them) can be used to put together slightly different operating systems. This kind of information about configurations and versions can be expressed in a module interconnection language (abbreviated "MIL" below; see [Prieto-Diaz & Neighbors 82] for a good survey of this field) and should be part of a library support system.

Ada itself provides some support for this with the specification part of a package or subprogram, and with **with** and **separate** clauses. However, these often tie units that ought to be more reusable to contexts that are far too specific. Another consideration is that it is often desirable to document what each part is supposed to do; this information might consist of formal specifications (i.e., sets of axioms) and/or less formal descriptions. The latter might be related to system requirements, which could also be stored in the library. Unless such knowledge of design objectives and decisions is stored with the code, it will not be available at later stages of the software life cycle; note that much of the cost of system maintenance is due to the difficulty of understanding existing code.

Ada's package construct is an especially nice way to modularize code. But formal specifications, documentation, designs and requirements should also be modularized, since they too will be difficult or impossible to understand if presented monolithically, especially for large

systems. We suggest that constructs similar to the Ada package should also be used for this purpose. Then a module interconnection statement could be used to assemble not only the final code, but also complete documentation and specifications for it.

Another general remark is that whatever is put into a library should be as widely applicable as possible. Ada generics provide a powerful mechanism for this at the code level, and we believe that similar mechanisms should be provided for specifications, designs, requirements, and documentation.

An elegant way to unify many of the above considerations is to view the process of programming as one of transforming a high level specification into lower level executable code [Burstall & Darlington 77], [Cheatham 83], [Scherlis & Scott 83], [Green *et al.* 81], [Feather 82], [Balzer 81], [Standish 83], [Goguen & Burstall 80]. The design history is then available as a sequence of transformations that can be applied to get the final system. Supporting this view of programming in an Ada environment would require the library to store both transformations and design histories. The latter could then be manipulated to construct other related systems, and in particular to reconstruct a given system after bugs have been corrected in its specification, or other modifications introduced. This should be especially useful during the maintenance phase of the software life cycle.

We will use the term **entity** to refer to anything that is stored in the library in a systematic way. Entities so far mentioned in this paper include program, subprogram, package, specification unit, documentation unit, requirement unit, transformation, and transformation sequence.

Figure 1 gives a tentative taxonomy of entities that might be stored in an Ada library system. The three main categories are: concrete, abstract, and managerial. Of course, having a place in this taxonomy does not mean that an entity should necessarily be included in a library. Notice that managerial information can be derived from abstract information, assuming that appropriate data is available from abstract entities, such as number of lines, date of completion, number of man-hours expended, performance and testing results, etc.

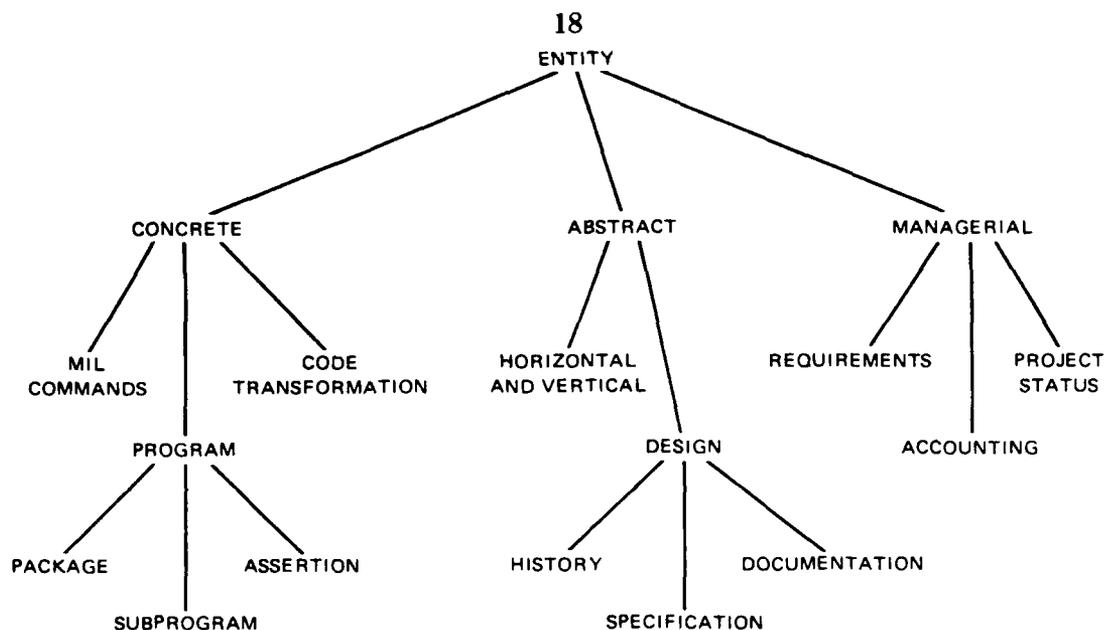


FIGURE 1 TAXONOMY OF LIBRARY ENTITIES

2.4 Program Composition

In brief, our approach to program composition is to provide a module interconnection language based upon a generalization of Ada's specification part for program units, enriched with commands for interconnecting components to form systems. This language, which we call LIL⁵ (for Library Interconnection Language), addresses certain important problems that cannot be handled by a simple Ada compiler⁶. It is only natural that Ada does not address these issues, since they arise at design time rather than at compile time, and require a "design database." These issues include the following:

1. Narrowing the interfaces of generics; for example, permitting the interface to demand particular user-defined types (such as **STACK**);
2. Allowing more flexible binding of compilation units (for example, composing two generics to get a third);
3. Allowing interactive version and configuration management;
4. Integrating hierarchical (which we call "vertical") design methodology (as in abstract or virtual machines [Parnas 72a]) with the "horizontal" structuring capabilities of a module interconnection language; see Section 2.4.6.

⁵This paper is in effect a preliminary design document for LIL; the language has not been implemented, nor even designed in detail at this time.

⁶However, we do not discuss all desirable features of a module interconnection language here. Instead, we emphasize features relating to program structure.

Of course, the main point, already addressed by an Ada compiler's library of compiled program units, is to avoid the time and confusion involved in recompiling every program unit of a large system every time components are changed or added, either during the construction phase or later during the maintenance phase; one wants separate, independent, incremental compilation. Thus, LIL packages and **make** clauses are (respectively) intended to designate and to manipulate compiled Ada units. It is our intention to do this with much greater power and flexibility than pure Ada can, and thus to support a much more powerful and flexible programming methodology. One key is that more than one Ada body can correspond to a given LIL package; these are its versions. For example, a generic list package **LIST[X]** might have three different bodies, **LIST.1**, **LIST.HACK**, and **LIST.EFFICIENT**. One could even go a little further, and allow versions in multiple programming languages. For example, one might have **LIST.1.ADA** and **LIST.1.PASCAL**. Of course, for an Ada library system, the default will be **.ADA**. LIL syntax is close to Ada syntax (but a little more abstract), and LIL statements can easily be translated into pure Ada for input to a compiler. It will be seen that Ada provides amazingly appropriate support for all this.

Some popular high level paradigms for program construction include top-down (from requirements), bottom-up (from selected library modules), transformation from an initial prototype or high level specification, instantiation of a existing generic program component, "data driven" programming in the sense of [Jackson 75] (which is really a systematic use of data abstraction), and dataflow [De Marco 78]. It is our intention to support all of these paradigms by providing high level specifications and very general parameterization mechanisms.

Among the many different specific techniques that might be useful under various circumstances for constructing new entities from old ones, are the following:

1. Set a constant (such as the maximum depth of a stack);
2. Substitute one entity for a "stub" or parameter in another;
3. Sew together two (possibly large) entities along a common interface;
4. Instantiate the parameters of a generic entity;
5. Enrich an existing entity with some new features;
6. Hide (abstract, or encapsulate) some features of an existing entity; this could include both data abstraction and control abstraction;
7. "Slice" an entity, to eliminate some unwanted functionality;
8. Implement one abstract entity using features provided by others (this leads to the notion of a *vertical hierarchy* of entities); and

9. Assemble existing entities over a skeleton. This skeleton might be either fixed or flexible; for example, it might be determined heuristically by an expert system. See Section 2.6.

The first four of these techniques each involve parameterization, a notion that is becoming well understood and to a considerable extent incorporated into modern programming and specification languages (e.g., Ada generics, and work of [Burstall & Goguen 77] on parameterized specifications in Clear). In fact, the first three parameterization methods can be seen as special cases of the fourth.

It is useful to distinguish between **horizontal** and **vertical** modes of composition: the latter has to do with the top-down and/or bottom-up hierarchy of levels of abstract machines; while the former has to do with modularization at a given level of detail; this is discussed in terms of an example in Section 2.4.6. A unified treatment of the modes of construction listed above is suggested by [Goguen & Burstall 80], using generic modular transformations that handle data representations and their associated operations, as well as more conventional transformations of program constructs such as recursion. The Ada package can be seen in this light as a method for realizing a given functionality if certain other functionality is provided (again see Section 2.4.6); notice that semantic specifications are needed in order to say exactly what functionalities are involved here.

There has been much previous inspiring work on module interconnection languages. The INTERCOL system [Tichy 79, Tichy 80] was specifically designed for Ada, and has also influenced the interconnection part of Gandalf environment [Habermann & Perry 81]. In comparison with LIL, INTERCOL and Gandalf lack the ability to specify (formally or informally) the semantics of packages, and also fail to distinguish between vertical and horizontal modes of composition. However, they do allow for module revisions and derived versions, and have experimental implementations.

Three concepts that go beyond Ada generic packages and still seem ripe for near term implementation are theories, views and interconnection commands. **Theories** are used to declare properties required of an actual parameter for it to be meaningfully substituted for the formal parameter of a given generic entity. **Views** are used to express that a given entity satisfies a given theory in a particular way (this is necessary because it is possible for some

entities to satisfy some theories in more than one distinct way). **Interconnection commands** are used to express how a system is to be built out of its components; these include **instantiating** a generic entity, using a particular view to produce a new entity. This approach to parameterization is inspired by the Clear specification language⁷, and is applied in [Goguen 83] to the OBJ programming language. A related topic is transformations, which can modify entities by adding, deleting, or renaming functionality, and can themselves be parameterized.

Before embarking on the technicalities of the following subsections, it seems a good idea to contextualize them with the following points, some of which are only explained in detail later:

1. We deliberately use a syntax for LIL that is closer to mathematics than Ada is.
2. The ordinary user of such a library system would probably not see LIL entities in the forms given below, which have been chosen to facilitate discussion of their basic properties. Rather, the user interface would hopefully involve natural language and/or interactive graphics. See Section 2.6.
3. Many features of Ada are not covered in the discussion below. In some cases, this is merely to simplify the discussion; for example, we discuss only functions, but procedures present no serious difficulties, and are illustrated in the example in Appendix III⁸. In other cases, further research would be needed to provide an adequate treatment; exceptions, access types, and tasking fall into this category.
4. We do not anticipate that most development projects will use of large or complex formal theories; rather, they would rely upon informal documentation and informal arguments about program properties. However, some applications might demand such a high standard of reliability as to warrant the effort of full mathematical verification. The approach suggested here would support both formal and informal specification and verification in an integrated manner, allowing whatever mixture seems most appropriate to the application.
5. Although the outlines of the project seem clear enough, and some parts, particularly LIL, could be implemented without much further research, there are other areas, such as transformations, that would require substantial further thought to provide an adequate foundation.

⁷In particular, the notion of view was developed in collaboration with R. M. Burstall for use in Clear, although it has not yet been published in that connection. Clear's approach was in turn inspired by some ideas in general system theory [Goguen 71].

⁸See [Goguen 82] and [Goguen & Meseguer 82a] for further examples and foundations.

2.4.1 Packages and the Using Hierarchy

The most important entity in LIL is the **package**, a generalization of the specification part of an Ada package⁹. There are two main ways that a LIL package differs from an Ada specification part: (1) **axioms** can be given for the operations that it declares; and (2) it is associated with (zero or more) **versions**, which are Ada packages that realize the behavior it describes. Thus, both semantic specification and version control are supported. Some additional capabilities of generic packages are discussed in the next subsection. The main point is that LIL packages help the library keep track of already compiled Ada code.

We wish to emphasize that the axioms in LIL packages do **not** have to be formal; they can be semiformal mathematics, or even informal natural language (see Section 2.5.1). It is also unnecessary to formally verify the axioms against code. We believe that to impose such a requirement would render a library system impractical except for a few applications requiring especially high reliability. However, we do propose a "truth management" system that will track the degree to which various assertions have been verified (again, see Section 2.5.1 for details); such a system would be useful for both testing and debugging.

New entities are often defined using others already defined, and possibly already compiled. We wish to carefully distinguish **using** entities from the process of instantiating generic entities, as well as from the vertical (hierarchical) development process in which an entity needs others in order to realize a given behavior (see Sections 2.4.3 and 2.4.6 respectively).

Because LIL's notion of horizontal hierarchy differs slightly from that of the Ada **use** clause, we introduce a slightly different notation, the **using** clause¹⁰. This is illustrated in the following example:

```

package COMPLEX_FUNCTIONS
  using MATH_FUNCTIONS is
  types COMPLEX

```

⁹LIL does not distinguish between specification and body parts of packages, because it does not provide executable code as such; that is given in Ada.

¹⁰Use imports Ada packages, whereas using imports (partial) descriptions. While an Ada function or procedure, once written, cannot be modified, a description *can* be modified, for example by adding new axioms.

functions

```
** : COMPLEX COMPLEX -> COMPLEX
```

```
.....
```

axioms

```
-- ** agrees with real exponentiation for real arguments
```

```
-- E ** I*R = COS(R) + I*SIN(R)
```

```
.....
```

```
end COMPLEX_FUNCTIONS
```

This package has a **using** clause that imports another package to provide the standard real mathematical functions; both the axioms and the associated Ada code can use these functions. Notice that this package has one axiom that is stated informally, and another that is stated formally but without formal declarations for its variables and constants; also note that these axioms will not hold exactly for any Ada code intended to realize it, because of roundoff error¹¹. The package **STANDARD** is considered to have been imported into every package with a **using** clause; all its operations are thus freely available.

Information hiding, as advocated by Parnas and others, can be accomplished by **hide-types** and **hide-ops** clauses; these render the mentioned types and ops invisible in any entities using the given package, even if the mentioned types and ops had been imported from another entity. The **using** and **hide** clauses lead to an acyclic graph expressing inclusions of the corresponding syntactic interfaces and sets of axioms of entities.

We conclude this subsection by noting another difference from Ada package specification parts: representations are not stated in LIL packages. The Ada code corresponding to the **COMPLEX_FUNCTIONS** package presumably represents the **COMPLEX** type as a record of two real types, and the Ada compiler surely needs to know this. But this information is hidden in LIL; in fact, it is not needed for describing which Ada units are to be interconnected, which is the primary purpose of LIL.

¹¹Somewhat different axioms could, however, take account of roundoff error.

2.4.2 Theories

Another basic entity of LIL is the **theory**, which expresses properties of entities or entity interfaces, but does not have any actual Ada code associated to it; its purpose is purely to express properties¹². In general, theories can be structured in the same ways as programs; the difference is that program units define executable structure, while theory units define not necessarily executable properties. In particular, theories can use other theories, can use data abstractions, can be parameterized, and can even have views (see Section 2.4.4 for this term).

Our first example is the trivial theory, **TRIV**, having just a single type **ELT** about which nothing is asserted.

```
theory TRIV is
  types ELT
end TRIV
```

For another example, the theory of partially ordered sets has a single type designated **ELT** plus a binary infix relation \leq that is reflexive, transitive and anti-symmetric. This theory is useful in describing the ordering among security classifications (as associated with classified documents) or in describing the interface of a sorting package, that is, in specifying the semantic requirement that the elements to be sorted have a suitable ordering relation.

```
theory POSET is
  types ELT
  functions  $\leq$  : ELT ELT -> BOOLEAN
  vars E1 E2 E3 : ELT
  axioms
    (E1  $\leq$  E1)
    (E1  $\leq$  E3 if E1  $\leq$  E2 AND E2  $\leq$  E3)
    (E1 = E2 if E1  $\leq$  E2 AND E2  $\leq$  E1)
end POSET
```

Finally, the theory of monoids. This will serve as a parameter requirement theory for a generalized iterator that can yield sums, products and other operations over lists (see Section 2.4.5).

¹²More technically, the axioms in a LIL package are to be interpreted "initially," i.e. in a standard model, while those in a LIL theory need not be so interpreted.

```

theory MONOID is
  types M
  functions * : M M -> M (assoc, id: I)
end MONOID

```

Here **assoc** indicates that the function ***** is infix and associative, i.e., satisfies the equation

$$(M1 * M2) * M3 = M1 * (M2 * M3)$$

and **id: I** indicates that it has an identity **I**. We mention again that theories can be defined and used even if not all their axioms are formal. Also note that **hide-types** and **hide-ops** can be used in LIL theories as well as in packages; moreover, **hidden** can be given as an attribute of an operation with the effect of hiding it.

2.4.3 Generic Entities

Ada lacks the ability to define semantic restrictions in the specification part of a generic; this is appropriate since an Ada could not use such information anyway. However, we wish to provide this information in the library system to increase reliability and to aid with problems of program understanding, retrieval and composition. The requirements that actual parameters of a generic entity should satisfy for the instantiated entity to behave as desired are given in theories. Of course, these theories must be defined before the entity can be instantiated. Again, we use a syntax closer to mathematics than Ada's syntax, to help distinguish these more abstract entities from the corresponding pure Ada entities¹³. To illustrate, here is a LIL generic list package:

```

generic package LIST[ELT :: TRIV] is
  types LIST
  functions
    . : LIST LIST -> LIST (assoc, id: NIL)
    EMPTY : LIST -> BOOLEAN
    HEAD : LIST -> LIST
    TAIL : LIST -> LIST
  vars E : ELT; L : LIST
  axioms
    HEAD(E . L) = L

```

¹³Of course, this is not essential, and a notation closer to pure Ada could be accommodated without any difficulty.

```
TAIL(E . L) = E
```

```
.....
end LIST
```

The attributes **assoc** and **id** of "." implicitly give some further equations, namely the associative law and two identity laws.

A major difference between this and the specification part of an Ada generic package is that all the parameters are collected together in one entity, called the **requirement theory**, enclosed in [...] after the package name, **LIST** here, telling not only what types, functions and procedures are needed, but also what properties they must satisfy. In this case, the formal parameter **ELT** must satisfy **TRIV**, i.e., no axioms. For a sorting package given later, the parameter must have a binary relation that is a partial ordering, i.e., it must satisfy the theory **POSET**. Here is an example of a parameterized theory, the theory of vector spaces over a field **F**.

```
generic theory VECTOR-SP [F :: FIELD] is
  types V
  functions
    + : V V -> V (assoc, comm, id: 0)
    * : F V -> V
  vars F F1 F2 : F;
       V V1 V2 : V
  axioms
    ((F1 + F2) * V = (F1 * V) + (F2 * V))
    ((F1 * F2) * V = (F1 * (F2 * V)))
    (F * (V1 + V2) = (F * V1) + (F * V2))
end VECTOR-SP
```

More generally, entities can have several parameters, indicated in the form

```
[X :: TH1; Y :: TH2]
```

and theories (with their corresponding formal parameters) can involve more than one type.

One can also write a package **using** the instantiation of a generic. The instantiation of generic entities is discussed in Section 2.4.5. The semantics of generic packages can be described by the methods of Clear, as discussed by [Litvinchouk & Matsumoto 83].

2.4.4 Views

The purpose of a view is to explicitly show *how* a given entity satisfies a given theory. For example, if **SORT** is a generic package for sorting **LISTS** of its parameter type (which must satisfy **POSET**), writing **SORT[NATURAL]** to define a package for sorting **LISTS** of **NATURALS** may be ambiguous, because there are many different order relations that could be used on the natural numbers. The most obvious is the usual "less-than-or-equal", but "divides" and "greater-than-or-equal" are other possibilities. Thus, a view of **NATURAL** as a **POSET**, which we write **POSET => NATURAL**, indicates just which order is to be used; the three choices of order mentioned above correspond to three different views. Note that **NATURAL** here denotes the LIL package, not the Ada type **NATURAL** in the Ada **STANDARD** package, which is here regarded as a version that realizes the LIL package.

More precisely, a **view** of an entity **A** as a theory **T** consists of a mapping from the types of **T** to the types of **A**, and a mapping from the operations¹⁴ of **T** to the operations of **A** preserving arity (which is the list of argument types), value type (if any), and operation attributes such as **assoc**, **comm** and **id**: (if any), such that every axiom in **T** is satisfied by **A**. Such a mapping of types can be expressed in the form

```
types (T1 => T11)
      (T2 => T21)
      .....
```

and the mapping of operations in the form

```
ops  (OP1 => OP11)
      (OP2 => OP21)
      .....
```

Thus, each mapping consists of two sets of pairs in what Ada calls "named parameter notation." Together they are called a **view body**. The syntax for defining a view adds to this names for the source and target entities, and a name for the view. For example,

```
view NATD :: POSET => NATURAL is
  types (ELT => NATURAL)
  ops (< => DIVIDES)
```

¹⁴We use the word **operations** to refer to either functions or procedures.

end NATD

defines a view called **NATD** of **NATURAL** as a **POSET**. We use the double colon **::** to emphasize that this concept is at a higher level of abstraction than that of just operations.

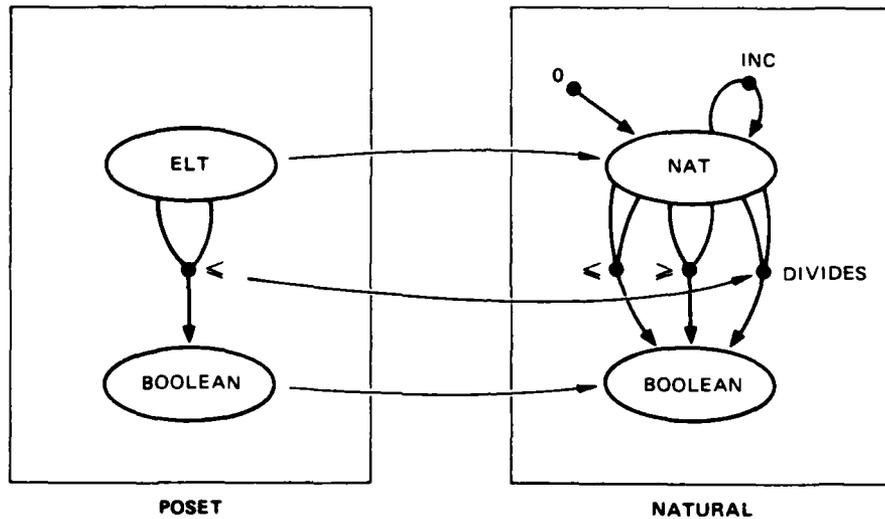


FIGURE 2 THE VIEW NATD:POSET \Rightarrow NATURAL

Each package and theory has a default view as **TRIV** using its first type (or the first type of the first entity that it is built upon if it doesn't have a first type itself, and so on backward recursively); this type is called the **principal type** of the entity. A **default view** is the one that is used unless another is explicitly provided instead. For example, if we were to write out the default view of **NATURAL** as a **POSET**, it would be

```
view NATV :: POSET => NATURAL is
  types (ELT => NATURAL)
  ops (< => <=)
end NATV
```

Thus, when there is just one type in the source theory **T** and the **types** line of a view is omitted, it is assumed that the type of the theory is paired with the principal type of the target. Moreover, pairs of the form **(T => T)** can be omitted from any view. There is a similar default convention for **ops**, namely correspondences of the form **(OP => OP)** can be omitted. For example, if the default view of **NATURAL** as a **MONOID** (according to these conventions) were written out in full, it would be

```

view NAT* :: MONOID => NATURAL is
  types (M => NATURAL)
  ops (* => *)
    (I => 1)
end NAT*

```

assuming that the LIL package for **NATURAL** tells us that **1** is an identity for *****, and then using the convention that a default view should preserve the **id:** attribute. The following is a non-default view of **NATURAL** as a **MONOID**.

```

view NAT+ :: MONOID => NATURAL is
  ops (* => +)
    (I => 0)
end NAT+

```

where **(I => 0)** could also be omitted by preservation of the **id:** attribute.

Finally, a view that involves a parameter:

```

generic view LISTM :: MONOID => LIST[X :: TRIV] is
  ops (* => .)
    (I => NIL)
end LISTM

```

A further generalization of the default rule permits omitting an operation pair of the form **(OP => OP1)** if the arity and type of **OP1** equal the translations (under the **types** mapping) of those of **OP**, and if **OP1** is the *only* operation (in its entity) having that particular arity and type.

A similar concept is used in Clu [Liskov et al. 79], where it is called a "binding." However, Clu bindings do not involve attributes or axioms, and provide a much weaker notion of default binding.

2.4.5 Instantiation

To actually use generic entities, it is necessary to instantiate their parameters with actual entities. This subsection shows how to do this with the **make** command, that uses a view to bind an actual to a formal. This is the command that makes LIL a module interconnection language for Ada, permitting more than one version to correspond to a given specification part (thus supporting "version management"), and permitting more than one way of organizing a given collection of program parts to co-exist in the library (thus supporting "configuration management").

For example, `SORT[X :: POSET]` can be instantiated using the view `NATD` from Section 2.4.4 by

```
make SORT-NATD is SORT[NATD] end
```

to get a package that sorts lists of `NATURALS` by the divisibility relation.

If the name of an entity is used instead of a view, the default view (if there is one) from the requirement theory of the parameterized entity to the named entity will be used for the binding. For example,

```
make NATLIST is LIST[NATURAL] end
```

uses the default view `TRIV => NATURAL` to instantiate the parameterized entity `LIST` with the actual parameter `NATURAL`. Similarly, we might have

```
make REAL-LIST is LIST[REAL] end
```

where `REAL` is the field of real numbers, using a default view `TRIV => REAL`; also

```
make REAL-VSP is VECTOR-SP[REAL] end
```

uses a default view `FIELD => REAL`, and

```
make REAL-VSP-LIST is LIST[VECTOR-SP[REAL]] end
```

uses two nested default views.

An example involving a default view between theories is given in Section 2.4.6: `LIST[X]` is introduced into `SORT[X :: POSET]` by a `using` clause, and takes the default view `TRIV => POSET` to ensure that `X` fits in `LIST` if it fits in `SORT`.

Here is an example that has some interesting instantiations:

```
generic package ITERATE[M :: MONOID]
using LIST[M] is
ops ITERATE : LIST -> M
vars E : M ; L : LIST
axioms
  (ITERATE(NIL) = I)
  (ITERATE(E . L) = E * ITERATE(L))
end ITER
```

using the default view `TRIV => MONOID`. We now use this entity in two other examples:

```
make SIGMA is ITERATE[NAT+] end
```

sums a list of numbers, i.e., computes $\Sigma L = \sum_{i=1}^n L_i$ where $L=L_1 \dots L_n$, while

make PI is ITERATE[NAT*] end

multiplies a list of numbers, i.e., computes $\Pi L = \prod_{i=1}^n L_i$ (noting that $\Sigma(\text{NIL}) = 0$ and $\Pi(\text{NIL}) = 1$). We think that these are impressively concise and clear ways to get these functions.

2.4.6 Package Stubs

Ada supports top-down program development through the use of body stubs. An Ada library system should support the further capability of *reusing* subunits, which is prevented by Ada's **separate** clause. Even in bottom-up development, Ada requires that a new program unit refer to something already compiled and in the library through a **with** clause; unfortunately, this significantly limits the reusability of the new program unit. This subsection suggests a more general notion of stub, based on the concept of view introduced previously, that overcomes these limitations. This notion is another part of our Ada library interconnection language LIL.

A capability missing from Ada, but that would sometimes be very useful, is to *compose* generic entities. For example, in Ada one cannot compose a generic **LIST[X]** with another **QUEUE[Y]** to get something corresponding to **LIST[QUEUE[Y]]**. One can form something corresponding to¹⁵ **LIST(QUEUE(INTEGER))**, although even this cannot be done in one step, but requires at least two. LIL supports the composition of arbitrary generics.

We noted above that the Ada package concept embodies the structural relationship of realizing one abstract machine (hereafter abbreviated "AM") with one (or more) others; this can be seen as a promise that a new behavior can be realized if certain other behaviors are provided. This relationship expresses the most important and characteristic vertical activity. LIL indicates interface requirements that need to be satisfied by a lower level AM with a **needs** clause. For example, here is a generic sorting package that **needs** a generic list package **LISTP**.

```
generic package SORT[X :: POSET]
  needs LISTP :: LIST[X] is
  functions
    SORT : LIST -> LIST
    SORTED : LIST LIST -> BOOLEAN
```

¹⁵We use parentheses in this expression, rather than square brackets, because it is a putative -- although not a real -- Ada expression.

```

vars L : LIST
axioms
    SORTED(SORT(L)) = TRUE
    .....
end SORT

```

Let us try to better understand vertical and horizontal structure in terms of this example. The components to be used are shown in Figure 3.

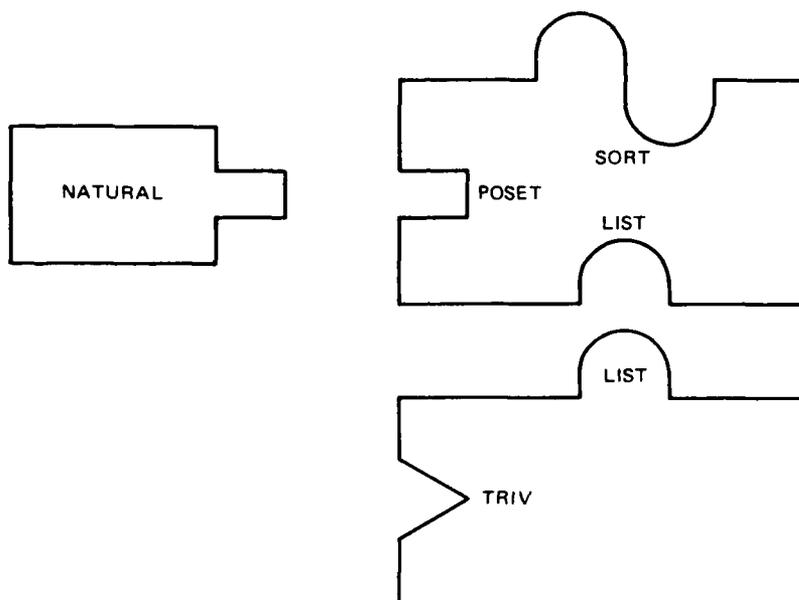


FIGURE 3 SOME SOFTWARE COMPONENTS

We will begin with the vertical structure. The **needs** clause in the generic **SORT** says that before a generic sorting function can really be supplied, we need to supply a generic Ada package **LISTP** that is a version of the LIL package **LIST[X]**; the fact that **X** is both the formal parameter of **SORT** and of **LIST[X]** indicates that the version is instantiated with the same **X** as **SORT[X]**. The advantage of this approach is that a generic Ada body for **LIST[X]** can now be reused, which would be impossible with the Ada **separate** clause. To actually get a version of **LIST** for use in **SORT**, one gives a module interconnection command indicating which version to use when compiling. LIL's syntax for this is

```

make SORT[X] needs LISTP => LIST.HACK end

```

where `LIST.HACK` is a particular generic body for the LIL generic package `LIST[X]`, i.e., a particular version of the `LIST[X]` capability. See Figure 4. The entity following the `::` in a `needs` clause is generally a LIL package¹⁶, as in the example above. Any actual horizontal parameters for the main package (`SORT` in this example) will also be supplied to the package version in the `needs` clause (here, `LIST.HACK`). This automatic management of the interactions of horizontal and vertical structure is one of the most novel features of LIL, and can greatly simplify the programmer's task in some cases.

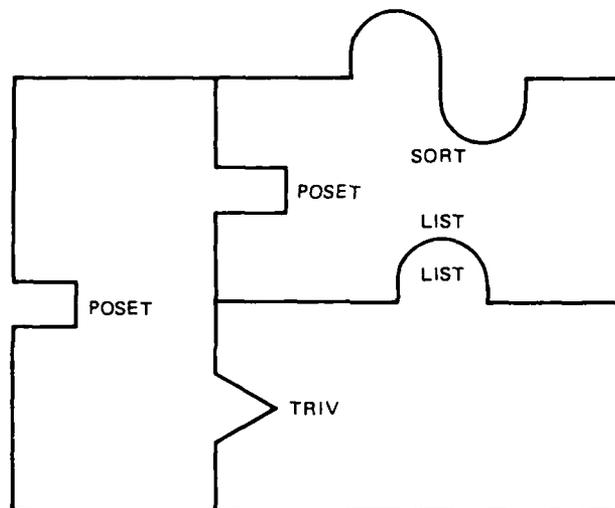


FIGURE 4 A VERTICAL COMPOSITION

The horizontal component of this example appears in the formal parameter `X`, which is required to satisfy the `POSET` theory. See Figure 5 for their horizontal composition.

A `make` command can accomplish both the vertical and the horizontal instantiation of `SORT` at once:

```
make SORT-NATD is SORT[NATD] needs LISTP => LIST.HACK end
```

See Figure 6.

¹⁶It could also be a LIL theory, but this case is not discussed in this note.

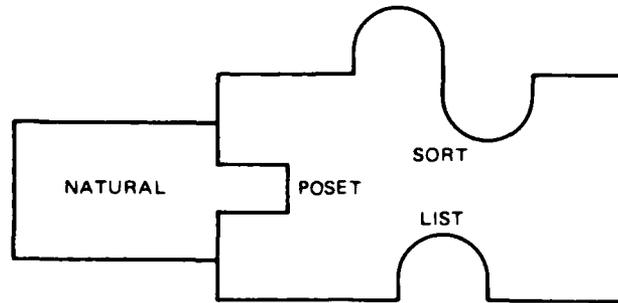


FIGURE 5 A HORIZONTAL COMPOSITION

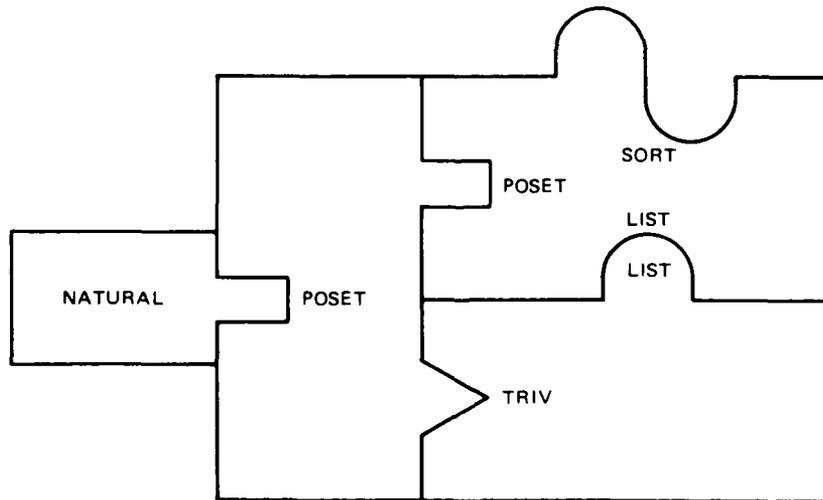


FIGURE 6 A REALIZATION OF SORT{NATURAL} WITH LIST{NATURAL}

Some appreciation of what is involved in a semantics for these constructs given with the methods of Clear can perhaps be gleaned from Figure 7, showing how the requirements theory of a generic package must be a subtheory of the AM that is realized by that package, and may

also be a subtheory of the stubs for the AMs that go into realizing it. Here **IN1** and **IN2** are the requirement theories for the lower level AMs; these are included in the body theory of the package. In addition, there is a theory of the behavior that is actually exported by the package; this AM may not be identical with the body theory because of some information hiding.

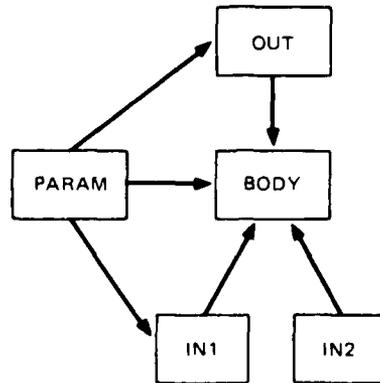


FIGURE 7 THEORIES INVOLVED IN A GENERIC PACKAGE

2.4.7 Environments

In a programming language, each statement, indeed each symbol, is interpreted in an "environment," which consists of the currently defined symbols and their current values (e.g., in a symbol table). Similarly, in LIL, there is an "environment" consisting of the currently defined entity names and their values; but there is also further information (not found in the "environments" of programming languages), namely relationships among these entities. More specifically, the inclusions of one theory or package in another, and also views, should be part of a LIL environment. These constitute, at any point in the processing of a LIL text, a diagram which the user might want to see displayed. For example, Figure 8 shows such a generalized environment for some of the packages, theories and views in the examples given in this paper. Views are drawn with solid arrow heads; the other arrows are inclusions of the kind shown in Figure 2.

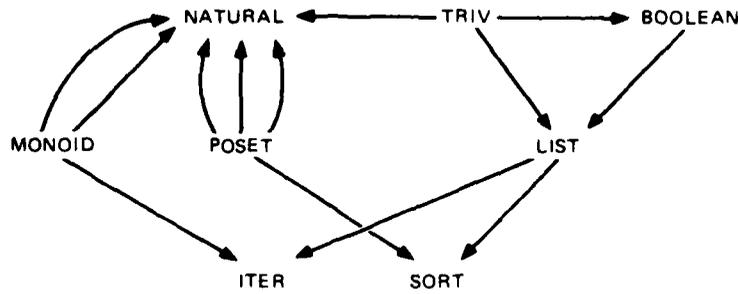


FIGURE 8 A LIL ENVIRONMENT

2.4.8 Transformations

The word "transformation" has been used both very loosely and in a wide variety of different more specific ways. Recall that "hyperprogramming" is our term for the methodology being sketched here. Figure 9 gives a tree for the taxonomy of activities that are involved in hyperprogramming. We hope that this classification scheme will aid in understanding and organizing various ideas already in the literature.

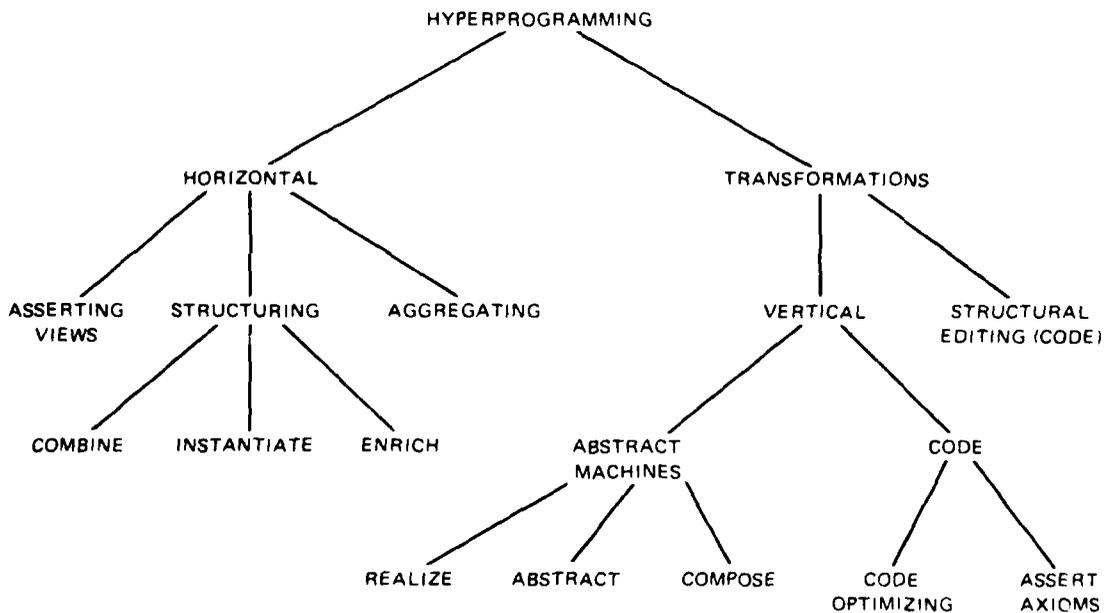


FIGURE 9 HYPERPROGRAMMING TAXONOMY

Let us begin with activities that are *not* transformations. These all fall under the "horizontal" classification and consist of activities that introduce structure into a specification, design or program; the inverse activity of unstructuring, or aggregating parts into wholes, is also considered horizontal (aggregation is sometimes needed before applying an optimizing transformation that involves more than one part), as is the activity of asserting a view, i.e., asserting that an entity satisfies some theory.

It should be noted that these structuring operations do not involve any commitment to either top-down or bottom-up development. Components can be put together (this does not mean aggregated) to form a structured whole; or an unstructured part can be broken into parts. However, it is often more natural to use the idiom of top-down development.

The activity of instantiating a generic has already been considered. It is important enough among horizontal activities to be called "horizontal composition" [Goguen & Burstall 80]. The reason for this importance is simply that the more generic entities are, the more they can be reused.

Perhaps the most basic horizontal activity is to **combine** two (or more) entities, which just means to consider them together. (For example, one might wish to have both **STACK[X]** and **ARRAY[X]**; the combination of these two is might then be written $C[X] = \text{STACK}[X] + \text{ARRAY}[X]$.) A particular point that must be handled carefully in this regard is that of **shared subentities**. These may arise implicitly through use of built-in entities (such as **INTEGER** and **BOOLEAN**) or explicitly by a **using** clause. For example, if both **STACK[X]** and **ARRAY[X]** involve **BOOLEAN**, we want to be sure that **C[X]** involves only one copy of **BOOLEAN** rather than two. Similarly, the formal parameter **X** (and its implicit requirement theory **TRIV**) are used in both **STACK[X]** and **ARRAY[X]**, and we want that **C[X]**, as the notation suggests, also has formal parameter **X** with requirement theory **TRIV**.

Two further horizontal activities are **enrich** and **derive**. The first of these adds some new functionality to an existing entity, while the second derives it from existing functionality. The activities discussed above are all part of the Clear specification language, and have been given a formal semantics in [Burstall & Goguen 80]. Since specifications give semantics for code, these formalisms can also be applied to programs. Indeed, it is our contention that the best way to

manipulate code is to manipulate the corresponding specifications, or even the corresponding informal documentation.

The purpose of all the above activities is to provide structure (i.e., design information) at a given level of abstraction. Activities that introduce concrete detail (called "commitments" by [Scherlis & Scott 83]) are discussed under our second major heading, that of transformations *per se*. We subdivide transformations further into two subcategories, those that preserve semantics and those that may not. Those transformations that preserve semantics we will call **vertical** activities, and we further divide these into two classes, corresponding to programming-in-the-large and programming-in-the-small.

Semantics-preserving transformations that manipulate modules (as opposed to statements or even lower level program phrases) are programming-in-the-large. Perhaps it is time to say in more detail what it is that activities at this level of granularity actually do: They manipulate entire abstract machines (AMs) in roughly the sense of [Parnas 72a, Parnas 72b]. We can distinguish three different kinds of activity involving abstract machines: The first of these corresponds almost exactly to the Ada package concept; it is a way of realizing a given behavior (i.e., AM) provided that certain other AMs are given. This is the most basic and important vertical activity, that of expressing a structural and algorithmic relationship between one level of abstraction and the next (either up or down). It is also of course the basic support for top-down programming in Ada; but the composability of such steps is very limited in pure Ada. A very special case of this is **hiding** some information, i.e., of abstracting the way that a particular behavior is realized by preventing implementation details from being visible outside the package.

It must be noted that Ada lacks the capability for specifying more than the syntax of the behaviors involved here. It should also be noted that any given vertical level may have non-trivial horizontal structure, that is, a given AM may be a (horizontal) composite of several simpler AMs. (See the example in Appendix III.)

Vertical layers of abstraction can be combined by **vertical composition**. This is simply the compounding of abstract realizations: if level $n-1$ realizes level n , and level n realizes level $n+1$, then level $n-1$ realizes level $n+1$. In terms of Ada packages, this is just the idea that supplying

all the stubs in a package means that you can actually execute the program now. It is a basic intuition of software engineering that this should be true independently of the horizontal structure existing at the various levels (a special case of this is the "double law" of [Goguen & Burstall 80] and [Goguen & Meseguer 82a]).

Among the semantics-preserving transformations on the code level (i.e., programming-in-the-small), there are two interesting subcategories of activity: (1) optimizing code; and (2) asserting axioms. The first subcategory has received considerable attention in the literature. It includes transformations that replace recursion by iteration, that eliminate unused variables, that insure common subexpressions are evaluated only once, etc. The early work of [Burstall & Darlington 77] is relevant, and the work of [Rich & Waters 83] also falls largely in this category.

The second subcategory includes work on program verification in which assertions about the state of the environment are inserted between (or possibly even inside of) statements; the best known approach is that of Hoare. The Anna language [Krieg-Bruckner & Luckham 80] embodies this approach for use with Ada. It should be noted that, although such an approach can be used to make assertions about the functions and procedures in an Ada package, it is conceptually quite different from the method of views and theories discussed above, which operates at the package level rather than at the operation and statement level¹⁷.

Our final category is that of transformations that do not preserve meaning. This includes all ordinary programming, which is just adding to or modifying other code (noting that a new project starts from the empty program). Modern structural editors make this a bit more elegant by preventing syntax errors, but the fact remains that these activities take no account of semantics at all. Of course, a programmer trying to debug a program is actually *trying* to change its semantics. But wouldn't it be better if he could get some help in understanding what it is supposed to do and how it is supposed to do it? In fact, providing sufficient semantic information to the programmer makes debugging the process of trying to find where, in the process of program production, semantics was *not* actually preserved.

¹⁷To be very technical about it, the assertion language approach views Ada as an "institution" in the sense of [Goguen & Burstall 84].

One very simple kind of transformation uses a view-body, i.e., a type mapping and an operation mapping, to create a new entity from an old one, with the old syntax modified as indicated in view-body; new types and operations can also be declared, and old ones can be hidden. We call this the **IMAGE** command; it can greatly increase the reusability of entities. Among possible modifications are: to enrich an entity, by adding to its functionality; to restrict an entity, by eliminating some of its functionality; and to rename parts of the interface of an entity. These support a number of useful (data type based) program transformations. This feature has been implemented at the command level of the OBJ programming language [Goguen, Meseguer & Plaisted 82].

We now consider how transformations relate to the horizontal hierarchy and to views. It is often desirable for transformations to include **using** clauses; indeed, this may be *necessary* in order to prevent the creation of new copies of packages that are not included in such a clause. If an old entity **M** has a view $V :: T \Rightarrow M$, then the image entity, say **M1**, should inherit a view $V1 :: T \Rightarrow M1$, except possibly if some functionality has been deleted or modified in constructing **M1**. Functionality can be deleted by **hide-types** and **hide-ops** clauses, much as for LIL package specifications.

It should be noted that transformations, like Ada packages, can be parameterized, instantiated, enriched, derived and combined. In fact, the whole range of horizontal activities can be applied to vertical entities. Moreover, it is clearly essential to provide some horizontal structure for program transformation activities, that is, some local context, if program transformations are to be used as an effective way of encoding design information.

2.4.9 Control Abstractions

The general idea of "control abstraction" is perhaps less familiar than that of data abstraction, but has many applications. An abstract control structure is a parameterized entity that describes some kind of control flow to be executed over instances of an actual parameter entity. For example, a generalized iterator is a parameterized entity that can be instantiated with any data structure having elements suitable for iterating over; suitable data structures include lists, sets, trees, stacks and queues. Many of the loops that arise in programming can usefully be seen as instances of this concept. Another example would be a generic backtrack programming module, that could be instantiated with any data structure having suitable notions of "next"

elements to try, and of the success of a try. Still another is the notion of "pattern-driven demons" found in many AI programming languages (originally in Hewitt's Planner language, and more recently in Prolog). Many other references could also be given, for example, to the languages Clu, SETL and ECL.

2.5 Library Organization

We now discuss library management issues. One suggestion is to use a hierarchical classification scheme (in the same sense that the Dewey decimal system is hierarchical), but with complex indices at different levels of detail and formality, ranging from keywords to formal axioms.

At the highest level, keywords might be organized by application domain, with specific library entities obtained by lower level key words. Examples of such domains are operating systems, relational data bases and compilers. For example, in the domain of operating systems, one might have modules for such functions as scheduling, spooling and checking capabilities. This calls for an acyclic graph organization, with as much sublibrary sharing as possible. For example, numerical routines and basic data types will be needed in of the many more specialized sublibraries.

The lowest levels of the classification hierarchy might contain descriptions of program properties in forms more readable by machines than by users. This would support a very limited analysis capability to search for entities that have relevant properties.

2.5.1 Truth Management

The axioms in a LIL package can be formal statements in a formal language, such as first order logic, or statements in informal mathematics, or sentences in ordinary English. Thus, validating a view (this means showing that the target entity of the view really satisfies the axioms given in the source entity) can be done in various ways, having various associated degrees of certainty. For example, an informal argument may have been given, a test suite may have been successfully run, an informal argument may have survived a formal presentation ("walkthrough"), or a proof checker may have accepted a formal proof.

The most important use of this information would be during debugging. Once a bug is found,

we suggest using critical path analysis and related techniques to find the weakest link in the argument (implicitly constructed along with the design) that there is no such bug. This will be the best place to look for something that needs to be changed; it may also be possible for the system to suggest what changes should be made.

2.5.2 Organization by Semantics

Library organization is a difficult problem. We suggest that LIL packages be used to index the information needed for retrieving appropriate entities from the library. LIL provides version management by permitting more than one Ada package body to be attached to a given LIL package. However, these package bodies need some further information attached to them for choosing among their body. Figure 10 shows how the library entities involved in two versions of a LIL package might be organized.

Another use of organizing a library by semantics and having available a library interconnection language, is that it should be possible to automatically build documentation for new entities that have been constructed from old ones, by using the documentation already given for the old entities. In fact, the first step of constructing a large system might be to structure its requirements, so that an appropriate interconnection statement would assemble the system requirements from the requirements for its components (of course, some system requirements could not be broken down in this way, since they apply only at the highest level, e.g., total system performance). Later steps would break these components down further.

Another potentially very valuable benefit of semantic organization (i.e., of LIL) is to provide a rapid prototyping capability by simply executing the axioms in LIL packages. This will be possible if all the axioms are given in an appropriate logical formalism, and if there are enough of them. For example, work done with OBJ [Goguen, Meseguer & Plaisted 82, Goguen & Meseguer 82b] shows that equational logic can provide such a capability; and Prolog [Colmerauer, Kanoui & van Caneghem 79] shows that it can be done with so-called Horn clause logic.

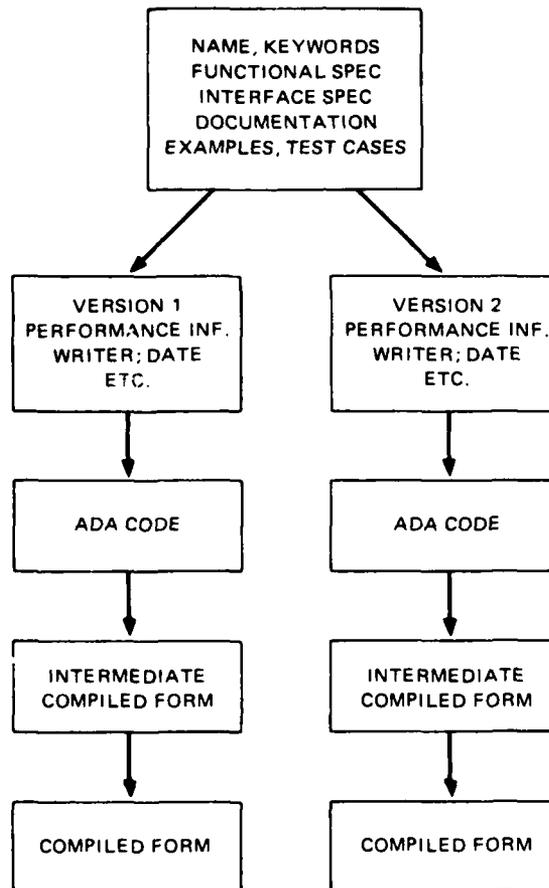


FIGURE 10 ORGANIZATION OF LIBRARY ENTITIES FOR A PACKAGE

2.5.3 System Families

The technique of "skeletons" (mentioned as point 9. at the beginning of Section 2.4) if properly developed, might provide a very general and flexible approach to software families [Parnas 76], permitting the system in effect to provide a "first draft" implementation (see also [Coopriider 79]). The intention here is that an expert system might embody some of the knowledge of the

Ada programming community on how to compose programs in some particular application domain. Such an expert system could be updated as experience with a given domain grows, and as library entities applicable to that domain accumulate. This would permit constructing individual family members by combining library entities with the aid of the expert system.

2.5.4 Cataloguing

Current techniques from information science and relational databases should also be relevant to the basic design of a library system. For example, there are results that help choose the optimal number of levels, and the optimal number of classifications at each level, for a given structure (e.g., Huffman coding, human factors experiments, and work of [Resnikoff 80] on library organization); one idea is to help the user to play "twenty questions" as well as possible. Relevant work on user interfaces to databases includes Query-By-Example and system R. One should also consider more experimental ideas, such as automatic indexing and classification schemes, for their possible relevance to Ada libraries.

Notice that the proposed design lends itself to a distributed implementation, so that users at multiple sites could share results. This could be achieved by having all indices available at all sites, but not necessarily all programs at all sites. (This is like having an inter-library loan system, with a system-wide catalogue at each library.) At the expense of greater retrieval time, it is also possible to avoid having all of the catalogue at each site.

2.6 User Interface and Management Issues

The user interface to the library should provide features not available directly in Ada, such as high resolution graphical, and possibly even audio, output of relevant information and structures. Choices could be presented to a user interactively with pop-up multimedia menus (these might involve icons that are suggestive pictures, words and/or phrases), with (optional) descriptions (that might be audio and/or visual) of the meanings of these choices available at various levels of detail. This would permit users at various levels of skill and experience to use the system, and to learn better how to use it as they do so. One can also imagine having computer generated sound movies that explain how a given system or component works.

It would seem worthwhile not only to enforce aspects of good programming style, but also to reinforce, encourage and teach it with built-in on-line system features. One thinks, for example,

of the powerful reinforcement techniques in video games. It would appear that a great deal could be done to support interactive learning about Ada, about the library system and about its methodology, while users are actually using them. There could be, for example, an on-line explanation capability derived from the expert systems already mentioned.

In the area of visual interfaces, an interesting possibility is to **animate** packages or other AMs in a system, that is, to generate a "cartoon" sequence of iconic representations for the abstract data types (ADTs) involved, such that changes in the display correspond to changes in the state of the AM. This is not so difficult as it might sound at first, since default iconic representations can be automatically generated from suitable equational specifications for the ADTs, and these might well already be in LIL packages for these types. These default icons consist of boxes containing data items, strung together in lines, or in trees. For many familiar data types, this gives the usual graphical representation. Among these are lists, queues, stacks, arrays and trees. For others, it does not, and some additional work would be required to get the usual representation. However, it should never be necessary to write complex display programs, but only to supply particular iconic representations for ADT constructors, since the way that they are to be put together is available from the algebraic representation. Going a little further, it should also be possible to automatically generate an audio commentary to accompany such a display sequence. The value of such animations for program understanding and debugging would certainly be immense.

Recent research on how people actually describe programs [Burstall & Weiner 80] might be utilized to make program documentation and the library classification scheme easier for users to understand and to use. This research shows that a number of different descriptive structures are used for different purposes, and that these are regularly embedded within one another in specific ways. The structures used include the following: **explanations**, e.g., for describing data structures; **plans**, e.g., for describing high level organization; **stories**, e.g., for giving historical information about program development; and (what have been technically called) **pseudo-narratives**, in which program entities participate in a story-like structure as "characters." While simple stylized formats will be adequate for small library entities, higher level schemes like those discovered by Burstall and Weiner will be needed to organize the documentation of larger assemblies of library entities. We propose that these descriptions be interactively system prompted, and include such features as interface syntax, input constraints,

exceptions, side effects, output properties, design decisions (e.g., space-time trade-offs) and performance.

It may seem anticlimactic to conclude a fairly technical paper with a discussion of management issues. However, experience makes it clear that unless such issues are handled properly, it will be difficult or impossible to effectively use software development tools.

It appears that each successive generation of software development tool has been significantly more expensive than the previous one. Compilers are much more complex than assemblers; and an Ada environment is an extraordinarily complex and expensive thing. However, these tools are still much less expensive than corresponding hardware tools, such as fabrication lines. Thus, it seems very strange that there is such great reluctance to invest significant amounts of money into research and development for software tools. This is especially true in view of the fact that more and more of the cost of real systems now lies in their software rather than their hardware. It should be noted that Japanese "software factories" have been reported to achieve remarkable rates of reusability, from 60% to 80%. Thus, it seems that unless these management policies change, both in government and industry, the United States may fall behind other countries in the important area of software productivity.

2.7 Acknowledgements

I would like to thank Drs. Ole Oest, Jose Meseguer, and Karl Levitt for their very valuable comments on drafts of this paper. Professor Rod Burstall is particularly thanked for his careful reading of a late draft.

2.8 References

- [Balzer 81] Balzer, R.
Transformational Implementation: An Example.
IEEE Transactions on Software Engineering SE-7(1):3-14, 1981.
- [Burstall & Darlington 77]
Burstall, R. M. and Darlington, J.
A Transformation System for Developing Recursive Programs.
JACM 24(1):44-67, 1977.

[Burstall & Goguen 77]

Burstall, R. M. and Goguen, J. A.
Putting Theories together to Make Specifications.
Proceedings, Fifth International Joint Conference on Artificial Intelligence
5:1045-1058, 1977.

[Burstall & Goguen 80]

Burstall, R. M., and Goguen, J. A.
The Semantics of Clear, a Specification Language.
In *Proceedings of the 1979 Copenhagen Winter School on Abstract Software
Specification*, , pages 292-332. Springer-Verlag, 1980.
Lecture Notes in Computer Science, Volume 86.

[Burstall & Weiner 80]

Burstall, R. M. and Weiner, J. L.
Making Programs more Readable.
1980.
Proceedings, International Symposium on Programming, Paris, April.

[Buxton & Druffel 81]

Buxton, J. N. and Druffel, L. E.
Requirements for an Ada Programming Support Environment: Rationale for
STONEMAN.
In Hunke, H. (editor), *Software Engineerin Environments*, , pages 319-330.
North-Holland, 1981.

[Cheatham 83]

Cheatham, T.
Reusability through Program Transformation.
In Biggerstaff, T. and Cheatham, T. (editors), *Proceedings, Workshop on
Reusability in Programming*, , pages 122-128. ITT, 1983.

[Cohen & Jackson 83]

Cohen, B. and M. I. Jackson.
*A Critical Appraisal of Formal Software Development Theories, Methods
and Tools*.
Technical Report, Standard Telecommunication Laboratories, Harlow,
England, June, 1983.
ESPRIT Preparatory Study.

[Colmerauer, Kanoui & van Caneghem 79]

Colmerauer, A., Kanoui, H. and van Caneghem, M.
Etude et Realisation d'un Systeme Prolog.
Technical Report, Groupe d'Intelligence Artificielle, U.E.R. de Luminy,
Universite d'Aix-Marseille II, 1979.

- [Cooprider 79] Cooprider, L. W.
The Representation of Families of Software Systems.
Technical Report, Carnegie-Mellon University, Computer Science Department,
1979.
Ph. D. Thesis.
- [De Marco 78] De Marco, T.
Structured Analysis and System Specification.
Yourdon, 1978.
- [DoD 80] United States Department of Defense.
Requirements for Ada Programming Support Environments.
February, 1980
- [Feather 82] Feather, M.
A System for Assisting Program Transformation.
ACM Transactions on Programming Languages and Systems 4(1):1-20, 1982.
- [Goguen 71] Goguen, J.
Mathematical Foundations of Hierarchically Organized Systems.
In E. Attinger (editor), *Global Systems Dynamics*, , pages 112-128. S.
Karger, 1971.
- [Goguen 82] Goguen, J. A.
Ordinary Specification of Some Construction in Plane Geometry.
In J. Staunstrup (editor), *Proceedings, Workshop on Program Specification*, ,
pages 31-46. Springer-Verlag, 1982.
Lecture Notes in Computer Science, Volume 134.
- [Goguen 83] Goguen, J. A.
Parameterized Programming.
In Biggerstaff, T. and Cheatham, T. (editors), *Proceedings, Workshop on
Reusability in Programming*, , pages 138-150. ITT, 1983.
To appear in *IEEE Transactions of Software Engineering*.
- [Goguen & Burstall 80] Goguen, J. A., and Burstall, R. M.
*CAT, a System for the Structured Elaboration of Correct Programs from
Structured Specifications.*
Technical Report, SRI, International; Computer Science Lab, 1980.
- [Goguen & Burstall 84] Goguen, J. A. and Burstall, R. M.
Introducing Institutions.
In E. Clarke and D. Kozen (editor), *Proceedings, Logics of Programming
Workshop*, , pages 221-256. Springer-Verlag, 1984.
Lecture Notes in Computer Science, volume 184.

- [Goguen & Meseguer 82a]
 Goguen, J. A. and Meseguer, J.
 Universal Realization, Persistent Interconnection and Implementation of
 Abstract Modules.
 In *Proceedings, 9th International Colloquium on Automata, Languages and
 Programming*, . Springer-Verlag, 1982.
 Lecture Notes in Computer Science.
- [Goguen & Meseguer 82b]
 Goguen, J. and Meseguer, J.
 Rapid Prototyping in the OBJ Executable Specification Language.
Software Engineering Notes 7(5):75-84, 1982.
 Proceedings of Rapid Prototyping Workshop.
- [Goguen & Meseguer 84a]
 Goguen, J. and Meseguer, J.
Equality, Types and Generics for Logic Programming.
 Technical Report Technical Report CSLI-84-5, Center for the Study of Logic
 and Information, Stanford University, March, 1984.
 Also to appear in *1984 Logic Programming Symposium*, Upsala, Sweden.
- [Goguen & Meseguer 84b]
 Goguen, J. A. and Meseguer, J.
An Initiality Primer.
 ?, 1984.
 To appear.
- [Goguen, Meseguer & Plaisted 82]
 Goguen, J. A., Meseguer, J., and Plaisted, D.
 Programming with Parameterized Abstract Objects in OBJ.
 In D. Ferrari, M. Bolognani and J. Goguen (editors), *Theory and Practice of
 Software Technology*, , pages 163-193. North-Holland, 1982.
- [Green *et al.* 81] Green, C. *et al.*.
Research on Knowledge-Based Programming and Algorithm Design.
 Technical Report, Kestrel Institute, 1981.
- [Habermann & Perry 81]
 Habermann, A. N. and Perry, D. E.
 System Composition and Verion Control for an Ada.
 In Hunke, H. (editor), *Software Engineerin Environments*, , pages 331-343.
 North-Holland, 1981.
- [Jackson 75] Jackson, M. A.
Principles of Program Design.
 Academic Press, 1975.

- [Krieg-Bruckner & Luckham 80]
Krieg-Bruckner, B. and Luckham, D.
Anna: Towards a Language for Annotating Ada Programs.
SIGPLAN Notices 15(11):128-138, November, 1980.
- [Liskov et al. 79] Liskov, B. H., Moss, E., Schaffert, C., Scheifler, B., and Snyder, A.
CLU Reference Manual.
Technical Report, MIT, Lab for Computer Science, 1979.
- [Litvinchouk & Matsumoto 83]
Litvinchouk, S. D. and Matsumoto, A. S.
Design of Ada Systems Providing Reusable Components.
In Biggerstaff, T. and Cheatham, T. (editors), *Proceedings, Workshop on Reusability in Programming*, , pages 198-206. ITT, 1983.
- [Parnas 72a] Parnas, D. L.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the Association for Computing Machinery 15, 1972.
- [Parnas 72b] Parnas, D. L.
A Technique for Software Module Specification.
Communications of the Association for Computing Machinery 15, 1972.
- [Parnas 76] Parnas, D. L.
On the Design and Development of Software Families.
IEEE Transactions on Software Engineering SE-2(1):1-9, 1976.
- [Prieto-Diaz & Neighbors 82]
Prieto-Diaz, R. and Neighbors, J.
Module Interconnection Languages: A Survey.
Technical Report, University of California at Irvine, August, 1982.
ICS Technical Report 189.
- [Resnikoff 80] Resnikoff, Howard.
Optimal Hierarchical File Organization.
Technical Report, National Science Foundation, 1980.
- [Rich & Waters 83]
Rich, C. and Waters, R. C.
Formalizing Reusable Software Components.
In Biggerstaff, T. and Cheatham, T. (editors), *Proceedings, Workshop on Reusability in Programming*, , pages 152-159. ITT, 1983.
- [Scherlis & Scott 83]
Scherlis, W. and Scott, D.
First Steps Towards Inferential Programming.
In Mason, R. E. A. (editor), *Information Processing 83*, , pages 199-212.
Elsevier, North-Holland, 1983.

- [Standish 83] Standish, T.
Software Reuse.
In Biggerstaff, T. and Cheatham, T. (editors), *Proceedings, Workshop on Reusability in Programming*, , pages 45-49. ITT, 1983.
- [Tichy 79] Tichy, W. F.
Software Development Control Based on Module Interconnection.
In *Proceedings, Fourth International Conference on Software Engineering*, , pages 29-41. IEEE, 1979.
- [Tichy 80] Tichy, W. F.
Software Development Control Based on System Structure Description.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1980.
Ph. D. Thesis.

3 Prepared Lectures

The following subsections contain summaries of the talks given in the workshop, in the order that they were given (see Appendix I). After each summary is a digest of the discussion that followed that talk. The slides from the talks that were available and reproducible are given in Appendix IV.

3.1 Why DoD Needs Software Environments

by Brian Schaar, AJPO and Jack Kramer, IDA

Brian Schaar first considered goals as they existed five to ten years ago. Congress was interested in why DoD weapons systems cost so much, and why these costs were continually escalating. It seemed that greater utilization of general purpose digital computers, as opposed to special purpose computers, would save money. In 1975, a Higher Order Software Working Group (HOLWG) was convened to consider language and environments. This led to Ada. In 1984, Ada will be required in DoD systems; this is the first time that DoD has required a *single* higher order language. AJPO will be concerned with supporting the transition, and will give increasing support to work on Ada education.

Jack Kramer then discussed prospects for the STARS program, and indicated the important role that Larry Druffel had played in both Ada and STARS. Even in the early HOWLG, there was skepticism that Ada alone would be a solution to the reusability problem; it was recognized that environments would be needed, including catalogues for libraries, and help to users in deciding which package is best. We still do not fully know what an environment is. But we feel the need for a software components industry, providing for example a basis for a rapid prototype that can later be modified. The hardware industry is relatively more successful, because it has standard interface conventions, catalogues of components and understandable specifications. Specification is more of a problem in software; and design for modifiability is an even greater problem. For example, testing should be integrated into environments and used throughout the lifecycle.

Brian Schaar then discussed various related projects, such as ALS (Ada Language System), CAIS (Common APSE Interface Set), AIE (Ada Integrated Environment), KIT/KITIA (KAPSE Interface Team/KAPSE Interface Team Industry and Academia) efforts.

3.1.1 Discussion

Tichy: Will free Ada compilers be available? It will be hard to use Ada if it is expensive.

Schaar: We are working on this.

Cohen: It is the intention that all government supported software tools would be provided as GFE (government furnished equipment).

Dempsey: What is happening with CAIS? KIT is seeking to integrate the Army and Air Force interfaces, and has produced a document, CAIS 1.1, containing guidelines for tool writers and identifying important interface problems.

Kramer: The goal of KIT is to identify a minimum set of tools, and to ensure that they can be moved from host to host.

3.2 Conceptual Architecture for a Software Engineering Environment

by Samuel T. Redwine, Jr., IDA

This lecture presented the results of an initial planning effort for the Joint Services Software Engineering (SEE) Environment Effort, and was described as having a preliminary character. The primary components of the architecture were presented in a three-dimensional diagram. The three dimensions of the diagram represented: the components of the environments; the linguistic entities used in the environment; and various factors in the context in which the environment is used. Among some specific points made, the use of standard metagrammars was recommended for capturing dynamic typing. A number of database concerns were also isolated, including: interaction with the rest of the SEE; what its contents and structure should be; how to describe the contents and structure; and standardization.

3.2.1 Discussion

Cohen: Is a universal translator practical for CAIS? Is CAIS an intermediate language?

Redwine & Kramer: Portability of software tools is a major goal of CAIS.

Rudmik: I am worried about standards. UNCOL, meant to be a standard intermediate language, failed. We should not try to standardize where there is not enough understanding.

Redwine: It is not clear when methodology technology will be ready. Trial use military standards might be useful, and we should also consider International Standards.

Kramer: DIANA is now accepted (or almost).

Redwine: It is important to have standardization in mind early.

3.3 An Overview of Ada Libraries

by Karl N. Levitt, SRI International

This talk was primarily an introduction to the SRI "strawman" approach to Ada Libraries. Technical details are treated in the talks by Goguen and Meseguer. The present talk considered three main topics: (1) our overall goals for a theory and practice of reusability; (2) available technology for Ada Libraries; and (3) needed new technology. Following a general motivation for reusability, features of Ada that support reusability were reviewed, and it was concluded that the Ada parameter mechanisms are a good basis, but there remains the question of how to put modules together, how to capture design issues more easily than is possible just with Ada, and how to document modules and systems. SRI's approach to these issues is considered in the paper and lecture by Goguen. It is emphasized that reusable code is not the complete answer; for example, generic designs and requirements capture decisions not conveniently statable in Ada. A major concern is how to document so that useful components can be identified and retrieved. Existing documentation technology was reviewed, and it was concluded that a standard database management system would be of considerable use as a first cut. However, it appears that more dynamic forms of documentation are needed for easier understandability and modifiability. Towards improved documentation methods, ongoing work at SRI on a system called PegaSys was described. PegaSys describes system structure (also called "form") using graphical methods in three ways: pictorial representations can be refined to yield more detailed forms; pictorial representations are linked formally to Ada code so that they are assured to have computational meaning; and execution results can be graphically displayed. PegaSys might be of great value in tracking changes to a system, since it identifies those system components that are impacted by changes to a single component. Similar methods are proposed for use in LIL, as described in the lecture by Meseguer.

3.3.1 Discussion

Cohen: How can you couple reusability to the software lifecycle?

Levitt: There is meaningful reusability at all stages of the lifecycle.

Livtintchouk: What is the form calculus?

Levitt: The form calculus can be used to describe objects and how they interconnect. The form calculus is used in the PegaSys visual programming environment being developed by Mark Moriconi *et al* at SRI. Standard relations are "call-by-value," "dataflow," etc., and new relations are derivable from the standard relations.

3.4 LIL: A Library Interconnection Language for Ada Programs

by Joseph A. Goguen, SRI International

The substance of Goguen's talk is included in his paper prepared for this workshop and given in Section 2 of this report. The following is an abstract for that paper:

This paper discusses problems, concepts and approaches relevant to an on-line library system supporting the creation, documentation and maintenance of Ada software systems. The ultimate goal of research in this area is to make Ada programming significantly easier, more reliable, and more cost effective by using previously written Ada code and previously accumulated programming experience to the maximum possible extent. The main suggestions made in this paper are as follows: systematic (but limited) use of semantics, by explicitly attaching theories (which may be informal) to program units by means of views (a new concept defined in this paper); use of library entities and a library interconnection language (called LIL) to assemble programs out of existing code; maximal use of generic library entities, to make them as reusable as possible; support for different levels of formality in both documentation and validation; and finally, facilitation of program understanding by animating abstract data types and module interfaces.

3.4.1 Discussion

Redwine: How does this relate to programming in the large?

Goguen: Programming in the large is the assembly of modules into larger programs, and typically involves many people and long lifetimes; programming in the small is concerned with writing the modules. The LIL language is specifically designed for programming in the large with components that are written in Ada. An environment based on LIL could automatically generate requirements documentation and management information from local documentation and the system design, as expressed in LIL.

Myers: Is LIL oriented to the package level, or to the system level?

Goguen: The entities that stored in the LIL database cover a very broad range, and LIL operations manipulate both packages and higher level descriptions; the purpose of these manipulations, of course, is to create systems. For example, LIL can "slice" a package to reduce its functionality, and at the same time, automatically slice the corresponding documentation. More generally, the operations of instantiation, enrichment and restriction are applicable to all kinds of library entities.

Dempsey: This raises the question of what should be stored in the database. There are some entities that should be treated separately.

Goguen: Yes. We need to be able to create relatively small views of a potentially very large database. It is obvious that code should be stored in the database, both in compiled and raw form. At the higher levels of information, having more of a structural or management function, it is less clear what should be stored and how it should be manipulated. Although LIL makes some concrete proposals, these issues need further research.

Dempsey: How do you deal with different kinds and levels of abstraction?

Goguen: One of the wonderful things about Ada is that it clearly distinguishes between vertical and horizontal abstraction. Horizontal abstraction has to do with parameterized modules, while vertical abstraction has to do with step-wise refinement. However, this power cannot be sufficiently exploited in Ada programming methodology, due to limitations of the Ada compiled program library concept. A good library interconnection language could solve this problem.

Rudmik: Is LIL really a good language from the user's point of view?

Goguen: The syntax chosen for LIL in the paper I wrote for this workshop is intentionally relatively close to mathematics, as well as to Ada. However, we have ideas for a much more natural user interface, involving menus, graphics, animations and so on; the user need not see LIL in the internal form described here. Meseguer will discuss aspects of this in his talk.

Levitt: Could you summarize some of the limitations of Ada that LIL helps to overcome?

Goguen: Ada's **separate** statement prevents reusing compiled code for stubs in more than one package; LIL has no such limitation, and instead relies on partially compiled code. Also, if $F(X)$ and $G(Y)$ are generics, Ada provides no way to compose them to get a new generic $G(F(X))$;

instead, one must first apply F to an actual A , and then one can get $G(F(A))$. LIL permits fully general composition of generics, both horizontally and vertically.

Litvintchouk: At what point do we go from LIL to Ada?

Goguen: One can think of LIL's modules as being like parameterized library cards; one can fill in parameters, and get various books; then one can combine these books to get still larger books. However, what is actually being put together is partially (or, when possible, fully) compiled Ada code. LIL is used to express system organization, rather than executable code *per se*.

Litvintchouk: It seems to be an advantage that this approach checks consistency between levels.

Goguen: Yes, the "theories" associated with interfaces in LIL permit one to declare both semantic and syntactic requirements for consistency. However, the semantic requirements can be expressed at a variety of levels of precision, from just English to some formal specification language. Then, when an actual is to be substituted into a generic, it can be made clear what properties that actual must have for things to work correctly. LIL uses the "view" concept as a "bridge" between the actual and the requirement theory of the generic, to say just how the requirement is actually satisfied. For example, with a SORTING generic, one should require that there is a partial ordering relation on the actual. But if we choose the naturals as actual, there is more than one such partial ordering available; suppose that we want to use "divisibility" as our ordering; then the view should express this, and perhaps also give an argument (formal or informal) for why it really is an ordering relation.

Babcock: Can all Ada code be stored in compiled form? Does LIL code compile into real code? For example, what if a stub is to be used in another package?

Goguen: LIL stores intermediate code, that is compiled when the parameters are available.

Babcock: Could DIANA deal with the intermediate forms required by LIL?

Goguen: That's a good question. I think that something like DIANA would be sufficient, but it will take more research to find out in detail exactly what is needed. It appears that generic addresses are needed, which I understand DIANA does not provide.

3.5 DCP Approach to Ada Libraries

by Andres Rudmik, GTE

This lecture summarized aspects of the DCP (Distributed Software Engineering Control Process) project at GTE Network Systems R&D in Phoenix, including the project goals and approach, and emphasizing the use of Ada libraries to support software reusability. Among the goals were that the DCP should be distributed and portable, should support centralized control of development, reduce software costs and improve software quality, support use of Ada for design and implementation, and should integrate software development tools. The approach involves using a relational database supporting configuration management, change tracking, program libraries, and document generation. Reusability is to be achieved by using Ada libraries and a "DCP Encyclopedia" which functions as a library catalog. It is necessary to have a reusability methodology, involving the specification, design, implementation, documentation and testing phases. Documentation is especially important.

3.5.1 Discussion

Tichy: What happens to old packages?

Rudmik: Configuration management keeps track of who uses what; notifies users of all changes; we are not building in a lot of rules just now.

Tichy: You need to build in *options* now, if not *rules*.

Levitt: What is your experience with DCP?

Rudmik: The project is only 9 months old. We are now using DCP to build DCP, and trying to demonstrate there are no performance degradation; also, we are trying to use Ada as a command language.

Litvintchouk: We should not be deluded by the success of the developers using DCP, since they know it, and are highly motivated to succeed. Perhaps you should try an experiment with two groups, package writers and package users.

Witte: Have you looked at menu-type access?

Rudmik: We are looking at that.

3.6 Flexibility vs. Efficiency for Reusable Components

by Allen S. Matsumoto, ITT

This lecture was concerned with the trade-off between flexibility and efficiency for reusable components, and more specifically, with the attempt to define reusability and generality. It was noted that a more general Ada package would involve fewer restrictions and therefore permit more instantiations; i.e., it would be more flexible. On the other hand, a more specific package would have more decisions already made, taking into account greater knowledge of context, and then permitting greater efficiency in that context. This was illustrated with various packages for sorting arrays, showing that early design choices, which may even have been made at specification time, can undesirably limit later implementation options. A tentative definition of generality (and thus of reusability) was offered in terms of the complexity of the graph of theories used in the specification of the package.

3.6.1 Discussion

Witte: How do you measure quality of design decisions; for example, formats, size of objects, and distribution of data?

Litvintchouk: It is hard to define "best" or even "better," because of the tradeoff between efficiency and reusability.

Kramer: One should maintain semantic equivalence throughout the life history of a design, so that one can backup and reconsider decisions.

Matsumoto: The complexity of the graph relates to how close one is to the base language.

Redwine: Another issue is how humans intuitively view complexity; what someone is used to is important.

Matsumoto: Sometimes a user sublimely ignores an issue, such as properties needed for the ordering relation on the elements to be sorted.

Cohen: When developing a secure system, it is convenient to view the system as having two parts: the **trusted** part that must be correct to assure security, and the rest of the system, that has no bearing on security. then, one wants to minimize the amount of code in the trusted part. This gives a different loss function.

Matsumoto: "Shortest" is not necessarily "best."

3.7 Mapping Clear Specifications to Ada Packages

by Steven Litvintchouk, Raytheon

The goal of the work presented in this lecture was the systematic development of Ada packages on skeletons which parallel the development of specifications written in the Clear specification language of Burstall & Goguen. The approach is to first develop system specifications in Clear, and then to drive the Ada design from this specification. This should facilitate the design process and also the reusability of Ada components. The use of features from Clear is intended to supplement the largely syntactic package interfaces provided by the Ada language itself. The mapping from Clear to Ada is natural and relatively simple, because of similarities in the way the two languages handle structure.

3.7.1 Discussion

Goguen: What are your more specific plans for this project?

Litvintchouk: In the next year, we hope to use dynamic logic in Clear, so as to be able to handle tasking. We believe that this is better than temporal logic for specifying Ada programs, because it is based on nondeterminism.

3.8 General Requirements for an Elementary Math Functions Library

by Bruno Witte, NOSC

This lecture discussed a draft document outlining requirements that should be satisfied by an Ada library of elementary mathematical functions. For example, functions that are likely to be used together should be packaged together. Separately for each elementary function, there should be accuracy tests of various kinds, such as with random arguments, range reductions, single-precision, separate test packages. Reference should be given to publications documenting the supporting mathematical theory for an algorithm.

3.8.1 Discussion

Babcock: Sometimes we want to have bodies written in assembly code in order to get efficiency. This is more difficult to do in Ada than in FORTRAN; also, it is harder to tap hardware hacks in Ada.

Goguen: It would be nice if documentation standards for other kinds of software were as well

developed as they are for numerical software. In particular, the close links to the theories provided by numerical analysis strikes me as worthy of imitation by other areas of software.

3.9 Knowledge Based Tools for Data Type Implementation

by Gordon Kotik, Kestrel Institute

This lecture discussed aspects of Kestrel's CHI environment that are relevant to data types, including a theory of data type implementation and a tool for implementing data types. The relevance to Ada libraries was also considered. CHI is a knowledge based programming environment having a programming language called "V", a database for representing V objects, and tools for reading, printing, structurally editing, and compiling (into Lisp) V objects. V is a wide-spectrum, very high level language for specifying, writing, and improving programs; it has program transformation rules, high level data types (sets, sequences, mappings, relations, products, unions, ...) and logic constructs. Programs and knowledge are expressed as rules in the CHI database; multiple contexts are maintained in a tree structure. V programs are compiled by successive rule applications. The data type facility of CHI is intended to cope with the fact that there may be a variety of different implementations with widely disparate efficiency characteristics, depending on usage patterns. The solution is to have an "efficiency expert" to estimate the resource requirements of V programs; such an expert can be used to guide the search for an appropriate implementation through a sequence of refinements.

3.9.1 Discussion

Mathis: What is the difference between having lots of generics and having rules?

Kotik: Perhaps having lots of generics is like storing good chess moves for each position, while having rules would help you deduce desirable positions.

Matsumoto: One can use a sequence of generics, reflecting a sequence of design decisions. Perhaps rules and generics are the same?

Kotik: Ada is clumsy with its parameterization mechanism, compared with what transformation rules can do.

Goguen: Vertical structure can provide a lot of the flexibility of rules. Gordon's rule system is probably equivalent to having a library of generics that is managed by an expert system, over a module (or library) interconnection language. Also, note that the problem of finding the "best implementation" is unsolvable (in the sense of recursive function theory) even if it is clear what cost function should be applied.

3.10 Library Organization and User Interfaces

by J. Meseguer, SRI International

In order to maximize reusability and provide good programming support, an Ada library environment should be organized to reflect the different levels at which users will interact with the library. Also, relationships between different levels, such as design and code, should be explicitly represented to facilitate "navigation" across levels; this is important to facilitate both program understanding and effective methods of library search.

This talk proposed a semantic-based library organization oriented towards maximal reusability not only of code, but also of designs. Thus, besides packages, designs (expressed in LIL), theories, specifications, views, horizontal and vertical structure, and managerial information should be stored. The organization suggested is that of a nested hierarchy consisting of three main levels:

1. Application domain hierarchy.
2. Module interconnection language hierarchical structure.
3. Ada package hierarchy (subdivided into specification and body sublevels).

Semantic relations between levels are also stored. Between levels 1 and 2 there are hierarchy preserving relations linking theories to application domains; between levels 2 and 3 one has views connecting LIL theories to Ada packages. Traveling across levels is guided by semantic relations, that also facilitate automatic or semiautomatic cataloguing and retrieval.

The user interface should exploit the graphical representation potential available in the nested hierarchy of levels. A multimedia interface, combining text, graphics, kinesthetics, and speech would be appropriate. Animation of specifications and programs could be provided by the interface to facilitate program understanding and testing. This possibility would be directly available if package specifications are written in an executable equational specification language like OBJ.

3.10.1 Discussion

Rudmik: How long would it take to implement a graphical editing system like the one you have described?

Goguen: Using the graphics capabilities already available in a Symbolics Lisp machine, or a Dolphin, perhaps something like four man years.

3.11 Version Control in Program Libraries

by Walter Tichy, Purdue

The development database is central for any programming environment, as it supports documentation, editing, execution and project management. Version control is an important technique for maintaining the consistency of this database. **Documents** are named, separately identifiable collections of information, and may be either **source** documents or **derived** documents. The latter can be fully automatically generated from the former. Attributes of source documents include author, data/time, phase, and type/language; attributes of derived documents include the source documents and generation process used, and the date/time of generation. A **revision** is a source document created by manually revising an existing document, and a **revision group** is a set of revisions related to one another by manual revision. To update a revision group, one must **checkout** all revisions to be modified, enter an edit/make/test-debug cycle for each, and then **checkin** the modified revisions. Temporary fixes, experimental modifications, update conflicts and parallel developments can all lead to branching of revision numbers. A **configuration** is a collection of related but individual documents or other configurations; examples include link configuration, test configuration, and program+documentation+manual. A **configuration description** is a collection of names of component documents of a configuration, possibly only partially resolved, and can be used for automatic generation of derived documents. Some incremental techniques for efficiently implementing databases of revisions and configurations were presented.

3.11.1 Discussion

Cohen: Is only one version active at a time?

Tichy: No, every tip node of a revision group graph is an active version.

Dempsey: How do you differentiate between checkin/checkout and just grabbing a copy? Would some AI-like rules be useful here?

Tichy: We do have a "copy" operation; also, the user can indicate a plan associated with checkout.

Dempsey: What makes global regeneration so expensive?

Tichy: It is because you have to recompile.

Mathis: Why not store text on a side branch instead of regenerating?

Tichy: Each delta is about 8%, and these add up.

Goguen: When $(1.08)^N$ is large enough, it will be cheaper to merge.

Schill: Stonebreaker is using C to get user definable datatypes for Ingress.

Rudmik: Database technology will give us even more performance than configuration management systems.

3.12 Using ANNA for Specifying and Documenting Ada Packages

by Friedrich von Henke, Stanford University

ANNA is an annotation language for Ada currently being developed at Stanford by the author, B. Krieg-Bruckner, D. Luckham and O. Owe. This lecture presented basic ideas underlying the ANNA design, and an overview of ANNA features useful for package specification and documentation. ANNA extends Ada with **formal comments** to express additional properties of packages, leaving Ada untouched; thus, an ANNA program is still a legal Ada program. A major goal of ANNA has been to provide precise (formalized) documentation, as a basis for processing by machines, e.g., formal verification; however, it can also be used as a basis for less formal validation. Ada concepts and syntax are used as far as possible. ANNA **virtual text** is used to introduce specification concepts, auxiliary functions, packages and semantic constraints for generics. An ANNA program is **consistent** if the constraints imposed by the annotations are satisfied by Ada text.

3.12.1 Discussion

Witte: Why would a human being bother to write ANNA annotations? Also, how to annotations differ from just comments?

von Henke: ANNA annotations can be formally checked, since they are subject to Ada syntax rules.

Cohen: For some applications, we need trusted processes, and ANNA can be used in specifying and verifying properties desired of trusted software.

Kotik: Could you use ANNA for the balanced binary tree example that I presented?

von Henke: Yes, it can specify the behavior of a package that would maintain a balanced binary tree.

Goguen: Wouldn't you need modal logic for exceptions? Also, what about tasking in ANNA?

von Henke: Yes, you need "strong logic" or modal logic for exceptions. ANNA does not currently support tasking.

Litvintchouk: ANNA is semantically equivalent to theories, as in Clear or LIL, but they are "flattened out" in ANNA, so you lose a lot of the information and simplicity of a hierarchical structure.

von Henke: Requirements on generics in ANNA are like "theories," but not reusable, they must be placed "in-line" each time.

Redwine: What about the EEC (European Economic Community) criticisms of ANNA?

von Henke: These were based on an earlier version; our language is now in better shape, we have a new manual, and their criticisms are no longer valid.

Dempsey: Can you compile ANNA into runtime checks?

von Henke: Yes, for many constructs of ANNA.

Kotik: I don't think that you can compile all ANNA features into Ada; for example, what about existential quantifiers?

Levitt: Why have two languages? For example, why have both Ada code and ANNA specs for a stack?

von Henke: ANNA has much greater expressive power than Ada, so the specs should be easier to understand.

Redwine: What about specifications? Why haven't they caught on?

von Henke: Specifications are being used in industry.

Levitt: Specifications are used mainly for security applications at present.

Goguen: Informal specifications, in contrast with formal specifications, are really very widely used; we should use computer technology to permit handling informal specifications in a more uniform and less burdensome way.

Redwine: Isn't this write-only notation?

Goguen: Techniques like those described by Meseguer for LIL will let the user see an animation of the action of a program, generated directly from the spec; the user doesn't have to read the math itself to get the benefit of its content.

Cohen: ANNA is justified by its use in WIS.

Kramer: Customers are worried about the trustworthiness of code; they want it to be verified if possible.

4 Reports of the Working Groups

Participants of the Ada Program Libraries Workshop spent their last day and a half in Working Groups focussed on special topics of particular interest. Seven working groups were originally proposed. Each working group chair was asked to prepare a list of at least five "initial questions" to introduce the area of interest to be covered by the working group to the whole workshop so people could choose which working group they wanted to attend. Four working groups were actually formed, and their reports are given in four subsections below. Each report includes its list of initial questions.

4.1 Library Documentation

This subsection contains the report of the Library Documentation Working Group, written by its chairman, Jack Kramer of IDA.

4.1.1 Participants

The following participated in the working group discussions; an ARPANET address is given for each.

- Jack Kramer, IDA - kramer@usc-eclb
- Joseph Goguen, SRI - goguen@sri-csl
- Dave Babcock, ROLM - babcock@usc-eclb
- Beverly Kedzierski, Kestrel - kedzierski@kestrel
- Friedrich W. von Henke, Stanford - fwh@su-ai
- Steve Litvintchouk, Raython - brunix!rayssd!sdl@ucb-vax
- Bruno Witte, NOSC - bwitte@usc-eclb

4.1.2 Initial Questions

The following questions were presented for the Library Documentation working group.

1. What is a component?

- A component could be anything from a complete database management system to a single Ada statement. The most commonly thought of component is the Ada package, but is this really the only definition of a component? Would a particular algorithm coded in Ada not also be a useful component?

- Does a component have to be only Ada code? Why couldn't it be a design, or a set of requirements? Does a component include the documentation for it, or is the documentation a component by itself?
2. Are components flat or layered?
 - Can some components be constructed from other components or do each of the parts and the whole need to have all of the same documentation?
 3. What information is necessary?
 - What kind of documentation needs to be provided for each component? Source code if the component is code. What about test data, design parameters and choices, the requirements the component is to satisfy, critical performance requirements and design decisions?
 4. What information is required to help the user (meta)?
 - There can be a lot of information which would be useful to a user both in selecting a particular component, but also in using it after it has been selected. How might this component best be integrated into a new system? Where might algorithms be altered without effecting the component structurally? Where and how should this component be tested as part of a larger system? Why was this component constructed instead of using some other component? Some idea of why the designer thinks this component is better than all the rest. How it was tested, were any formal techniques used?
 5. Life cycle of components and their associated documentation?
 - We are fairly careful about identifying the life cycle and documentation requirements for DoD systems. Are not the same reasons applicable to the software and documentation of components in a library? Are we going to have to worry about versions of components and tracking users and saving old versions, etc?
 6. User confidence in the product? What tests?
 - Should the documentation include a detailed discussion of what kind of testing, how it was done, the tests themselves, the test results, and a statement by the designer of the completeness of the testing? Will there be some sort of standard testing format and requirements so that a "consumer survey" type of organization will be able to consistently judge components?

4.1.3 Initial Working Group Discussions

1. Initial discussions centered around the questions above, and then began to focus on certain issues. Generally, it was agreed that a library mechanism needed to address both the near term and the long term solutions. It was also agreed that the software industry was much more a "cottage industry" at present than the more formalized and controlled hardware

components industry. There are some good reasons to believe that there will always be a large number of "cottage" inputs to a software components library. This will probably influence the requirements for documentation and the tools for capturing it.

2. There is a requirement for documentation to be somehow attached to the code. This might take the form of a LIL-like capability in some cases. There are also going to be different problems associated with documentation for existing systems verses what we can expect to have in the future library system.
3. It was felt that existing library systems provide a good starting place for determining a software components library system. Can we have branch libraries and interlibrary loans? How should components be charged for? Is there a central registrar like the Library of Congress? Who is going to do the cataloging? Will we need specialists in the early years? On this question the feeling was that we probably would. There would be a lot of pump priming required, and assistance in helping initial users. This was both to make sure the system wasn't harder to use than the benefit accrued to the user, but equally as important, to learn how the system is actually used and what needs to be improved and tools developed.
4. It was also felt that there would initially, and probably always, be a requirement to rate "reusability" of components. This would be a long term learning experience. We would need to "have" a library in order to really understand what to do in the future. In order for this to be effective though, we would need to learn from use patterns why certain components were selected, were they useful, and what information would have helped in finding the correct component. Can we construct cataloging criteria by monitoring the search requests and by interacting with the user? This will be an important aspect of the "librarian's" job. We may need to contractually encourage our DoD contracting community to use the system and feed it.
5. Cataloging must be automated and at a low cost to the inputer if the system is to be successful. The library system must also have adequate ability to collect royalties or in some way encourage the construction and input of components.

4.1.4 Assumptions

After several false starts, the working group agreed that we needed to understand the assumptions that a software components library would be working under. This would influence both the mechanics and the documentation required.

1. We need a short term solution which can be transitioned to the long term.
2. The short term library system will be passive, but the long term system must be Active.
3. Libraries will be disjoint but cooperating.

4. Ada is the implementation language for the software stored.
5. Different user needs must be satisfied. Different users will have different degrees of sophistication with respect to the use of the library, computer science technology and the application area.
6. The library must always deal with incomplete information. We need to be able to change both the data in the library system and the system itself.
7. The mechanisms must provide solutions to access, issue control, and schema problems. Not all people should be permitted to see all things in the library.
8. Software is a cottage industry, users have a diverse background, feedback will be hard to capture, and not all users of the library will have equal equipment with which to access the library. Information required to make the library useful will be hard to capture for the initial inputs to the library because the designers of the components will not be "experts" in documentation.

4.1.5 Scenarios

The working group then developed some alternate scenarios for how a software component library might be constructed and operated.

1. There would be differences between the short term and long term solutions, but both solutions must provide cataloguing, updating and retrieving capabilities. Careful attention must be paid to the transition from the short term to the long term library system solution.
2. Short Term.
 - The short term will be "passive" with emphasis on retrieving and searching. Components will be registered with the system "after the fact" of development. Documentation will have to be captured and then the component will be catalogued.
 - For a component to be useful it must be well documented. Quality is much more important than quantity. There should be standards for the quality and quantity of the documentation required for each component.
 - The "card catalog" will be critical. It was felt that existing database technology could be used. A hierarchical schema could provide an easy ability to add, delete and update the catalog. Most importantly, the retrieval language could be simple and oriented towards its use.
 - A simple Ada system could be constructed to meet short term needs.
 - Instrumentation tools are important right from the start. We must instrument the system now to be able to learn how to build the system for the future.
 - A librarian would be required. It is critical that a follow up debriefing system be

implemented with the initial system. This could be as simple as recording the name of a retriever and having the librarian call "N" months afterward.

- A mechanism such as the "Animal" program available on Apple and other systems should be implemented to determine and capture what differentiates between components from the user's point of view. The proper classification schema for components could thus be learned as we go.

3. Long Term.

- The long term system must be an active part of the user's everyday work environment. Where possible the "system" should automatically construct the necessary documentation and appropriate cataloging information when a component is registered. It should take minimal user effort to add a component to the library.
- Reusability must become an integral part of our future system development methodologies and also must become central to our software engineering environments. Reusability must be natural and not something that is forced by management.
- Information and presentation mechanisms "higher" than code will be mandatory for quick user understanding of a component and how it might fit into his system.
- The system should know something about the user, available components, and the application area in order to help the user find the best component for his needs. There probably should be some sort of working set kept for each user and application area.
- A wide physical and organizational dispersion of potential users will require some form of automatic feedback mechanism. The feedback mechanism should be part of the user's environment and be capable of automatically forwarding information to the branch and central libraries where appropriate.

4.1.6 Documentation

The working group then spent some time trying to understand what documentation should be captured and how it might be captured.

1. There will be differences between what can be expected from documentation captured as part of a software engineering environment and that which must be captured off line after the component is developed.
2. We must allow for unconventional documentation facilities, video or sound, but there must be some minimal documentation which is always available on all types of devices. For that we must assume a relatively unsophisticated hardware capability, such as hard copy terminal. There may also be multiple representations of the properties of a component (Ada, ANNA, LIL), but these must be kept consistent.

3. Our understanding of what to ask for in the way of documentation is uneven. We better understand the documentation requirements for an algorithm than we do a "design".
4. Some components will be part of a larger whole. For these we will need proper configuration management of the context information as well as the component and its associated documentation. There may be many different "bodies" associated with a particular Ada package specification.
5. All documentation must be "useful". It must be required, meaningful and designed to become part of a whole. If possible the documentation scheme should be uniform across both the short term and long term library systems. It should also be a goal that the documentation of a component can easily be integrated into the documentation of the new system being constructed by the user.
6. There is a strong possibility that the system and user will be subject to information overload. This means that the system must be designed with careful attention to providing only what is necessary at each step of use. The system should also be able to generate information where possible to reduce redundancies and the consistency problem that redundant information causes over time. Some information will need to be archived based on usage and some should be discarded because it can be regenerated.
7. Feedback is absolutely critical to the system. We must find out what is useful documentation for the many varied uses to which it will be put, how to best present the information, and where critical information was missing. There are difference between local and global information feedback requirements.

4.1.7 Policy and Non-Technical Issues

The working group spent part of the last day talking about some of the issues which must be addressed if a software component library is to be successful. These issues were of a nontechnical nature, but the group felt they may have at least as much impact on the success of such a system as the technical issues.

1. Pump priming will be necessary. We might require the use of the library as part of DoD contracts. Contractors could then be rated as to the reuse of components from the library and contribution to the library as the contract proceeds. The contractor could be rewarded for reuse and contributions. This rewarding need not cease at delivery of the user system, but could continue for some period afterwards. In addition, the direct procurement of "useful" components will probably be necessary.
2. User confidence in the product is critical. Several mechanisms should be available such as software acceptance tools, user experience ratings, a "Good Housekeeping" seal of approval, and software reviews. Degrees of validation of a component along with

statistics on critical path and flow analysis should be available as they apply to components. Standard tools should be available for application to components when they are registered with the library system.

3. Proprietary issues must also be addressed. The working group felt that an appropriate and effective mechanism for providing economic incentives and royalties would be the best way to encourage library use and insertion. The solution to this problem must address issues such as the levels of documentation to be provided and the various products to be provided for different fees.
4. The issue of warranty must be addressed.
5. The question of who can and who should operate the library and various branches must be addressed.

4.2 Methodology

This subsection contains the report of the methodology working group, written by its chairman Allen S. Matsumoto of ITT Programming Technology Center.

4.2.1 Introduction

The group on methodology was formed to consider the basic notions underlying library components, specifically, reusable Ada components. This group was composed of:

- Allen Matsumoto, ITT Programming
- Gordon Bradley, Naval Postgraduate School
- Paul Cohen, Defense Communication Engineering Center - pcohen@usc-eclb
- Gordon Kotik, Kestrel Institute - kotik@kestrel

4.2.2 Issues

Our basic position was that an understanding of the notion of reusability is necessary to "solve" the problems of structuring, searching and using component libraries. Analyzing the current knowledge of reusable components is a first step toward designing a component library. However, any libraries which are built in the near term will necessarily be incompletely designed, and should be constructed to be extensible once more is learned about reusability.

Libraries designed for adaptability must try to foresee which types of requirements are likely to change to allow for such changes. This will be possible only after the currently promising approaches to reusability are identified. Careful consideration of candidate approaches will enable the building of libraries which can accommodate results from research in these areas.

We believe it is now possible to define a research program to experiment with approaches to reusability. Concurrently, libraries of reusable components can be constructed which will be useful in the near term and which can be extended (at least in some directions) as reusability research progresses.

4.2.3 Initial Questions

The following list of questions was presented for consideration by the subgroup. These illustrate the types of questions which must be addressed in setting up component libraries and in defining further research into reusability.

- Which program construction paradigms support reusability?
 - Parameterization
 - Inheritance
 - Specialization
 - Transformation/tailoring
- When are these paradigms most easily/powerfully used during development?
- How can we describe (specify) how to include a component in a system?
 - Parameter semantics
 - Which actual parameters
 - Is modification required?
- What is the effect of using (reusing) a component?
 - Function of the component
 - New types and operations provided
- How much generality for generic components?
 - How to describe (measure) generality
 - Relate more general and more specific implementations
 - Differentiate more and less general implementations

4.2.4 Preliminary Report

The initial subgroup discussion resulted in agreement on several points. We agreed that the problems of reusability can be factored into meaningful subproblems. We also felt that some definite approaches appear promising for attacking several of these.

The preliminary report contained the following comments:

1. Reuse requires understanding
 - a. effect of component
 - b. interface of component with system

- c. how to modify component
- d. effect of modifying component
- 2. General vs. Specific
 - a. More specific components
 - i. easier to understand
 - ii. more rigid
 - iii. ease of reuse vs. value of reuse
 - b. More parameters vs. more generality
 - i. independence of parameters
 - ii. aggregation (structuring) of parameters
- 3. Programming techniques
 - a. Parameterization
 - i. fixed capability
 - ii. semantics?
 - Inheritance
 - i. reusable objects
 - ii. adding capability
 - b. Transformation
 - i. How much?
 - ii. Is editing reuse?
- 4. Multi-directional choices, e.g., between early inefficient prototypes vs. efficient (late) implementation, and between very general vs. a very specific implementations.

4.2.5 Final Report

The subgroup's final report contained the following specific recommendation:

1. Experiment with Ada implementation of standard paradigms.
 - a. Generic components with a fixed set of data type classes. This will be a low risk domain specific approach to reusability. The goal of such experiments is to gain experience with Ada generic packages, more than to extend knowledge about reusability.
 - b. Inheritance with limited subclassing. A generic package may implement a parameterized object (as in object-oriented programming). Another package may inherit its capabilities using the "with" clause and override or add data types or operations.
 - c. Nested generics. Outer levels of generic packages may be useful for transforming information between the interior of the component and the outer local environment. Some such mechanism is necessary to incorporate generally reusable components with Ada's linear elaboration.

- d. Domain independent components.
2. Develop standard formalisms for describing components.
 - a. Necessary for cataloging and retrieving.
 - b. The package specification + formalized comments may contain sufficient information (e.g., ANNA).
 - c. Formal (mathematical) specification of components may be used if automated tools can be provided.
 3. User support via automated tools.
 - a. Check compliance to standard form. Check for sufficiency of information. Such tools would perform many quality assurance functions as well as aid in document preparation.
 - b. Process information for ease of retrieval (storage, search, display). The structure of the library will depend upon the knowledge representation techniques chosen and upon the ability to acquire and codify that knowledge.
 - c. Knowledge-based retrieval capability. An intelligent library management system will be supported by knowledge-based techniques. In fact, until a formal specification of library components is developed, such techniques will be the only ones available to provide automated assistance.

4.3 Library Searching

This subsection contains the report of the Library Searching Working Group, written by its chairman, Andres Rudmik of GTE Network System R&D. Other participants in this working group included:

- Tom Brown, Kestrel - brown@kestrel
- Mary Forthofar, IBM FSD
- Timothy Gill, Wang
- Jose Meseguer, SRI - meseguer@sri-ai

4.3.1 Introduction

This subsection summarizes the discussions of the subgroup on Ada library searching. We have taken a broader perspective in considering the reusability of all program related objects. We concentrated our attention on what kinds of information would be needed to support searching and how this searching might be accomplished.

The model that we adopted consisted of a catalogue or encyclopedia that contains descriptions

of the program objects stored in the library. This catalogue could be implemented as a database against which queries can be made to retrieve descriptions of objects. Most of our discussion centered on searching for Ada packages.

There are two kinds of retrievals that should be considered. First, one may search for program units that exactly provide the desired function or second, one could search for program units that satisfy certain constraints and that can be adapted for the function at hand.

4.3.2 Initial Questions

The following are the questions initially posed for the library search working group:

1. What items should be in the library?
2. What kinds of information can be used as a basis for library searching?
3. How to structure information on which searching is based?
4. How would different Ada library users search the library? Non-Ada user?
5. What tools can be developed to support searching?
6. Where does this fit into the Software Engineering Methodology?

4.3.3 Library Objects

We tried to identify what items would be stored in a library. The following is a list of the items identified:

- Packages: Each package would have a unique spec but there could be many package bodies for a given package spec.
- Designs: There could be several levels of program design. The design could include descriptions expressed in a module interconnection language.
- Formal specifications expressed as theories. These specifications could be embedded or separate.
- Command procedures used to run programs.
- Design histories recording the design steps.
- Note that Ada subprograms were not considered as appropriate units for reuse. Although many existing libraries contain subprograms.
- Program units in other languages must be considered (i.e. Fortran, COBOL).
- Test plans associated with program unit. Each unit which is a candidate for reusability should be separately testable. The test plans also include all necessary test drivers.
- Performance information: This may be needed to assist in the selection of the package bodies.
- Test data on which the program was tested.

- Managerial information.
- Configuration information.
- Program skeletons that can be used to build new packages.

4.3.4 Basis for Searching

Next we examined the kinds of information that could be used as a basis for searching.

- Package classification -- can be used to further refine packages into categories with well defined properties. The advantage of this approach is that categorization can be used to partition the set of packages into a smaller more manageable set.
- Keywords -- can be used to partition the objects by application, function, date, author, the implementation language.
- Summary descriptions -- brief descriptions of the object.
- Detailed description -- provide a complete but informal description of the object.
- Package specifications -- these can also be examined.
- Formal specification -- e.g., ANNA, LIL.
- By generality -- How general is the package?
- Selection of package bodies -- Since there can be many bodies for a given package specification, the search for package bodies will typically be based on implementation or environmental issues such as:
 1. performance (space/time)
 2. complexity of implementation
 3. host dependencies
 4. data dependencies
- Production position -- status of object ie. development, test, or production
- Cost
- Access rights -- are we allowed to use it?

The following scenario describes how an Ada Library might be used. First, one would specify the desired function and query the database to find the candidate set of items that will support this function. Typically this set will contain several alternatives. The choice of alternative will depend on more global constraints based on the contexts in which the item is to be used, the data that it operates on, how well it fits into the current program design, the target host environment, level of confidence (validation), internal efficiency, and even issues such as royalty fees to the original designers.

As described above, the choice of library items will be made in two contexts.

- The identification of the library object will typically be done in some local context based on function rather than application. This selection process will produce a set of candidate items.
- The choice of a specific item will typically be based on the context in which the item is to be used. The selection of the specific item will require a good understanding of the application and the environment of the application.

4.3.5 Catalogue Information Structuring

The information in the catalogue must be structured to reflect the way in which the catalogue is used. Within the catalogue there may be many hierarchies of information that can be used to search for objects. For example, in searching for packages one may

- Define a theory and then search for a package specification that matches the theory. Within the library, there can be many package specs for each theory. The choice of package spec may be based on generality of the package as an example.
- Similarly, there may be many package bodies for each package spec.

The important concept is that the catalogue should allow the user to go from the abstract to the specific or from the specific to the abstract.

4.3.6 Characterization of Ada Library Users

We classified the Ada library user in two ways:

- By user task: Depending on what the user was doing, the library would be used to provide different kinds of information and perhaps support different kinds of queries. Some examples of how the user tasks might be categorized are:
 1. Application users - searching for programs, job control programs etc.
 2. Managers - needs information on packages developers, royalties, and other costs.
 3. System specifiers - developing and using formal specifications.
 4. Designers - developing and searching for package/program designs.
 5. Implementers - developing and searching for package bodies.
 6. Maintainers - being able to locate all program documentation.
- By level of user expertise: Need various kinds of query support depending on user background, training, and experience.

4.3.7 Tools Supporting Searching

The tools that support searching could range from database query languages to sophisticated knowledge based interactive query systems supported by graphics capabilities. Some of the obvious methods that might be used are:

- Ad-hoc queries.
- Interactive queries: menus, navigator, etc.
- Interactive rule based derivation of specifications - supported by synthesis and verification techniques (whatever that means).

In practice, there will be several Ada libraries, using different database management systems, running on different hosts and even using different ways to document and classify the library items. There will be a need to standardize on a common subset of item descriptors that will be common to all Ada Libraries. On the other hand it is important to recognize that there will be a great deal of evolution in the library systems as we continue to develop better ways to document and retrieve the library items.

4.3.8 Impact on Software Engineering Methodology

Finally, we examined what impact the notion of software component reusability would have on software engineering methodology. First, we recognized that building systems from reusable components placed a greater emphasis on being able to specify the function and design of a system, so that its parts could be obtained from the library and assembled into programs. This concept may be thought of as programming in the large or "hyper-programming", a new term referring to the concept of program development through composition of smaller components (usually Ada packages).

We also identified the need to support the reuse of designs that are similar to what we want but must be modified before they can be used.

We felt that the concepts needed to support programming for reusability are new concepts and not embedded in most current programming practice. Considerable training would be needed to support the development of reusable components and more training will be required to effectively use Ada libraries.

We felt that each site should have a Librarian whose job would be to maintain the Ada library,

add new items to the library, and to support library searching. The librarian position would be a senior position requiring substantial expertise and perhaps a strong background in reading formal specification would be an asset. The effectiveness of a programming department will be largely dependent on how well the Ada library is utilized.

4.4 Applications

This subsection contains the report of the Applications Working Group, chaired by Samuel Redwine. The report was written by Karl Levitt, and the working group consisted of:

- Gordon Bradley, Naval Postgraduate School
- Paul Cohen, Defense Communications Engineering Center -- pcohen@usc-eclb
- James Dempsey, GTE R&D
- Mary Farthofer, IBM FSD
- William Johnson, Naval Postgraduate School
- Steve Leung, ESL/TRW
- Karl Levitt, SRI International -- levitt@sri-csl
- Carol Morgan, AJPO -- morgan@usc-eclb
- Samuel Redwine, Mitre Corporation - redwine@mitre
- Bruno Witte, NOSC -- bwitte@usc-eclb

4.4.1 Initial Questions

The initial questions posed for consideration by the working group were:

1. What are the available technologies?
 - Component Libraries
 - Program Composers
 - Very-High-Level (Application-Oriented) Languages
 - Application Generators
 - Knowledge-Based Systems
2. What are criteria for determining if an application area is ready for reusability?
3. What are approaches for matching a reusability technique with an application?
4. Does the supporting software engineering environment have to be application-specific?
5. What investment strategies will facilitate the introduction of reusability?
6. Are there special problems with the suitability of Ada for application areas?

4.4.2 The Issue of Incentives

It is clear that in the long run, reusability must be cost effective if it is to be a viable approach to system development. However, in the short term it may be necessary for DoD to induce contractors into making reusability a part of their projects. A similar approach is now being taken with Ada itself; for example contractors are most likely increasing their bids for jobs that require Ada -- either as an implementation language or as a PDL -- in order to cover the time required to learn Ada. The question we discussed is how DoD can prime the pump to get reusability quickly into the market place. Among the approaches are:

- AJPO, STARS and other agencies fund the initial creation of libraries and supporting environments, fund the initial documentation and maintenance of entities, and establish clearinghouses that would eventually be self-supporting.
- DoD agencies identify specific systems as strong candidates for reusability. Initially, these efforts would try to be concerned with creating systems that could be reusable; subsequent efforts would be mandated to use the previously constructed entities.
- If DoD is to have a chance of transferring responsibility for the library to the private sector, the royalty system must be carefully worked out. One obvious approach is to give royalties for contributions that are used. The question is how what to charge? In the long run, "best sellers" will survive -- as in any other venture.
- Allow contributors to retain proprietary rights. Again this is obvious.
- Remind contributors that by contributing to the library, they will get free validation. This incentive is likely to be important while the clientele is being built up.

4.4.3 Impediments and Potential Disadvantages to Reusability

It is feared that the program managers for large systems will always take a short-term view, rejecting the long-term benefits of reusability. Also, it should be noted that these managers often do not last for very long periods.

4.4.4 General Approaches to Application-Oriented Reusability

We spent considerable time in reviewing the available approaches to reusability that seem to have promise in the short term. In identifying approaches, we treated reusability as a technique for reducing the amount of work required of a developer as compared with the conventional manner of his creating high level code from scratch. Our emphasis was on techniques at the level of implementation, in part keeping with the Ada theme of the workshop. Consequently, we did not discuss in detail approaches based on reusability of designs or of requirements. Among the approaches we discussed were:

- **Component Libraries.** This is perhaps the most obvious approach, and the most similar to the practice of reusability in hardware -- at least at the level of integrated circuits. Components in Ada are likely to be packages, and as discussed extensively in Goguen's paper for the workshop, generality is achieved through the use of generics. However, there are other approaches to making components reusable, even beyond as intended by their developers. For example, a *preprocessor* could transform data from a particular application into a form acceptable as input to the component, and a *postprocessor* could transform the data output of the component back into the required form. Clearly, efficiency considerations could preclude the use of this technique for critical inner loops.
- **System Families.** A system family is a description of a system from which a number of specific implementations (family members) can be realized. To make the concept more concrete, the realization is to take place by specific *instantiations* of the family, for example, assigning a value to a constant, or eliminating certain operations. With such relatively straightforward instantiations, the system family can be viewed itself as being an implementation or very close to an implementation, but a family member is a sub-implementation that is better suited (i.e., with respect to performance) than the system family for a particular application. An Ada package with generics could be viewed as a system family, but a more interesting family is likely to consist of many packages, e.g., an operating system family.
- **Very-High-Level Languages (VHLL).** A VHLL is usually a language providing very powerful constructs that are oriented to a particular domain or application. Prolog might be considered a VHLL, supporting backtracking and Horn-clause deduction as one might use in developing a mechanical theorem prover. Of course, the output of a Prolog compiler could be assembly code, but any language can be the target. Usually, no manipulation of the object code is possible, but if the object language is a reasonably high level language (e.g., Ada), then the object code might be optimizable beyond that achievable by the Prolog compiler. Spread Sheets and Report Generators are other examples of VHLLs, as are view languages for a relational database. It is often assumed that hand-crafted object code is superior to mechanically compiled code from a VHLL. However, this need not be universally true. For example, an optimizer for a particular construct of a VHLL could be designed to produce very efficient code. Furthermore, special purpose hardware could lead to even more efficiency. For example, one construct in Prolog is concerned with the unification of terms. This task is well-suited to special purpose hardware -- and easily allocated to such hardware when it arises in the execution of the program.
- **Application Generators.** These differ from VHLLs in a number of ways. First, the user is more involved in making decisions that impact the efficiency of the object code. In this

sense, the application generator is more of an environment than just a VHLL. An example of an application generator is a parser generator, which will produce a parser from the production rules of a language. Although, a parser can be generated from any grammar in a particular form (e.g., LALR-1), some kinds of productions will lead to more efficient parsers. A compiler-compiler is a more complex example of an application generator where extensive user interaction is required, particularly in the code generation phase.

- Knowledge-Based Systems (KBS). A KBS is an approach to system design that requires still more user interaction than an application generator. In its most general form, the user input consists of a database and rules to transform the database, the output of the KBS being an executable program. What is particularly attractive about the approach is user's ability to produce a more comprehensive generator by adding more rules or making the existing rules more general. The research on program synthesis and transformation is in this spirit, although the goals are ambitious: the generation of efficient programs independent of application. More short-term work is concerned with more limited domains, e.g., searching particular spaces, or planning.
- Combination of Methods. It is likely that no single method will suffice when the goal is a significant-size program, hence the need for combining the available methods. As in SRI's "strawman" approach (see the paper by Goguen), we envisioned three levels in a system, each with its own kind of reusability. At the lowest level, there would be component reusability, where the identifiable components would be common data types (such as lists, queues, etc.), common data transformers (such as digital filters) or simply common functions (such as mathematical packages). At the next higher level there would be more complex packages (such a statistical package or a linear programming package), which could be viewed as employing the system families approach to reusability. An operating system family would be another good example for this level. At the highest level would be the application interface. Application generators or knowledge-based systems would be most useful here.

4.4.5 Initial Candidate Applications

We devoted considerable discussion to identifying candidate applications that would lead to a large class of reusable entities and would determine the feasibility of applying reusability to real applications. The applications should not be so complex that excessive effort is required to just carry out the development. On the other hand, toy efforts will not be convincing. The list of applications we came up is the following, although the interests of the working group members clearly influenced the choices: communications systems (e.g., protocols such as TCP), direction

finder, navigation, avionics, simulation, and decision support. The interfaces required by these applications would be the "highest level" in our hierarchical approach. At the intermediate level, we strongly recommended consideration of an operating system family together with a collection of instantiations for producing family members.

4.4.6 Discussion

Johnson: Mary, could you explain to us what were the problems that IBM Federal Systems had with reusability?

Farthofer: There are some cost-benefit tradeoffs associated with reusability. It is hard to reuse parts that were made with obsolete technology [Editor's note: e.g., Fortran], or parts that were not originally intended to be reused. New products should take advantage of new technology. Also, if problems show up late in the lifecycle, they can be very expensive to correct; we would have discovered these problems earlier if we had not tried to reuse these old components. Reuse has to be supported.

I. Schedule of Workshop

Tuesday, November 1

- 9:00-9:30 Schaar & Kramer "Why DoD Needs S/W Environments"
 9:30-10:00 Redwine "Conceptual Architecture for a S/W Engineering Environment"
 10:00-10:30 Break
 10:30-12:00 Levitt "An Overview of Ada Libraries"
 12:00-1:30 Lunch
 2:00-2:30 Goguen "LIL: A Library Interconnection Language for Ada Programs"
 2:30-3:00 Break
 3:30-4:00 Rudmik "DCP Approach to Ada Libraries"
 4:00-4:30 Matsumoto "Flexibility vs. Efficiency for Reusable Components"
 5:00-5:30 Discussion
 5:00-5:30 Litvintchouk "Mapping Clear Specifications to Ada Packages"
 5:30-6:00 Witte "General Requirements for an Elementary Math Functions Libraries"

Wednesday, November 2

- 9:00-9:30 Kotik "Knowledge Based Tools for Data Type Implementation"
 9:30-10:00 Meseguer "Library Organization and User Interfaces"
 10:00-10:30 Break
 10:30-11:00 Tichy "Version Control in Program Libraries"
 11:00-11:30 von Henke "Using ANNA for Specifying and Documenting Ada Packages"
 11:30-1:30 Lunch
 1:30-3:00 Working Group Meetings
 3:00-3:30 Working Group Reports

Thursday, November 3

- 9:00-10:00 Working Group Meetings
 10:00-10:30 Break
 10:30-12:30 Working Group Reports
 12:30-1:30 Lunch
 1:30-3:00 Discussion

II. Names and Addresses of Participants

This appendix gives the names and addresses of participants, with their Arpanet addresses, if available.

Charles Arnold
Naval Underwater System Center
New London, CT 06320
(203) 447-4319

David J. Babcock
ROLM Corp. - MSC Division
Manager, Software Development
M/S 150
One River Oaks Place
San Jose, California 95134
(408) 942-7702
babcock@usc-eclb

Gordon Bradley
Naval Postgraduate School
Computer Science Department
Monterey, CA 93940

Tom Brown
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304
(415) 494-2233
brown@kestrel

Paul M. Cohen
Defense Communication Engineering Center
Code R620
1860 Wiehle Avenue
Reston, VA 22090
(703) 437-2176
pcohen@usc-eclb

James B. Dempsey
GTE R&D
2500 W. Utopia Road
Phoenix, Arizona 85027
(602) 582-7532

Mary Forthofer
IBM FSD
MC 70
1322 Space Park Drive
Houston, Texas 77058
(713) 333-3300

Timothy Gill
Wang Institute of Graduate Studies
Tyngsboro, MA 01879
(617) 649-9731

Joseph A. Goguen
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-5454
goguen@sri-csl

LCDR W. C. Johnson
SMC 1509
Naval Postgraduate School
Monterey, CA 93940
(408) 372-4602

Beverly I. Kedzierski
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304
(415) 494-2233 x2132
kedzierski@kestrel

Gordon Kotik
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304
(415) 494-2233
kotik@kestrel

Jack Kramer
Institute for Defense Analysis
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2263
kramer@usc-eclb

Steve Leung
ESL/TRW, MS 302
495 Java Drive
Sunnyvale, CA 94086
(408) 738-2888 x5372

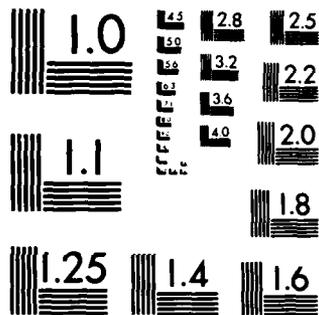
Karl N. Levitt
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-4172
levitt@sri-csl

Steve Litvintchouk
Raytheon Company
P. O. Box 360
Portsmouth, RI 02871
(401) 847-8000 x4018
brunix!rayssd!sdl@ucb-vax

Bob Mathis
Institute for Defense Analysis
1801 N. Beauregard Street
Alexandria, VA 22311
(703) 845-2263
kramer@usc-eclb

Allen S. Matsumoto
ITT Programming
1000 Oronoque Lane
Stratford, CT 06497
(203) 375-0280 x501

Jose Meseguer
SRI International
Computer Science Lab
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 859-3044
meseguer@sri-ai



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Carol Morgan
AJPO
Room 3D139 (400AN), Pentagon
Washington, DC 20301
(202) 694-0210
morgan@usc-eclb

Gilbert Myers
Naval Ocean Systems Center
Code 8322
San Diego, CA 92152
(619) 225-7401
myers@usc-eclb

Samuel T. Redwine, Jr.
Mitre Corporation
1820 Dolly Madison Blvd.
McLean, VA 22102
(703) 827-6080
redwine@mitre

Andres Rudmik
GTE R&D
2500 W. Utopia Road
Phoenix, Arizona 85027
(602) 582-7518

Brian Schaar
Ada Joint Program Office
3D139 (400AN) Pentagon
Washington, DC 20301
(202) 694-0280
schaar@usc-eclb

John Schill
Naval Ocean Systems Center
Code 8322
San Diego, CA 92152
(619) 225-2264
schill@isia

Roger Smeaton
Naval Ocean Systems Center
Code 8321
San Diego, CA 92152
(619) 225-2083
smeaton@nosc-tecr

Walter Tichy
Purdue University
Department of Computer Science
West Lafayette, IND. 47907
(317) 494-1998
tichy@purdue

Friedrick W. Von Henke
Stanford University
Computer Science Laboratory
Stanford, California 94040
current address: SRI International
333 Ravenswood Avenue
Menlo Park CA 94025
(415) 859-2560
vonhenke@sri-csl

Bruno Witte
NOSC
Code 8315
San Diego, CA 92152
(619) 225-7945
bwitte@usc-eclb

III. An Example of LIL

This appendix gives a somewhat longer example in LIL, a generic resource manager. Most of the work on this example was done by Dr. Jose Meseguer.

```
theory TRIV is
  types ELT
end TRIV
```

```
theory POSET is
  types ELT
  functions < : ELT ELT -> BOOLEAN
  vars E1 E2 E3 : ELT
  axioms
    (E1 < E1)
    (E1 < E3 if E1 < E2 AND E2 < E3)
    (E1 = E2 if E1 < E2 AND E2 < E1)
end POSET
```

```
theory EQV is
  types ELT
  functions == : ELT ELT -> BOOLEAN
  vars E1 E2 E3 : ELT
  axioms
    (E1 == E1)
    (E1 == E3 if E1 == E2 AND E2 == E3)
    (E1 == E2 if E2 == E1)
end EQV
```

-- for any POSET, there is a natural way to define an equality;
 -- that is the content of the following, which involves a derived operation:

```
view EQ :: EQV => POSET is
  vars E1 E2 : ELT
  ops (E1 == E2 => E1 < E1 AND E2 < E1)
end EQ
```

-- now an example of top-down reusable development, RESOURCE using
 -- TABLE, which has not yet been defined

```
generic package RESOURCE[ACCESSOR :: EQV; X :: TRIV] is
  using TABLEP :: TABLE[ACCESSOR, X] is
  functions
    ACC-OK : ACCESSOR -> BOOLEAN
  procedures
    WRITE : ACCESSOR X
    READ : ACCESSOR -> X
    ....
  exceptions
    WRONG-ACC
  vars A : ACCESSOR; X : X
  axioms
    WRITE(A,X) = PUT(A,X) if ACC-OK(A)
    WRITE(A,X) = WRONG-ACC if NOT ACC-OK(A)
    ....
end RESOURCE
```

```
generic package TABLE[ENTRY :: EQV; X :: TRIV] is
  state TABLE initially EMPTY
  procedures
    PUT : ENTRY X
    LOOKUP : ENTRY -> X
  vars E : ENTRY; X : X
  axioms
    PUT(E,X); LOOKUP(E) = X
    ....
end TABLE
```

```
generic package SEC-MEMORY-ACCESSOR[LEVEL :: POSET]
  needs MEMORYP :: MEMORY is
  types S-MEM-ACCESSOR is
  record
```

CELL : CELL; -- cell is part of the MEMORY package

ACCESSING-LEVEL : LEVEL;

ACCESSED-LEVEL : LEVEL;

end record

functions

S-MEM-ACC-OK : S-MEM-ACCESSOR -> BOOLEAN

vars SMA : S-MEM-ACCESSOR

axioms

ACC-OK(SMA) = S-MEM-ACC-OK(SMA)

....

end SEC-MEM-ACCESSOR

make SECURE-MEM-MANAGER-0[LEVEL :: POSET; X :: TRIV] is

RESOURCE[SEC-MEM-ACCESSOR[LEVEL], X] needs TABLEP => TABLE.HASH1

end SECURE-MEM-MANAGER-0

make SECURE-MEM-MANAGER[LEVEL :: POSET; X :: TRIV]

using SEC-MEM-TABLE = TABLE[IDENTIFIER, PAIR[CELL, LEVEL]] (hidden)

using SECURE-MEM-MANAGER-0[LEVEL, X] (hidden) is

procedures

REQUEST-WRITE : IDENTIFIER X LEVEL

REQUEST-READ : IDENTIFIER LEVEL -> X

....

end SECURE-MEM-MANAGER

IV. Slides from Prepared Lectures

This appendix reproduces the slides used in the prepared lectures that were presented to the workshop, when these are available and reproducible.

IV.1 Conceptual Architecture for a Software Engineering Environment

SAMUEL T. REDWINE, JR.

CONCEPTUAL ARCHITECTURE
FOR A
SOFTWARE ENGINEERING ENVIRONMENT

Samuel T. Redwine, Jr.
1 NOV 1983

MOTIVATION/STATUS

- Aid Initial Planning of Joint Service Software Engineering Environment Effort
- Analogous to Architect's Preliminary Sketches
- No Official Status

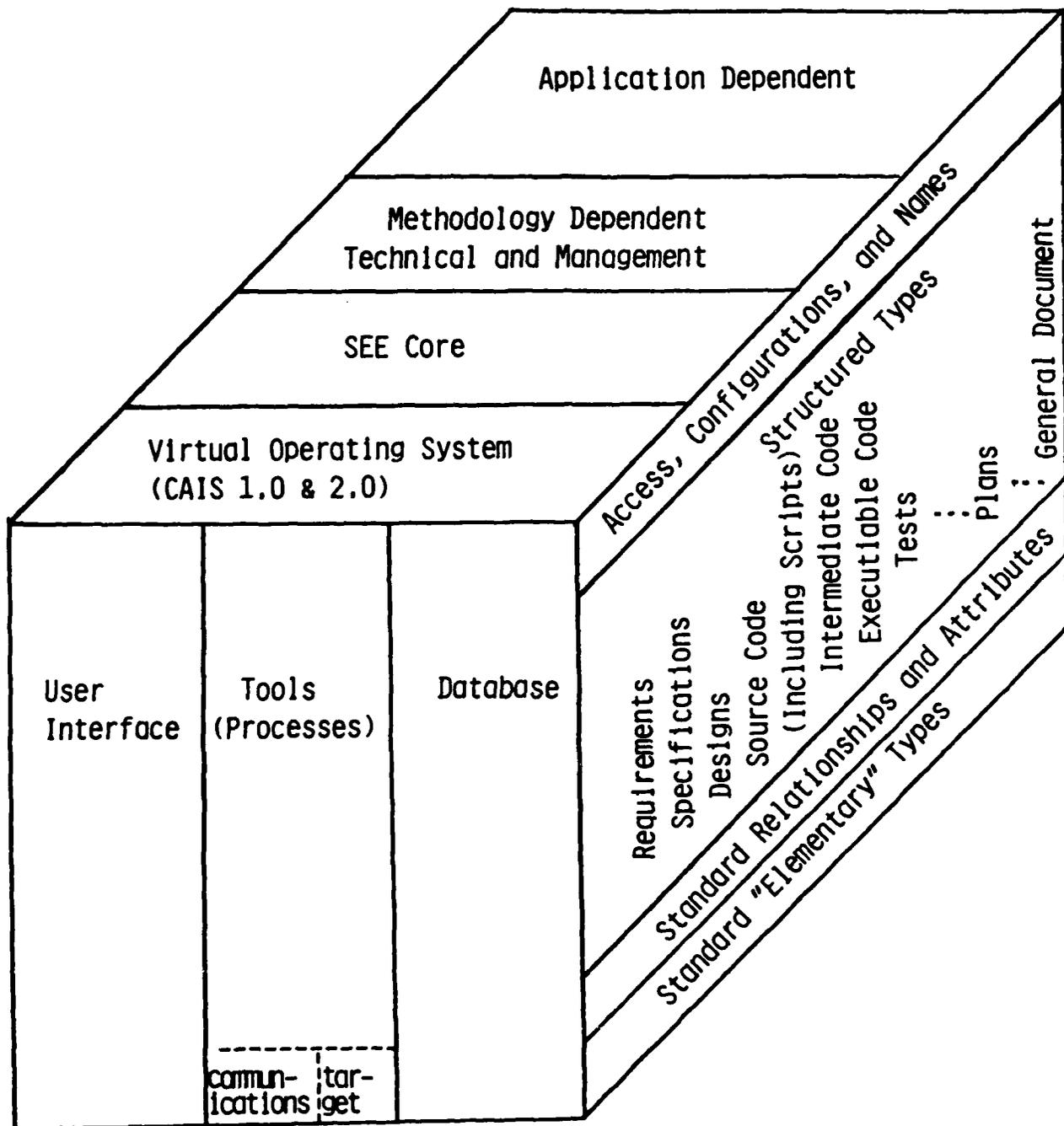


FIGURE 1
SOFTWARE ENGINEERING ENVIRONMENT ARCHITECTURE SKETCH

Sam Redwine
6 Oct 83

TWO-DIMENSIONS WITH EXAMPLES

	<u>User Interface</u>	<u>Tools (Processes)</u>	<u>Database</u>
Virtual OS	Virtual Terminals	Process Nodes Process Observer	File Nodes Exchange Formats
SEE Core	User Interface Management System	Process Mgt. System Office Automation	Well Understood Work Products Database Maintenance & Inquiry/Reporting Content/Structure Definition Facility
Methodology Dependent	SADT Graphics	HDM Too Stri ESD Management Reporting	OBJ Specification
Application Dependent	Heads Up Display Program Generator Display Mapper	Application Generator	Application-Oriented Language Source Code

STRUCTURE

- Access, Configuration, and Name Management
- Structured Types (Work Products)
 - Systems Level
 - Software Requirements/Specifications
 - Software Designs
 - Code
 - Tests
 - User/Operator Documentation
 - Management Data
 - (- Standard STARS Measurements)
- Standard Relationship and Attributes
- Standard "Elementary" Types
 - Standard Meta-Grammar(s)

DATABASE/SEE CORE

Components/Functions Include:

- **Data Maintenance (Connect Directly to User Interface Management System)**
 - **Entry** - **High Level**
 - **Change** - **Low Level**
 - **Inquiry/Reporting** - **Graphics**
 - **Integrity Checking**
- **Content/Structure Definition Facility**
 - **Dynamic Typing**
 - **Grammar for Defining Notations**
- **Dictionary/Directory**
- **Software Catalog/Warehousing/Distribution**

DATABASE/VIRTUAL OS

Components/Functions Include

- File Node Capabilities
- Exchange Format Standards Between SEE Instantiations
- Database Management System (Partial)
 - At Least Bootstrap Loader of Program Library
 - At Most DBMS Primitives

Issues Include

- How Much of DBMS Functions Here Versus SEE Core

DATABASE/SEE CORE

Issues Include:

- Database Models Used
- Form and Power of Definitional and Generative Facilities
 - Data Maintenance
 - Metrics
 - Consistency and Completeness Analysis
- Database Must Reflect the Dynamic and Iterative Nature of Software Process (IST)
- User Needs to be Able to Maintain Clear Views of Database at Both Coarse and Detailed Levels (IST)
- Danger Exists of Wasteful Use of Storage Because Large Amounts of Data Will Differ Only in Trivial Aspects (IST)

SUMMARY

- SEE Architecture Gives One Type of Context for Database
- Database Concerns Include:
 - Interaction With Remainder of SEE
 - Contents/Structure
 - Content/Structure Description Facility and Related (Generative) Services
 - Appropriate Standardization

IV.2 An Overview of Ada Libraries

KARL LEVITT

A N O V E R V I E W O F A D A L I B R A R I E S

- TOWARDS A "THEORY" AND "PRACTICE" OF REUSABILITY
- AVAILABLE TECHNOLOGY
- NEEDED NEW TECHNOLOGY

Karl Levitt
Joe Goguen
Jose Meseguer

SRI INTERNATIONAL

WHY REUSABILITY

- REDUCED COST
- ALLOW LARGER SYSTEMS TO BE BUILT
(PARKINSON'S LAW)
- BETTER RELIABILITY
- REAL ENCOURAGEMENT FOR CRACK SYSTEM BUILDERS
- PROVIDE JOBS FOR UNEMPLOYED MATHEMATICIANS
(FORMAL SPECIFICATION AND VERIFICATION?)

QUESTIONS TO BE DISCUSSED

- WHAT SHOULD BE IN A LIBRARY?
PROGRAM UNITS, DESIGNS, DOCUMENTATION...
- HOW CAN LIBRARY ENTITIES BE COMPOSED?
MODULE INTERCONNECTION LANGUAGE...
- HOW TO RETRIEVE LIBRARY ENTITIES?
CATALOGING, SEARCHING, ...
- IS THE ABSTRACTION MECHANISM PROVIDED BY ADA ADEQUATE?
NECESSARY BUT NOT SUFFICIENT
- WHAT ABOUT ADA + APSEs ?
NOT QUITE
- HOW WILL USERS UNDERSTAND LIBRARY ENTITIES?
INFORMAL DOCUMENTATION, SPECS, ANIMATION, ...

- HOW TO COUPLE REUSABILITY WITH THE LIFECYCLE?

REQUIREMENTS + DESIGN METHODOLOGIES

- WHAT ABOUT MANAGEMENT ISSUES?

INVESTMENT, QUALITY CONTROL, ENCOURAGEMENT

- WHAT EXPERIMENTS SHOULD BE CONDUCTED?

OPERATING SYSTEMS, DATABASE SYSTEMS, PROCESS CONTROL

- IS THE CURRENT TECHNOLOGY ADEQUATE?

STANDISH -- YES

WORKSHOP -- ??

ANALYSIS

FUNCTIONAL DESIGN

DETAILED DESIGN

IMPLEMENTATION

VALIDATION

REQUIREMENTS MODELS

PROJECT MANAGEMENT

FAMILIES

DESIGN METHODOLOGIES

PACKAGES

MODULES

SPECS FORMAL INFORMAL

PACKAGE BODIES

TRANSFORMATIONS

CODE SKELETONS

TEST FAMILIES

VERIFICATION



WHAT ADA PROVIDES

- PACKAGES " SPECIFICATION + BODY
- SEPARATE COMPILATION
- PRIVATE DATA TYPES
- REUSABLE NAMES AND OVERLOADING
- GENERICS AND PARAMETERIZATION

WHAT APSES WILL ADD

- BASIC TOOLS
- VERSION CONTROL

WHAT ELSE IS NEEDED

"TRUE" REUSABILITY

RESTRICTIONS ON PARAMETER INSTANTIATION

CLEARER SEPARATION OF DESIGN AND
IMPLEMENTATION

GUIDELINES ON WHAT SHOULD BE IN
LIBRARY

DOCUMENTATION AND RETRIEVAL

Constructing New Entities From Old

1. Set a constant (such as the maximum depth of a stack);
2. Substitute one entity for a "stub" or parameter in another;
3. Sew together two (possibly large) entities along a common interface;
4. Instantiate the parameters of a generic entity;
5. Enrich an existing entity with some new features;
6. Hide (abstract, or encapsulate) some features of an existing entity; this could include both data abstraction and control abstraction;
7. "Slice" an entity, to eliminate some unwanted functionality;
8. Implement one abstract entity using features provided by others (this leads to the notion of a vertical hierarchy of entities);
and
9. Assemble existing entities over a skeleton. This skeleton might be either fixed or flexible; for example, it might be determined heuristically by an expert system.

HYPERPROGRAMMING

=

DESIGN

+

IMPLEMENTATION

- SELECT PACKAGE SPECS

- ASSEMBLE INTERFACES

- VALIDATE COMPOSITION

- SELECT OR PRODUCE PACKAGE BODIES

HYPERPROGRAMMING

HORIZONTAL

VIEWS

STRUCTURING

AGGREGATING

COMBINE

INSTANTIATE

TRANSFORMATIONS

VERTICAL

EDITING

ABS MACHINE

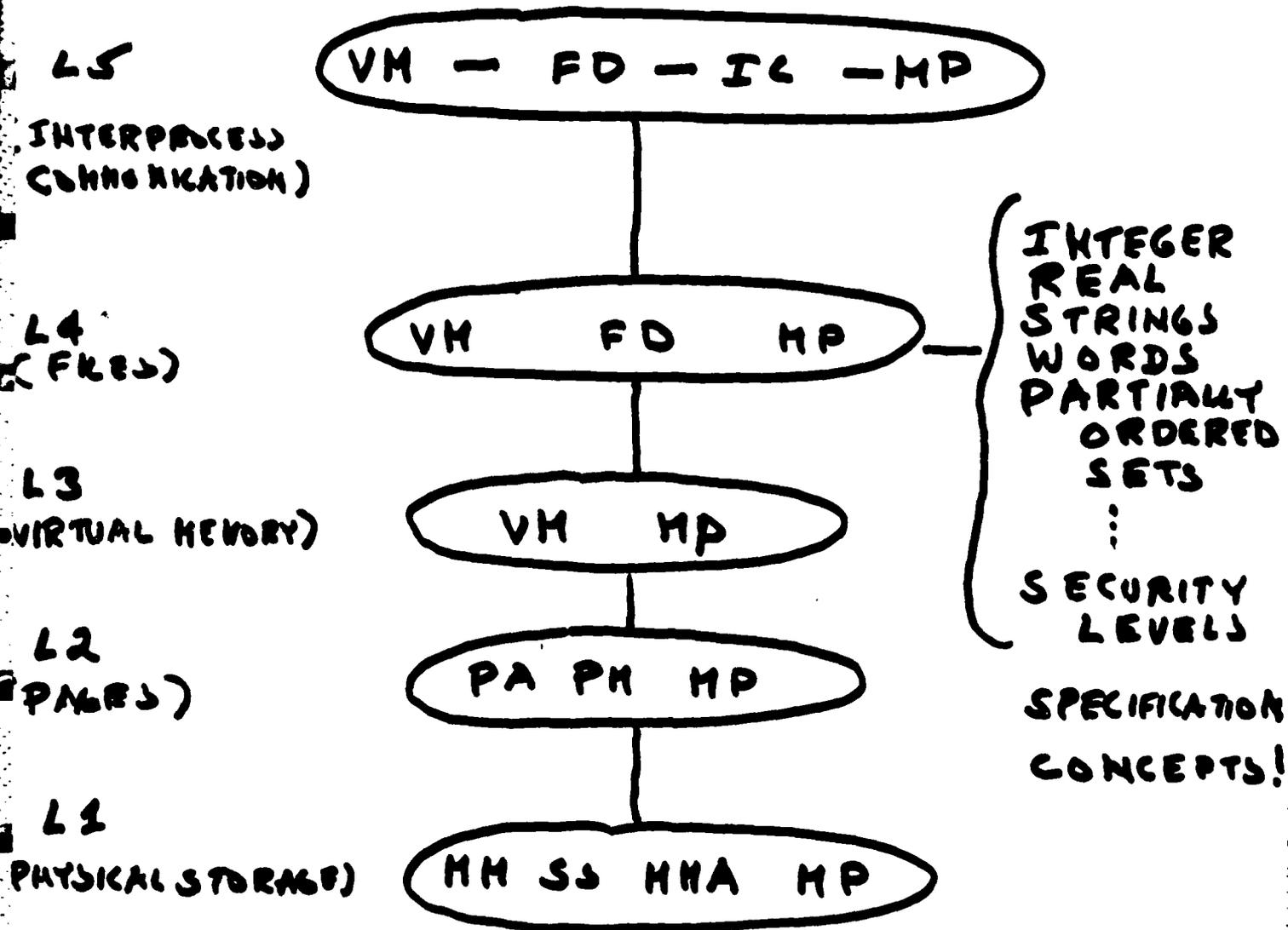
CODE

REALIZE

COMPOSE

OPTIMIZE

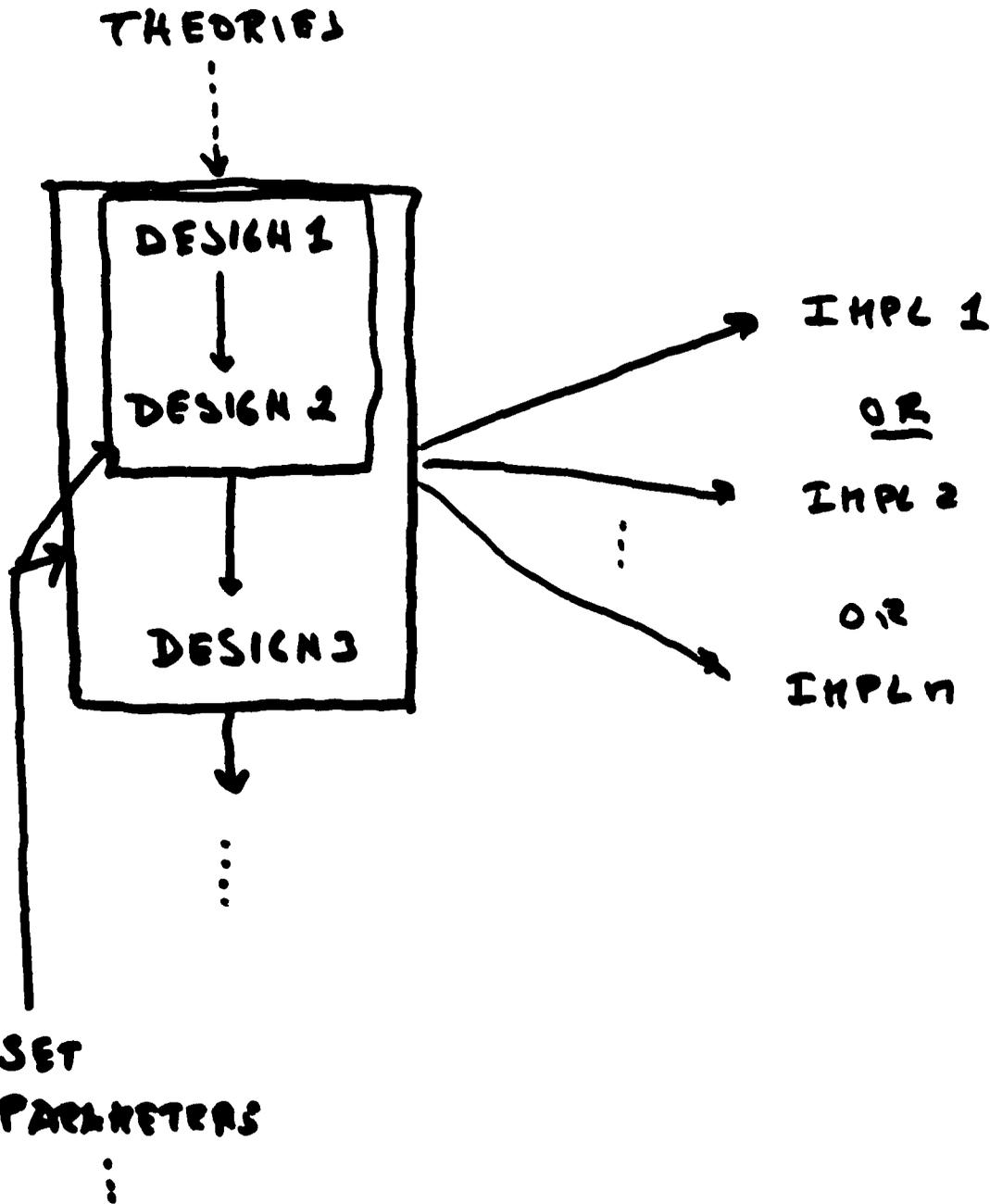
HORIZONTAL AND VERTICAL DECOMPOSITION



ANNOTATION

- | | | | |
|----|--------------------|-----|-------------------|
| VM | VIRT. MEM | SS | SECONDARY STORAGE |
| FD | FILE DIRECTORIES | MMA | MEMORY MAPPING |
| IC | INTERPROCESS COMM. | | |
| MP | MULTIPLE PROCESSES | | |
| PA | PAGES | | |
| PH | PAGE MAPPING | | |
| MM | MAIN MEMORY | | |

WORKING CODE FROM ENTITIES



ENTITIES IN A DESIGN/IMPLEMENTATION

LIBRARY

PACKAGE	SPECIFICATION PART + DOCUMENTATION IMPLEMENTATION VERSIONS
THEORIES	NON-IMPLEMENTED ENTITIES MAINLY FOR EXPLANATION
GENERIC ENTITIES	RESTRICTIONS ON GENERICS
VIEWS	HOW AN ENTITY SATISFIES A THEORY
INSTANTIATION	INSTANTIATE PARAMETERS WITH ENTITIES
PACKAGE STUBS	HORIZONTAL :: DESIGN VERTICAL :: CODE
TRANSFORMATIONS	HORIZONTAL :: COMBINE, ENRICH, .. VERTICAL :: ABSTRACT MACHINE CORRECTNESS PRESERVING
CONTROL ABSTRACTION	ITERATION OVER SETS, ..
CODE SKELETONS	UNINTERPRETED FUNCTIONS

EXAMPLE

DEVELOP A RESOURCE MANAGER OF
SECURE DATA.

WRITE: INSERT AN ENTRY INTO A TABLE;
ALLOWED ONLY IF $SLW \leq SLT$;

READ: RETURN ENTRY FROM TABLE;
ALLOWED ONLY IF $SLT \leq SLR$

NOTES:

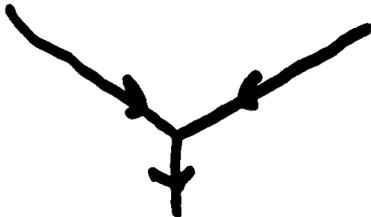
1. A SECURE RESOURCE MANAGER IS TOO
SPECIALIZED FOR A LIBRARY ENTITY.
2. BETTER TO PRODUCE IN A SERIES
OF STEPS FROM A GENERAL
RESOURCE MANAGER

DESIGN

IMPLEMENTATION

EQUIVALENCE

POSET



VIEW (EQ → POSET)



SECURITY LEVELS



SECURE-MEMORY-ACCESSOR



RESOURCE (GENERIC EXCEPTIONS)



TABLE (INFINITE)

HASHING



- THEORY
- PACKAGE SPEC
- PACKAGE BODY

A Generic Secure Resource Manager in LIL

```
theory TRIV is
  types ELT
end TRIV
```

```
theory POSET is
  types ELT
  functions  $\leq$  : ELT ELT -> BOOLEAN
  vars E1 E2 E3 : ELT
  axioms
    (E1  $\leq$  E1)
    (E1  $\leq$  E3 if E1  $\leq$  E2 AND E2  $\leq$  E3)
    (E1 = E2 if E1  $\leq$  E2 AND E2  $\leq$  E1)
end POSET
```

```
theory EQV is
  types ELT
  functions == : ELT ELT -> BOOLEAN
  vars E1 E2 E3 : ELT
  axioms
    (E1 == E1)
    (E1 == E3 if E1 == E2 AND E2 == E3)
    (E1 == E2 if E2 == E1)
end EQV
```

-- for any POSET, there is a natural way to define an equality:

```
view EQ :: EQV => POSET is
  vars E1 E2 : ELT
  ops (E1 == E2 => E1  $\leq$  E1 AND E2  $\leq$  E1)
end EQ
```

-- now a top-down reusable development, RESOURCE using
-- TABLE, not yet been defined

generic package RESOURCE[ACCESSOR :: EQV; X :: TRIV] is
using TABLEP :: TABLE[ACCESSOR, X] is

functions

ACC-OK : ACCESSOR -> BOOLEAN

procedures

WRITE : ACCESSOR X

READ : ACCESSOR -> X

....

exceptions

WRONG-ACC

vars A : ACCESSOR; X : X

axioms

WRITE(A,X) = PUT(A,X) if ACC-OK(A)

WRITE(A,X) = WRONG-ACC if NOT ACC-OK(A)

....

end RESOURCE

generic package TABLE[ENTRY :: EQV; X :: TRIV] is
state TABLE initially EMPTY

procedures

PUT : ENTRY X

LOOKUP : ENTRY -> X

vars E : ENTRY; X : X

axioms

PUT(E,X); LOOKUP(E) = X

....

end TABLE

```

generic package SEC-MEMORY-ACCESSOR[LEVEL :: POSET]
  needs MEMORYP :: MEMORY is
  types S-MEM-ACCESSOR is
    record
      CELL : CELL; -- cell is part of the MEMORY package
      ACCESSING-LEVEL : LEVEL;
      ACCESSED-LEVEL : LEVEL;
    end record
  functions
    S-MEM-ACC-OK : S-MEM-ACCESSOR -> BOOLEAN
  vars SMA : S-MEM-ACCESSOR
  axioms
    ACC-OK(SMA) = S-MEM-ACC-OK(SMA)
  ....
end SEC-MEM-ACCESSOR

make SECURE-MEM-MANAGER-0[LEVEL :: POSET; X :: TRIV] is
  RESOURCE[SEC-MEM-ACCESSOR[LEVEL], X] needs TABLEP => TABLE.HASH1
end SECURE-MEM-MANAGER-0

make SECURE-MEM-MANAGER[LEVEL :: POSET; X :: TRIV]
  using SEC-MEM-TABLE = TABLE[IDENTIFIER, PAIR[CELL, LEVEL]] (hidden)
  using SECURE-MEM-MANAGER-0[LEVEL, X] (hidden) is
  procedures
    REQUEST-WRITE : IDENTIFIER X LEVEL
    REQUEST-READ : IDENTIFIER LEVEL -> X
  ....
end SECURE-MEM-MANAGER

```

USING VISUALIZATION TO DOCUMENT AND UNDERSTAND LIBRARY ENTITIES

DESIGN LEVEL

WHAT SYSTEM DOES

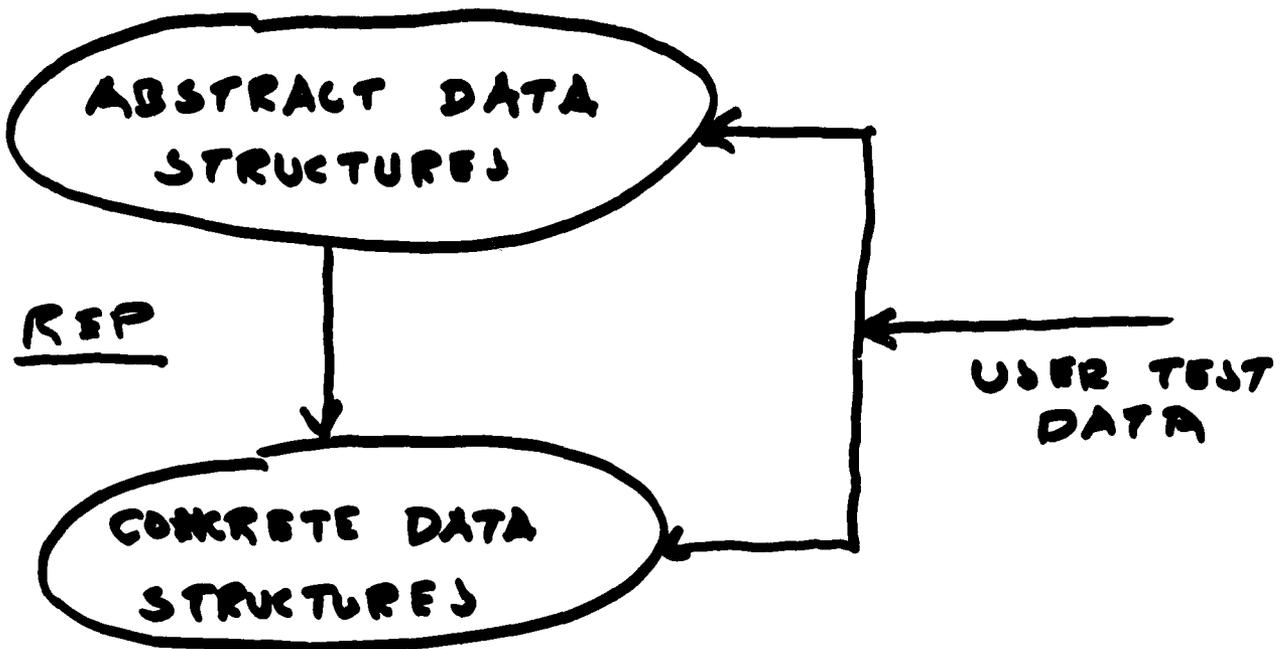
USE ABSTRACT DATA
STRUCTURES

IMPLEMENTATION
LEVEL

HOW SYSTEM IS STRUCTURED

USE CONCRETE DATA
STRUCTURES

SHOW RELATIONSHIP AMONG
PROGRAM ENTITIES



ABSTRACT TABLE : TABLE : WORD \rightarrow BOOL



REP



CONCRETE ARRAY : ARRAY : INTEGER \rightarrow WORD

REP: TABLE (WORD)

\equiv

$\exists ! i : \text{ARRAY}(i) \in \text{WORD}$

\vdots

IV.3 LIL: A Library Interconnection Language for Ada Programs

JOSEPH A. GOGUEN

LIL:
A Library Interconnection Language
for Ada

J. A. Goguen

SRI International

Plagiarize! Plagiarize!

Let no one else's work evade

Your eyes!

-- Tom Lehrer

Parameterize! Parameterize!

Let no one else's code evade

Your eyes!

-- LIL

LIL is a Module Interconnection Language for Ada.

In particular, it respects the general structure of Ada,

including

- * use clauses,
- * generics,
- * separate compilation.

Note that the Ada separate clause supports top-down

program development and is different from Ada generics.

However, this feature prevents the reuse of the stub package.

Issues

1. What should be in a library? Beyond compiled Ada code are: corresponding uncompiled Ada texts, version and configuration information, requirements, specifications, documentation, transformations, histories and management information.
2. What program composition techniques take maximum advantage of Ada and the library concept? E.g., instantiating, enriching and restricting entities.
3. How to construct families of related programs? (Would transformations and expert systems help?)
4. What documentation and specification techniques yield clear yet mechanizable program descriptions?
5. How to find library entities most relevant to a user needs? What cataloging services (e.g., taxonomies) and reference services (e.g., search strategies)?
6. How to integrate libraries into an APSE? (e.g., with module test, linkage and interpretation facilities)

7. How to best present information to users? What about multi-media (e.g., graphics and natural language) for program composition, retrieval (e.g., clever use of menus and icons), documentation and modification?
8. What about management issues? E.g., policies for investment, quality control, and distributing and encouraging documentation?
9. What experiments could be performed to test the viability of various approaches to these problems?

Main Ideas

1. Systematic (but limited) use of semantics; in particular, explicitly providing theories (which are just sets of axioms) attached to program units via views.
2. A variety of different methods for program construction, so that the process of programming will consist, as much as possible, in the application of these methods, rather than in just writing code; we call this hyperprogramming
3. Maximal use of generic (i.e., parameterized) library entities. This is intended to make them as reusable as possible.
4. Support different levels of formality in axioms, and degrees and kinds of validation (such as informal arguments, testing, and formal proofs); this should support a practical user interface and also aid in pinpointing weak spots during debugging.
5. Facilitation of program understanding by animating abstract data types, and otherwise illustrating and explaining behavior at module interfaces.

Constructing New Entities From Old

1. Set a constant (such as the maximum depth of a stack);
2. Substitute one entity for a "stub" or parameter in another;
3. Sew together two (possibly large) entities along a common interface;
4. Instantiate the parameters of a generic entity;
5. Enrich an existing entity with some new features;
6. Hide (abstract, or encapsulate) some features of an existing entity; this could include both data abstraction and control abstraction;
7. "Slice" an entity, to eliminate some unwanted functionality;
8. Implement one abstract entity using features provided by others (this leads to the notion of a vertical hierarchy of entities);
and
9. Assemble existing entities over a skeleton. This skeleton might be either fixed or flexible; for example, it might be determined heuristically by an expert system.

Some LIL Design Decisions

1. LIL syntax is closer to mathematics than Ada is.
2. The ordinary user of such a library system should not see LIL entities as shown here; the user interface should involve natural language and/or interactive graphics.
3. Several features of Ada are not treated.
 - a. Some are omitted to simplify the discussion; for example, we discuss only functions, but procedures present no serious difficulties
 - b. Others would require further research to provide an adequate treatment; e.g., exceptions and tasking.
4. Most development projects will never use large or complex formal theories, but would rely on informal documentation and informal arguments about program properties. We wish to support both formal and informal specification and verification in an integrated manner, allowing whatever mixture seems most appropriate to the application.
5. LIL could be implemented without significant further research; but other areas, such as transformations, would require substantial further thought to provide an adequate foundation.

The LIL Package

```
package COMPLEX_FUNCTIONS
  using MATH_FUNCTIONS is
  types COMPLEX
  functions
    ** : COMPLEX COMPLEX -> COMPLEX
    .....
  axioms
    -- agrees with real exponentiation for real arguments
    -- E ** I*R = COS(R) + I*SIN(R)
    .....
end COMPLEX_FUNCTIONS
```

1. A LIL package can have zero or more corresponding Ada bodies.
2. Axioms need not be completely formal.
3. Types and Operations can be hidden.

Theories

```
theory TRIV is
  types ELT
end TRIV
```

```
theory POSET is
  types ELT
  functions  $\leq$  : ELT ELT -> BOOLEAN
  vars E1 E2 E3 : ELT
  axioms
    (E1  $\leq$  E1)
    (E1  $\leq$  E3 if E1  $\leq$  E2 AND E2  $\leq$  E3)
    (E1 = E2 if E1  $\leq$  E2 AND E2  $\leq$  E1)
end POSET
```

This describes the interface of a sorting package, including that the elements to be sorted have a suitable ordering relation.

```
theory MONOID is
  types M
  functions * : M M -> M (assoc, id: I)
end MONOID
```

Here assoc indicates that the function * is infix and associative, i.e., satisfies the equation

$$(M1 * M2) * M3 = M1 * (M2 * M3)$$

and id: I indicates that it has an identity I.

Generics

```
generic package LIST[ELT :: TRIV] is
  types LIST
  functions
    . : LIST LIST -> LIST (assoc, id: NIL)
    EMPTY : LIST -> BOOLEAN
    HEAD : LIST -> LIST
    TAIL : LIST -> LIST
  vars E : ELT; L : LIST
  axioms
    HEAD(E . L) = L
    TAIL(E . L) = E
    .....
end LIST
```

The attributes `assoc` and `id` of `"."` implicitly give some further equations, namely the associative law and two identity laws.

All parameters are collected together in the requirement theory, telling what types, functions and procedures are needed, and what properties they must satisfy.

Here is a parameterized theory, vector spaces over a field F.

generic theory VECTOR-SP[F :: FIELD] is

types V

functions

+ : V V -> V (assoc, comm, id: 0)

* : F V -> V

vars F F1 F2 : F;

V V1 V2 : V

axioms

$((F1 + F2) * V = (F1 * V) + (F2 * V))$

$((F1 * F2) * V = (F1 * (F2 * V)))$

$(F * (V1 + V2) = (F * V1) + (F * V2))$

end VECTOR-SP

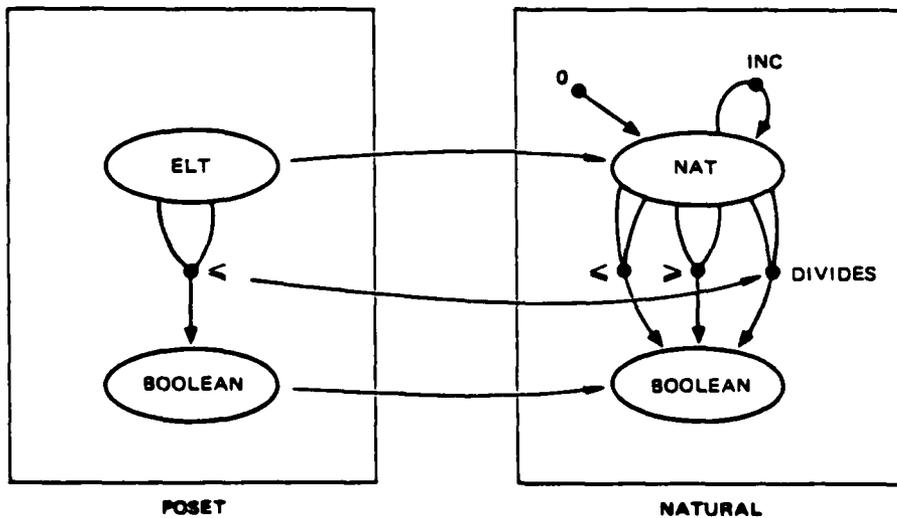
Views

```
view NATD :: POSET => NATURAL is
  types (ELT => NATURAL)
  ops (< => DIVIDES)
end NATD
```

A default view is the one that is used unless another is explicitly provided instead.

```
view NATV :: POSET => NATURAL is
  types (ELT => NATURAL)
  ops (< => <)
end NATV
```

```
view NAT+ :: MONOID => NATURAL is
  ops (* => +)
  (I => 0)
end NAT+
```



THE VIEW NATD:POSET \Rightarrow NATURAL

Instantiation

`SORT[X :: POSET]` can be instantiated using the view `NATD` by

```
make SORT-NATD is SORT[NATD] end
```

to get a package that sorts lists of `NATURALS` by the divisibility relation.

```
make NATLIST is LIST[NATURAL] end
```

uses the default view `TRIV => NATURAL` to instantiate the parameterized entity `LIST` with the actual parameter `NATURAL`.

```
make REAL-LIST is LIST[REAL] end
```

where `REAL` is the field of real numbers, uses a default view `TRIV => REAL`.

```
make REAL-VSP is VECTOR-SP[REAL] end
```

uses a default view `FIELD => REAL`, and

```
make REAL-VSP-LIST is LIST[VECTOR-SP[REAL]] end
```

uses two nested default views.

Here is an example with some interesting instantiations:

```
generic package ITERATE[M :: MONOID]
  using LIST[M] is
  ops ITERATE : LIST -> M
  vars E : M ; L : LIST
  axioms
    (ITERATE(NIL) = I)
    (ITERATE(E . L) = E * ITERATE(L))
end ITER
```

Using the default view TRIV => MONOID.

make SIGMA is ITERATE[NAT+] end
sums a list of numbers.

make PI is ITERATE[NAT+] end
multiplies a list of numbers.

$$\Sigma L = \sum_{i=1}^N L_i$$

$$L = (L_1 \dots L_N)$$

$$\Pi L = \prod_{i=1}^N L_i$$

$$\Sigma \text{NIL} = 0$$

$$\Pi \text{NIL} = 1$$

Package Stubs

```
generic package SORT(X :: POSET)
  needs LISTP :: LIST[X] is
  functions
    SORT : LIST -> LIST
    SORTED : LIST LIST -> BOOLEAN
  vars L : LIST
  axioms
    SORTED(SORT(L)) = TRUE
    .....
end SORT
```

The needs clause says: to provide a generic sorting function, we need a generic Ada package LISTP that is a version of the LIL package LIST[X].

That X is both the formal parameter of SORT and of LIST[X] indicates that the version is instantiated with the same X as SORT[X].

The advantage of this approach is that a generic Ada body for LIST[X] can now be reused, which would be impossible with the Ada separate clause.

To actually get a version of LIST for use in SORT, one gives a module interconnection command indicating which version to use when compiling.

```
make SORT[X] needs LISTP => LIST.HACK end
```

where LIST.HACK is a particular generic body for the LIL generic package LIST[X].

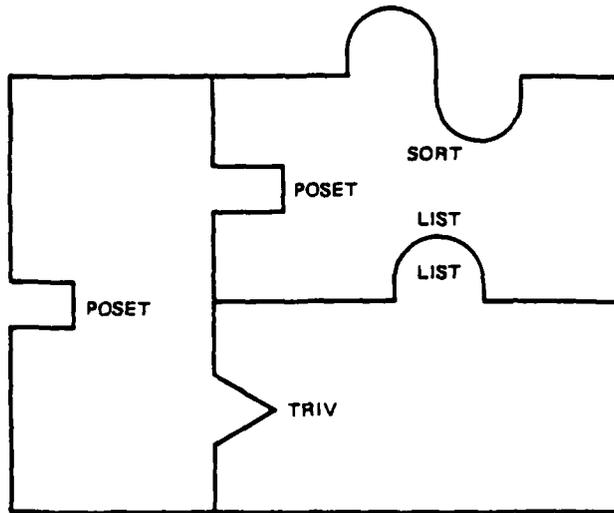
Any actual horizontal parameters for the main package (SORT in this example) will also be supplied to the package version in the needs clause (here, LIST.HACK).

This automatic management of the interactions of horizontal and vertical structure is one of the most novel features of LIL, and can greatly simplify the programmer's task in some cases.

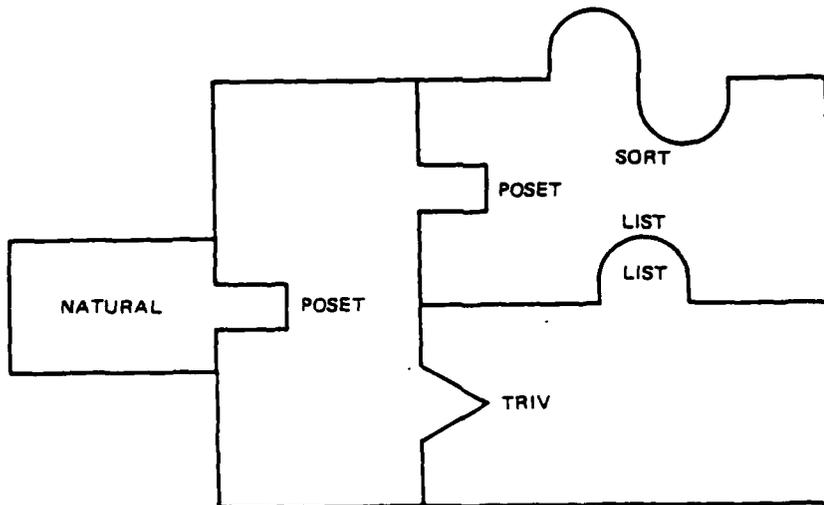
The horizontal component of this example appears in the formal parameter X, which is required to satisfy the POSET theory.

A make command can accomplish both the vertical and the horizontal instantiation of SORT at once:

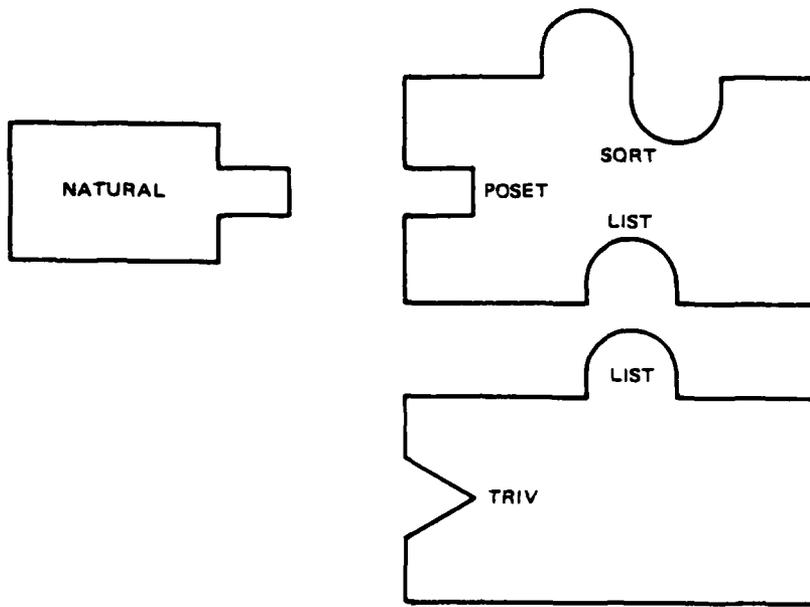
```
make SORT-NATD is SORT[NATD] needs LISTP => LIST.HACK end
```



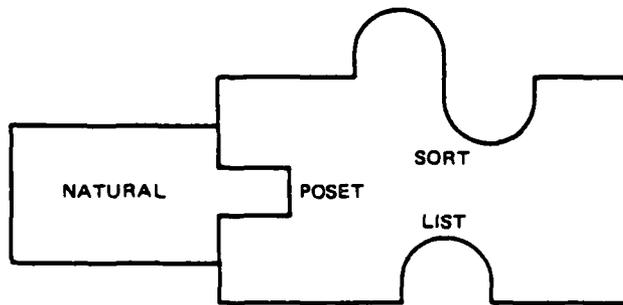
A VERTICAL COMPOSITION



A REALIZATION OF SORT(NATURAL) WITH LIST(NATURAL)



SOME SOFTWARE COMPONENTS

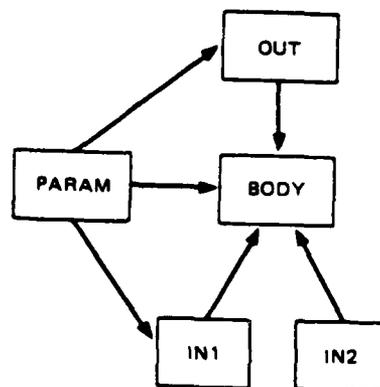


A HORIZONTAL COMPOSITION

The following figure shows that the requirements theory is a subtheory of the AM realized by that package, and may also be a subtheory of the stubs for the AMs that go into realizing it.

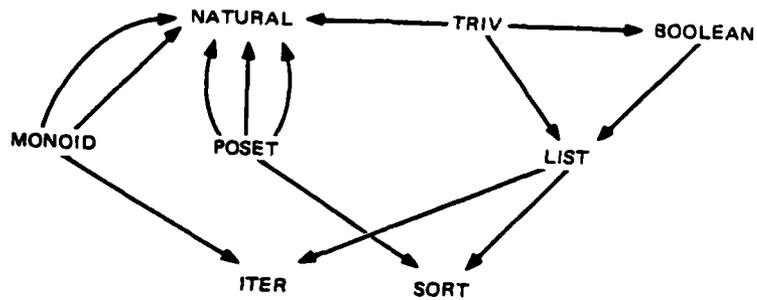
Here IN1 and IN2 are the requirement theories for the lower level AMs; these are included in the body theory of the package.

In addition, there is a theory of the behavior that is actually exported by the package; this AM may not be identical with the body theory because of some information hiding.

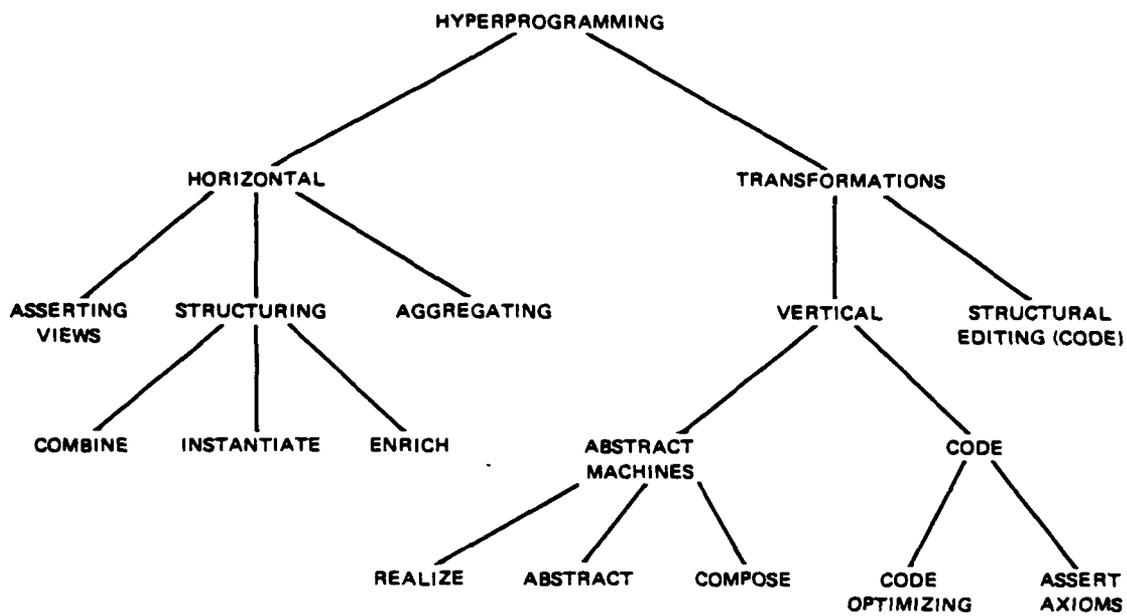


LIL Environments

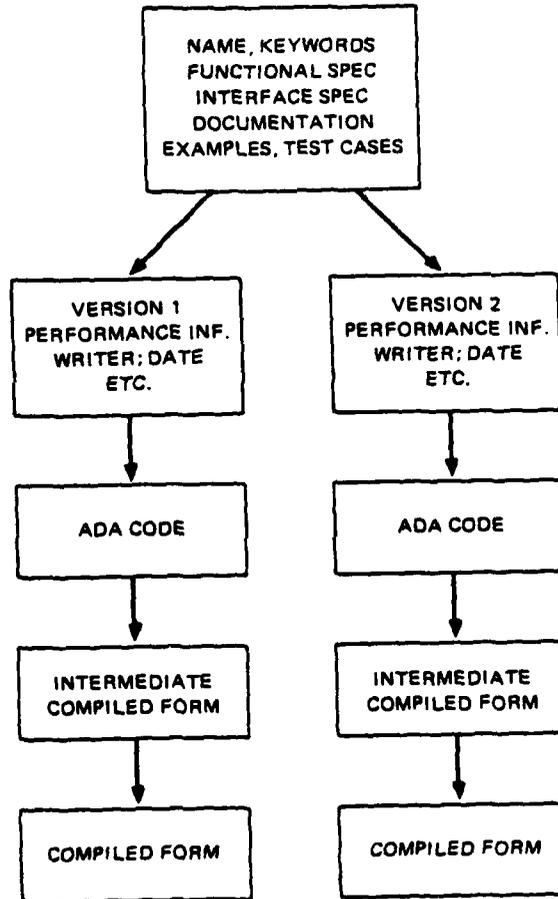
A LIL environment consists of the currently defined entity names and their values, plus the relationships of inclusion among these entities, and their views.



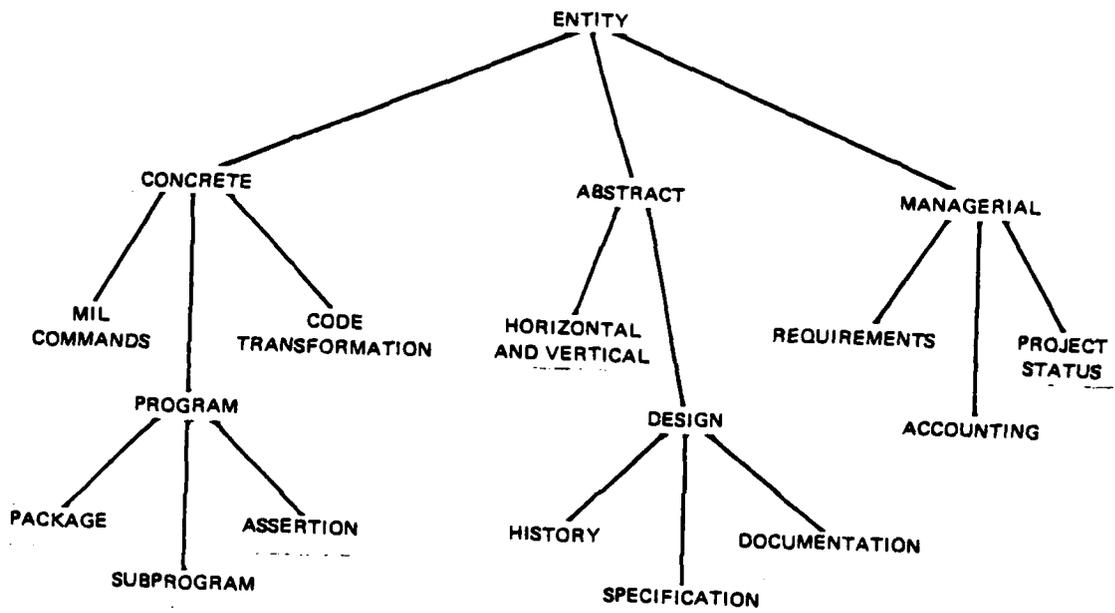
Hyperprogramming Taxonomy



Organization of Library Entities for a Package



Taxonomy of Library Entities



The Components of LIL

(1) Declarations

(a) Packages -- generalize the spec part of Ada packages
by including:
 semantics;
 versions;
 vertical & horizontal parameterization.

(b) Theories -- purely semantic, do not have bodies

(c) Views -- tells how an entity satisfies a theory

(2) Commands

-- tell how to put entities together to form
systems; both vertical and horizontal
instantiation can be used

Organize by Semantics

- cataloging & retrieval
- understanding
- animation
- symbolic execution
- rapid prototyping
- version control

A Generic Secure Resource Manager in LIL

```
theory TRIV is
  types ELT
end TRIV
```

```
theory POSET is
  types ELT
  functions  $\leq$  : ELT ELT -> BOOLEAN
  vars E1 E2 E3 : ELT
  axioms
    (E1  $\leq$  E1)
    (E1  $\leq$  E3 if E1  $\leq$  E2 AND E2  $\leq$  E3)
    (E1 = E2 if E1  $\leq$  E2 AND E2  $\leq$  E1)
end POSET
```

```
theory EQV is
  types ELT
  functions == : ELT ELT -> BOOLEAN
  vars E1 E2 E3 : ELT
  axioms
    (E1 == E1)
    (E1 == E3 if E1 == E2 AND E2 == E3)
    (E1 == E2 if E2 == E1)
end EQV
```

-- for any POSET, there is a natural way to define an equality;

```
view EQ :: EQV => POSET is
  vars E1 E2 : ELT
  ops (E1 == E2 => E1  $\leq$  E1 AND E2  $\leq$  E1)
end EQ
```

```
-- now a top-down reusable development, RESOURCE using
-- TABLE, not yet been defined
```

```
generic package RESOURCE[ACCESSOR :: EQV; X :: TRIV] is
  using TABLEP :: TABLE[ACCESSOR, X] is
  functions
    ACC-OK : ACCESSOR -> BOOLEAN
  procedures
    WRITE : ACCESSOR X
    READ : ACCESSOR -> X
    ....
  exceptions
    WRONG-ACC
  vars A : ACCESSOR; X : X
  axioms
    WRITE(A,X) = PUT(A,X) if ACC-OK(A)
    WRITE(A,X) = WRONG-ACC if NOT ACC-OK(A)
    ....
end RESOURCE
```

```
generic package TABLE[ENTRY :: EQV; X :: TRIV] is
  state TABLE initially EMPTY
  procedures
    PUT : ENTRY X
    LOOKUP : ENTRY -> X
  vars E : ENTRY; X : X
  axioms
    PUT(E,X); LOOKUP(E) = X
    ....
end TABLE
```

```

generic package SEC-MEMORY-ACCESSOR[LEVEL :: POSET]
  needs MEMORYP :: MEMORY is
  types S-MEM-ACCESSOR is
  record
    CELL : CELL; -- cell is part of the MEMORY package
    ACCESSING-LEVEL : LEVEL;
    ACCESSED-LEVEL : LEVEL;
  end record
  functions
    S-MEM-ACC-OK : S-MEM-ACCESSOR -> BOOLEAN
  vars SMA : S-MEM-ACCESSOR
  axioms
    ACC-OK(SMA) = S-MEM-ACC-OK(SMA)
  ....
end SEC-MEM-ACCESSOR

make SECURE-MEM-MANAGER-0[LEVEL :: POSET; X :: TRIV] is
  RESOURCE[SEC-MEM-ACCESSOR[LEVEL], X] needs TABLEP => TABLE.HASH1
end SECURE-MEM-MANAGER-0

make SECURE-MEM-MANAGER[LEVEL :: POSET; X :: TRIV]
  using SEC-MEM-TABLE = TABLE[IDENTIFIER, PAIR[CELL,LEVEL]] (hidden)
  using SECURE-MEM-MANAGER-0[LEVEL, X] (hidden) is
  procedures
    REQUEST-WRITE : IDENTIFIER X LEVEL
    REQUEST-READ : IDENTIFIER LEVEL -> X
  ....
end SECURE-MEM-MANAGER

```

Programming-in-the-Large

Most work in programming is still centered at the statement level.

For large programs, it is essential to break them into meaningful pieces!

Procedures (or even functions) are not the right level of granularity -- these in fact are statements.

An Abstract data type, with the data representation hidden, and with all associated functions, is one reasonable module.

Module Interconnection Languages are the assembly language level of Programming-in-the-Large.

Reusability

- of code
- of specs
- of requirements
- of designs
- of documentation
- of transformations

Implies that we need

- libraries
- parameterization
- version & configuration information
- help in retrieving
- good cataloging
- methods for interconnecting modules
- module interface specification
- management encouragement

IV.4 DCP Approach to Ada Libraries

ANDRES RUDMIK

DISTRIBUTED SOFTWARE ENGINEERING CONTROL PROCESS

D C P

OVERVIEW

- DCP PROJECT
- DCP APPROACH TO ADA SOFTWARE REUSABILITY
- DCP SUPPORT FOR ADA LIBRARIES

GOALS

- DISTRIBUTED AND PORTABLE DCP.
- CENTRALIZED CONTROL OF DEVELOPMENT.
- SUPPORT THE DEVELOPMENT OF PORTABLE SOFTWARE.
- REDUCE SOFTWARE COSTS AND IMPROVE SOFTWARE QUALITY.
 - AUTOMATION
 - REUSABILITY
- SUPPORT THE USE OF ADA AS A DESIGN AND IMPLEMENTATION LANGUAGE.
- INTEGRATION OF SOFTWARE DEVELOPMENT TOOLS.

APPROACH

RELATIONAL DATABASE SUPPORTING

- CONFIGURATION MANAGEMENT
- CHANGE TRACKING
- DOCUMENTATION GENERATION
- ENCYCLOPEDIA - REUSABILITY
- HOST TRANSPARENT DIRECTORY
- PROGRAM LIBRARIES
- DISTRIBUTED DEVELOPMENT

APPROACH CONT'D

UNIFORM ADA ENVIRONMENT

- ADA DESIGN LANGUAGE
- ADA IMPLEMENTATION LANGUAGE
- ADA DESIGN AND IMPLEMENTATION TOOLS
- ADA COMMAND LANGUAGE

APPROACH CONT'D

DCP PORTABILITY

- ADA IMPLEMENTATION
- VIRTUAL INTERFACES

APPLICATION PORTABILITY

- ADA IMPLEMENTATION
- DESIGN METHODOLOGY AND STANDARDS
- AVAILABILITY OF DCP VIRTUAL INTERFACES FOR REUSE BY APPLICATIONS.

DCP APPROACH TO ADA REUSABILITY/ADA LIBRARIES

- WHAT TO PUT IN ADA LIBRARIES
- METHODOLOGY FOR REUSABILITY
- DESIGNING FOR REUSABILITY
- DOCUMENTING FOR REUSABILITY
- DCP ENCYCLOPEDIA -- LIBRARY CATALOGUE
- DCP DIRECTORY SUPPORT OF ADA LIBRARIES
- DCP USE OF ADA LIBRARIES

METHODOLOGY FOR REUSABILITY

REUSABLE SOFTWARE IS PRODUCED AS A RESULT OF A CONSCIOUS EFFORT TO PRODUCE REUSABLE SOFTWARE PROGRAMS AND PACKAGES.

- SYSTEM SPECIFICATION FOR REUSABILITY.
- GENERALIZATION OF FUNCTIONS
- SPECIFY PRIMITIVE FUNCTIONS
- DESIGN FOR REUSABILITY
- MODULAR DESIGN.
- OBJECT ORIENTED DESIGN.
- REVIEW FOR REUSABILITY
- ADA PDL SUPPORTING REUSABILITY

METHODOLOGY FOR REUSABILITY CONT'D

- IMPLEMENT FOR REUSABILITY
- USE ADA GENERIC FEATURES
- GENERALIZE IMPLEMENTATION
- REVIEW FOR REUSABILITY
- TESTING SUPPORTING REUSABILITY
- REUSABLE COMPONENTS MUST BE INDEPENDENTLY TESTABLE.
- TEST PLANS ASSOCIATED WITH REUSABLE COMPONENTS.
- TRAINING REQUIRED

DOCUMENTING FOR REUSABILITY

PACKAGE LEVEL

- PACKAGE CLASSIFICATION BY ABSTRACTION.
- KEYWORD CLASSIFICATION BY APPLICATION.
- SUMMARY DESCRIPTION.
- DETAILED DESCRIPTION.
- DATA FLOW.
- INTERFACE SPECIFICATIONS AND DESCRIPTIONS.
- EXCEPTIONS.

PROGRAM LEVEL

- SIMILAR TO PACKAGE.
- UNIQUE PROGRAM PACKAGE.

METHODOLOGY FOR REUSABILITY CONT'D

- IMPLEMENT FOR REUSABILITY
- USE ADA GENERIC FEATURES
- GENERALIZE IMPLEMENTATION
- REVIEW FOR REUSABILITY
- TESTING SUPPORTING REUSABILITY
- REUSABLE COMPONENTS MUST BE INDEPENDENTLY TESTABLE.
- TEST PLANS ASSOCIATED WITH REUSABLE COMPONENTS.
- TRAINING REQUIRED

DOCUMENTING FOR REUSABILITY

PACKAGE LEVEL

- PACKAGE CLASSIFICATION BY ABSTRACTION.
- KEYWORD CLASSIFICATION BY APPLICATION.
- SUMMARY DESCRIPTION.
- DETAILED DESCRIPTION.
- DATA FLOW.
- INTERFACE SPECIFICATIONS AND DESCRIPTIONS.
- EXCEPTIONS.

PROGRAM LEVEL

- SIMILAR TO PACKAGE.
- UNIQUE PROGRAM PACKAGE.

DCP ENCYCLOPEDIA -- LIBRARY CATALOGUE

- USE RELATIONAL DATABASE -- INGRES
- PROVIDES CM ON LIBRARY OBJECTS.
- PROGRAM DOCUMENTATION
 - EXTRACTED FROM ADA PDL
 - OTHER DOCUMENTS
- CONSISTENT AND UP-TO-DATE
- INQUIRY SUBSYSTEM
- CHANGE TRACKING.
- SUPPORTS SHARING OF PACKAGES.

DATABASE ORGANIZATION OF ADA PROGRAM AND
PACKAGE INFORMATION

PROGRAM

1. BODY - MAIN PROCEDURE
2. PACKAGE SPEC. DEFINING ARGUMENTS
PASSED TO PROGRAM.
3. PROGRAM PACKAGE UNIQUE TO PROGRAM.
4. REMAINING PACKAGES DETERMINED BY
WITH CLAUSES ASSOCIATED WITH PROGRAM
PACKAGE.
5. EXECUTABLE LOAD MODULE.
6. COMMAND PROCEDURE USED TO INVOKE
PROGRAM.

DCP DIRECTORY SUPPORT OF ADA LIBRARIES

- ENCYCLOPEDIA MAINTAINS LOGICAL NAMES
- LOGICAL NAMES MAP TO A HOST DEPENDENT PHYSICAL NAME.
- ALL DCP USERS ACCESS LIBRARY OBJECTS USING LOGICAL NAMES.
- DISTRIBUTED AND POSSIBLY REPLICATED.

DCP USE OF ADA LIBRARIES

- DCP CM IMPLIES THAT THERE IS ONLY ONE INSTANCE OF AN ADA PACKAGE AT A GIVEN DEVELOPMENT POSITION.
- REUSE OF PACKAGE IN ANOTHER PROGRAM STILL IMPLIES THAT ONLY ONE COPY OF THE PACKAGE EXISTS IN THE SYSTEM.
- MODIFICATION OF PACKAGE IMPLIES THE CREATION OF A NEW VERSION OF THAT PACKAGE.
- DCP AUTOMATICALLY CONSTRUCTS ADA PROGRAM LIBRARY REQUIRED BY ADA COMPILER.
- AUTOMATIC CONSTRUCTION OF ADA PROGRAM PARAMETER INTERFACE PACKAGE FROM COMMAND PROCEDURE.

IV.5 Flexibility vs. Efficiency for Reusable Components

ALLEN S. MATSUMOTO

FLEXIBILITY VS. EFFICIENCY
FOR REUSABLE COMPONENTS

MORE GENERAL:

FEWER RESTRICTIONS ON SYSTEM

MORE POSSIBLE INSTANTIATIONS

⇒ FLEXIBILITY

MORE SPECIFIC:

MORE DECISIONS ALREADY MADE

GREATER KNOWLEDGE OF CONTEXT

⇒ EFFICIENCY

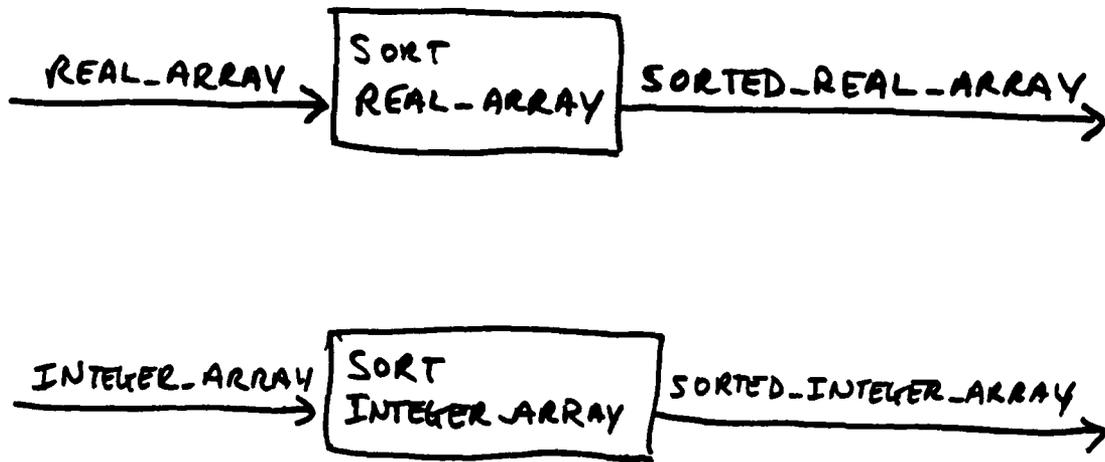


Fig. 1. Specific Sort Components

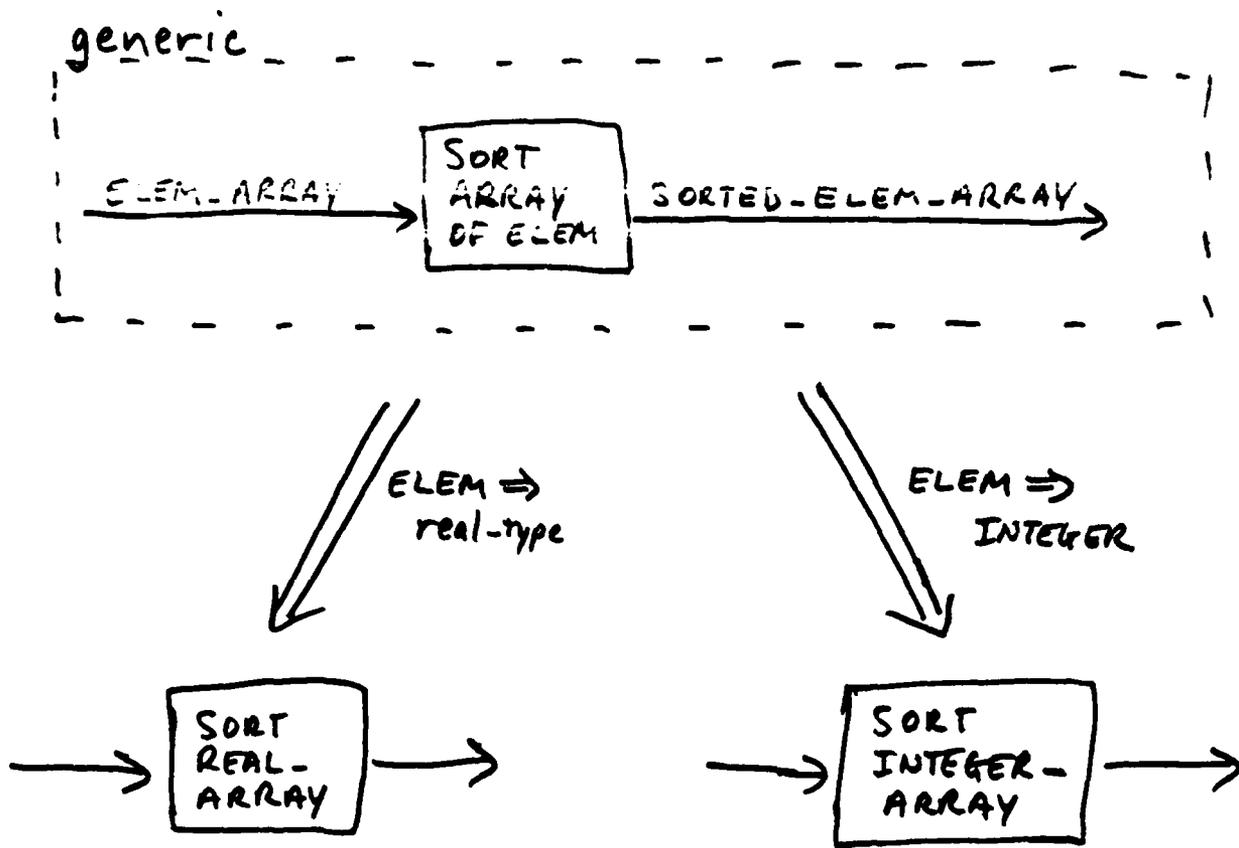


Fig. 2. Instantiations of Generic Sort Component

generic

```
type ELEMENT_TYPE is private;  
with function LESS ( A, B: ELEMENT_TYPE )  
  return BOOLEAN;
```

package SORT_ELEMENT_ARRAY is

```
type ELEMENT_ARRAY is array (integer range <> )  
  of ELEMENT_TYPE;  
function SORT ( A: in ELEMENT_ARRAY )  
  return ELEMENT_ARRAY;
```

end SORT_ELEMENT_ARRAY;

generic

```
type ELEMENT_TYPE is private;  
with function LESS ( A, B: ELEMENT_TYPE )  
  return BOOLEAN;
```

```
type INDEX is (<>);
```

```
package SORT_ELEMENT_ARRAY is
```

```
  type ELEMENT_ARRAY is array (INDEX range <> )  
    of ELEMENT_TYPE;
```

```
  function SORT ( A: in ELEMENT_ARRAY )  
    return ELEMENT_ARRAY;
```

```
end SORT_ELEMENT_ARRAY;
```

generic

```
type ELEMENT_TYPE is private;  
with function LESS ( A, B: ELEMENT_TYPE )  
  return BOOLEAN;
```

```
type INDEX is (<>);
```

```
type LIST is limited private;  
with procedure CREATE ( HEADER: out LIST );  
with procedure INSERT ( HEADER: in out LIST,  
  PLACE: INDEX,  
  ITEM: ELEMENT_TYPE );  
with procedure GET ( HEADER: in out LIST,  
  PLACE: INDEX,  
  ITEM: out ELEMENT_TYPE );
```

```
package SORT_LIST is
```

```
  function SORT ( L: in LIST ) return LIST;
```

```
end SORT_LIST;
```

generic

```
type ELEMENT_TYPE is private;  
with function LESS ( A, B: ELEMENT_TYPE )  
  return BOOLEAN;
```

```
type LIST is limited private;  
with procedure CREATE ( HEADER: out LIST );  
with procedure APPEND ( HEADER: in out LIST,  
  ITEM: ELEMENT_TYPE );  
with procedure GET_FIRST ( HEADER: in out LIST,  
  ITEM: out ELEMENT_TYPE);  
with procedure GET_NEXT ( HEADER: in out LIST,  
  ITEM: in ELEMENT_TYPE )  
  NEXT: out ELEMENT_TYPE );
```

package SORT_LIST is

```
  function SORT ( L: in LIST ) return LIST;
```

end SORT_LIST;

generic

```
type ELEMENT_TYPE is private;  
with function LESS ( A, B: ELEMENT_TYPE )  
  return BOOLEAN;
```

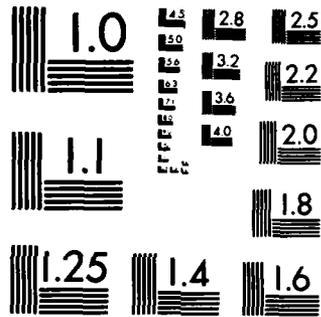
```
type LIST is limited private;  
with procedure CONS ( A, B: ELEMENT_TYPE,  
  HEADER: out LIST );  
with procedure CONS ( A: ELEMENT_TYPE,  
  B: LIST,  
  HEADER: out LIST );  
with procedure CONS ( A, B: LIST,  
  HEADER: out LIST );
```

```
with procedure GET_FIRST ( HEADER: in out LIST,  
  ITEM: out ELEMENT_TYPE);  
with procedure GET_REST ( HEADER: in out LIST,  
  REST: out LIST );
```

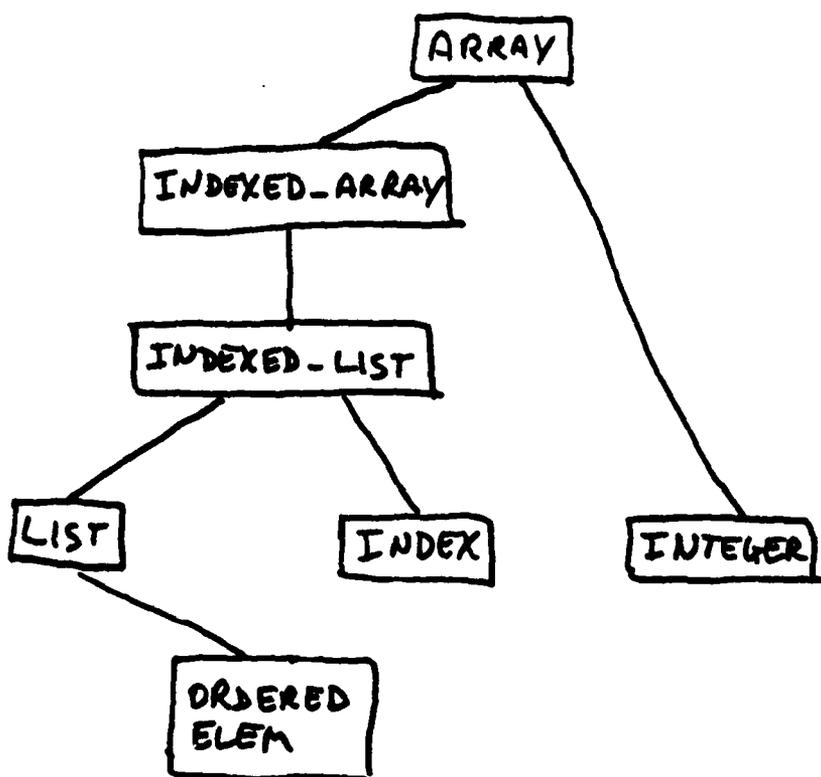
package SORT_LIST is

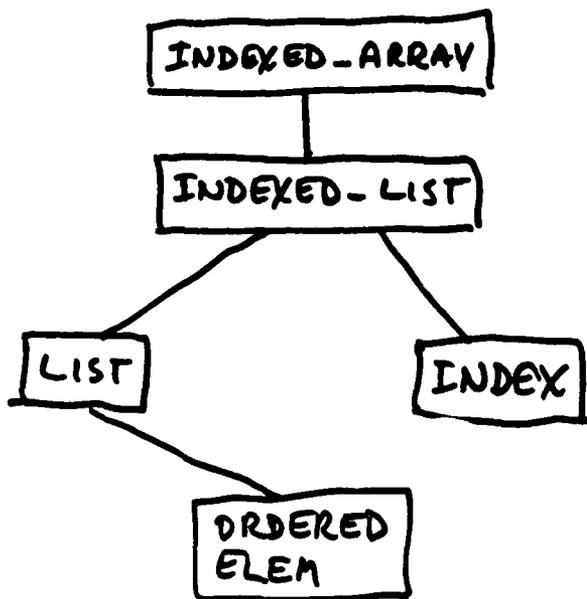
```
  function SORT ( L: in LIST ) return LIST;
```

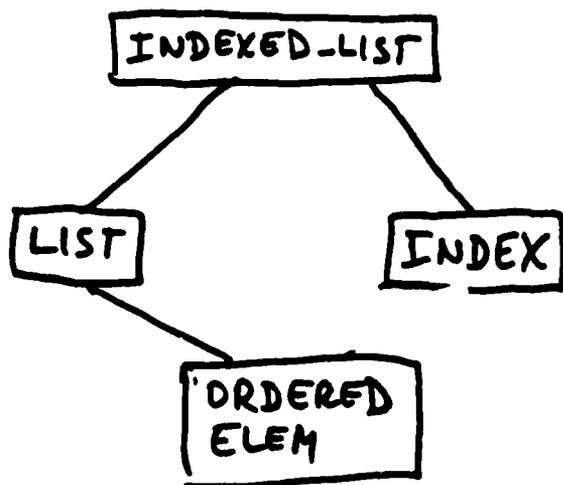
end SORT_LIST;

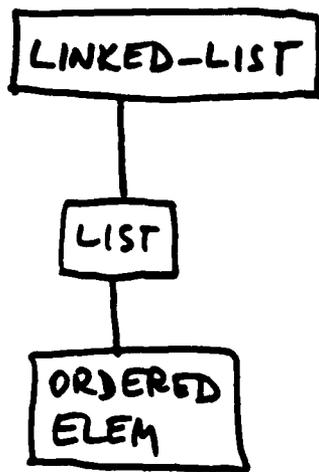


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

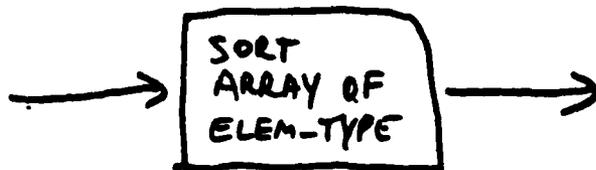
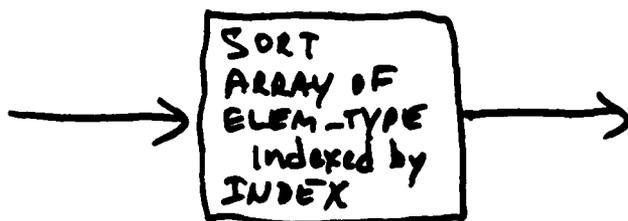
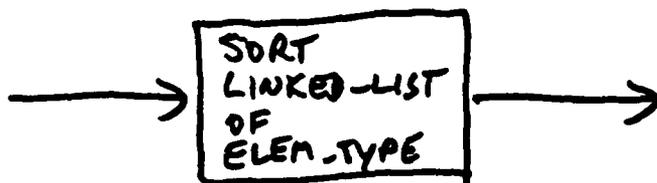
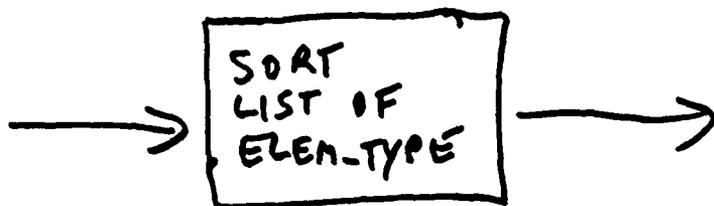






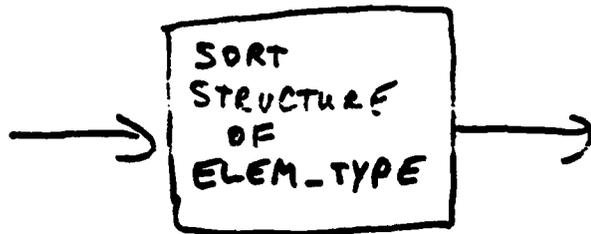


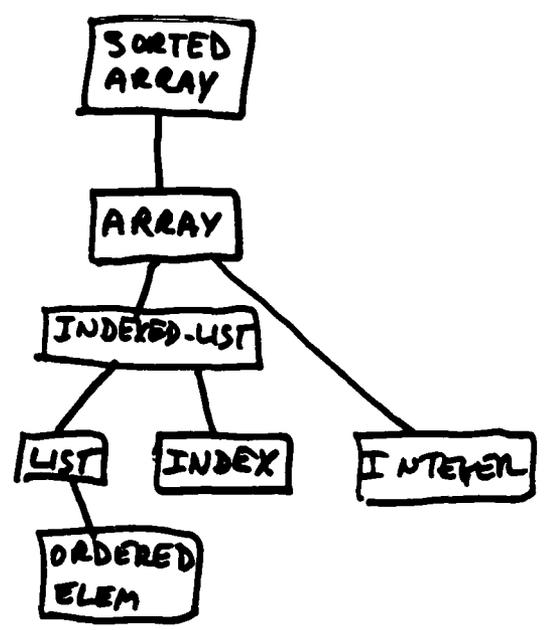
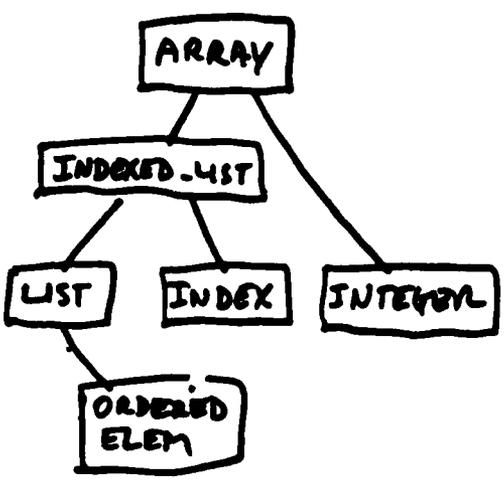
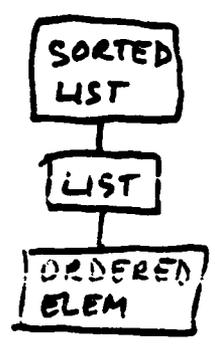
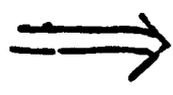
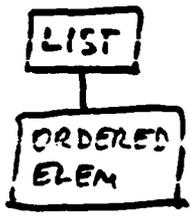


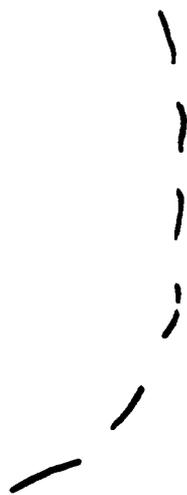


generic

- ELEM-TYPE WITH ORDER
- STRUCTURE AND OPERATIONS









FLEXIBILITY

IS DESIRABLE DURING DESIGN

EFFICIENCY

IS IMPORTANT IN IMPLEMENTATION

CAN WE ENCOURAGE

FLEXIBILITY FOR DESIGN

WITHOUT PREVENTING

EFFICIENT IMPLEMENTATION ?

IV.6 Mapping Clear Specifications to Ada Packages

STEVEN D. LITVINTCHOUK

MAPPING CLEAR SPECIFICATIONS TO ADA PACKAGES

STEVEN D. LITVINTCHOUK
RAYTHEON COMPANY

GOAL: SYSTEMATIC DEVELOPMENT OF ADA PACKAGE
 SKELETONS WHICH PARALLEL DEVELOPMENT
 OF CLEAR SPECIFICATIONS

PURPOSE: FIRST DEVELOP SPECIFICATIONS IN CLEAR;
 THEN "DRIVE" ADA DESIGN BY SPECIFICATION

-- FACILITATE DESIGN

-- FACILITATE MANAGEMENT OF REUSABLE
 ADA COMPONENTS

CLEAR AND ADA -- BASIC FEATURES

CLEAR SPECIFICATION LANGUAGE

- RIGOROUS SEMANTIC DEFINITION BASED UPON FORMAL ALGEBRA
- FORMAL SPECIFICATION OF SOFTWARE
- SPECIFICATIONS BUILT IN MODULAR, STRUCTURED MANNER

ADA PROGRAMMING LANGUAGE

- NO WIDELY ACCEPTED FORMAL SEMANTIC DEFINITION
- ADA PACKAGE: MODULE PROVIDING A COLLECTION OF RELATED FEATURES
- GENERIC PACKAGES CAN IMPLEMENT PARAMETERIZED ABSTRACT DATA TYPES
- BUT PACKAGE INTERFACES ARE MOSTLY SYNTACTIC

CONSTANT (NONPARAMETERIZED) THEORIES

CONST X = THEORY . . . ENDTH

DISCUSSION

-- CANONICAL VS. LOOSE THEORIES

-- THE DATA OPTION

-- "INSTITUTIONS"

EXAMPLE

```
CONST BOOL =  
  THEORY  
    DATA  SORTS  BOOL  
           OPNS  TRUE, FALSE : BOOL  
           EQNS  NOT : BOOL -> BOOL  
                NOT (TRUE) = FALSE  
                NOT (NOT (P)) = P  
  
  ENDTH
```

MAPPING TO ADA: DISCUSSION

- SIGNATURE IN VISIBLE PART OF PACKAGE SPECIFICATION
- REPRESENTATION TYPE IN PRIVATE PART
- IMPLEMENTATION OF FUNCTIONS IN PACKAGE BODY
- CHOICE OF IMPLEMENTATIONS

```
PACKAGE BOOL_PACKAGE IS
```

```
    TYPE BOOL IS LIMITED PRIVATE;
```

```
    FUNCTION TRUE (X: BOOL) RETURN BOOL;
```

```
    FUNCTION FALSE (X: BOOL) RETURN BOOL;
```

```
    FUNCTION "NOT" (X: BOOL) RETURN BOOL;
```

```
    FUNCTION EQUAL (X, Y: BOOL) RETURN BOOL;
```

```
        -- "NOT" IS AN ADA RESERVED WORD
```

```
        -- BECAUSE OF THE DATA OPTION
```

```
PRIVATE
```

```
    TYPE BOOL IS . . .      -- CHOSEN IMPLEMENTATION OF BOOL
```

```
END BOOL_PACKAGE;
```

```
PACKAGE BODY BOOL_PACKAGE IS
```

```
        -- IMPLEMENTATION OF FUNCTIONS
```

```
    FUNCTION TRUE (X: BOOL) RETURN BOOL IS . . .
```

```
    FUNCTION FALSE (X: BOOL) RETURN BOOL IS . . .
```

```
    FUNCTION "NOT" (X: BOOL) RETURN BOOL IS . . .
```

```
    FUNCTION EQUAL (X, Y: BOOL) RETURN BOOL IS . . .
```

```
END BOOL_PACKAGE;
```

COMBINING THEORIES

X + Y CREATES "UNION" OF THEORIES X AND Y

DISCUSSION

- CLEAR SUBTHEORIES ARE SHARED (YOU GET ONLY ONE COPY)
- SO ARE ADA LIBRARY UNITS (FORTUNATELY)
- "BASE" OF A THEORY: RECORDS DEPENDENCE ON ALL GLOBAL SUBTHEORIES
- ENVIRONMENT MUST KEEP TRACK OF THIS INFORMATION
- USE ADA CONTEXT SPECIFICATION ("WITH") CLAUSE
 TO RECORD SAME KIND OF INFORMATION
- "LOCAL" SUBTHEORIES ARE ALSO POSSIBLE

```
(WITH {BASE_OF_X} U {BASE_OF_Y});  
PACKAGE X_PLUS_Y IS  
  
    [PACKAGE X_PACKAGE IS....     --IF X LOCAL  
      END X_PACKAGE;]  
    [PACKAGE Y_PACKAGE IS....     --IF Y LOCAL  
      END Y_PACKAGE;]  
  
END X_PLUS_Y;
```

ENRICHMENTS OF THEORIES

ENRICH E BY X ENDEN

-- ADDS NEW CAPABILITIES AS SPECIFIED BY X, TO EXISTING THEORY, E

EXAMPLE

```
CONST BOOLOPNS =  
  ENRICH BOOL BY  
    OPNS AND, OR, --> : BOOL, BOOL -> BOOL  
    EQNS P AND TRUE = P  
        P AND FALSE = FALSE  
        P OR TRUE = TRUE  
        P OR FALSE = P  
        P --> Q = NOT (P AND NOT (Q))  
  ENDEN
```

```
WITH BOOL PACKAGE;  
PACKAGE BOOLOPNS_PACKAGE IS
```

```
  FUNCTION "AND" (X, Y: BOOL) RETURN BOOL;  
  FUNCTION "OR" (X, Y: BOOL) RETURN BOOL;  
  FUNCTION IMPL (X, Y: BOOL) RETURN BOOL;
```

```
END BOOLOPNS_PACKAGE;
```

```
PACKAGE BODY BOOLOPNS_PACKAGE IS
```

```
  FUNCTION "AND" (X, Y: BOOL) RETURN BOOL IS  
  BEGIN  
    IF Y = TRUE THEN  
      RETURN X;  
    ELSE  
      RETURN FALSE;  
    END IF;  
  END "AND";
```

```
  FUNCTION "OR" (X, Y: BOOL) RETURN BOOL IS . . .  
  FUNCTION IMPL (X, Y: BOOL) RETURN BOOL IS . . .
```

```
END BOOLOPNS_PACKAGE;
```

ANOTHER ENRICHED THEORY

CONST NAT =

ENRICH BOOL BY

DATA SORTS NAT

OPNS \emptyset : NAT

SUCC : NAT \rightarrow NAT

\dagger : NAT, NAT \rightarrow NAT

EQNS $\emptyset + M = M$

SUCC (N) \dagger M = SUCC (N + M)

ENDEN

WITH BOOL_PACKAGE;

PACKAGE NAT_PACKAGE IS

TYPE NAT IS LIMITED PRIVATE;

FUNCTION ZERO RETURN NAT;

FUNCTION SUCC (X: NAT) RETURN NAT;

FUNCTION "+" (X: NAT) RETURN NAT;

PRIVATE

TYPE NAT IS . . .

END NAT_PACKAGE;

DERIVING ONE THEORY FROM ANOTHER

DERIVE D [USING E₁, E₂, ...] FROM F BY M: D --> F ENDDE

SELECTS A SUBSET OF THEORY F
RENAMED AS SPECIFIED BY M (CALLED A SIGNATURE MORPHISM)
TO YIELD A THEORY WITH SIGNATURE D
(CAN ALSO USE E₁, E₂, . . . AS SUBTHEORIES)

EXAMPLE

```
CONST NATORD -  
  ENRICH NAT BY  
    OPNS 1, 2, 3, 4, 5 : NAT  
          <= : NAT, NAT -> BOOL  
    EQNS . . . .  
ENDEN
```

```
CONST SECURITY =  
  DERIVE SORTS LEVEL  
    OPNS UNCLASSIFIED, CONFIDENTIAL,  
          SECRET, TOP_SECRET : LEVEL  
          <=, == : LEVEL, LEVEL -> BOOL  
  USING BOOL  
  FROM NATORD  
  BY LEVEL IS NAT,  
      UNCLASSIFIED IS 0,  
      CONFIDENTIAL IS 1,  
      SECRET IS 2,  
      TOP_SECRET IS 3  
ENDDE
```

```
WITH NATORD_PACKAGE, NAT_PACKAGE, BOOL_PACKAGE;  
PACKAGE SECURITY_PACKAGE IS
```

```
    TYPE LEVEL IS LIMITED PRIVATE;  
    FUNCTION UNCLASSIFIED RETURN LEVEL;  
    FUNCTION CONFIDENTIAL RETURN LEVEL;  
    FUNCTION SECRET RETURN LEVEL;  
    FUNCTION TOP_SECRET RETURN LEVEL;
```

```
    FUNCTION "<=" (X, Y: LEVEL) RETURN BOOL;  
    FUNCTION EQUAL (X, Y: LEVEL) RETURN BOOL;
```

```
PRIVATE
```

```
    TYPE LEVEL IS NEW NAT;  
END SECURITY_PACKAGE;
```

```
PACKAGE BODY SECURITY_PACKAGE IS
```

```
    FUNCTION UNCLASSIFIED RETURN LEVEL IS  
    BEGIN  
        RETURN LEVEL(ZERO);  
    END;  
    .  
    .  
    .  
END SECURITY_PACKAGE;
```

THEORY PROCEDURES (PARAMETERIZED THEORIES)

PROCEDURE P (FML₁: REQT₁, ...) = T

WHERE REQT₁: REQUIREMENT FOR FML₁

TO APPLY PROCEDURE:

P (ACT₁ [M₁], ...)

WHERE M₁: REQT₁ --> ACT₁ (FITTING MORPHISM)

ACTS LIKE TEXTUAL SUBSTITUTION INTO T:

- ACTUAL PARAMETERS FOR FORMAL PARAMETERS
- SORTS/OPERATORS OF ACTUAL PARAMETERS FOR THOSE OF REQUIREMENTS

DISCUSSION

- ADA GENERICS CAN ACCEPT SIGNATURES OF ABSTRACT DATA TYPES AS ARGUMENTS
- BUT SHARING OF SUBTHEORIES MAY RESULT IN POSSIBLE "ALIASING" INVOLVING ACTUAL PARAMETERS AND SUBTHEORIES IN T; ADA GENERICS PROHIBIT SUCH ALIASING
- CAN PLACE RESTRICTIONS ON DEFINITION AND APPLICATION OF CLEAR PROCEDURES TO PROHIBIT ALIASING (PERHAPS REQUIRING THAT SUBTHEORIES OF PROCEDURE BE "RESPECTED")
- CONTROLLING "PROLIFERATION" OF THEORY APPLICATIONS WITH IDENTICAL ACTUAL PARAMETERS CONFLICTS WITH ADA RULES FOR MULTIPLE GENERIC INSTANTIATIONS

EXAMPLE: THEORY OF TOTALLY ORDERED LISTS OF ELEMENTS

```
CONST TRIV =  
  THEORY SORTS ELEM  
  ENDTH
```

META ORDERED-ELEMENTS =

ENRICH BOOLOPNS BY

SORTS ELT

OPNS ==, <= : ELT, ELT -> BOOL

EQNS A==A = TRUE

A==B = B==A

A==B AND B==C --> A==C = TRUE

A<=A = TRUE

A<=B AND B <= A --> A==B = TRUE

A<=B AND B <= C --> A<=C = TRUE

A<=B AND B <= A = TRUE

ENDEN

PROCEDURE ORDERED-LIST (X: ORDERED-ELEMENTS) =

ENRICH LIST (X[ELEM IS ELT]) + BOOLOPNS BY

OPNS ORDER: LIST -> LIST

ORDERED: LIST -> BOOL

EQNS ORDER(NIL) = NIL

:
:
:

-- AXIOMATICS OF ORDER & ORDERED

ENDEN

TO APPLY TO THEORY NATORD,

YIELDING ORDERED LISTS OF NATURAL NUMBERS:

ORDERED-LIST(NATORD[ELT IS NAT, <= IS <=, == IS ==])

```
WITH BOOL_PACKAGE, BOOLOPNS_PACKAGE, LIST_PACKAGE;  
GENERIC
```

```
TYPE ELT IS LIMITED PRIVATE;  
WITH FUNCTION EQUAL (A, B: ELT) RETURN BOOL IS <>;  
WITH FUNCTION "<=" (A, B: ELT) RETURN BOOL IS <>;
```

```
PACKAGE ORDERED_LIST_PACKAGE IS
```

```
PACKAGE NEW_LIST_PACKAGE IS NEW LIST_PACKAGE (ELEM => ELT);  
-- NOTE IMPLEMENTATION OF  
USE NEW_LIST_PACKAGE; -- APPLICATION OF LIST THEORY!  
FUNCTION ORDER (L: LIST) RETURN LIST;  
FUNCTION ORDERED (L: LIST) RETURN BOOL;  
END ORDERED_LIST_PACKAGE;
```

```
TO APPLY (INSTANTIATE) NATORD_PACKAGE:
```

```
NEW ORDERED_LIST_PACKAGE (ELT => NAT_PACKAGE.NAT,  
EQUAL => NAT_PACKAGE.EQUAL,  
"<=" => NATORD_PACKAGE."<=");
```

COPIES OF THEORIES

COPY E [USING F₁, F₂, . . .] ENDCO
MAKES NEW COPY OF E,
WITH SHARABLE SUBTHEORIES F₁, F₂, . . .

APPROACH TO ADA IMPLEMENTATION:
REPLACE EACH ADA PACKAGE BY INSTANTIATION OF
EQUIVALENT GENERIC PACKAGE OF NO ARGUMENTS

INSTEAD OF CREATING PACKAGE E_PACKAGE AS:

```
PACKAGE E_PACKAGE IS
  {TYPES/FUNCTIONS OF E}
END E_PACKAGE;
```

CREATE IT AS INSTANTIATION OF GENERIC PACKAGE OF
NO ARGUMENTS:

```
GENERIC
PACKAGE E_TEMPLATE;

  {TYPES/FUNCTIONS OF E}
END E_TEMPLATE;
WITH E_TEMPLATE;
PACKAGE E_PACKAGE IS NEW E_TEMPLATE;
```

⌘ MORE COPIES

```
PACKAGE E1_PACKAGE IS NEW E_TEMPLATE;
PACKAGE E2_PACKAGE IS NEW E_TEMPLATE;
.
.
.
```

FOR THE FUTURE.....

- FIX REMAINING PROBLEMS, INCLUDING:
 - "ALIASING" INVOLVING PROCEDURE PARAMETERS AND BASE
 - " PROLIFERATION" OF IDENTICAL PROCEDURE APPLICATIONS
- FORMALIZE AND VERIFY CORRECTNESS OF THIS PROCESS
- INVESTIGATE ROBUSTNESS FOR OTHER "INSTITUTIONS"
- DEVELOP ADA OPTIMIZATIONS FOR USEFUL CASES
- DRAW UP REQUIREMENTS FOR SUPPORT ENVIRONMENT
- APPLY RESEARCH ON AUTOMATED GENERATION OF ABSTRACT DATA TYPES

IN CONCLUSION...

- IT IS FEASIBLE TO DO ADA DESIGNS WHICH PARALLEL
THE STRUCTURE OF CLEAR SPECIFICATIONS

- BUT REQUIRES:
 - SOME RESTRICTION ON USE OF CLEAR

 - SIGNIFICANT ENVIRONMENT SUPPORT

IV.7 General Requirements for an Elementary Math Function Library

BRUNO WITTE

GENERAL REQUIREMENTS FOR AN ELEMENTARY MATH FUNCTIONS LIBRARY

(Bruno Witte, NOSC)

San Diego, CA, 10/24'83

.....

- (a) Structure.
- (b) Angles.
- (c) Names.
- (d) Exception handling.
- (e) References.
- (f) Generic packages.
- (g) Accuracy tests.
- (h) Verifications and demos.
- (i) Documentation.

a. **STRUCTURE.** The library of functions and facilities shall be partitioned into Ada packages in such a manner that the following observations are taken into account:

1. **Functions used together.** Functions which are likely to be used together, or are generated together, should be packaged together.
2. **Overloading of library units.** Names of library units cannot be overloaded. Thus, if unpackaged functions were to become library units, their names could not be overloaded.
3. **Advantage of overloading.** Many function names were purposely overloaded to avoid overloading people's memories.
4. **Suboptimal compilers.** When an Ada package consists of several functions, and a user needs only one of them, an ideal compiler will give him only that one. However, such optimizing compilers may be years away. Meanwhile, compilers and loaders are likely to burden programs with unnecessary functions by supplying entire packages named in WITH clauses, even when only a fraction of the package contents is needed.
5. **Minimum package sizes.** It follows that users who object to having their programs burdened at execution time with functions they do not need, should be helped in some other way for the time being. For example, if a math functions library were structured so packages contain only one, or a few functions (or other components), this would be helpful to such users.
6. **Number of WITH clauses.** However, to follow this advice in (5) by placing only one function into most packages, would cause a user to write more WITH and USE clauses.

7. **Package of packages.** But, then again, users can themselves place several smaller packages into a larger one to get a more readable and a better structured program, and to reduce the number of WITH and USE clauses.
8. **Package sizes.** Another factor influences the program structure in Ada: When lowest-level packages or compilation units are too large, they tend to become "unwieldy" from the point of view of the programmer who has to write or maintain them. "Unwieldy" means a combination of lesser readability, clumsier compiling and program development, more chain reactions when changes or corrections are made, etc.
9. **Summary.** The upshot seems to be that the library should be implemented in terms of a hierarchy of packages, with each of the lowest-level packages including only functions which most users will want to use together. An illustration is given below.

 | Illustration of a hierarchy of package trees |
for the functions of a math library.

Package Name	Functions and/or Procedures included in package	Other packages named in a WITH clause
PKG_SQRT	SQRT	
PKG_CBRT	CBRT	
PKG_LOGS	NAT_LOG COM_LOG BIN_LOG	
PKG_TRIG	SIN COS TAN	
PKG_COTAN	COTAN	
PKG_ARCSIN_ARCCOS	ARCSIN ARCCOS	
PKG_ARCTAN	ARCTAN	
PKG_E_EXP	EXP	PKG_LOGS
PKG_GEN_EXP	**	PKG_E_EXP PKG_LOGS

PKG_HYPERBOLICS

SINH
COSH
TANH

PKG_INV_HYPERBOLICS

INV_SINH
INV_COSH
INV_TANH

PKG_POLAR_CARTESIAN

POLAR (X)
POLAR (X1,X2)
POLAR (RHO,PHI)
CARTESIAN (R)
CARTESIAN (X1,X2)
CARTESIAN (RHO,PHI)
ABSCISSA (X)
ABSCISSA (R)
ABSCISSA (RHO,PHI)
ORDINATE (X)
ORDINATE (R)
ORDINATE (RHO,PHI)
RADIUS (X)
RADIUS (R)
RADIUS (X1,X2)
ANGLE (X)
ANGLE (R)
ANGLE (X1,X2)

PKG_SQRT
PKG_ARCTAN
PKG_TRIG

PKG_CARTESIAN_COMPLEX

CARTESIAN (X1,X2)
POLAR (X1,X2)
REAL_PART (U)
IMAG_PART (U)
CONJUGATE (U)
RADIUS (U)
ANGLE (U)
POLAR (U)
+, -, *, /
GET (U)
PUT (U)

PKG_SQRT
PKG_ARCTAN

PKG_POLAR_COMPLEX

POLAR (RHO,PHI)
CARTESIAN (RHO,PHI)
CONJUGATE (S)
CARTESIAN (S)
REAL_PART (S)
IMAG_PART (S)
RADIUS (S)
ANGLE (S)
+, -, *, /
GET (S)
PUT (S)

PKG_TRIG

PKG_FULL_COMPLEX

PKG_COMPLEX_SQRT

SQRT (U)
SQRT (S)

PKG_CARTESIAN_COMPLEX
PKG_POLAR_COMPLEX
PKG_FULL_COMPLEX

PKG_COMPLEX_LOG	NAT_LOG (U) NAT_LOG (S)	PKG_FULL_COMPLEX PKG_LOGS
PKG_COMPLEX_EXP	EXP (U) EXP (S)	PKG_POLAR_COMPLEX PKG_TRIG
PKG_COMPLEX_TRIG	SIN (U) SIN (S) COS (U) COS (S) TAN (U) TAN (S)	PKG_FULL_COMPLEX PKG_HYPERBOLICS PKG_TRIG
PKG_COMPLEX_INV_TRIG	ARCSIN (U) ARCSIN (S) ARCCOS (U) ARCCOS (S) ARCTAN (U) ARCTAN (S)	PKG_FULL_COMPLEX PKG_ARCSIN_ARCCOS PKG_ARCTAN PKG_LOGS PKG_SQRT
PKG_COMPLEX_HYPERBOLICS	SINH (U) SINH (S) COSH (U) COSH (S) TANH (U) TANH (S)	PKG_FULL_COMPLEX PKG_HYPERBOLICS PKG_TRIG
PKG_COMPLEX_INV_HYPER	INV_SINH (U) INV_SINH (S) INV_COSH (U) INV_COSH (S) INV_TANH (U) INV_TANH (S)	PKG_COMPLEX_INV_TRIG
PKG_GEN_COMPLEX_EXP	U**V S**T	PKG_FULL_COMPLEX PKG_COMPLEX_LOG

b. **ANGLES.** Before explicitly or implicitly referring to angles in the context of trigonometric functions, coordinate transformations, or complex arithmetic, the user shall have the option to decide whether he wants angles in degrees or in radians.

c. **NAMES.** Names of packages, functions, procedures, and constants, which are used in the code, shall be as given.

d. **EXCEPTION HANDLING.** For most functions and procedures there shall be "catch-all" exception traps and handlers which report their function or procedure name to the calling program, raise the exception, and do nothing else (no printing, no value assignments, etc.). The calling programs must decide what to do, among options such as [a] printing that there is trouble in function XYZ, [b] halting program execution, [c] re-executing the calling routine (for expl., recursively) with dif-

ferent parameter values, [d] choosing a substitute routine, and resuming execution, [e] sounding an alarm at the control center, or [f] re-raising the exception, etc. When the calling program is itself part of one of the library packages, then the developer shall decide on the action to be taken by the exception handler in the calling program. If, in a specific case, a decision is made to re-raise the exception, then the name of the calling program, too, shall be reported back to the parent routine, in addition to the name of the subprogram where the problem started; etc.

e. **REFERENCES.** All literature references shall be by author's last name, year of publication, and a letter when several publications are listed for the same author and year. For example, "Wilkinson (1959b)" would refer to the second of the following fictitious entries in a list of references:

Wilkinson, J.H. (1959a), "Error Analysis of Floating-point Computation", Numer.Math., vol.2, pp.319-340.

Wilkinson, J.H. (1959b), "The Evaluation of Zeros of Ill-conditioned Polynomials", Numer.Math., vol.1, pp.150-166, 167-180.

f. **GENERIC PACKAGES.** Most functions shall be declared inside generic packages, with statements like

```
generic type REAL is digits <>;
...

package XXX_XXX is
...
function ANY (X:REAL) return REAL;
function ETC (X:REAL) return REAL;
...
```

g. **ACCURACY TESTS.** There shall be separate tests of the "single-precision" and the double-precision evaluations, as well as tests of the single-precision results by comparing them with double-precision ones. There shall be random argument accuracy tests as well as tests with selected specific arguments (near the extremes of the function domains, near the zeros of the functions, near zero-arguments, etc.). The different kinds of tests are labelled Tests [A], Tests [B], Tests [C], etc., and each of these kinds of tests shall satisfy its own set of requirements as illustrated further below for Tests [A].

Tests [A]: random arguments,
with range reductions,
separate test packages,
no double-precision tests.

Tests [B]: random arguments
no range reductions,
separate test packages,
no double-precision tests.

Tests [C]: random arguments,
one integrated test program,
only double-precision tests.

Tests [D]: random arguments,
one integrated test program,
no double-precision tests,
but double-precision comparisons.

Tests [E]: selected arguments,
separate test packages,
single- and double-precision tests.

Tests [A]

Tests of type [A] do not apply to double-precision functions. They use random arguments, and check how well certain identities are satisfied.

1. Tests [A] shall be in the form of separate test packages for each function.
2. Packages for Tests [A] shall become available to the users in the same way as function approximations, so that users can repeat the tests, either with the given functions or with their own.
3. Each Test [A] package shall display the results in the form shown in paragraph (10) below.
4. The Test [A] packages shall be prepared with the same care and in the same style, and with the same level of documentation, as is required for the function approximations themselves.
5. Tests [A] shall be random argument accuracy tests, to check how well certain functional identities are satisfied.
6. Random arguments in the Tests [A] shall for most functions have a logarithmic distribution, in intervals for which the functions are defined.
7. Each Test [A] shall use a repeatable sequence of 100,000 random arguments, with seed N=1.
8. For each Test [A] the required number of significant decimals, K, shall be FLOAT'DIGITS.

9. For each random argument accuracy Test [A], the contractor shall compute the following quantities:

- [1] R = ABS(E), absolute value of the relative error E, for each argument;
- [2] MAX_R = maximum R, see [1];
- [3] X_MAX_R = argument associated with MAX_R, see [2];
- [4] AVG_R = average value of R, see [1];
- [5] MIN_DIG = $-\text{COM_LOG}(\text{MAX_R})$ = minimum number of correct decimal digits obtained with any argument in the test, see [2];
- [6] AVG_DIG = $-\text{COM_LOG}(\text{AVG_R})$ = average number of correct decimal digits obtained with the arguments in the test, see [4];
- [7] MAX_MISS = $K - \text{MIN_DIG}$ if $K > \text{MIN_DIG}$
= 0 if $K \leq \text{MIN_DIG}$

maximum number of significant decimal digits missing on required K decimal digits, see (8) and [5];
- [8] AVG_MISS = $K - \text{AVG_DIG}$ if $K > \text{AVG_DIG}$
= 0 if $K \leq \text{AVG_DIG}$

average number of significant decimal digits missing on required K decimal digits, see (8) and [6];

10. Table arrangement illustration for results of tests of type [A]. All numerical values are fictitious. However, they indicate what results may be expected, i.e., some unavoidable inaccuracies when users' DIGITS specifications correspond to maximum machine accuracy, and K is set to this DIGITS value.

Type [A] random argument accuracy test of:						SQRT
Location of test site:		XYZ Co., City, State				
Computer:		DEC-360				
Date:		10/5 '83				
Number of arguments:		100,000				
argument distribution:		logarithmic				
argument range:		0..9.876543210E65				
relative error:		E = [SQRT(X**2)-X]/X				
K	MIN_DIG	AVG_DIG	MAX_MISS	AVG_MISS	X_MAX_R	
10	8.9	9.5	1.1	0.5	1.222333444E-55	
K	-- FLOAT'DIGITS = maximum number of significant decimal digits available;					
MIN_DIG	-- minimum number of significant figures actually returned for any of the random arguments;					
AVG_DIG	-- average number of significant figures actually returned for all random arguments;					
MAX_MISS	-- K-MIN_DIG (if positive), 0 (otherwise);					
AVG_MISS	-- K-AVG_DIG (if positive), 0 (otherwise);					
X_MAX_R	-- argument causing the maximum relative error.					

h. VERIFICATIONS AND DEMOS

- (1) Mathematical constants.
- (2) Coordinate transformations.
- (3) Complex arithmetic and I/O.
- (4) Complex functions.
- (5) Double-precision.
- (6) Scalar operations.
- (7) Array operations.
- (8) Linked lists.
- (9) Random numbers.
- (10) Exception handling.

i. DOCUMENTATION.

- (1) Electronic transmittals.
- (2) Package hierarchy overview.
- (3) Summaries of Ada packages.
- (4) Source code listings.
- (5) Output listings.
- (6) Test evaluations.
- (7) General comments.
- (8) References.

1. **Electronic transmittals.** The entire documentation shall be stored in the form of files on hard or floppy disks of the preparer's computer, and shall be transmitted electronically to a dedicated special directory on an ARPANET host computer, to be designated by the Ada Joint Program Office. Each section of the reports which starts on a new page shall correspond to a different disk file on the preparer's computer, and vice versa, and the transmittals of these files shall be by uploading from the preparer's (micro-) computer(s) to the ARPANET host.

2. **Package hierarchy overview.** In a file of its own, there shall be an overview of the hierarchical package structure of the Ada packages and subprograms in this library, in the style illustrated in (a.9).

3. **Summaries of Ada packages.** For each individual Ada package, there shall be a separate abstract in English prose, stating such items as domains of input variables of subprograms in the package, range of output values of the subprograms, approximation methods, range reduction procedures, exception handling, generic aspects, literature references, etc. There shall be one such summary for each package listed in the above overview (2) under the heading "Package Name", and the summaries shall be in the same order as in this listing, each in a file by itself, and none exceeding the equivalent of an ordinary typewritten text page.

4. **Source code listings.**

- (4.1) Source code of Ada packages.
- (4.2) Source code of verifications.
- (4.3) Source code of accuracy tests.
- (4.4) Source code of timing program.
- (4.5) Source code of random number tests.

4.1 **Source code of Ada packages.** For each package in the overview (2), there shall be a separate file with its Ada source code.

4.2 **Source code of verifications.** For each verification program in (h), there shall be a separate source code file.

4.3 Source code of accuracy tests. For each individual accuracy testing program (devised according to section (g)), there shall be a separate source code file.

4.4 Source code of timing program. There shall be a separate file with the timing program source code.

4.5 Source code of random number tests. There shall be a separate source code file for each of the five test programs for evaluating the UNIFORM_RANDOM number generator.

5. Output listings.

- (5.1) Two parallel sets.
- (5.2) Verification outputs.
- (5.3) Accuracy test outputs.
- (5.4) Timing test outputs.

5.1 Two parallel sets. There shall be two parallel sets of files of output listings for the outputs described below in (5.2), (5.3) and (5.4), one set obtained with runs on one computer, the other with runs on another computer, and the two computers shall be by different manufacturers. Each output listing shall identify the computer, its location and the date of the run; and each set shall contain all of the following output files. See (5.2, 5.3, and 5.4).

5.2 Verification outputs. There shall be separate files for each of the following output listings:

- the output listing of the verifications program for the mathematical constants;
- the output listing of the verifications program for the coordinate transformations;
- the output listing of the verifications program for the complex arithmetic and I/O;
- the output listing of the verifications program for the evaluation of elementary functions of complex argument;
- the output listing of the verifications program for the double-precision capabilities;
- the output listing of the verifications program for the various scalar operations;
- two output listings from the two verification programs for array operations;
- two output listings from the two verification programs for operations on linked lists;

- ten frequency plots obtained with the program for verifying the performance of random number generators; each of these plots shall be considered a separate output listing.
- the five program outputs for verifying exception handling shall all be in the same output listing.

5.3 Accuracy test outputs. There shall be files with the outputs of the accuracy testing programs of section (g), (see also (4.3)). Thus, there shall be 62 (or more) output tabulations of the test results, and each shall be a separate output listing, as follows:

- twenty tabulations for Tests [A], i.e., one for each of the elementary functions;
- twenty tabulations for Tests [B], i.e., the repeats of Tests [A] for primary ranges;
- one tabulation for all Tests [C], the accuracy tests of the double-precision function evaluations;
- one tabulation for all Tests [D], the accuracy tests of the single-precision function evaluations, by comparing with double-precision results;
- twenty (or more) tabulations for Tests [E] with deliberately chosen arguments.

5.4 Timing test outputs. There shall be files with the output listings of the timing tests.

6. Test evaluations.

- (6.1) Accuracy test evaluations.
- (6.2) Timing test evaluations.
- (6.3) Random number test evaluations.

6.1 Accuracy test evaluations. Separately for each elementary function there shall be a summary of the results of the accuracy tests, a cursory comparison with results of corresponding tests in FORTRAN, as well as comments on such items as the extra loss of accuracy due to range reductions (Tests [A] versus Tests [B]), the adequacy of tests which rely on identities rather than double-precision comparisons (Tests [A] versus Tests [D]), any accuracy loss at the endpoints of the domain (Tests [E]), etc. Each summary shall begin on a new page, and none shall exceed one page.

6.2 Timing test evaluations. There shall be a summary of the timing test results, and a cursory comparison with results of corresponding FORTRAN tests. All twenty functions shall be discussed in the same summary, starting on a new page.

- 6.3 **Random number test evaluations.** There shall be a summary of the results of the tests of the generator of uniformly distributed random numbers, including comparisons with corresponding FORTRAN results on the same equipment, and including statements (for each of the tests) telling whether Ada results are better (more random) or worse (less random) than FORTRAN results.

7. **General comments.** There shall be some brief comments on programming approaches, and on difficulties encountered during the implementation of this library, and the developer shall feel free to make suggestions for other programmers of large Ada software systems.

8. **References.** There shall be a collection of all references in this part of the documentation, compiled in the manner as is described in (e).

IV.8 Knowledge Based Tools for Data Type Implementation

GORDON KOTIK

Knowledge Based Tools For
Data Type Implementation

by

Gordon Kotik

of

Computer and Information Science Dept.
University of California , Santa Cruz

and

Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304

KESTREL INSTITUTE

OUTLINE

I. What does a data type user want?

II. How can a programming environment help?

A. CHI, a Knowledge Based
Programming Environment

B. A Theory Of Data Type Implementation

C. A Tool For Implementing Data Types

III. Relevance To ADA Libraries

A SHOPPING LIST FOR DATA TYPE USERS

I. Expressive Power

II. Efficient Implementation

III. Correctness

KESTREL INSTITUTE

EXPRESSIVE POWER

-- Standard Low Level Types

- Integers
- Booleans
- Floating Point
- Characters

.

-- High Level Type Constructors

- Sets
- Sequences
- Mappings
- Relations
- Product Types

.

-- Powerful Operators

- Set Constructors
- Reduction operators
- Relational operators
- Quantifiers

.

-- User Defined Types

- Data Abstraction
- Hidden Implementations

KESTREL INSTITUTE

POWERFUL DATA TYPES ALLOW SIMPLE SPECIFICATIONS

Primes:

$(x \text{ in } 2..Max \mid \forall y \text{ in } 2..(x - 1) [\sim(y|x)])$

Business Programming:

type Employee = record

 Fname, Lname : string;
 Salary : integer
end;

var Employees : set of Employee;

 TotalWages : integer;

 IsBossOf : relation on Employee x Employee;

begin

 TotalWages <-

 Reduce(x in Employee, x.Salary, +);

 Overpaid <-

 (E in Employees |

 ($\exists B$ in Employees [IsBossOf(B,E) &

 E.Salary > B.Salary])

) ;

end ;

KESTREL INSTITUTE

EFFICIENCY

- Default implementations are not good enough

- Data type implementations should be chosen according to
 - Context of use (operation frequencies)
 - User defineable cost function

KESTREL INSTITUTE

KNOWLEDGE BASED TOOLS FOR

DATA TYPE IMPLEMENTATION

I. CHI: A Knowledge Based
Programming Environment

II. A Theory Of Data Type Implementation

III. Data Structure Implementation in CHI

IV. An Example: Derivation of a Hash Table
Implementation For Sets

KESTREL INSTITUTE

CHI

- A Programming Language "V"

- A database for representing V objects

- Tools for
 - Reading and Printing V objects
 - A structure-based editor for V
 - Compiling V into Lisp

KESTREL INSTITUTE

V: A WIDE SPECTRUM, VERY-HIGH-LEVEL LANGUAGE

- Includes standard low level data types and control constructs

- Very high level constructs
 - Program transformation rules
 - High level data types
 - Sets
 - Sequences
 - Mappings
 - Relations
 - Products
 - Unions
 - Logic constructs
 -
 -
 -

KESTREL INSTITUTE

THE CHI DATABASE

- Instances of V constructs are represented in the CHI Database as sets of assertions
- Programs, Knowledge (expressed as rules)
- Uniform Representation Of All Assertions as (Property, Object, Value) triples
- Stored and computed properties
- Maintains multiple contexts in a tree structure

KESTREL INSTITUTE

V PROGRAMS ARE COMPILED BY

SUCCESSIVE RULE APPLICATIONS

-- V Rules Are Source to Source
Program Transformations

-- Rules operate on the database

-- Rules have form

$P \rightarrow Q$

where P and Q are predicates on the
state of the database, and are
interpreted as

"if P is true in the current state
of the database, then make Q true
in the next state of the database"

KESTREL INSTITUTE

DATA TYPE IMPLEMENTATION IN CHI

GOAL: Intelligent selection of implementations
for high-level V data types

PROBLEM:

1. High Level types in V have many distinct implementations.
2. Distinct implementations have widely disparate efficiency characteristics.
3. Many different usage patterns.

KESTREL INSTITUTE

TOOLS FOR SOLUTION

1. A system which generates a broad class of data type implementations.
2. An "efficiency expert", which estimates the resource requirements of V programs.
3. A control structure for searching the space of alternative implementations, guided by the efficiency expert.

KESTREL INSTITUTE

KNOWLEDGE BASED SYNTHESIS OF

----- DATA TYPE IMPLEMENTATIONS -----

IDEA: Complex data type implementations embody the results of a sequence of independent implementation decisions.

DESIGN PLAN:

1. Factor knowledge about data structure implementation into a small set of general, orthogonal representational techniques ("data type refinements").

"Set of X" can be represented by
"mapping from X to boolean"

"Mapping from I to Y" can be represented by
"Array(I) of Y", if I is an integer subrange.

2. Derive distinct data type implementations by composing data type refinements.

"Set of 1..1000"

=> "Mapping from 1..1000 to boolean"

=> "Array(1..1000) of boolean"

DATA TYPE REFINEMENTS

For data types

$$X = (X\text{-Vals}, X\text{-Ops})$$
$$Y = (Y\text{-Vals}, Y\text{-Ops})$$

a refinement

$$I: X \rightarrow Y$$

of type X to type Y is a pair

$$I = (\text{Abs}, \text{Trans})$$

where:

Abs: $Y\text{-Vals} \rightarrow X\text{-Vals}$ (abstraction function)

Trans: Relation on Terms \times Terms
(translates relation)

KESTREL INSTITUTE

PALO ALTO, CA 94304

COMPOSING DATA TYPE REFINEMENTS

For refinements

$$I: A \rightarrow B, I = (\text{Abs-I}, \text{Trans-I})$$
$$J: B \rightarrow C, J = (\text{Abs-J}, \text{Trans-J})$$

the refinement $K = J \circ I$,

$$K: A \rightarrow C, K = (\text{Abs-K}, \text{Trans-K})$$

obtained by composing J and I , is defined by

$$\text{Abs-K} = \text{Abs-I} \circ \text{Abs-J}$$
$$\text{Trans-K} = \text{Trans-J} \circ \text{Trans-I}$$

KESTREL INSTITUTE

IMPLEMENTATION OF DATA TYPE REFINEMENTS

A refinement

$$I: X \rightarrow Y, I = (\text{Abs-I}, \text{Trans-I})$$

is implemented as

- (1) A V rule which specifies choice of refinement I , and
- (2) A set of (term rewriting) V rules which specify the legal translations of X -Terms into Y -Terms.

- Each rule in (2) induces a relation on Terms \times Terms.
- The translates relation Trans-I is then the union (over these rules) of the induced relations.

KESTREL INSTITUTE

AN EXAMPLE DATA TYPE REFINEMENT:

SETS REPRESENTED AS SEQUENCES

-- Abstraction Function: $Abs(Seq) = Range(Seq)$

-- Rule which chooses the refinement:

```
rule Set-Rep-As-Sequence params(*)
  *:'set of X' -> *.Impl='Set-Rep-As-Seq
  & *.Rep-As=SeqType
  & SeqType:'sequence of X'
```

-- Some rules which translate operations:

```
rule Set-Rep-As-Sequence-Member-Test params(*)
  *:'Y in S' & S.Type.Impl='Set-Rep-As-Seq
  & S.Rep-Var=Repv ->
  *:'Ex I [I in 1..size(S) & (Repv(I)=Y)]'
```

```
rule Set-Rep-As-Sequence-Size params(*)
  *:'size(S)' & S.Type.Impl='Set-Rep-As-Seq
  & S.Rep-Var=Repv -> *:'size(Repv)'
```

EXAMPLE SYNTHESIS OF A DATA TYPE IMPLEMENTATION:

SETS REPRESENTED BY HASH TABLES

▶ var S : set of Integer; x in S



Choose a characteristic function implementation

▶ S1 : mapping from Integer to boolean; S1(x)

	⋮	
-1	→	False
0	→	True
1	→	False
2	→	False
	⋮	
105	→	True
	⋮	

► S1 : mapping from integer to boolean; S1(x)

↓ Choose a hashed impl.
for characteristic
function

► function Hash (i : integer) : HashIndexType;

S2 : mapping from HashIndexType to
mapping from integer to boolean;

S2(Hash(x))(x)

↓ Choose HashIndexType
to be 0..100; choose
a hash function

► function Hash (i : integer) : HashIndexType;
begin return(i mod 101) end;

S2 : mapping from 0..100 to
mapping from integer to boolean;

S2(Hash(x))(x)

▶ S2 : mapping from 0..100 to
mapping from integer to boolean;

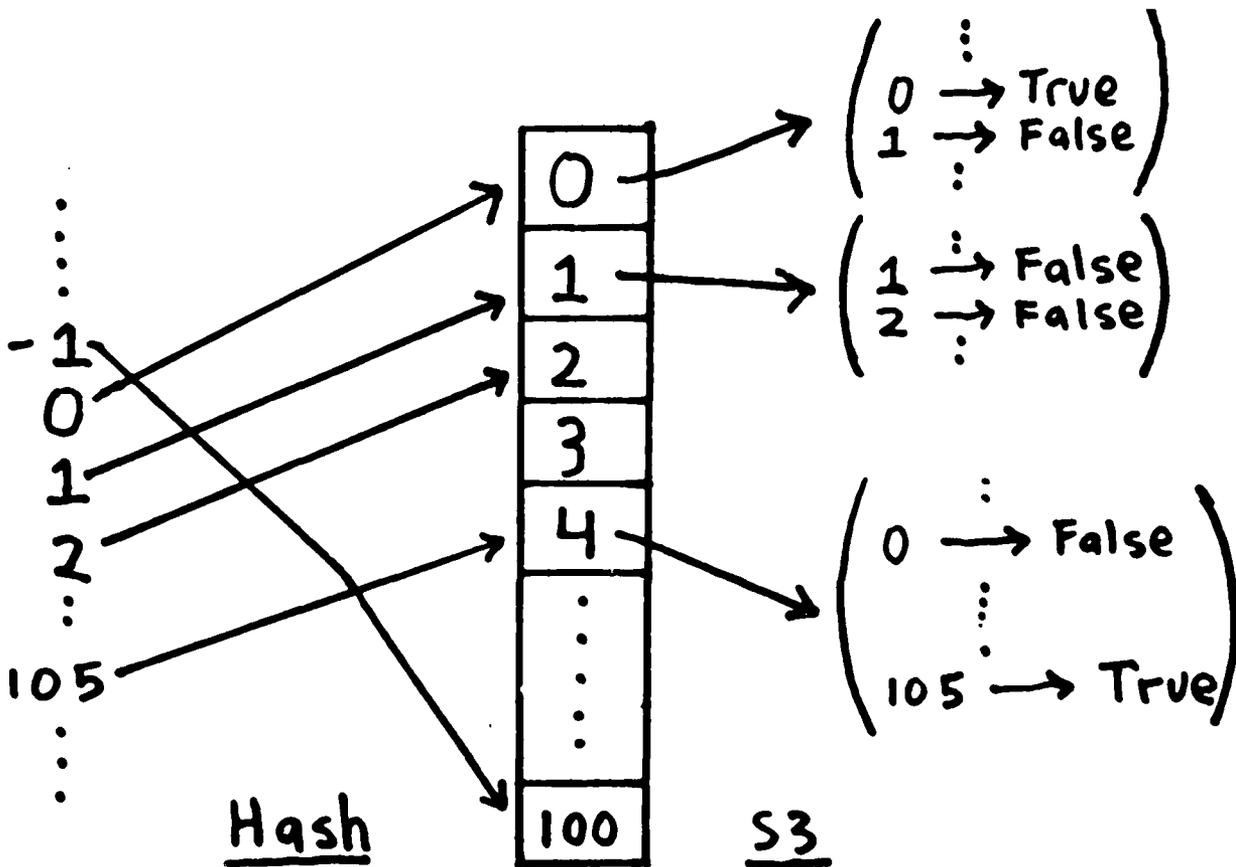
$$\underline{S2(\text{Hash}(x))(x)}$$



Represent the top-level
mapping as an array

▶ S3 : array(0..100) of
mapping from integer to boolean;

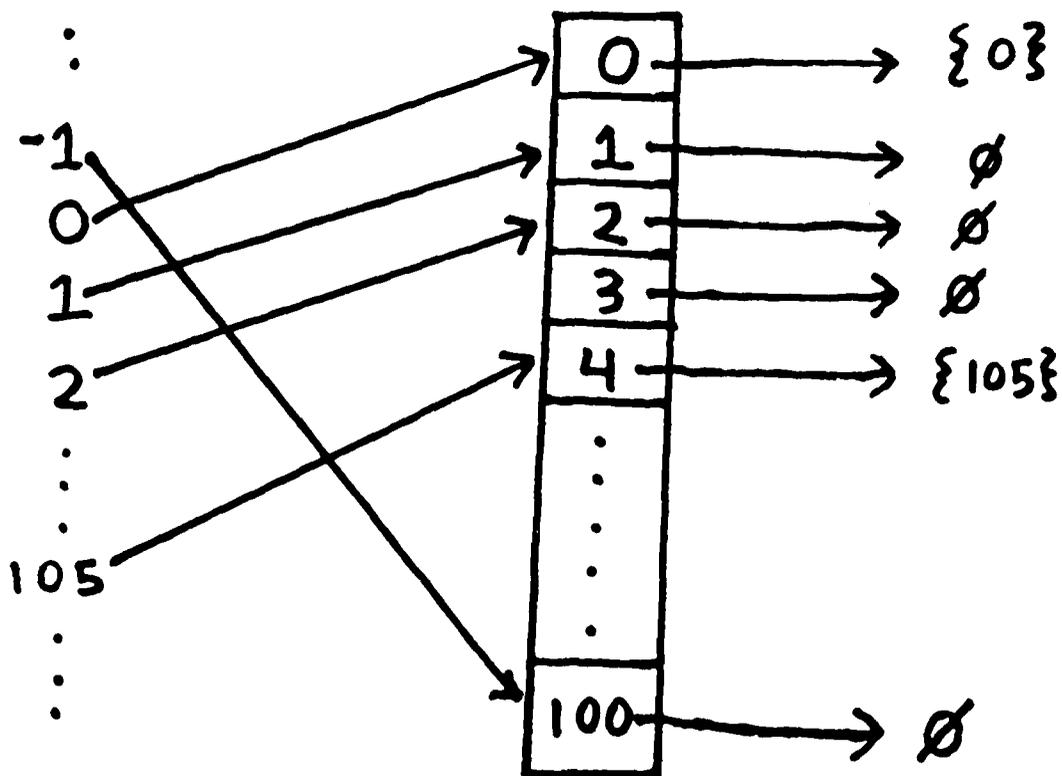
$$\underline{(\text{ELT } S3 \text{ Hash}(X))(x)}$$



- ▶ S3 : array(0..100) of mapping from integer to boolean;
(ELT S3 Hash(X))(x)

↓ Represent the range type of the array as sets

- ▶ S4 : array(0..100) of set of integer;
x in (ELT S4 Hash(x))



▶ S4 : array(0..1000) of set of integer;

x in (ELT S4 Hash(x))



Choose a sequential
representation for
the set type

▶ S5 : array(0..1000) of sequence of integer;

Ex y [y in 1..size(ELT S5 Hash(x))
& (ELT S5 Hash(x))(y)=x]

-- The sequence type can now be refined into
linked lists or trees or....

ADA DATA TYPE LIBRARIES

- ADA users should have high level types

- High level types necessitate the use of many distinct target implementations

- Storing a large library of hand-coded data type implementations is not the best way
 - Implicit, informal derivations
 - Tremendous redundancy

- A better method is to have a computed library
 - Generate ADA packages for high level data types based on efficiency analysis and resource constraints

IV.9 Library Organization and User Interfaces

JOSE MESEGUER

LIBRARY ORGANIZATION
AND
USER INTERFACES

J. MESEGUER

SRI

GOAL:

MAXIMIZE REUSABILITY.

PROBLEMS

1. Hard to get programmers to document their work
2. Too much unstructured information (e.g. 30.000 lines of FORTRAN)
- etc.

SUCCESS of a LIBRARY depends on providing good answers to these problems will FAIL if:

- a PACKAGE wasn't found because not enough **INFORMATION** existed to retrieve it
- a PACKAGE was found but with little information about **WHAT** it does or **HOW** to use it.

MAIN IDEA:

ORGANIZATION

BY (+WITH +OF!)

SEMANTICS

i.e., to exploit as much as possible the semantic info.

available in

ADA packages { spec. ^{not version}
body (allowing diff.)

LIL+ADA { . Configuration info.
. semantic requirements for
param. instantiation
. horizontal + vertical str.

Taking advantage of built-in semantic structure a lot can be done to

1. Help the programmer document his work (interactively)
2. Facilitate programming understanding + communication

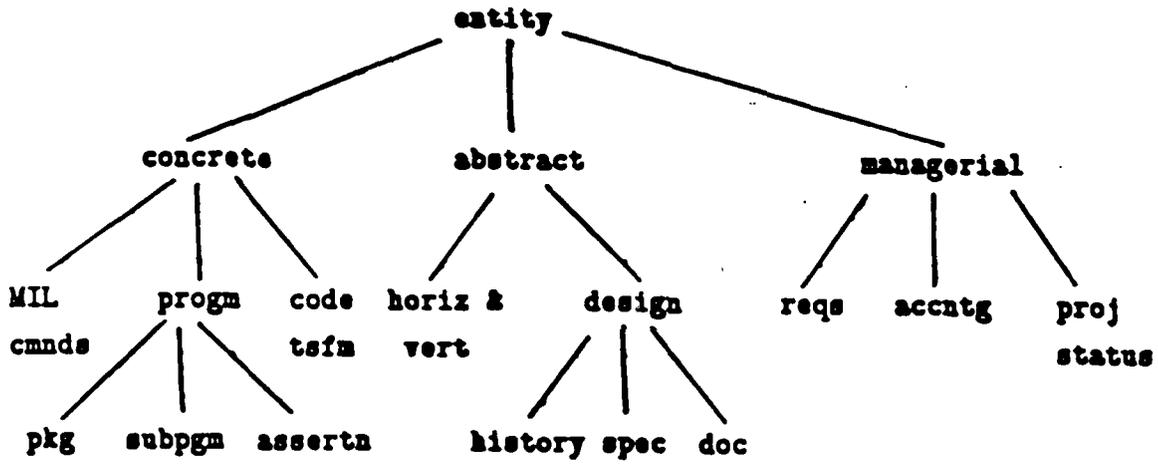
WHAT TO STORE IN THE LIB.

Again, goal: to maximize reusability
not only of COMPILED ADA PACKAGES

but also: {
- DESIGNS
- VIEWS + TRANSFORMATIONS

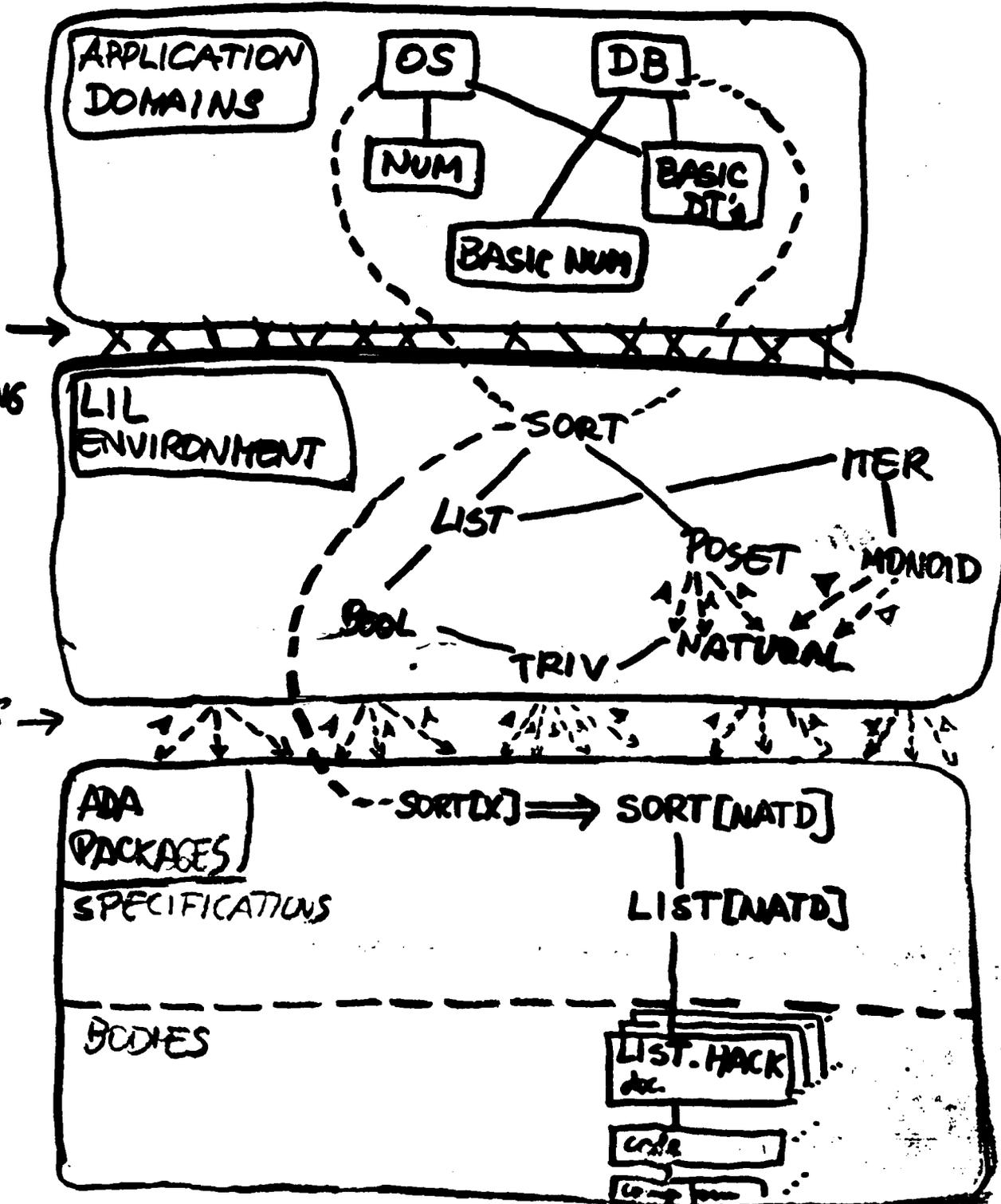
- DESIGNS (e.g. OS's design structure
in LIL + documentation (parameterized))
- THEORIES, SPECIFICATIONS, VIEWS (+view states)
- PACKAGES (esp. generic ones)
with relevant info about { validation } ver. (testing)
{ performance }
• version control
- SEMANTIC RELATIONSHIPS BETWEEN
DIFFERENT LEVELS OF ABSTRACTION
HORIZONTAL + VERTICAL STRUCTURE (provided
by LIL) very important for understanding + maint.
- MANAGERIAL INFORMATION (for a team)

Taxonomy of Library Entities



HAVE : NESTED HIERARCHICAL STRUCTURE

("BOXES WITHIN BOXES")



CATALOGING + RETRIEVAL

KEYWORDS {

- (a) data (stack, file, graph..)
- (b) function (sort, search, multiplication)
- (c) application domain (OS, DB, NUMER)
- (d) other (date, author, algorithm author)

(a) + (b) available in THEORIES, VIEWS, PACKAGE

(c) obtainable by interactive abstract production in dialogue with hierarchy of expert systems parallel to appl. doms.

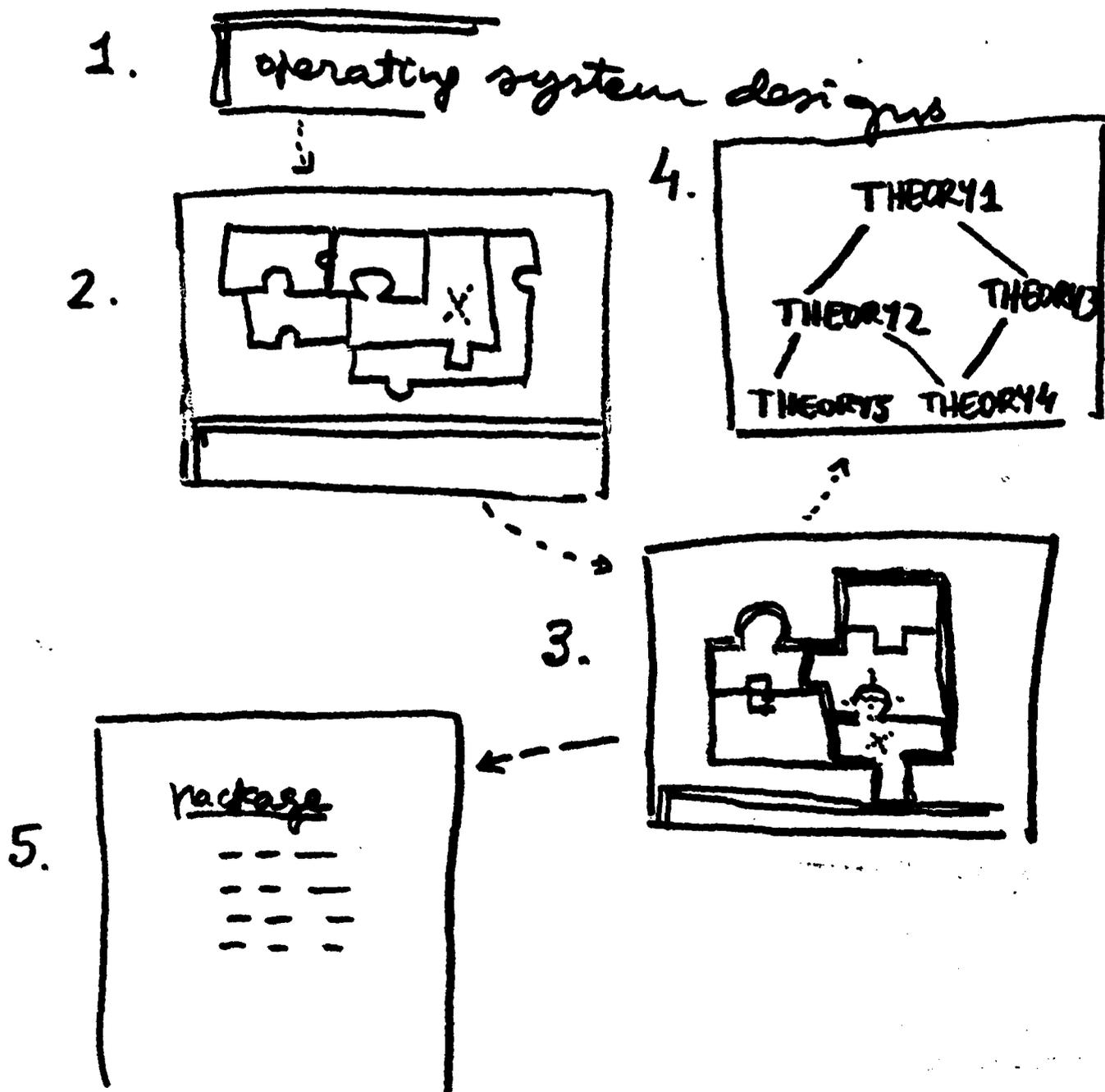
(d) available in the package bodies + other doc.

ALL THE ABOVE INFORMATION REFLECTS THE
NESTED HIERARCHICAL STRUCTURE + PROVIDES
CROSS-REFERENCING

SHIRIFF

1. → RETRIEVAL: Interactive, menu-driven query system, helping the progr. refine his queries/insia

NAVIGATING THE LIBRARY BY USING THE NESTED HIERARCHICAL STRUCTURE + MENUES + KEYWORDS



USER INTERFACE

- Nested hierarchical structure supports different levels of abstraction and has a natural GRAPHICAL representation that can be used for:
 1. - program understanding
 2. - navigation and retrieval
 3. - "meccano set" hyperprogramming
 - not esoteric math, but pictures that help the intuition
 - LIL operations correspond naturally to graphical manipulation of the icons
- Wand • MULTIMEDIA: text, graphics, mouse, voice
 - ANIMATION: for understanding + testing specs + bodies

ANIMATION

Providing PACKAGES with executable algebraic specifications à la OBJ:

1. Can do design testing + rapid prototyping before the code is written
2. Can improve dramatically the amount of testing by eliminating the need for a human "oracle" to interpret test results.
3. Can understand the behavior of specifications and bodies by ANIMATION. Idea: use constructors. By observing the errors dynamically: can more easily spot their sources.

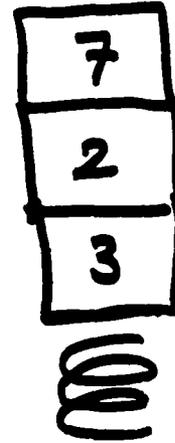
For STACK

Constructor is PUSH

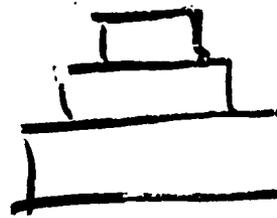
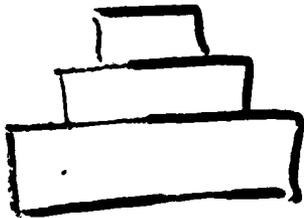
Standard Animation:

PUSH - 7
|
PUSH - 2
|
PUSH - 3
|
EMPTY

Fancier:



Similarly: Towers of Hanoi



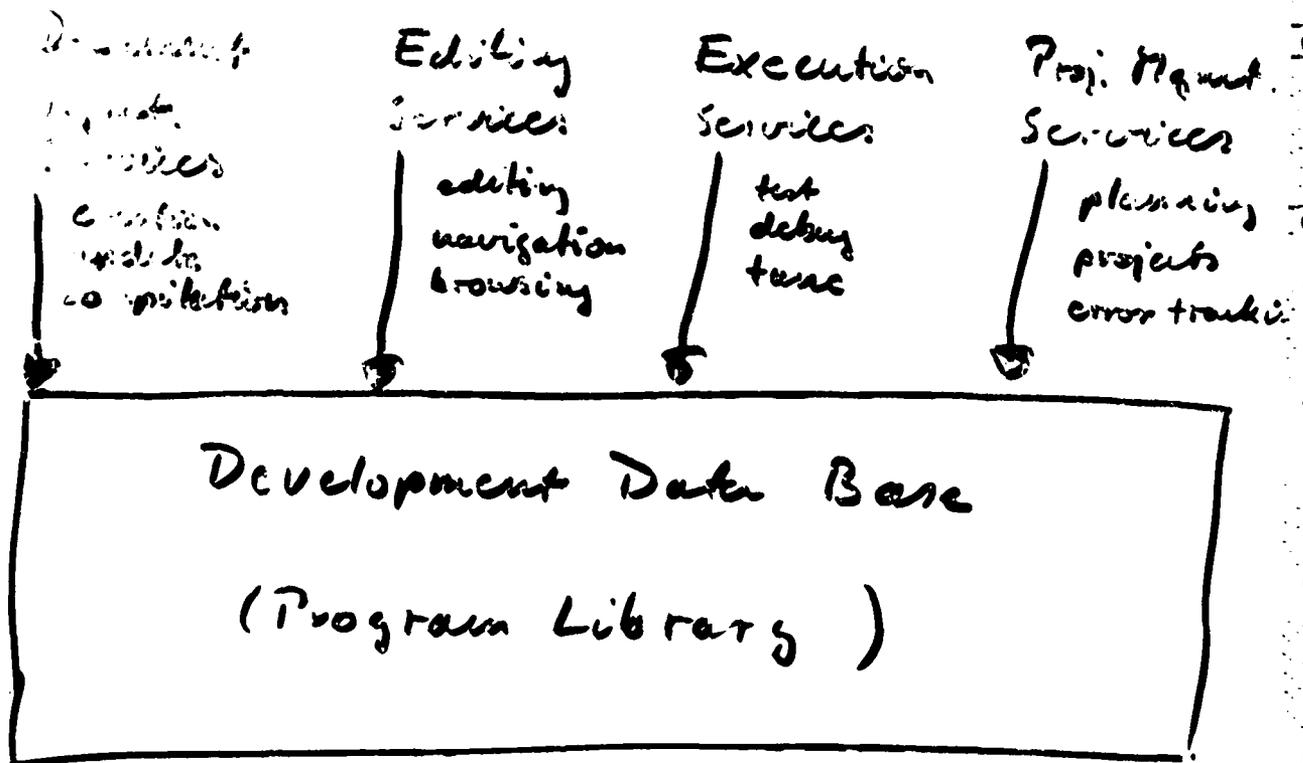
IV.10 Version Control in Program Libraries

WALTER TICHY

Version Control
in
Program Libraries

Walter Tichy
Purdue University

The Development Data Base is the central data structure for any programming environment.



Outline

Motivation

Version Control Concepts

" " Functions

Implementation techniques

Document:

A named, separately identifiable collection
of information

Source Document

manually generated

history attribute:

- author
- date/time
- (- documents used)

other attributes:

state attribute
phase attribute
type/language

Derived Document

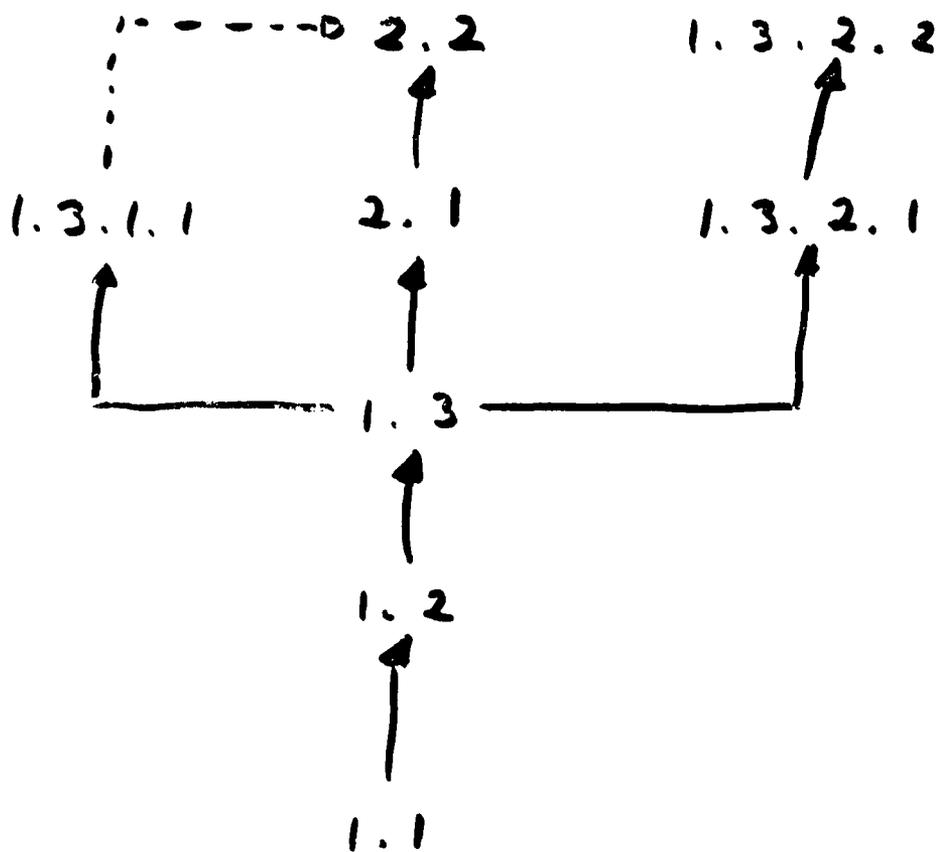
fully automatically
generated

history attribute:

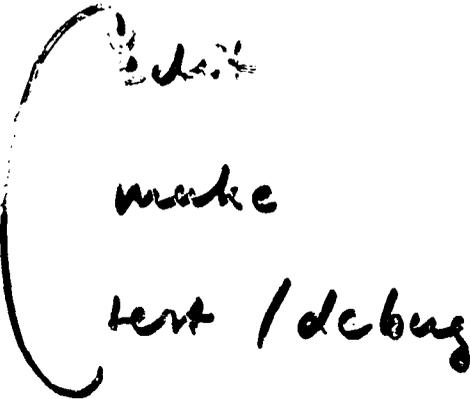
- documents generated
from
- generation process
- date/time

Revision: A source document that was created by manually revising an existing document.

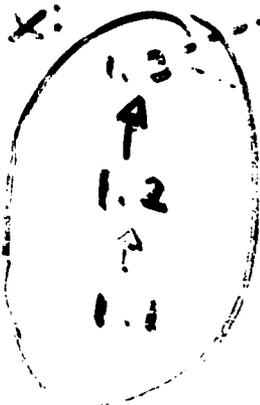
Revision group: A set of revisions that evolved from each other under manual revision.



Update of a Revision Group

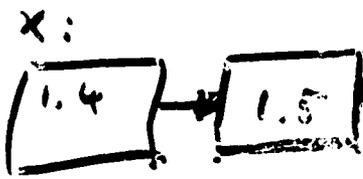
1. checkout all revisions to be modified
2.  edit
make
test / debug
3. checkin changed revisions
(log message requested)

Revision groups



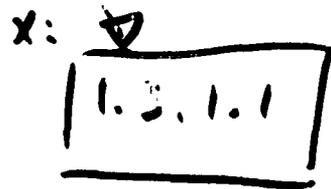
checkout

Workspace W1



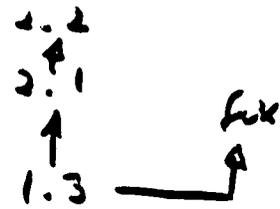
cancel
free rc
checkin

Workspace W2

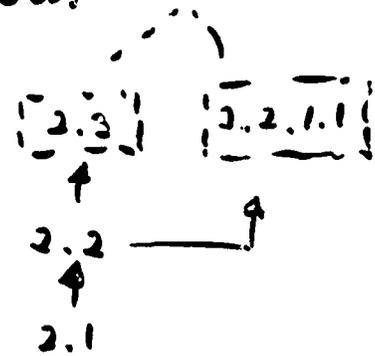


Reasons for branching:

1. temporary fixes

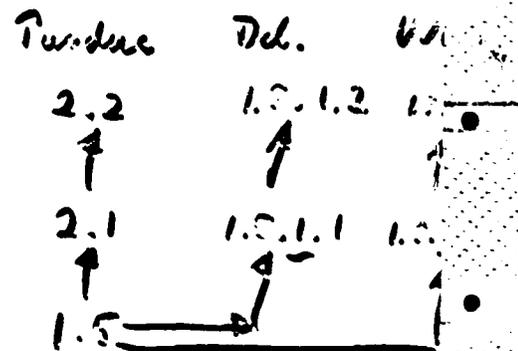


2. experimental modifications



3. Update conflict

4. Parallel development



Configuration:

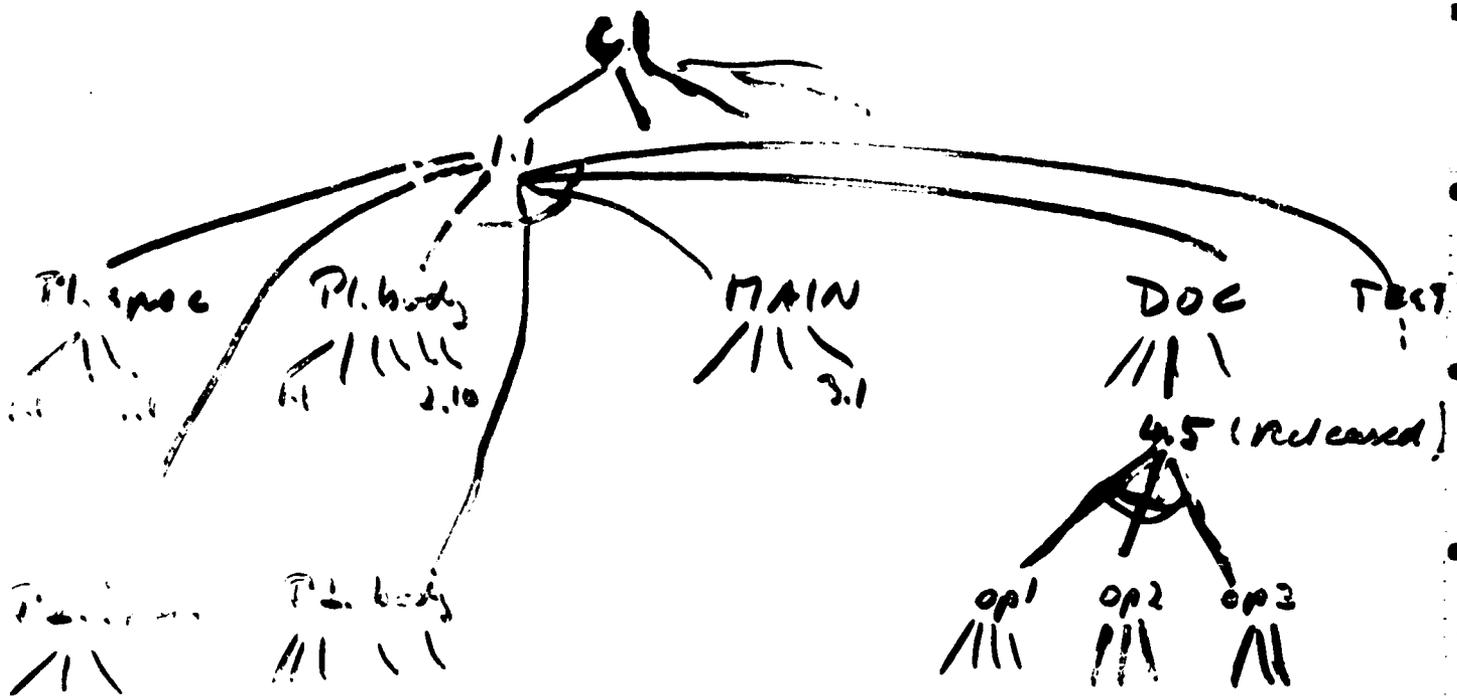
collection of related, but individual documents or other configurations.

Examples: link configuration
test configuration
program + documentation + manual

Configuration Description:

list of names of components,
possibly partially resolved.

C1 = (P1.spec, P1.body,
(1.1) P2.spec (1.1), P2.body (1.2))
MAIN (3.*),
Doc (Released)
Test (3.*)



 Revision groups
 (OR-mode)

 Configuration description
 (AND-mode)

Configuration Descriptions

can be collected into Revision Groups.

Selection from Revision Groups:

- Development Mode
(use co'ed revisions were possible)
- Global Selection Mode
(revision number,
revision name,
date, author, state
)
- Interactive Mode

Automatic Generation of Derived Documents

Derivation rules:

document type	→ processor	document type
(source or derived)		(derived)
(configuration or atomic)		(confg. or atomic)

Regeneration options:

- explicit command
- immediate regeneration (one step)
- global regeneration (transitive)

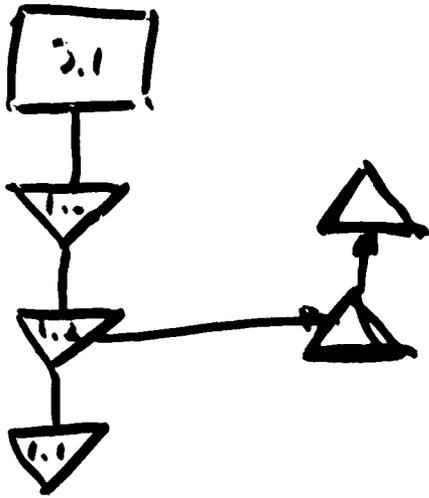
Regeneration events:

- Update of checked-out revision
- Freeze
- Unfreeze
- State attribute change

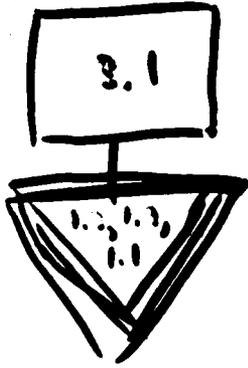
Implementation Techniques

- Difference Techniques for
Revision groups

Representations:



Fast regeneration
of latest revision



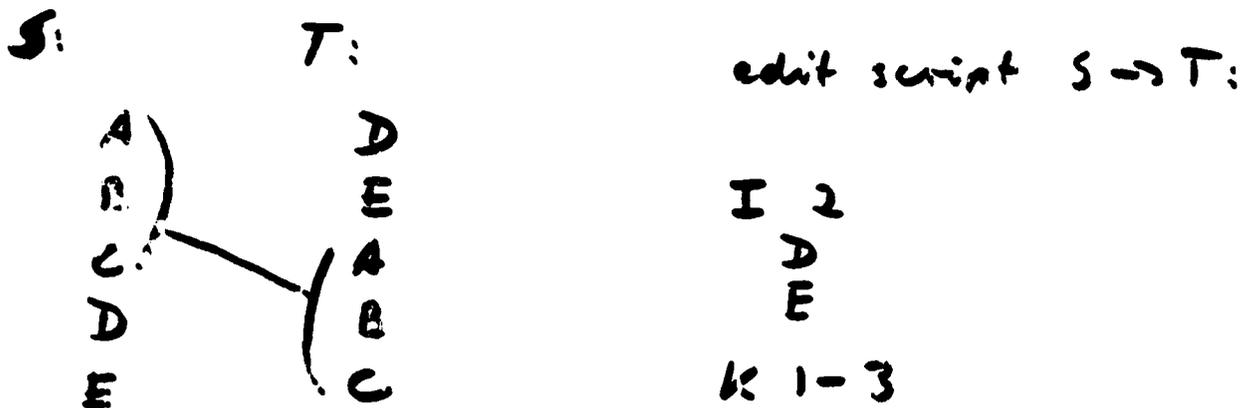
merged backward
deltas.
Fast regeneration of all
revisions.



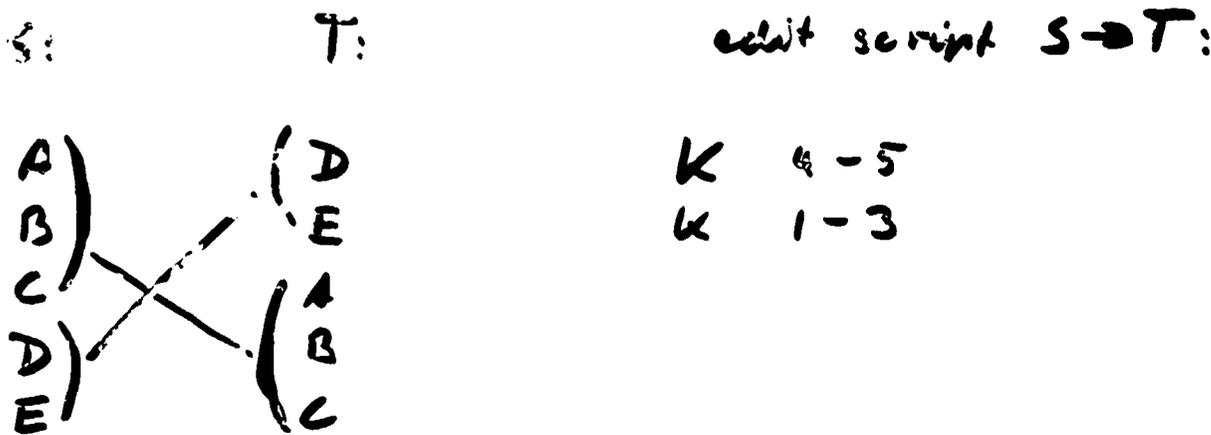
merged
forward
deltas:
poor
performance

Computing Differences

Longest Common Substring:



Block Moves:



Algorithms for text, trees, graphs exist.

Using a DBMS to
implement the DDB

Advantages:

Simplified implementation

- I/O
- high-level queries
- access control
- synchronization
- stable storage
- distribution (?)

Disadvantages:

Slow (Order of magnitude)

Data Structuring extremely
restrictive.

Not extensible

(Not suitable for long strings.)

2 languages (EQUEL & C)

Future Research

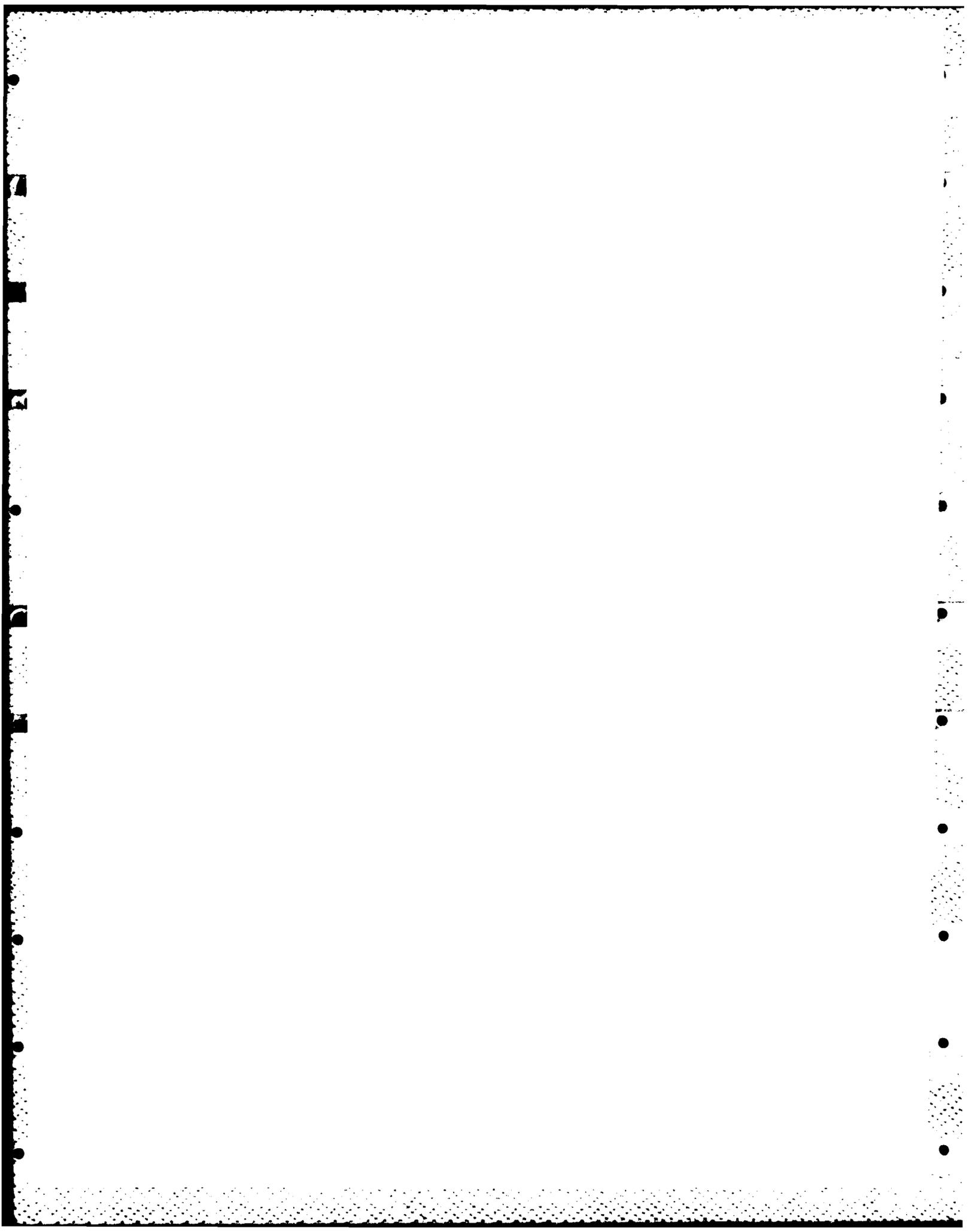
Implement DDB with an
object-oriented file system.

- passivate / activate
replaces I/O
 - synchronization provided
 - full flexibility of host language.
- ... language system.

Future Research

Implement DDB with an
object-oriented file system.

- passive/active
replaces I/O
- synchronization provided
- full flexibility of host language
- in language system.



AD-A149 570

REPORT ON ADA (TRADEMARK) PROGRAM LIBRARIES WORKSHOP
HELD AT MONTEREY CAL. (U) SRI INTERNATIONAL MENLO PARK
CA J A GOGUEN ET AL. 03 NOV 83 N00014-83-M-0088

4/4

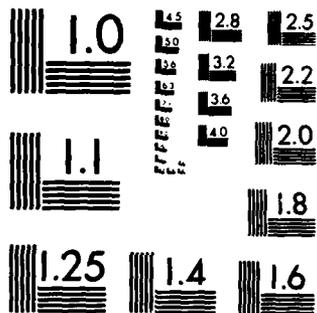
UNCLASSIFIED

F/G 9/2

NL



			END
			FILED
			DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

IV.11 Using ANNA for Specifying and Documenting Ada Packages

FRIEDRICH W. VON HENKE

Using ANNA for Specifying and Documenting Ada Packages

Friedrich W. von Henke
Stanford University

ANNA - Annotated Ada
an annotation language for Ada,
currently being developed at Stanford
with B. Krieg-Brückner
D. Luckham
O. Owe

- (1) Basic ideas underlying the Anna design
and overview
- (2) Anna features useful for package specification
and documentation

2

Anna is an annotation language for Ada

- extends Ada by "annotations"
- leaves Ada untouched
an Anna program is still a legal Ada program
- annotations may express:
restrictions, claims, specifications,
requirements, formalized comments,
"documentation"
- annotations range from low-level (statements,
objects) to high-level (packages, concepts, "theories")

Goals

- provide precise (formalized) documentation basis for processing by machines
- extend Ada expressibility
- provide (semantic) specification of packages independently of package body
- provide uniform and fitting framework for application of current specification techniques to Ada
- provide basis for (formal) verification and (less formal) validation
- not an attempt to develop new specification techniques

Anna has been designed "to go with Ada"

- using Ada concepts as basis for syntax and semantics as far as possible

An Ada programmer should not have to learn yet another language.

- basic concept: extend Ada by formal comments, marked by special comment delimiters:

--: virtual Ada text

--| formal annotations

Virtual Text

- for definition of specification concepts, auxiliary functions/packages/computations
- follows Ada rules (with very minor exceptions)

Expressions in Annotations

simple extensions of Ada expressions:

- conditional expressions

if $x < y$ then y else x end if

- propositional operators

$A \rightarrow B$ $A \leftrightarrow B$

- quantifiers

for all $X: \text{NATURAL} \Rightarrow$

exist $Y: \text{NATURAL} \Rightarrow Y \leq \text{SQRT}(x) < Y+1$

- modifiers to denote initial and final values of objects

in $x + y$
out ($x \geq$ in x);

- states of composite objects and package states

STACK'INITIAL [PUSH(x); PUSH(y); POP(z)]

Annotations

- syntax close to that of Ada
- semantics designed so that it agrees with Ada semantics
 - in many instances derived from Ada
 - attempts to be realistic
 - e.g. allows for partially defined expressions
 - annotations have a scope of application
 - annotations generalize the Ada concept of type constraint:
 - In general, an annotation constrains certain values in certain program states.
- An Anna program is considered consistent if the Ada text is consistent with the annotations, i.e. if all constraints imposed by the annotations are satisfied.

Kinds of Annotations

- subtype annotations

subtype SMALL is INTEGER range 1..16;

subtype SMALL is INTEGER;

-- | Where $X: \text{SMALL} \Rightarrow 1 \leq X \leq 16$;

type INTERVAL is record

LB, UB: INTEGER;

end record;

-- | Where $I: \text{INTERVAL} \Rightarrow I.LB \leq I.UB$;

- object annotations (in declarative parts)

-- | $X < Y$;

- statement annotations

- assertions (same syntax as object annotations)

- "invariants" over a scope

-- | with ORDERED(A, I, J);

while J < K loop

 J := J + 1;

end loop;

- subprogram annotations

- annotations on parameters (input (output conditions))

- result annotations

function Sqrt(X: NATURAL) return NATURAL;

-- | return Y: NATURAL => $Y+2 \leq X \leq (Y+1)^2$;

- annotation of exception propagation

- package annotations

- axioms

- abstract package states

- package invariants

- annotation of exception propagation

- "weak": raise E => state = in state;

- "strong": S.Length = max => raise OVERFLOW;

- context annotations

with P;

-- | limited to P.A, P.B;

package Q is

-- "restriction" of
-- visibility

- annotations of generic units

- constraints on generic parameters

- templates for annotations in instances of unit

- currently no facilities for dealing with tasking

Specification and Documentation of Packages

extend package specification (the 'invisible part')
so that it contains all the desired information
(that is not expressed in the Ada text)

- semantics of subprograms:
 - subprogram annotations
 - package axioms
- constraints on subprogram parameters:
 - input/output annotations of subprograms
- package state (properties, constraints):
 - object annotations
 - subprogram annotations
- use of global objects:
 - which ones are actually used:
 - context annotations
 - how are they accessed/modified:
 - object annotations
- abstract data types:
 - package axioms
- exceptions:
 - which may be propagated
 - under what conditions will an exception be propagated
 - what conditions (on state) hold when an exception is propagated
 - } propagation annotations

generic

type ITEM is private;

MAXSIZE: POSITIVE;

package STACK is

OVERFLOW, UNDERFLOW: exception;

-- function LENGTH return NATURAL;

-- function "=" (S, T: STACK'TYPE) return BOOLEAN;

procedure PUSH (E: ITEM);

-- where

-- STACK.LENGTH = MAXSIZE => raise OVERFLOW;

-- raise OVERFLOW => STACK = in STACK;

procedure POP (E: out ITEM);

-- where

-- STACK.LENGTH = 0 => raise UNDERFLOW;

-- raise UNDERFLOW => STACK = in STACK;

-- axiom

-- for all S: STACK'TYPE, X, Y: ITEM =>

S [PUSH(X); POP(Y)] = S,

S [PUSH(X)]. POP'OUT.E = X,

S [PUSH(X)]. LENGTH = S.LENGTH + 1,

-- S [POP(X)]. LENGTH = S.LENGTH - 1;

-- STACK'INITIAL.LENGTH = 0;

end STACK;

for all S: STACK'TYPE => 0 <= S.LENGTH <= MAXSIZE

Virtual packages

for packaging documentation concepts
virtual functions/predicates, "predicate abstractions"
"specification theories"
re-usability, pre-definition, use of libraries as for
regular packages

Annotation of Generic Units

- annotations in the body are templates for annotations of each instance (similar to Ada text)
- annotations of generic parameters constrain instantiation
- effect of "theory parameters"

```

--: generic
--:   type ITEM is private
--:   type INDEX is (<>);
--:   type ROW is array (INDEX range <>) of ITEM;
--: package PERM is
--:   function SWAP(A: ROW; I, J: INDEX) return ROW;
--:     where
--:       return A[I => A(J); J => A(I)];
--:   function PERMUTATION(A, B: ROW) return BOOLEAN;
--:   axiom
--:     for all A, B: ROW; I, J: INDEX =>
--:       PERMUTATION(A, A),
--:       PERMUTATION(A, B) -> PERMUTATION(B, A),
--:       PERMUTATION(A, B) and PERMUTATION(B, C)
--:         -> PERMUTATION(A, C),
--:       PERMUTATION(SWAP(A, I, J), A);
--: end PERM;

```

```
--: with PERM;
```

```
generic
```

```
type ITEM is private;
```

```
with function "<=" (X, Y: ITEM) return BOOLEAN is <>;
```

```
type INDEX is (<>);
```

```
type ROW is array (INDEX range <>) of ITEM;
```

```
--| for X, Y, Z: ITEM =>
```

```
--| X <= X,
```

```
--| X <= Y and Y <= X -> X = Y,
```

```
--| X <= Y and Y <= Z -> X <= Z,
```

```
--| or Y <= X;
```

```
package SORTING is
```

```
--: package P is new PERM (INDEX, ROW);
```

```
-- use P;
```

```
-- function ORDERED (R: ROW) return BOOLEAN;
```

```
--| where
```

```
--| return for all I, J: R' RANGE =>
```

```
--| I < J -> A(I) <= A(J);
```

```
procedure SORT (R: in out ROW);
```

```
--| where
```

```
--| out (ORDERED (R) and PERMUTATION (R, in R));
```

```
...
```

```
end SORTING;
```

- Preliminary Reference Manual is almost completed
- Anna frontend is under development
- axiomatic semantics
- Checking implementation:
Transformation of annotations into (runtime) checking code
- extension/modification of Anna for application to
"earlier stages in the life cycle":
Towards an Ada PDL

END

FILMED

2-85

DTIC