

DTIC FILE COPY

2

RADC-TR-90-59
Final Technical Report
May 1990



AD-A224 797

**4TH ANNUAL KNOWLEDGE BASED
SOFTWARE ASSISTANT CONFERENCE
PROCEEDINGS**

IIT Research Institute

Nancy L. Sunderhaft

DTIC
ELECTE
JUL 31 1990
S E D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

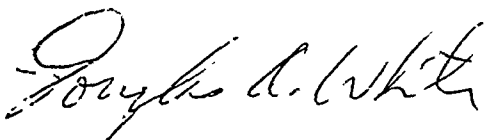
90 07 31 026

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

Permission has been granted to publish papers in this report (suitable for public release).

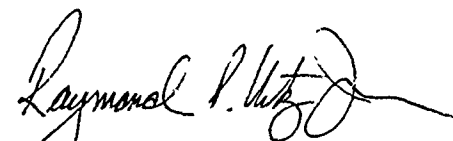
RADC-TR-90-59 has been reviewed and is approved for publication .

APPROVED:



DOUGLAS A. WHITE
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



IGOR G. PLONISCH
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

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

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE May 1990		3. REPORT TYPE AND DATES COVERED Final 12 to 14 Sep 89	
4. TITLE AND SUBTITLE 4TH ANNUAL KNOWLEDGE BASED SOFTWARE ASSISTANT CONFERENCE PROCEEDINGS				5. FUNDING NUMBERS C - F30602-87-D-0094 PE - 63728F PR - 2532 TA - QC WU - 08	
6. AUTHOR(S) Nancy L. Sunderhaft					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) IIT Research Institute Beeches Technical Campus RT 26 North Rome NY 13440-2069				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Air Development Center (COES) Griffiss AFB NY 13441-5700				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RADC-TR-90-59	
11. SUPPLEMENTARY NOTES RADC Project Engineer: Douglas A. White/COES/(315) 330-3564					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Rome Air Development Center (RADC), Command and Control Directorate (CO), sponsored the 4th Annual RADC Knowledge Based Software Assistant (KBSA) Conference held September 12-14, 1989 in Syracuse, New York. The annual conference provides a forum for exchanging technical and managerial views on the progress of the RADC program for developing a knowledge-based life cycle paradigm for large software projects, the Knowledge Based Software Assistant. The mature KBSA will provide intelligent assistance to system builders in producing quality mission-critical computer resources (MCCR). Software developed using the KBSA is expected to be more responsive to changing requirements, to be more reliable, and to be more revisable than software produced using current practices. The KBSA will improve software practices by providing machine-mediated support to decision makers, formalizing the processes associated with (Continued on Reverse)					
14. SUBJECT TERMS Artificial Intelligence, Software Assistant, Knowledge Based Systems, Automatic Programming, Knowledge Based Programming				15. NUMBER OF PAGES 390	
				16. PRICE CODE (P)	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED		18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	
				20. LIMITATION OF ABSTRACT UL	

Block 13 Continued:

software development and project management, and providing a corporate memory for projects. The KBSA will utilize artificial intelligence, automatic programming, knowledge-based software engineering, and software environment technology to achieve the goal of providing an integrated environment for developing MCCR systems.

Ch-1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100

NOTES

Although this report references the following limited documents, no limited information has been extracted.

KBSA Performance Estimation Assistant, Aug 89; USGO agencies and their contractors.

KBSA Framework (Phase I), Oct 88; USGO agencies and their contractors.

Knowledge Based Requirements Assistant (Vols 1 & 2), Oct 88; USGO agencies and their contractors.

The Knowledge-Based Specification Assistant (Vols 1 & 2), Feb 89; USGO agencies & private individuals or enterprises eligible to obtain export-controlled technical data IAW regulations implementing 10 U.S.C. 140C.

3RD Annual Knowledge-Based Software Assistant Conference, Mar 89; USGO agencies & private individuals or enterprises eligible to obtain export-controlled technical data IAW regulations implementing 10 U.S.C. 140C.

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



4th Annual RADC KBSA Conference

Monday, 11 September 1989

5:00 p.m. to 9:00 p.m.

Registration & Reception

Ramada Inn; Executive Room
1305 Buckley Road
Syracuse, New York
(315) 457-8670

Tuesday, 12 September 1989

7:45 a.m. to 8:45 a.m.

Breakfast Tutorial (Reservation Required)

Ramada Inn; Syramada Room
Mr. Douglas A. White
Rome Air Development Center

7:30 a.m. to 9:00 a.m.

Registration & Continental Breakfast

Ramada Inn; Victorian Room

9:00 a.m. to 9:30 a.m.

Welcome & Opening Remarks

Dr. Fred L. Diamond, Chief Scientist
Mr. Douglas A. White
Rome Air Development Center

9:30 a.m. to 10:00 a.m.

KBSA User Interface Environment

Mr. Joshua Glasser
Honeywell, Inc.

10:00 a.m. to 10:30 a.m.

Reusing Software Developments

Dr. Allen Goldberg
Kestrel Institute

10:30 a.m. to 10:45 a.m.

Refreshments Break

10:45 a.m. to 11:15 a.m.

Building Evolution Transformation Libraries

Dr. W. Lewis Johnson
USC/Information Sciences Institute

11:15 a.m. to 12:00 p.m.

Coordinators and Coordinator Generators

Dr. Walter G. Morris
Software Options, Inc.

12:00 p.m. to 1:30 p.m.

Luncheon Buffet

Syramada Room

1:30 p.m. to 2:15 p.m.

The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems

Mr. Raymond C. McDowell
Unisys Defense Systems

2:15 p.m. to 2:45 p.m.

A Software Assistant for Automatic Test Equipment Engineering

Mr. David R. Harris
Sanders Associates, Inc.

2:45 p.m. to 3:00 p.m.

Refreshments Break

Spin-Off Technologies

4th Annual RADC KBSA Conference

Tuesday, 12 September 1989 (Continued)

3:00 p.m. to 3:30 p.m. **KBSA for Automated Software Analysis, Test Generation and Management**
Mr. Gordon B. Kotik & Mr. Lawrence Z. Markosian
Reasoning Systems, Inc.

3:30 p.m. to 5:00 p.m. **Software Demonstrations**
Board Room & Executive Room

KBSA Development Assistant: Dr. Allen Goldberg, Kestrel Institute
A demonstration of the knowledge-based support of code development. This demonstration will show the interactive development of a simple, easily understood problem, that of computing elementary statistics from a sample of data (mean, frequency, variance, etc.). The demonstration also shows the recording and replay of the derivation. That is, after deriving an implementation, the specification will be modified and the derivation replaced on the modified specification.

Romulus: Dr. David Rosenthal, Odyssey Research Associates
A demonstration of the construction of specifications and verification of a system.

6:00 p.m. to 9:30 p.m. **Cocktails & Conference Dinner Theater**
The Glen Loch Mill (circa 1827)
4626 North Street
Jamesville, New York
(315) 469-6969

6:00 p.m. to 6:45 p.m. **Cocktails; Cash Bar**

6:45 p.m. to 7:15 p.m. **System Integration Technology; Mr. Laszlo A. Belady**
Microelectronics and Computer Technology Corporation

7:15 p.m. **Dinner Theater; It Had to Be You**

Wednesday, 13 September 1989

9:00 a.m. to 9:30 a.m. **Progress on the Knowledge-Based Software Assistant**
Dr. Richard Jullig
Kestrel Institute

9:30 a.m. to 10:15 a.m. **Gala: An Object-Oriented Framework for an Ada Environment**
Mr. Donald Vines
Honeywell, Inc.

10:15 a.m. to 10:30 a.m. **Refreshments Break**

10:30 a.m. to 11:00 a.m. **KBSA's Requirements Assistant and Aerospace Industry Needs**
Dr. Douglas A. Abbott
McDonnell Douglas Corporation

11:00 a.m. to 11:30 a.m. **Knowledge-Based Specification, Analysis and Synthesis of Communication Protocols**
Mr. Lawrence Z. Markosian
Reasoning Systems, Inc.

Spin-Off Technologies

4th Annual RADC KBSA Conference

Wednesday, 13 September 1989 (Continued)

11:30 a.m. to 12:00 p.m. **Knowledge Based Reverse Engineering for Re-engineering and Reuse**
Mr. Philip Newcomb
Boeing Computer Services

12:00 p.m. to 1:30 p.m. **Luncheon Buffet**
Syramada Room

1:30 p.m. to 3:00 p.m. **Panel Discussion: Standards: Enabling or Crippling?**
The panelists will discuss emerging standards and how they may influence the future research and development activities of the RADC KBSA Program.

XWindows
Mr. Joshua Glasser
Honeywell, Inc.

IRDS
Dr. Henry C. Lefkovits
AOG Systems Corporation

CLOS
Mr. Aaron Larson
Honeywell, Inc.

MACH
Mr. James Spoerl
Encore Computer Corporation

CommonLisp
Ms. Ellen Waldrum
Texas Instruments Incorporated

3:00 p.m. to 3:15 p.m. **Refreshments Break**

3:15 p.m. to 4:45 p.m. **Software Demonstrations**
Board Room & Executive Room

Gais: An Object Oriented Framework: Mr. Donald Vines, Honeywell, Inc.
A demonstration of an object-oriented support environment for the Ada language.

KBSA Project Management Assistant: Ms. Marilyn Daum & Dr. Richard Jullig, Kestrel Institute
A demonstration of the knowledge-based support of code development. The current version of the Project Management Assistant, which now runs on Sun workstations, will be demonstrated. The primary new feature is a policy editor, which allows a manager to easily (graphically) state a policy for the system to monitor. Violations of policies automatically send messages to the appropriate manager. Other features to be demonstrated include project task and component planning, project scheduling, monitoring of progress through budget charts, assignment of personnel to tasks, personal "things to do" lists, task access control, and semi-automated communication of problems.

Knowledge Based Requirements Assistant: Mr. David Harris, Sanders Associates, Inc.
A demonstration of knowledge-based support for the evolutionary development of software system requirements using multiple text or graphic viewpoints.

Reusability Library Framework: Mr. Raymond C. McDowell, Unisys Defense Systems
A demonstration of a prototype software librarian based on cooperating knowledge-based systems.

Spin-Off Technologies

4th Annual RADC KBSA Conference

Thursday, 14 September 1989

9:00 a.m. to 9:15 a.m.	Administrative Remarks
9:15 a.m. to 9:45 a.m.	Relating Formal and Informal Descriptions of Systems Dr. W. Lewis Johnson USC/Information Sciences Institute
9:45 a.m. to 10:15 a.m.	KBSA Technology Transfer: An Industrial Perspective Mr. Chunka Mul Andersen Consulting
10:15 a.m. to 10:30 a.m.	Refreshments Break
10:30 a.m. to 11:00 a.m.	KBSA for Software Maintenance and Re-engineering Mr. Gordon B. Kotik & Mr. Lawrence Z. Markosian Reasoning Systems, Inc.
11:00 a.m. to 11:30 a.m.	An Approach to the Support of Dynamic Extensibility in Nonstop, Distributed Target Environments Dr. Charles W. McKay University of Houston-Clear Lake
11:30 a.m. to 11:45 a.m.	Closing Remarks Mr. Douglas A. White Rome Air Development Center
11:45 a.m. to 1:00 p.m.	Luncheon Buffet Syramada Room

Conference Committee

Mr. Douglas A. White
Rome Air Development Center

Ms. Nancy L. Sunderhaft
MIT Research Institute

Ms. Penny Muncaster-Jewell
McDonnell Douglas Space Systems

Ms. Mary Anne Overman
National Security Agency

Spin-Off Technologies

AN OVERVIEW OF RADC'S KNOWLEDGE BASED SOFTWARE ASSISTANT PROGRAM

Donald M. Elefante

Rome Air Development Center (COES)

Griffiss AFB, NY 13441-5700

Abstract

In 1983, Rome Air Development Center (RADC) published "Report on a Knowledge-Based Software Assistant." ¹⁴ That report integrated key ideas on how artificial intelligence (AI) might be used to design, develop and maintain software over the complete life-cycle. Since then, RADC initiated the first of three multiple-contract iterations intended to develop both a Knowledge-Based Software Assistant (KBSA) and its supporting technologies. This paper provides a KBSA overview and describes the most interesting results to date.

What is KBSA?

The Knowledge-Based Software Assistant is a formally based, computer-mediated paradigm for the specification, development, evolution, and long term maintenance of computer software. The paradigm captures system evolution history, providing a corporate memory of how parts interact, what assumptions were made and why, the rationale behind choices, and how requirements were satisfied. It also features an explanation facility that supports developers in grasping the project's state at any time and tying it to the specific application at hand. The current, evolving version of KBSA achieves these functions with a set of software modules or "facets" that can be thought of as sub-assistants. In the future, the facet functionality will be integrated through a common, underlying "framework" that coordinates life-cycle activities. Facets (and their progeny) currently under development support project management, requirements definition, system specification, program development and performance maximization. Facets under consideration for future development will support documentation, testing, and other areas yet to be defined.

The KBSA concept formalizes all activities and products of the software life-cycle, and all life-cycle activities are machine mediated and supported. In this connection, it exhibits four main features which depart from the most common software engineering paradigm, the "waterfall" paradigm (and its close derivatives).

Distinguishing KBSA Features

The first distinguishing feature is the use of incremental, executable and formal specifications. "Incremental" means that the specifier may gradually add detail to the specification as pertinent knowledge becomes available. This detail can be added by

interacting with appropriate specification presentations that help manage process complexity. One need not start out with a complete, or even an accurate, specification. "Executable" means that the specification "runs" as a high-level prototype and portrays the states and activities that will occur in the modeled system as it is exercised. This portrayal helps the specifier validate the specification against user intent. "Formal" means that the specification is expressed in a language exhibiting precise semantics. This avoids natural language ambiguity and renders a specification accessible to machine reasoning.

The second distinguishing KBSA feature is formal implementation. This means that all decisions made during system implementation are captured in formal descriptions and justified by analyzing those descriptions. This formality supports the use of meaning-preserving transformations for creating lower and lower implementation levels. The result is a verified implementation, where implementation validity arises from the very process by which the implementation is developed.

The third distinguishing KBSA feature is a project management policy that is formally stated and enforced. That is, project policy defines relationships between various software development objects (e.g., requirements, specifications, code, test cases, bug reports, etc.), and KBSA ensures that the relationships are held throughout development.

The last main, distinguishing feature of KBSA is that maintenance is done at the requirements and specification levels. This is important since maintenance activities are normally a result of new or better-defined user requirements. Requirements and specification maintenance then drives changes in the lower-level code through meaning-preserving transformations of the specifications. This high-level maintenance approach permits integrating old and new design decisions and validating the package before new code is generated. In addition, the formal capture of requirements, specifications and high-level maintenance activities permits simply "replaying" the development after modifying specifications or refinement decisions. Retention of the original design history helps to identify which parts of a fielded system must be modified. This makes the maintenance and upgrade of fielded systems simpler and more reliable.

These four features indeed capture the larger vision of the KBSA concept. However, system integration is also a major component of the larger vision. Additionally, supporting technologies and their evolution have played a critical role in KBSA development success.

System Integration

System integration in KBSA will involve defining and building a support environment that supplies full life-cycle-support services to KBSA users. The environment's role will be to embody the framework, facet, user-interface and other support functionality deemed necessary for a KBSA. However, this should be accomplished with a seamless, well integrated package that eliminates arbitrary facet distinctions. Therefore, an ideal support environment interface will have five characteristics. First, it will be implementation independent. Second, it will exhibit a well-conceived interface and communication protocol suite. Third, its services will be present at many granularity levels to support both commonality in usage and specialization. Fourth, it will exhibit quality explanation and help facilities. Finally, it will support extendibility to provide the integration of new functionality in a uniform way. These service characteristics are largely analogous to those provided by a kernelized, "portable" operating system.

Supporting Technologies

Evolving technologies that have been instrumental in supporting KBSA development fall into three main categories: the wide-spectrum language, general inferential systems and domain-specific inferential systems.

An ideal wide-spectrum language (WSL) is a single language that lets the user capture the formal semantics--at any level of detail or at any step in the development cycle--of the system under development. It permits uniform expressibility regardless of what is being described (e.g., requirements, specifications, code, test cases, project management policy, etc.), and it does so in a way that is syntactically and semantically consistent at and between all abstraction levels. The WSL is intended to cover the spectrum from requirements to implementation, from management to maintenance. A mature WSL for KBSA should also cover graphic and schematic system representations as well as conventional, written representations. A great deal of headway has been made on all these fronts.

A general inferential system is one that supports reasoning and is applicable to all problem domains. We must be concerned in KBSA with this reasoning efficiency and the data representations used. We need modular expansion facilities to accept domain-specific inference modules or more efficient decision procedures. And the user interface must make the system useful for practical applications. Mechanisms that support inheritance, constraint propagation and planning must be present, and theorem provers may be integrated with the inference engines used to perform meaning-preserving transformations of specifications. All of these qualities will be seen in the KBSA facets described later.

Domain-specific inferential systems extend general inferential systems to address the unique software development features within a given domain. Three dimensions comprise the character of a domain-specific inferential system: the application domain, the life-cycle phase being addressed, and the functional area (e.g., presentation domain, structured text domain, evolving system description). For each problem domain, special reasoning and solution-finding rules apply. We know that inferential systems inherently based on special domain rules are more efficient than general inferential systems applied to encodings of the special domain. In KBSA, this technology area has focused on achieving good knowledge representation of software development objects and inference rules; also, how they can be formally represented and used for further reasoning. Therefore, three sub-areas of investigation are also pertinent here: formal semantic models of the software development environment, the knowledge representation and management of software development objects in the domain, and specialized inferential systems that incorporate rules tuned to a specific problem or task.

When the KBSA report ¹⁴ first came out in 1983, supporting technologies were not adequately developed. Neither was the software development process well understood. To address these shortfalls, an evolutionary, three-iteration contractual approach was undertaken. The next section describes the basic approach and key accomplishments of each contract, and shows that KBSA supporting technologies have matured along with the facet developments.

KBSA Facet Developments and Related Work

The first of the three iterations was initiated in 1985 and is now largely complete. Some overlap with the second is also taking place. Iteration 1 has focused on designing individual KBSA facets and pulling the supporting technologies as needed. Universal solutions haven't been sought. Rather, solutions unique to each facet have been pursued to help facet developers understand and formalize their chosen life-cycle phases. Because of

the need to better understand the software-development process with respect to KBSA, initial facet partitionings were intentionally chosen to parallel phase designations of the waterfall paradigm. However, over the long haul, KBSA will unveil its own identity and adopt the broader vision of an AI-inspired computer assistant shorn of unnecessarily restrictive notions about life-cycle partitionings or phased development.

Although the formalizations derived under iteration 1 have centered on the products of individual phases (e.g., requirements, specification, and code), they have more significantly focused on the process of how these products came about. The sections that follow elaborate on this.

The first-iteration KBSA efforts are: Project Management Assistant (PMA) 1 and 2, Requirements Assistant (RA), Specification Assistant (SA), Performance Assistant (PA), Development Assistant (DA) and KBSA Framework (KF). Efforts marking the start of iteration 2 are the Requirements/-Specification Assistant (Req/Spec), and the KBSA Concept Demo.

Project Management Assistant

Work on a formalism definition for PMA and the construction of a prototype began in 1985^{8,9,23} by Kestrel Institute. The goal of PMA within the software development life-cycle was to provide knowledge-based help to users and managers in project communication, coordination, and task management.

PMA capabilities fall into three categories: project definition, project monitoring and user interface. Project definition consists of decomposing the project into individual, manageable tasks and then scheduling and assigning them. Once manageable tasks have been defined for the project, the project must be monitored. In PMA, this monitoring primarily takes the form of cost and schedule constraints, but the enforcement of specific management policies (e.g., DOD-STD-2167, rapid prototyping, KBSA, etc.) is also included. Finally, the PMA user interface supports interactions involving project monitoring and definition, and uses direct queries and updates, Pert charts, and Gantt charts.

Although the above capabilities are important, we would expect them of any project management tool. What sets PMA apart from its predecessors is its expressibility and flexibility. Not only does PMA handle user defined tasks, but it also understands their products and the implicit relationships between them (e.g., components, tasks, requirements, specification, source code, test cases, test results, and milestones). Also captured in PMA are software development objects unique to programming-in-the-large: versions, configurations, derivations, releases, and people.

From a technical perspective, the advances made in PMA include: the formalization of the software development objects enumerated above, the development of a powerful time calculus for representing temporal relationships between software development objects, and a mechanism for directly expressing and enforcing project policy.

PMA formalizes software development products (e.g., components, tasks, milestones, requirements, specifications, test cases etc.) from a project management perspective, and provides a language to describe the process by which these products came about. Current software engineering literature calls this concept "process programming,"²⁹ but it has been a part of PMA since its inception. Thus, PMA demonstrates that a software project can be described formally and reasoned about.

PMA also demonstrates WSL applicability within the project management domain. The WSL used for the PMA prototype is Reasoning Systems' REFINE. REFINE is best described as a programming language that provides constructs for doing specification in a variety of styles (e.g., functional, logical, procedural, and object oriented). These constructs include the following list: user-defined object classes, set-theoretic data types, constructs from first order logic, relations defined by assertions, transforms, a pattern language, and conventional programming constructs.²⁰

In addition to the above, REFINE supports the definition of new constructs in PMA, i.e., assertion types used to maintain knowledge base integrity. One example is CHECKING, a trigger that looks for particular conditions and then flags the system when they arise. A second is COMPLAINING, which is often used with CHECKING. It interacts with the project manager to explain what has gone wrong. MAINTAINING is a third assertion type that automatically insures that a given condition remains true. In general, PMA has shown that a single WSL can uniformly handle both software programming constructs and process programming constructs. This is a consequence of REFINE's expressibility and extensibility.

The PMA effort spent considerable resources developing a highly expressive, generalized time calculus. The calculus, an extension of James Allen's interval calculus for reasoning about time,¹ was developed by Peter Ladkin. Allen's relational algebra has 13 atoms and is therefore of finite size. PMA's time calculus, on the other hand, uses an infinite algebra²⁵ whose temporal logic primitives are implemented directly in REFINE. It is the infinite algebra that gives the calculus its unique expressive power in terms of time representation.

Some of the advantages cited by Ladkin include capturing the metaphysics of time and presenting a natural way for representing time. This includes allowing for the joining of seconds into minutes and days into weeks or months such that there is a linear hierarchy of standard units.

The initial PMA effort was completed in 1986.²³ A follow-on PMA contract was initiated in November 1987. Its goals are two-fold: First, the evolution of PMA with the purpose of expanding the formalized knowledge of project management and providing enhanced capabilities. Second, implementation of PMA as an integral part of a full-scale software engineering environment called SLCSE (Software Life-Cycle Support Environment).

Requirements Assistant

The work on the Requirements Assistant (RA)^{6,7,31} began in 1985 by Sanders Associates. The greatest RA challenge was to adapt to the informal nature of the requirements process. It also had to allow users to enter requirements in any desired order or level of detail. Therefore, RA's responsibility is to do the necessary bookkeeping to support user requirements manipulation, and to maintain consistency among requirements as they become known. RA's key features are (1) complexity management through the use of multiple presentations, including formal and informal expression modes, (2) reusability at the requirements level, and (3) the use of constraint propagation technology for supporting requirements traceability and trade-off analysis.

RA features a convenient user interface that spotlights multiple presentation modes, any of which can be used at any time. The user chooses the mode or modes that seem the

most natural for the task. Internally, only a single representation exists to maintain consistency, for multiple presentations are typically used to capture and validate user requirements.

The presentation modes available within RA are:

a. Intelligent note pad -- Diary notes on requirements collection. Support is provided through limited natural language, structure editing, and feedback on word recognition in a lexicon.

b. Graphical presentations -- These include context diagrams, system function diagrams, internal interface diagrams, functional decomposition diagrams, data flow diagrams, state transition diagrams, and activation tables (which provide an active-function list for each system mode).

c. Calculator-spreadsheet -- The tabular display of non-functional properties for each system function (tied to an underlying constraint propagation system for bi-directional propagation, retraction and explanation).

d. Requirements document -- Natural language requirements document.

Support for reusable domain knowledge components are of particular note in RA. As the user enters requirements, the knowledge base is updated. However, requirements needn't be described entirely from scratch. RA provides mechanisms for merging "reusable requirements components" into the evolving system description. Requirements components are analogous to Rich's and Water's cliches in KBEmacs.³² Like cliches, reusable requirements components are manually encoded with descriptions of standard system requirements derived from specific application domains (in the case of Sanders' effort, the air-traffic-control domain). When merged into the evolving system requirement, they become a source of system expectations. Expectations support the critiquing of the evolving specification. Capabilities associated with critiquing include: detecting inconsistencies and duplicate objects, checking for reasonable values, and detecting missing information.

RA provides two mechanisms for merging specific reusable requirements components into the evolving system requirements. The first mechanism employs inheritance along user defined relations. Here, when the user defines an object as an instance of some supertype, supertype expectations are inherited (supertypes are examples of reusable components).

The second mechanism employs automatic classification. Here, the system tries to establish supertype relations of the most specific type possible, thus inferring the greatest amount of information possible and providing the most specific expectations on the evolving system. RA does automatic classification by consulting built-in tables within each supertype to determine appropriateness of that supertype. These tables are generally driven by requirements analysis tradeoffs, i.e., time, space, and other resource considerations.

Unique requirements constructs were necessary, for RA to have the capabilities described thus far. Sanders used SOCLE,¹⁵ their wide-spectrum-language based on frame and constraint inference, and derived from Roberts' and Goldstein's FRL, and Steele's constraint based language. Expressed in SOCLE, three classes (realms) of requirements constructs were necessary: Presentations, Structured Text, and Evolving System Description.

A presentation architecture³ handles objects that contain information about (1) how to display requirements in any of the presentation modes, (2) the presentation mode semantics, and (3) the impact of editing these presentations. The requirements/presentation boundary is crossed by presenters and recognizers. Presenters are concerned with the object's physical properties (i.e., the object being presented, screen coordinates, sizes, alignments, labels, and relevant graphical editing options), while recognizers are concerned with updating the evolving system description when the user edits the current presentation.

Structured Text is an object class that allows informally and formally related textual information to be managed as such. Groupings of related text are not necessarily dependent on the presentation mode, though presentation modes rely on structured text to represent text strings and their relationship to the current presentation. Initially, structured text objects are created interactively as the user enters more information. The groupings between these text objects evolve as the system is better defined. Finally, additional text objects are automatically created to paraphrase requirements statements first expressed in other non-textual presentations.

The Evolving System Description is the requirements repository in RA. Presentations and Structured Text merely reflect the Evolving System Description and, as a result, are tightly tied to it. Within this requirements construct class is an object-types hierarchy. The most general is the requirements object. A requirements object may have no more information than an associated Structured Text Unit that contains an uninterpreted text string and recognized key words.

At the next level down are activity, event, data, transition, and constraint object types. Each of these object types have more expectations on the requirements being expressed. Farther down the hierarchy, and thus more specialized, are specific reusable requirements components. Evolving-system-description components are type instances in this taxonomy.

The RA work was completed in October 1988.¹⁶ A prototype was delivered which is now serving a technology transfer function as well as feeding in to the new Requirements/Specification Assistant effort (to be briefly described below).

Specification Assistant

The main goal of the KBSA Specification Assistant^{2,21,22,24} is to yield a formal specification of the system under development and then validate it against user intent. Formal specification development must be supported in an incremental fashion, modeling the way developers typically construct specifications. Validation must be done by exposing the specification to the user at the earliest opportunity and continuing the exposure throughout the construction process. The effort to develop a KBSA Specification Assistant began in 1985.

In SA, a user begins specification development by describing a high-level GIST²¹ specification of the intended system. This specification is highly idealized, is usually incomplete, and does not include exceptional-case behavior. SA then provides the user with high-level transformation commands (HLTC's) to refine the high-level specification to a low-level specification where: (1) exceptional behavior is described, (2) all agents within the application are enumerated, (3) data boundaries are clearly defined (i.e., removal of the perfect knowledge assumption by each agent), and (4) all necessary functionality is enumerated and described in a semantically complete fashion.

HLTC's are similar to transforms in transformational systems, but are not necessarily meaning-preserving. In fact, most high-level transformation commands intentionally modify the semantics of the specification under development to achieve desired revisions. These semantics-modifying commands are referred to as evolutionary transformations. They explicitly formalize and carry out the evolution process which is an essential KBSA feature. Evolutionary transformations would typically be used to define new specifications, modify existing specifications, or merge existing specifications into larger ones.

When an evolutionary transformation is executed, it determines the impact of the change elsewhere in a specification and propagates further changes accordingly. HLTC's can thus assist in the process of integrating new specification components, such as those that might be retrieved from a library of cliches, into an existing specification. At present, over 100 high-level transformation commands exist.

An important effect of the user's applying HLTC's is that they formally capture specification evolution. Furthermore, typographical errors are less common because specification modification is automated. The real payoff for using HLTC's will come in the form of more mature application strategies surfacing earlier in the development cycle. Examples of new questions that can now be sensibly asked are: What does the developer want to do? Does a selected operation make sense within the current context? This work is just beginning.

In addition to incrementally refining and elaborating the specification under development, a user often wishes to incorporate previously developed specifications or portions of specifications. SA provides the user a package facility, called "views," that encapsulates specification components. Views are used both for reusing previously developed specifications and for focusing SA analysis tools.

View extraction is essential for removing unnecessary detail lingering from original definitions. It is accomplished when the user identifies critical specification definitions to be reused (with SA adding additional definitions discovered while tracing along dependency links). SA can also extract the transitive closure of any dependencies (e.g., retrieving all invoked procedures along with procedures that they, in turn, invoke).

Once a view has been extracted, it may be reused by merging it into a specification under development. Current merge capabilities are fairly simple. They can handle views containing no conflicting object definitions. However, new analysis tools are needed to deal with views containing conflicting definitions. In particular: When an object is unconstrained in one view and deterministically constrained in the other, under what condition can these constraints be reconciled when merged?

To simplify specifications, a focusing capability takes advantage of SA's view extraction facilities plus some HLTC's. Once a view has been defined it may be passed to any analysis tool. Such a tool will analyze the extracted view only. This allows the user to focus on small parts of the overall specification without being overwhelmed by the details of the full specification.

So far, only the sequential development and elaboration of specifications has been described. This deals effectively with a series of events, one occurring after another. But many times the sequential ordering is arbitrary--a function of when the developer happened to handle particular problems. In contrast, SA's parallel elaboration supports non-sequential specification development. In those cases where no interaction between parallel

activities exists, the solutions to supporting parallel elaboration are trivial. However, when minor interactions are called for, SA can integrate parallel development activities.

SA achieves parallel elaboration by customarily saving the development histories of the HLTC's used to arrive at a specification elaboration associated with any sequential elaboration path. If a user wants to integrate multiple elaboration paths, SA retreats to the common starting point and tries to interleave each development history. User guidance is needed and, in some cases, certain HLTC's may have to be reapplied in light of development histories interaction.

During specification development, SA permits the user to add more information about the specification--information that can't be adequately expressed in GIST. Rather than revising GIST, SA supports an annotation language. One example of its use: annotating specification statements according to whether they are requirements or goals. The distinction is that requirements are inviolable constraints, whereas goals describe general behavior that need not always be accurate and, in some cases, may involve exceptional cases not currently covered by the goal. Therefore, HLTC's first look at specification statement annotations to see whether they denote requirements or goals. Requirements will only be edited by HLTC's that are meaning preserving so that requirements satisfaction is insured. Goals, on the other hand, might be "compromised" by HLTC's to handle exceptional cases. This of course implies that non-meaning-preserving transformations are allowed to operate on statements annotated as goals.

SA is built on top of an integrated support environment called the Specification Service. The Specification Service was, in turn, built on top of three tools previously developed at ISI: AP5, ⁵ ISI's wide-spectrum language; CLF, ⁴ an object base built on top of AP5; and POPART, ³³ a tool for building and manipulating language parse trees given the language's BNF grammar definition (in this case, GIST). Within the SA effort, incompatibilities between these three tools were overcome so that all three could simultaneously work together on the same specification. SA also provides several integrated analysis tools for validating and debugging the specification. These include a symbolic evaluator, a behavior explainer, an influence graph generator, a GIST paraphraser, a static type analyzer and a resource analyzer.

Performance Assistant

The Performance Assistant ^{10,11,12,13} work began at Kestrel Institute in 1985. Technical issues addressed so far in the effort are (1) data structure selection (DSS) using symbolic and heuristic techniques, and (2) the development of PERFORMO, a functional specification language with set-theoretic data types. PERFORMO is similar to VAL, ²⁷ developed at MIT, and SISAL, ²⁸ developed at Lawrence Livermore Laboratory. PERFORMO is intended for DSS work, but is expressive enough to be a good initial specification language for further research on performance and implementation strategies.

Long term goals for a performance assistance are to guide software performance decisions at many levels in the software development cycle--from requirements specifications in very high-level programs to low-level code. The strategy is to combine heuristic, symbolic, and statistical approaches to provide capabilities for symbolic evaluation, data structure analysis and advice, and algorithm design analysis and advice. This effort is focusing on data structure selection and optimizations which include finite differencing, iterator inversion, flattening, operator elaboration, membership-test removal, loop fusion, and dead variable and copy elimination.

Briefly, finite differencing calculates a new expression-value within a loop by using the old expression-value in conjunction with a delta function (rather than recomputing the new expression-value from scratch). Iterator inversion is a special case of finite differencing that focuses on duplicate iterator expressions that can often be reformulated into a single iterator expression. Flattening, which is necessary for both finite differencing and iteration inversion, removes all nested function calls and provides names for all possible subexpressions. Operator elaboration transforms high-level operations into lower-level operations. Membership-test removal eliminates explicit membership tests when they can be inferred a priori. Loop fusion transforms multiple loops into a single loop to reduce overhead code and (sometimes) iteration requirements. Dead variable and copy elimination is done following automatic transformations, which often leave behind useless or redundant code.

PA's data-structure-selection strategy involves supplying refinement decisions to the implementation generator while reducing PERFORMO specifications to efficient Ada-class implementations. When PA needs a refinement decision, it determines which program properties are necessary to make a satisfactory selection. Properties include how a specific variable will be used and some its characteristics (like size and containment). Properties of a variable could include whether it is random access, ordered, enumerated, dynamic, or possibly empty. Based on such properties, specific implementation decisions can be made.

DSS's approach is to use basing as the representation methodology, when possible--particularly for complex data structures. Basing is a representation method used within the SETL data structure optimizer. Basing introduces a dual representation consisting of the object (data structure) and an accessor to the object. Multiple objects may share the same base. The base may also be some intermediate object upon which data access can be efficiently carried out. To determine the applicability of basing and what the appropriate base should be, the following analysis techniques are employed: Containment, 1-1, Bound, Sparseness, and Operation Analyses.

Framework

The KBSA Framework (KF) effort 17,18,26 brings a global perspective to KBSA. Initial work on KF began at Honeywell Systems and Research Center in early 1986. This effort seeks two goals: (1) to develop an integrated KBSA demonstration and (2) to propose the specification of a framework through which all facets must interact, communicate and eventually be built upon. The purpose of (1) is to provide a concept demonstration that would be intuitively obvious to the most casual observer. The purpose of (2) is to promote a tightly coupled interaction between facets. KF will provide a common reference for each facet developer and allow information sharing. KF interaction will be a requirement for all iteration-2 contracts. The future result will be a more tightly integrated KBSA.

The main technical challenges in this effort are to (1) define the minimum functionality that the framework must provide to all facets, (2) define a common interface for the facets to interact with each other, (3) extend the framework functionality to include a distributed environment, (4) support programming-in-the-large concepts like configuration control, traceability, and access control, and (5) provide consistent user interface services.

The first KF iteration will yield a demonstration of two independent facets tightly coupled to the KF. The facets may exist on a separate machine but will communicate via the KF. KF will be responsible for maintaining traceability between software development

objects and keeping facets updated about the status of those objects. The two facets, on the other hand, will be responsible for establishing the initial traceability (e.g., the relationship between requirements objects and specification objects) via the use of services supported by KF.

The difficult issue within the KF has been to characterize what indeed constitutes a framework definition. At the conceptual level, where agreement exists between all developers, KF should have or support the following characteristics:

- a. An object base.
- b. Logical relations and inference mechanisms that map to the object base.
- c. A distributed knowledge base.
- d. Access control.
- e. Configuration control.
- f. Transactions (i.e., collecting related operations into a single transaction to prevent the recognition of temporary knowledge base inconsistencies resulting from intermediate operations)
- g. Object permanence.
- h. A flexible user interface.

No deeper-level descriptions for the KF have been agreed upon to date. However, such a description must become available under the second iteration phase.

Early in the facet-development efforts, the importance of a common example was recognized. For the KBSA Framework demonstration, the domain will be substantial in that the air traffic control (ATC) problem (also analyzed by Sanders Associates and covered in the RA and SA) will be addressed. Although the demonstration will not solve the ATC problem or focus on its full scope, the rich set of requirements inherent in ATC will serve as an excellent driver for KBSA. ATC involves a variety of real world issues such as real time requirements, data base management, user interaction, interaction with the outside world, and changing or not-too-well-defined requirements.

Planning and Plan Recognition

As part of the Northeast Artificial Intelligence Consortium, which is sponsored by RADAC, the University of Massachusetts has conducted research in planning and plan recognition as it relates to the software development process. The primary role that this research will play in future KBSA will be in the areas of activities coordination and the "replaying" of developments.

A "plan" is a collection of partially ordered actions or operations that achieve one or more goals, given an initial state of the world. Each operator is tied to (a) one or more pre-conditions that define the state which must hold for the action to be legal, and (b) one or more post-conditions that define the state changes which result from performing the action.

"Planning" offers a mechanism for formalizing the software development process. It can be used to assist the programmer in achieving some end goal, to explain the current

project status in terms of completed or partially completed plans, or to automatically trigger an action for which all of the pre-conditions are satisfied.

"Plan Recognition" uses plans to infer user end goals from the actions that are being performed. The plan recognizer is able to act as an automated apprentice that "looks over the user's shoulder," correcting simple programmer errors and suggesting alternative actions. This mechanism will enable the construction of user interfaces that reduce the complexity of determining what operations to apply in a given situation. It can also minimize errors that result from performing inappropriate operations.

Development Assistant

The Development Assistant (DA) effort began in November 88 by Kestrel Institute. This effort will define a formalism that captures the knowledge and decision making processes used by software programmers and designers. The effort's goal is a knowledge-based system that derives efficient, executable code from a completed specification, automating wherever possible (via automatic transformation), and capturing user supplied design decisions otherwise. DA's assistance will include providing search trees of possible implementations, and permitting the user-in-the-loop to fine-tune his implementation on the basis of various performance factors, maintenance considerations, code reusability, etc.

Requirements/Specification Assistant

This iteration-2 contractual effort was awarded in May 1989 to the Information Sciences Institute of the University of Southern California (who also subcontracted with Sanders Associates). The effort's mission is to join the activities of requirements analysis and program specification by using a common knowledge base. This will provide a smooth transition from informal user requirements to formal, low-level specifications, thus enabling greater tracing and requirements validation. This has grown out of the recognition that both early requirements validation and the need for informal methods to support specification evolution are important.

Activities Coordination Formalism Design

In large software projects (and similar cooperative-work programs), a major share of the effort is spent either on coordinating the activities of participants or on repairing the damage caused by poor communication and coordination. Either way, costs grow explosively as projects grow larger. This contract, awarded to Software Options, Inc. of Cambridge, MA in October 88, will design a view formalism for project communication (which, here, means any exchange of information between project participants or agencies that could reasonably be computer mediated, even if it is not handled electronically in present practice). Views in this formalism will allow project participants to see the project's established communication and coordination protocols without having to study the detailed process model.

Because of its specific focus on project communication and the enactable model derivation, work under this effort will complement the activity coordination and project management work already under way in KBSA. One of the main objectives is to show that activity coordination by communication management is a useful technique throughout the military systems acquisition process, from the pre-award stage in which an RFP is generated to fielded systems maintenance. Therefore, protocol sketch notation will be made accessible to a broad spectrum of project personnel. Another central objective is to enable managers and other non-programmers to play a direct role in the creation and customization of enactable process models, and in their evolution over a project's life. To that end,

techniques will be developed to elaborate and refine protocol sketches into detailed process models. The design will show how tools could be provided to partially mechanize this derivation. However, human participation will still be needed as well, both to fill in details that are not represented in the sketch abstractions and to establish links with process model aspects that do not bear directly on communication.

KBSA Concept Demonstration

Targeted for contract award in the third quarter of 1989, the KBSA Concept Demonstration will first design then implement a "broad" and "shallow" software life-cycle processing system based on the KBSA paradigm. This system will be broad in the sense that it will demonstrate support for the entire system life-cycle, including both development and evolution. It will be shallow in the sense that it will provide less powerful assistance and functional completeness than will be required of eventual productized versions of a KBSA.

This effort will differ from the Framework effort in that the loose coupling and facet integration is not a goal. Rather, Concept Demo will provide essential experience with the KBSA paradigm as a whole, particularly the gathering of insights into requirements for an integrated high-to-medium-level interface suite, and high-to-medium-level process mediation and coordination. Furthermore, it will provide an additional vehicle for technology transfer to members of RADC's KBSA Technology Transfer Consortium, which was instituted in 1988 to promote KBSA technology transfer to industry. Therefore, the KBSA Concept Demonstration might be thought of as a vanguard for iteration 2 in the sense that it will attempt to give us insight into the software development process that takes us farther, still, from the waterfall paradigm and closer to the goals expressed in the KBSA vision. In particular, it should provide critical insight for formulation of the iteration-2 framework definition.

Standards and Technology Transfer

Throughout the KBSA program, efforts have been made to encourage the use of common conventions that will increase components reuse during KBSA development, and ensure a smooth technology transfer. Three communities have emerged that play a role in defining these conventions and serve as vehicles for transferring the technology:

1. KBSA developers -- Meetings of all the players that directly contribute technology to the program (government, industry, university) are held on a regular basis. This ensures coordination and participant awareness of new technologies and prototype components that might be shared.

2. KBSA Consortium -- RADC established the KBSA Technology Transfer Consortium to ensure a fertile base to transfer KBSA technology. The consortium, coordinated by RADC, is comprised of KBSA developers and a competition-derived set of "alpha" sites from industry, small business and academia. Each consortium member is committed to active technology transfer through reports and KBSA prototype evaluations, and the feedback of technical results from trial applications and studies. The actual technology transfer process as it relates to this consortium is as of much interest to RADC as the results because the operational agreement that participants signed up to is based upon synergy and perceived benefit rather than a legal arrangement or the exchange of funds for service.

3. External Community -- External KBSA program awareness is maintained through yearly KBSA Conferences that focus on KBSA and related knowledge-based technologies. Four such conferences have been held to date.

Within each of these technology transfer communities, the use and development of standard components has been emphasized. Each community has been a source of influence in developing "common" components that are referred to as "KBSA conventions."

To minimize effort duplication, the KBSA developers have been identifying and discussing common components that might be shared among two or more KBSA component developers. To date, standardization efforts have focused on user interface components, with the following three areas being discussed:

1. Editor standard -- Text manipulation is performed by the SA, KBRA, and the KF. A common editor convention is being defined based on GNU Emacs.

2. Natural Language Generation -- Explanation of the knowledge-based state is a necessary capability of several KBSA components. An initial natural language generation facility developed by one of the facet developers will be used as a basis for this convention.

3. Graphics toolkit -- The user interface to the KBSA will be highly graphical. Each of the existing facets and framework make extensive use of graphics and were developed using advanced workstations. A graphics toolkit based on the X-Windows package from MIT is now being proposed. The toolkit will provide a highly object oriented interface using many concepts originally defined in the Common Lisp User Environment (CLUE).

To facilitate technology transfer to consortium members, a limited set of hardware platforms have been identified for use as initial demonstration platforms. This set consists of: Sun 3, Symbolics 3600, TI Explorer II, VAX 11/780, and Macintosh II. These platforms have emerged from the initial platforms available to RADC and the current KBSA developers. The limited list is intended to reduce platform diversity and the number of hardware configurations needed to demonstrate the KBSA.

Currently, all KBSA developers have constructed systems using Common Lisp. With the inclusion of the Common Lisp Object System (CLOS) into the ANSI Common Lisp Standard, the KBSA developers are evaluating the CLOS adoption as a basis for future development. Currently, only the KF has been constructed using CLOS. Tentative plans call for CLOS to be used as well in future releases of the SA and the PMA.

Conclusion

Though significant progress is being and has been made in providing machine-in-the-loop assistance for specific life-cycle activities, we are still short of the technology needed for a true KBSA. In iteration 2, we will be focusing on more closely coupled assistants that do not fall neatly within the arbitrary life-cycle boundaries one sees in the waterfall model or between iteration-1 assistants. Instead, the second iteration will focus on continued development of a framework specification and assistants that are targeted at specific users--assistants that span the entire software life-cycle.

Acknowledgements

I extend my appreciation to Mr. David Harris of Sanders Associates, Mr. Steve Huseh of Honeywell, and Dr. Lewis Johnson of the University of Southern California Information Sciences Institute--all participating KBSA developers--who provided edifying comments and text to help insure that this paper is as accurate and up-to-date as possible. Special thanks go to Mr. Kevin Benner, a defacto collaborator in this paper. Kevin was an enthusiastic and tireless purveyor of the KBSA philosophy during his commission as a junior officer at RADC, and is responsible for gathering and organizing the bulk of the technical information found herein. He is now pursuing his Ph D at the University of Southern California and is employed part time by the Information Sciences Institute of USC.

References

- 1 Allen, J.F., "Maintaining Knowledge about Temporal Intervals," Comm. A.C.M.26(11), November 1983, 832-843.
- 2 Balzer, R. et al., "Knowledge-Based Specification Assistant," Interim Technical Report, RADC, Griffiss AFB, NY, Dec., 1986.
- 3 Cicarelli, E., "Presentation Based User Interfaces," TR 794, MIT Artificial Intelligence Lab, 1984.
- 4 CLF Project, "CLF Overview," USC/Information Sciences Institute, Mar, 1986.
- 5 Cohen, D., "AP5 Manual, ' User Manual, USC/Information Sciences Institute, Oct 19, 1987.
- 6 Czuchry, A. J. Jr., "Where's the Intelligence in the Intelligent Assistant for Requirements Analysis?," RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.
- 7 Czuchry, A., Harris, D., "The Knowledge-Based Requirements Assistant: A New Paradigm for Requirements Engineering," IEEE Expert, Nov. 1988.
- 8 Gilham, L., "KBSA-PMA Program Specification," Documentation, RADC Contract F30602-84-C-0109.
- 9 Gilham, L., "KBSA-PMA User Manual," Documentation, RADC Contract F30602-84-C-0109.
- 10 Goldberg, A. and Smith, D., "Towards a Performance Assistant," Interim Technical Report, RADC, Griffiss AFB, NY, Nov., 1986.
- 11 Goldberg, A. and Smith, D., "Performance Estimation for a Knowledge-Based Software Assistant," RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.
- 12 Goldberg, A., "Technical Issues for Performance Estimation," RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.
- 13 Goldberg, A., et al, "KBSA Performance Estimation Assistant Program Specification," Documentation, RADC Contract F30602-86-C-0026.

- 14 Green, C. et al., "Report on a Knowledge-Based Software Assistant," RADC Tech. Report TR-83-195, RADC, Griffiss AFB, NY, Aug, 1983.
- 15 Harris, D., "A Hybrid Structured Object and Constraint Representation Language," Proceedings of National Conference on Artificial Intelligence, Aug. 1984.
- 16 Harris, D., Czuchry, A., "Knowledge Based Requirements Assistant," RADC Final Technical Report TR-88-205 (two volumes), October 1988.
- 17 Huseth, S., "Analysis of Knowledge-Based Frameworks, Interim Technical Report, RADC/COES, Griffiss AFB, NY, Nov, 1987.
- 18 Huseth, S. and King, T., "A Common Framework for Knowledge-Based Programming," RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.
- 19 Huff, K., Lesser, V., "Plans and Meta-plans in an Intelligent Assistant for the Process of Programming", 2nd KBSA Conference, August 1987.
- 20 Huseth, Steve et al. (Honeywell, Inc.), "KBSA Framework (Phase I)," RADC Final Tech. Report, TR-88-204, October 1988.
- 21 Johnson, W. J., "Overview of the Knowledge-Based Specification Assistant," RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.
- 22 Johnson, W. J., "Turning Ideas into Specifications," RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.
- 23 Jullig, R., et al., "KBSA Project Management Assistant," Final Technical Report, RADC, Griffiss AFB, NY, July 1987, TR-87-78 (two volumes).
- 24 "Knowledge-Based Specification Assistant User Manual," Documentation, RADC Contract F30602-85-C-0221, June, 1988.
- 25 Ladkin, P., "Primitives and Units for Time Specification," AAAI-86, Philadelphia, PA, Aug. 11- 15, 1986.
- 26 Larson, A. and Huseth, S., "KBSA Common Framework Implementation," RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.
- 27 McGraw, J. R., "The VAL Language: Description and Analysis," ACM Transactions on Programming Languages and Systems, Jan., 1982.
- 28 McGraw, J. R., et. al, "SISAL: Streams and Iteration in a Single Assignment Language," Technical Report M-146, Lawrence Livermore Laboratory, Mar., 1985.
- 29 Osterweil, L., "Software Processes Are Software Too," Proceedings, 9th Annual ICSE, Monterey, CA, 30 March - 2 April 1987.
- 30 Reasoning Systems, "REFINE User's Guide," June 15, 1986.

- 31 Sanders Associates, "Knowledge-Based Requirements Assistant," Final Technical Report, RADC Tech. Report, TR-88-205 (two volumes), Griffiss AFB, NY, Oct., 1988.
- 32 Waters, R., "KBEmacs: Where's the AI?," The AI Magazine, Spring, 1986.
- 33 Wile, D., "POPART: Producer of Parsers and Related Tools Systems Builders' Manual," USC/Information Sciences Institute, July 1987.

Knowledge-Based Software Assistant

(KBSA)

**THE BASIS FOR A NEW SOFTWARE PARADIGM -- SHIFTING
FROM INFORMAL PEOPLE-BASED DEVELOPMENT TO FORMALIZED
COMPUTER-ASSISTED DEVELOPMENT.**

**DOUGLAS A. WHITE
4TH ANNUAL KBSA CONFERENCE
12 SEPTEMBER 1989**

WHY KBSA?

Software is Important to the Air Force!

- o **Computer Software Dominates the Functioning of Most Military Systems**
- o **Computer Software is The Problem in 7 Out of 10 Troubled Systems**
- o **Demand for Software is Growing at the Rate of 12% per Year**

CURRENT PARADIGM TECHNOLOGY

PROBLEMS:

- escalating cost of large software systems
- inability to field adequate software systems
- inability to do desired enhancements (MAINTENANCE)

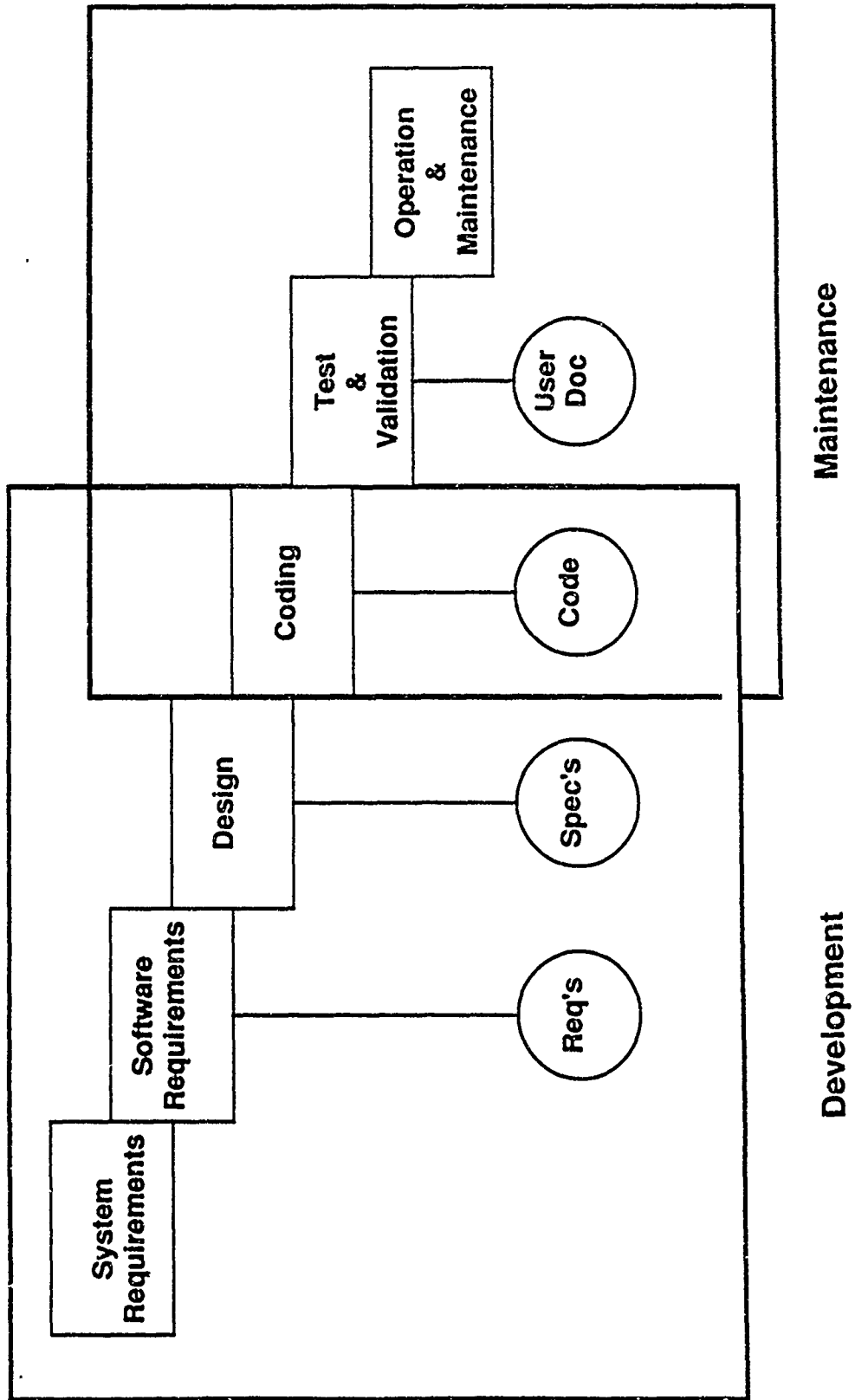
SOLUTIONS:

- new high level languages
- program support environments
- improved measurement of software
- better management

GAINS:

- modest

CURRENT SOFTWARE PARADIGM



CURRENT PARADIGM DEFECTS

Very Little Technology Exists for Managing the Knowledge Intensive Activities that Constitute the Software Development and Maintenance Process. The Existing Technology Deals Primarily With the Products.

- products are relatively independent
- the process is undocumented
- little is formalized
- history of lifecycle activities and rationales is not captured
- maintenance is concentrated on the implementation
- management is difficult

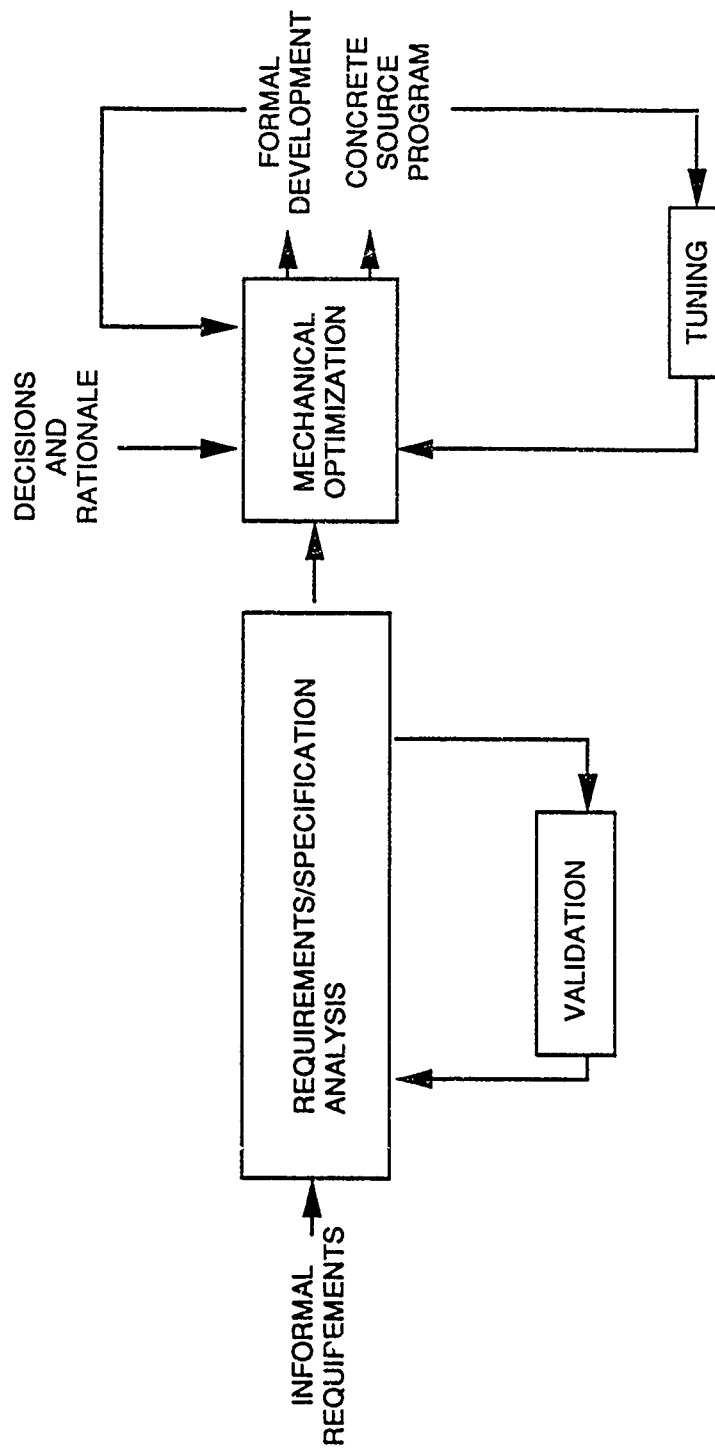
KNOWLEDGE-BASED SOFTWARE ASSISTANT

"Process Oriented"

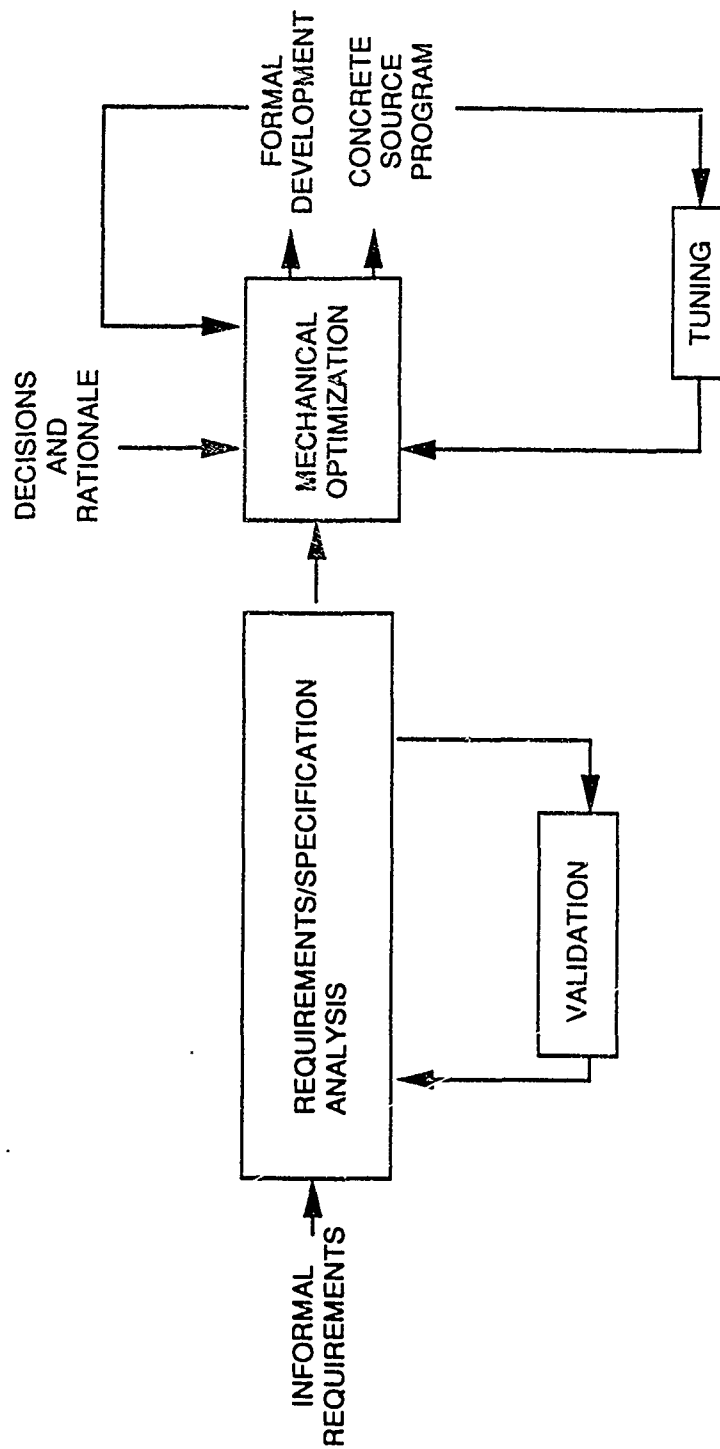
The Machine Is "In-The-Loop" and All Lifecycle Activities Are Machine Mediated And Supported

- capture the history of system evolution: the "corporate memory"
- formalize all aspects of lifecycle activities
- provide many facets of assistance
- imbed in an integrated system support environment

KBSA DEVELOPMENT PARADIGM



KBSA MAINTENANCE PARADIGM



KBSA vs CURRENT PRACTICE

- | | |
|--|--|
| o Formal Specifications | o Informal Specifications |
| o Automatic Prototype | o Prototype Unusual |
| o Specification Validated Against Intent | o Code Validated Against Intent |
| o Prototype Transforms to Implementation | o Prototype Is Implementation |
| o Implementation Automated | o Manual Implementation of Specification |
| o Testing Eliminated | o Code Tested Manually |
| o Formal Specification Maintained | o Code Maintained |
| o Automatic Documentation | o Product Documented, Design Lost |
| o Maintenance by Replay | o Maintenance by Code Patching |

AREAS OF MAJOR CHANGE

Features Which Distinguish the KBSA Paradigm From Evolution of the Current Paradigm.

Specification Validation

- executable specifications

Implementation

- derived from specifications

Project Management

- policies and procedures that provide activity coordination

Maintenance

- done at the requirements and specification level

VALIDATION OF SPECIFICATIONS

The Executable Specification is a Prototype of the System.

- specification is incremental**
- specification is derived from requirements**
- implementation is derived from specification**
- consistency with intent is demonstrated**
- several prototyping cycles economically feasible**

IMPLEMENTATION

The Implementation is Derived Through a Process of Refinement of the Specification via Formal Manipulations.

- controlled and selected by developer
- carried out by machine

Validity of Implementation Arises From the Process.

- the development itself constitutes a demonstration of correctness
- need for testing for consistency with specification eliminated

Reusability of the Specification

- system evolution
- families of programs

MAINTENANCE

Even Systems Matching the Original Intent are Never Static Because Users' Needs Evolve.

- improved capabilities
- new environment

Maintenance is Accomplished by Modifying the Requirements/Specifications and/or refinement decisions and reimplementing by "replaying" the development.

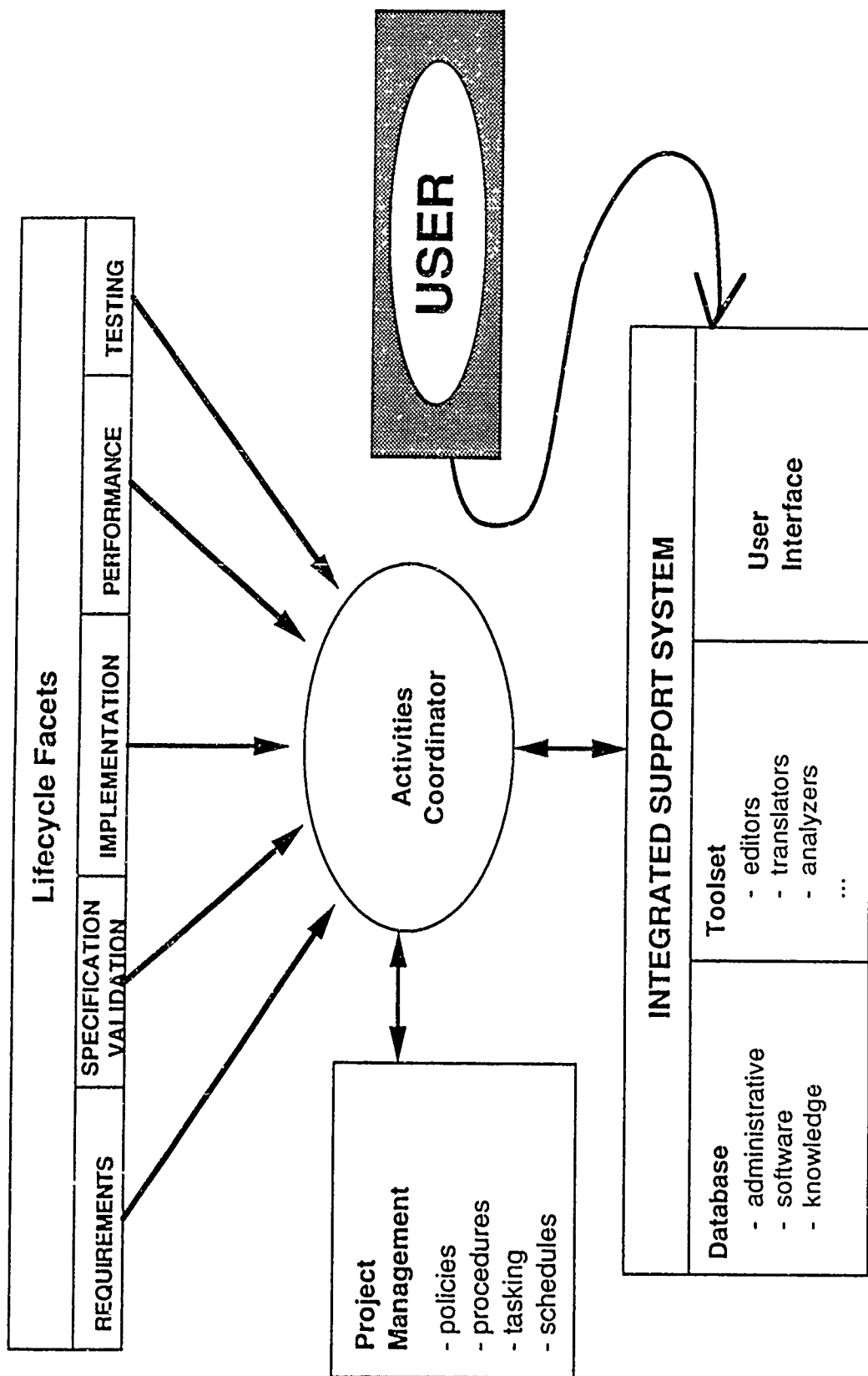
- maintenance activity parallels the original development
- eliminate the need to "patch" the (optimized) implementation
- increased development automation facilitates the "replay"

PROJECT MANAGEMENT

The KBSA Has The Means To Make Software Development Manageable.

- necessary data will be available
(all activities will be formalized, operations mediated and supported,
and progress recorded.)
- coordination of activities will be formalized
(project organization, resource allocation, states and choices,
transitions and authorization of these transitions)

KBSA ARCHITECTURE



BENEFITS

Systems can be

- more cost effective
- better integrated
- the result of many small evolutionary steps

Systems will remain "SOFT"

- modifiable/enhanceable

Systems can have greater complexity

- evolution central activity/prototyping standard

The whole will become more than the sum of its parts.

- cooperation among knowledge-based components
- reduce dependence upon people

KBSA PLAN

Short Range

- concentrate on individual facets
- support several frameworks
- research supporting technologies

Mid Range

- spin-off to conventional environments
- integrate facets on common framework
- continue facet technology development

Long Range

- fully integrated KBSA
- mature support technologies

KBSA

ISSUE: Application of AI Technology to Software Development

CONTRIBUTION:

- New Paradigm
- Power of Machine Turned Back Upon Itself
- Intelligent Assistant
- Corporate Memory

CONTRACT: Jun 82 - Jun 83

C. Green, D. Luckham, R. Balzer, T. Cheatham, C. Rich

REPORT: AD# A134 699

Report On A Knowledge-Based Software Assistant

Project Management Assistant

ISSUE: Project Planning and Monitoring

CONTRIBUTION:

- Task Definition and Work Assignments
- Cost Estimation
- Task Monitoring
- Project Communication
- Time Model of Tasks

CONTRACT: May 84 - Nov 86

Kestrel Institute

ENVIRONMENT: Symbolics, Refine, Lisp

DOCUMENTS: AD# A189 286 & B117 268

KBSA Project Management Assistant Vol 1&2

AUTOMATED MULTI-SOURCE KNOWLEDGE ACQUISITION AND MANAGEMENT

ISSUE: Automated assistance for differing user types in a distributed environment.

CONTRIBUTION:

- Planning technology
- Knowledge acquisition techniques
- Plan recognition
- Plan exception handling
- Distributed cooperating systems
- Man/machine interaction

CONTRACT: Nov 84 - Oct 89

University of Massachusetts at Amherst

ENVIRONMENT: TI Explorer, Macintosh

REPORTS: AD# A210 330

Northeastern Artificial Intelligence Consortium Annual Report

KNOWLEDGE-BASED SPECIFICATION ASSISTANT

ISSUES: Formal Specification and Specification Validation

CONTRIBUTION:

- Incremental Specification Language (GIST)
- Executable Specifications
- Formalization of Change
- Specification Evaluation
- Explanation

CONTRACT: Aug 85 - Aug 88

University of Southern California - Information Sciences Institute

ENVIRONMENT: Symbolics, TI Explorer, AP-5, Lisp

REPORTS: AD# B132 716 & B132 717

The Knowledge-Based Specification Assistant

KNOWLEDGE-BASED REQUIREMENTS ASSISTANT

ISSUE: Acquisition and analysis of requirements

CONTRIBUTION:

- Review and analysis through multiple presentations
(incremental, informal, "catch-as-catch-can")
- Reusable knowledge components
- Reasoning processes
(inheritance, automatic classification, constraint propagation)
- Knowledge representation - SOCLE (Frames and Constraints)

CONTRACT: Aug 85 - Feb 88

Sanders Associates

ENVIRONMENT: Symbolics Genera 7.2, Lisp

REPORT: AD# B130 358 & B130 359

Knowledge-Based Requirements Assistant, Vol 1&2

KBSA PERFORMANCE ASSISTANT

ISSUE: Assistance in specification refinement for algorithm and data structure design.

CONTRIBUTION:

- Automatic/interactive transformations
- Finite differencing
- Iterator Inversion
- Flattening
- Operator elaboration
- Performance estimation analysis
- Flow analysis & data structure selection

CONTRACT: Mar 86 - Nov 88
Kestrel Institute

ENVIRONMENT: Sun 4, Refine, KIDS

REPORT: RADC TR-89-98

KBSA Performance Estimation Assistant

KBSA FRAMEWORK

ISSUE: COMMON SUPPORT ENVIRONMENT

CONTRIBUTION:

- Support requirements of all facets
- Object-oriented programming
- Logic/Functional programming
- Multiple distributed facets
- Common user interface
- Demonstration of facet coupling

CONTRACT: May 86 - Jun 90
Honeywell, Inc

ENVIRONMENT: Symbolics or Sun, Franz Common Lisp

REPORT: AD# B130 699

KBSA Framework (Phase 1)

ACTIVITIES COORDINATION FORMALISM DESIGN

ISSUE: Abstract "process programming" language.

CONTRIBUTION:

- Formal syntax and semantics
- Visual formalism
- Long term execution
- Database connect
- Accessible
- Efficient

CONTRACT: Aug 88 - Feb 90
Computer Sciences Corporation/ Software Options, Inc.

ENVIRONMENT: Sun 3, Macintosh

REPORT: None

PMA for ADA

ISSUE: Extend PMA formalism and transfer technology to software engineering environment.

CONTRIBUTION:

PMA Model Enhancements

- alternative plan generation and analysis
- risk management
- cost estimation
- flexible communication
- user modifiable constraints and notice of constraint violation
- alternative modes of presentation
- variable granularity

Integrate with SLCSE

Adaptable prototype

CONTRACT: Nov 87 - Apr 90
Kestrel Institute

ENVIRONMENT: Sun 4, Refine, Common Lisp

REPORT: None

KBSA DEVELOPMENT ASSISTANT

ISSUE: Transformation of formal specifications to code.

CONTRIBUTION:

- Expand upon performance assistant capabilities
- High level transformation abstractions
- History/replay capability

CONTRACT: Sep 88 - Jun 91
Kestrel Institute

ENVIRONMENT: Sun 4, Refine, KIDS

REPORT: None

REQUIREMENTS/SPECIFICATION ASSISTANT

ISSUE: Enhancement and merging of requirements and specification facet technology.

CONTRIBUTION:

- Integrated capability for end-user and designer
- Traceability
- Prototyping
- Common interface
- Common Knowledge Base
- Distributed Capability

CONTRACT: Jun 89 - Dec 91

University of Southern California - Information Sciences Institute
Sanders Associates

ENVIRONMENT: TI Explorer, Common Lisp, AP-5

REPORT: None

KBSA CONCEPT DEMONSTRATION

ISSUE: Demonstration of the KBSA paradigm for entire lifecycle.

CONTRIBUTION:

- Broad lifecycle coverage
- Shallow facet expertise
- Uniform User Interface
- Integration of previously developed technology
- Vehicle for technology transfer

CONTRACT: Aug 89 - Feb 92

Andersen Consulting

University of Southern California - Information Sciences Institute

ENVIRONMENT: Symbolics, TI Explorer, Sun, Common Lisp, CLOS, AP-5

REPORT: None

SUPPORT FOR PERSISTANT OBJECTS IN A DISTRIBUTED SOFTWARE ENVIRONMENT

ISSUE: Persistent, distributed object-base management.

CONTRACT: SBIR, Aug 88 - Feb 89
Incremental Systems, Corp.

REPORT: N/A

KBSA User Interface Environment

This section describes a proposed design for the KBSA User Interface Environment (KUIE), a portable system for constructing user interfaces. The environment is highly object-oriented and is based on the Common Lisp Object System (CLOS). This specification is not complete and is expected to evolve as a result of prototyping and review by KBSA developers.

The toolkit consists of a set of object classes and methods that may be used to construct application user interfaces. Instances of objects created are active. They are aware of how to display themselves on the user's screen and their relationship to other objects being displayed. The power of expression and versatility of CLOS makes this approach a powerful design and development tool for the construction of complex user interfaces.

Several of the characteristics of the KUIE design have been principally drawn from Common Lisp User Environment (CLUe)[3]. The design has also been influenced by the Knowledge Base Requirements Assistant (KBRA) presentation mechanism[5] and other object-oriented user interface systems. KUIE expands these systems by making extensive leverage on the advanced object-oriented concepts provided in CLOS. This has simplified the interface and increased its flexibility.

In addition to our interest to maintaining a strong object-oriented flavor and compatibility with CLOS, we have also attempted to avoid dependence on any single window system such as X or Common Windows to increase portability. Several window systems have emerged with no obvious champion. Although a user interface subcommittee was formed within the ANSI X3J13 Common Lisp Standards Committee, little activity has occurred to date. We view this to be indicative of the quandary that the industry is currently in.

2.1 Window Systems

In the selection of an implementation platform, several key factors were considered important:

Portability Applications which use KUIE should be easily portable to any hardware/software environment which hosts KBSA. Thus the implementation platform must be highly portable.

Flexibility KUIE is intended to support the development of a wide variety of user interface styles. In particular, it should be possible to use KUIE to implement any of the user interfaces found in current Lisp development environments. Both graphical and textual interfaces should be easy to create. In order to achieve this, the implementation platform must be highly flexible.

Extensibility KUIE should provide the ability to define and deploy new types of user interface objects which refine and extend the behavior of more basic object types. KUIE provides this ability through the methodology of CLOS. Thus, the implementation platform should be readily extensible.

Compatibility The implementation platform for KUIE must be compatible with related software systems which will be separately standardized.

We examined several window systems in the design of, and choice of an implementation platform for KUIE. These include Common Windows, CUEForms, Symbolics Flavors, CLUE/CLX. The following tables show how the window systems we examined compare in the selected key areas.

Window System	Portable	Compatibility	Extensible	Public Domain Specifications
Common Windows	Yes	Yes	Yes	Yes
CUEForms	Yes	No	Yes	No
Symbolics Flavors	Yes	No	Yes	No
CLUE/CLX	Yes	Yes	Yes	Yes

Window System	Network Support	Level of Detail Ease of Use	Flexibility	Monolithic/ Multiple Screens
Common Windows	No	Low	Medium	Monolithic
CUEForms	No	Low	Low	Monolithic
Symbolics Flavors	No	Low	High	Monolithic
CLUE/CLX	Yes	High	High	Multiple Screens

The principal contenders for a standard Common Lisp user interface fall into two camps: the X-based systems such as CLX and CLUE, and Common Windows. Within each of these camps, minor differences exist that make applications that use one version of CLX or Common Windows not operate on another. We will briefly discuss each of the two systems and their future prospects.

The X-window system, or simply X, originated from the Athena Project at MIT. It is a C-based implementation that has been adapted to several other languages including Ada and Lisp. X consists of low level graphical services that provide a high degree of flexibility in constructing user interfaces, but the interfaces tend to be highly complex. No support is provided for common graphics structures such as menus, title bars, and pop-up windows.

The Lisp interface to X is called CLX. The Common Lisp User Environment (CLUE) [3] is an X toolkit developed by Texas Instruments to provide higher level object-oriented abstractions for constructing user interfaces. There is an ongoing debate as to whether the CLUE-level or the CLX-level is the most appropriate to provide the object-oriented abstractions. This discussion will inevitably result in changes to both interfaces.

Common Windows was first produced by Intellicorp to increase portability to the Knowledge Engineering Environment (KEE) system. It is one of the earlier Lisp-based user interface toolkits and has since been embraced by other Lisp software vendors. Franz Corporation has committed to a complete development environment on Common Windows implemented on top of an X base. This however, has required Franz to make custom modifications to both Common Windows and CLX reducing the portability of any applications using it.

2.2 KUIE Prototype Implementation Platform

Due to this state of flux, we have attempted to minimize our dependence on a specific window system. It is our intention that the complete KUIE system will provide a sufficiently rich number of primitive object classes for creating user interfaces that direct calls to the underlying window system will be minimal.

For the initial prototype implementation, we chose Common Windows for its high level window operations, availability, and general acceptance. Common Windows provided the necessary flexibility and performance. However, after discussions with several other KBSA developers, it became apparent that Common Windows would not serve our long term needs. The wide spread use of Common Windows is very much in doubt. We believe that it will become extinct due to the large support that has been given to X-windows. The number of platforms supporting X-windows and the scope of the X-windows standard (OSF, ANSI/ISO) means that it delivers a portability that Common Windows will never achieve.

As a result we have moved to using the CLUE system constructed on top of CLX as the window system platform. CLX alone represents relatively low-level functionality of the X-windows protocol. Using the CLUE toolkit makes the task of constructing KUIE interfaces significantly easier.

The kind of interoperability made possible by X means that user interface policy can be provided by special window manager programs, independent of both client programs and the window system. Common Windows is not prepared for this kind of world; it will end up conflicting with or duplicating the control of window titles, window geometry, etc., that is

provided by an X window manager.

Although we may have to work harder short term, the adoption of CLX as the basis of the window environment will ensure compatibility with emerging standards and commercial products.

2.3 Design Goals

In designing KUIE, three principle goals were important:

Object-Oriented Abstractions The benefits of object-oriented abstractions for user interfaces have been demonstrated in several operational systems [10, 9]. The approaches taken have proven to be capable of managing the complexity inherent in user interfaces constructed for applications, yet be highly flexible for constructing novice user interface features [8].

CLOS Compatibility The ratification of CLOS as part of the Common Lisp language [2] has provided an industry-wide base for object-oriented programming. Close association with the CLOS standard will ensure its use and acceptance by a larger community.

Portability Applications should be easily portable to any hardware/software environment which supports CLOS and an associated window system. We have attempted to avoid specific dependencies upon an underlying window system thus enabling KUIE interfaces to be constructed on a variety of window systems. (Note: A high degree of portability is a KUIE design goal, but differences between hardware and window system platforms result in different implementations of window coordinate systems, handling of regions, sizes and resizing, etc., which make achieving this goal difficult. Ease of portability in KUIE will come with the correct choice of a window system as an implementation platform.)

Extensibility The interface should provide the ability to define new types of user interface objects which refine and extend the behavior of the basic object types. KUIE provides this ability through the CLOS features of class specialization, method mixing, and inheritance.

Modularity KUIE should comprise a well-defined and self-sufficient layer of the user interface programming system. Using KUIE, an application programmer should be able to implement most types of user interfaces without accessing underlying software layers and without knowledge of the implementation internals of KUIE objects.

2.4 Overview

An interactive application program can be considered to consist of a collection of functions, some of which perform the processing that is essential to the application's purpose (e.g. text editing, knowledge base management, etc.). Other functions exist solely for the purpose of communicating with the application's human user. In KUIE, such human interface functions are represented by specializations of the *contact* object class. The contact class defines a primitive graphical object that is capable of being positioned and displayed on a graphics screen. Contacts are responsible for presenting application information to the user on the display screen, and for informing the application of input sent by the user via interactive input devices (such as the keyboard and the pointer). A contact generally embodies a component of the user interface that knows how to:

- display its contents,
- process input events that are directed to it, and
- report its results (if any) back to the application.

A contact provides a relatively high-level abstraction for user interface programming. Objects physically placed on the user's screen are subclasses of the contact class. The purpose of such an abstraction is twofold:

- To simplify and raise the level of the dialog between the application and the user. A contact insulates the application programmer from detailed behavior of a user interface component (such as displaying its contents and acquiring its input). As an "agent" of the application, a contact object can directly communicate with the user in terms closer to the application's domain.
- To define a uniform framework within which many different types of user interface objects can be combined. The contact class raises to a higher level the commonality between a great variety of interface objects — menus, forms, dials, scroll bars, buttons, dialog boxes, text entry, etc.

The subclassing and inheritance properties of CLOS are important to the use of contacts. A contact subclass implements a specific interface technique for input and output. Thus, a contact subclass can represent either an extension of a technique (such as a hierarchical-pop-up-menu which is a subclass of a pop-up-menu class), or it can provide a variation in style (as in a drop-shadow-pop-up-menu subclass). This protocol is expected to lead to the development of contact "libraries", providing a rich repertoire of interface techniques and a choice of several functionally interchangeable styles.

2.5 The User Interface Programming System

Contact objects represent an intermediate level of abstraction within a large user interface programming system. KUIE relies upon the services of a lower-level subsystem typically referred to as a *window system*. The window system provides programmer interfaces for controlling interactive I/O hardware such as the display screen, the keyboard, and the pointer.

Currently, KUIE is defined in terms of CLUE/CLX using the X Window System. Most window systems, including Common Windows, contain a component that is commonly called the *window manager*. The window manager is the part of the window system that provides a user interface to various operations on windows (i.e., changing a window's position, size, visibility, etc). We have assumed the existence of an underlying window system that performs these services and therefore has not been included in the KUIE specification.

KUIE distinguishes two different aspects of programming the user interface:

- defining a contact, and
- using a contact

The application programmer, who instantiates and uses a contact object, does not need to know how the class and methods of the contact were implemented by the contact programmer. In particular, the window system interfaces used by the contact programmer need not be visible to the KUIE application programmer. This distinction contributes to the separation of the application programming from the user interface programming.

2.6 Constructing User Interfaces With KUIE

KUIE defines two categories of user interface builders: the application programmer and the contact programmer. The application programmer has knowledge of the application that is being constructed and defines how the screen should appear to the user of the application. The application programmer will use and combine the existing contacts to achieve the desired user interface. The contact programmer constructs new contacts and specializations of existing contacts that may be used in new applications. The needs of the contact programmer are different than those of the application programmer in that he must be aware of the specific protocols that must be maintained for contacts and how to use the inheritance hierarchy to achieve the desired results. This difference in perspective between contact programmers and application programmers separates knowledge of the application from knowledge of the user interface. This is a primary goal of user interface management systems.

New contacts that require complex screen management operations may require the contact programmer to use underlying window system calls. Extensions of this form may introduce window system level dependencies that will limit the portability of applications using the

special contact. Consequently, contact programmers should avoid directly using window system services where possible.

KUIE graphical objects know how to organize themselves for presentation. They know how to present themselves, and how to unpresent themselves. Further, in our model there are mixins that combine these graphical objects and their presentations with additional attributes, such as mouse sensitivity, repositioning, etc. These graphical objects may be simple, such as an object representing a simple dot, or label (text string), or the object may be a composition of several parts, such as a line, a labeled box, or a graph.

In the case of an object representing a simple thing, the object's organization is to center its graphical element in a region, the dimensions of which are such that it is the smallest rectangular region that can fully encompass the graphical element.

In the case of a composite object, the object organizes itself by first calling down to all of its parts to organize themselves. Each organization of a component part results in that component's region being returned by the Organize method. The composite object then organizes these subregions into a new region.

When an object is presented, it displays its graphical elements in the region defined by the object of which it is a part (unless the object presenting itself is a stream, and the stream presenting itself is not a part of any other stream). The object then calls down its list of its parts to present themselves.

2.7 Network Support

Since KUIE is implemented on top of CLUE/CLX, the place to start examining the issues of networking and network support in KUIE is at the CLX level. KUIE provides an object oriented interface to the basic X paradigm found in CLX and shared by CLUE. This basic X paradigm is one of "one client, one display" (where "display" means a CLX display object). Refer to the X, CLX, and CLUE documentation for a complete description of this paradigm. In summary, KUIE windows operate in a network of separate servers, clients, and managers, each (in general) with its own address space.

Since all the KUIE objects are also KBSA objects and thus are distributed, not only can a client use multiple displays, a client can also share a single display object among multiple processes.

There are still some unanswered questions regarding the handling of a distributed independent "mouse process" which arises when a client is sharing an event stream with other client programs.

Protocols are being investigated to address the following kinds of networking issues:

- How will the mouse process be aware of all client displays since any client can create

its own display event stream?

- What if both the mouse process and the client process are interested in enter/leave events?
- Who removes events from the queue?
- How will the mouse process be able to find the CLOS object associated with each client window id?
- How are remote clients handled?
- How does mouse process feedback fit in with the window manager and other user interface policy arbiters for the server?

2.8 Event Dispatch

KUIE events include user input events, client synthesized events, and window system events. The KUIE event translation scheme can be extended to handle other types of events. In general, these events are handled by handle-event methods specialized on contacts.

2.9 The Contact Class

The fundamental properties of a contact are defined by the contact class.

Contact

Class

```
(defclass contact ()
  ((parts
    :type (listof contact)
    :accessor contact-parts
    :initform nil
    :initarg parts)
   (parent
    :type (or null contact)
    :accessor contact-parent
    :initform nil
    :initarg parent)
   (x
    :type integer
    :accessor contact-x
    :initform 0
    :initarg x)
   (y
    :type integer
    :accessor contact-y
```

```

: initform 0
: initarg y)
(width      : type integer
            : accessor contact-width
            : initarg width
            : initform 60)
(height     : type integer
            : accessor contact-height
            : initarg height
            : initform 50)
(state      : type (member '(nil organized exposed))
            : accessor contact-state
            : initform nil
            : initarg exposed)
(stream     : type stream
            : accessor contact-stream
            : initform nil
            : initarg stream)))

```

The Contact class defines the most primitive graphical object. It has a region which it occupies, a window stream upon which it is drawn, a presentation state, and a list of graphical objects it is related to in some class specific manner.

A contact may be a singleton or an aggregate. Aggregate contacts are graphical objects composed of other graphical objects. The constituents are specified in the *parts* slot of the contact and are physically drawn within the region defined by the parent contact. For each operation applied to a contact, the operation will be performed on the contact, and then recursively performed on each of contacts in the parts slot.

For example a graph made up of labeled boxes will be represented by a top level graph contact object. The parts slot will contain box contact objects and arrow contact objects. The parts slot of each box contact will contain a single label contact.

The slots of the contact class are described below.

parts

The list of constituents that make up the contact. If this slot is non-NIL, the contact is an aggregate and contains a list of contacts that make up this contact. By building hierarchies of contacts, complex graphical structures may be composed. If the slot is *NIL*, then the contact is a singleton.

stream

The stream object that the contact is displayed on. A contact is always associated with a stream object. All of the parts of an aggregate contact have the same stream object.

state

The state variable which controls the visual effect of the contact. A contact may be in one of three states: *organized*, *presented*, and *nil* (see section 2.10). The *organized* state denotes that the contact has been positioned on the stream object. The *presented* state denotes that the object has been physically displayed on the screen. Contacts may be *unpresented* to remove or hide the object.

x,y,width,height

These geometrical attributes of a contact window are pixel values that describe the position of the contact within the stream object. The height and width must be specified before the contact is *organized* or *presented* (see section 2.10). When a contact is *organized*, the *x* and *y* values of any constituent contacts are set. This establishes the position of the constituent contacts within the region of the parent contact.

2.10 Manipulating Contacts

The creation of a contact object is actually a two-step process in which a contact instance is first *organized* and then *presented*. Organization consists of setting the attributes of the contact and positioning the objects in the contact's parts slot. Presentation consists of drawing the contact on the graphics screen.

make-instance

Function

```
(make-instance contact-class-name initargs)
```

A contact is created using the CLOS *make-instance* function. An instance of the class specified will be created and initialized.

organize

Method

```
(defmethod organize ((c contact)))
```

The organize method positions the constituents of an aggregate contact within the region defined by the parent. Organize assigns positions to objects in the parts list of a contact and recursively calls the parts to organize themselves. The positioning algorithm used is class specific.

present Method

```
(defmethod present ((c contact)))
```

Present physically draws a contact object on the stream object specified by the contact and calls the objects in the contact's parts list to present themselves.

unpresent Method

```
(defmethod unpresent ((c contact)))
```

The unpresent removes the object from the users screen. This method may be used to provide pop-up icons that are removed after some user action.

stream Class

```
(defclass stream (contact))
  ((x-extent      :type integer
                  :accessor stream-x-extent
                  :initform 0
                  :initarg x-extent)
   (y-extent      :type integer
                  :accessor stream-y-extent
                  :initform 0
                  :initarg y-extent)
   (height-extent :type integer
                  :accessor stream-height-extent
                  :initform 0
                  :initarg height-extent)
   (width-extent  :type integer
                  :accessor stream-width-extent
                  :initform 0
                  :initarg width-extent)
   (border-width  :type integer
                  :accessor stream-border-width
                  :initform 0
                  :initarg border-width)))
```

The stream class defines a basic window object for displaying contact objects. The window may optionally have an extent that is larger than the size of the viewing area. In such cases, scroll bars are automatically added to the window.

Each contact object must be assigned a stream before it can be presented. All contacts within the same aggregate hierarchy must have the same stream.

The slots of the stream class are described as follows:

x-extent,y-extent

These slots define the placement of the window on the parent window. If no parent window is defined, then the values are relative to the user's screen. The corresponding x and y values inherited from the contact class define the position of the viewing region of the window.

height-extent,width-extent

These slots define the height and width of the extent of the window. These correspond to the height and width slots inherited from the contact class which refer to the size of the visible area within the extent. The height-extent and width-extent values must be greater than or equal to the height and width values. If the height-extent and width-extent are greater than the stream height and width, scroll bars will be included in the window.

2.11 Handling User Input

User input is handled using "mixin" classes and through the generation of special event objects.

sensitive

Class

```
(defclass sensitive ()
  (active      :type boolean
               :accessor sensitive-active
               :initform t
               :initarg active))
```

The sensitive class enables events to be generated by making contacts mouse sensitive. The sensitive class is not intended to be instantiated but instead is a "mixin" which is used as a superclass for a user defined contact class. When a contact is presented that is sensitive,

the underlying window level mechanisms are established to make the contact on the users screen mouse sensitive and consequently capable of generating events.

Whenever a user action occurs on a sensitive contact, an event object is generated. The contact associated with the event is sent the handle-event message with the associated event and stream objects. The application developer is able to customize the handling of events by specializing new handle-event methods for the event of interest. By using the class hierarchy, a wide range of flexibility and tailorability is possible.

event

Class

```
(defclass event () ())
```

The following is a partial list of pre-defined event classes exist and are generated as a result of the corresponding mouse operations:

```
(defclass button (event) ())
(defclass button-down (event) ())
(defclass button-up (event) ())
(defclass left-button (button) ())
(defclass left-button-up (left-button button-up) ())
(defclass left-button-down (left-button button-down) ())
(defclass middle-button (button))
(defclass middle-button-up (middle-button button-up) ())
(defclass middle-button-down (middle-button button-down) ())
(defclass right-button (button))
(defclass right-button-up (right-button (button-up)) ())
(defclass right-button-down (right-button button-down) ())
```

handle-event

Method

```
(defmethod handle-event ((self contact)(event event)(stream stream))
```

The handle-event method is sent to the contact that is responsible for the event. By defining new handle-event methods, the user is able to tailor the event handling mechanism. For example, by defining the method

```
(add-method #'handle-event
  (make-instance 'method
    :lambda-list '(contact event stream)
    :specializers (list (find-class 'my-contact)
                        (find-class 'left-button))
```

```

                                '(eql ,*current-stream*))
: function #'(lambda (contact event stream) ...)))

```

The left-button-up and left-button-down event will be handled but not middle or right button events. The expression (eql *current-stream*) is a CLOS construct that enables methods to be specialized on specific object instances, in this case is the current window.

moveable

Class

```
(defclass moveable (sensitive))
```

The moveable class is a mixin that enables objects to be moved on the user's screen. The mouse may then be moved to indicate the contact's desired new position. Clicking left on a moveable object unpresents the object from the screen and replaces the object with an outlined box enclosing the region occupied by the object. Clicking left again will re-present the object on the screen at the new position. Any constituent contacts will be reorganized with respect to the the new region.

2.12 Ongoing Expansion of Classes

The next two sections describe the classes defined in the prototype KUIE. Many more classes are under development. More sophisticated windows and menus are notable among the on going development: classes, methods, and protocols for windows with Symbolics style pointer/documentation capabilities, scrolling, more presentation-types, and display-lists, as well as accepting-values menus and greater support for menus with complex items.

2.13 Predefined Contacts

The following section contains the interface specifications to predefined contacts that have been created to permit limited user interfaces to be constructed without references to the supporting window system.

icon

Class

```

(defclass icon (contact)
  ((bitmap :accessor icon-bitmap
           :initform nil
           :initarg bitmap)))

```

The icon class contains a bitmap object that is directly displayable on the users screen.

pop-up-menu

Class

```
(defclass pop-up-menu (contact)
  ((selection-list :type list
                   :accessor pop-up-menu-selection-list
                   :initform nil
                   :initarg selection-list)
   (result :type t
            :accessor pop-up-menu-result
            :initform nil
            :initarg result)))
```

The pop-up-menu class defines objects to draw pop-up menus. The selection-list contains a property list of string/value pairs. The strings will be displayed on the menu for selection. The value corresponding to the string selection made is returned in the result slot.

box

Class

```
(defclass box (contact)
  (line-width :type integer
              :accessor box-line-width
              :initform 1
              :initarg line-width))
```

The box class defines an unfilled rectangle on the region defined by the contact. The outline of the box is drawn with width *line-width*.

filled-box

Class

```
(defclass filled-box (box))
```

The filled-box class defines a filled rectangle on the region defined by the contact.

label

Class

```
(defclass label (contact)
  (string :type string
           :accessor label-string
           :initform ""
           :initarg string))
  (font :type string
         :accessor label-font
         :initform nil
         :initarg font))
```

The label defines a string. Contacts may be labelled by associating the contact with a label contact.

line Class

```
(defclass line (contact)
  (point1      :type integer
                :accessor line-point1
                :initform 0
                :initarg point1)
  (point2      :type integer
                :accessor line-point2
                :initform 0
                :initarg point2)
  (width       :type integer
                :accessor line-width
                :initform 3
                :initarg width))
```

This class defines a line between point1 and point2.

arrow Class

```
(defclass arrow (line))
```

The arrow class is a specialization of a line that draws an arrow from point1 to point2.

circle Class

```
(defclass circle (contact)
  (radius      :type integer
                :accessor circle-radius
                :initform 10
                :initarg radius))
```

The circle class defines a circle that has a size defined by the radius slot.

filled-circle Class

```
(defclass filled-circle (circle))
```

A filled-circle is a specialization of a circle that is filled in.

DAG Class

```
(defclass DAG (contact))
```

The DAG class defines a directed acyclic graph. The parts of the DAG must be instances of the box class and arrow class.

text-window Class

```
(defclass text-window (contact))
```

The text-window defines a formatted text output window whose size is defined by the contact region.

lisp-listener Class

```
(defclass lisp-listener (contact))
```

The lisp-listener class defines a window running a lisp-listener whose size is defined by the contact region.

KBSA User Interface Environment (KUIE):

- a portable system for constructing user interfaces
- a collection of predefined user interface components and protocols

Why Have a KUIE

- a flexible interface to accommodate existing facets
- a common interface for developing future facet UI's

Design Goals

Object-Oriented Abstractions

CLOS Compatibility

Portability

Extensibility

Modularity

Implementation Platform Selection Criteria:

Portability

Flexibility

Extensibility

Compatibility

Modularity

Other Window Systems

The following tables show how the window systems we examined compare in the selected key areas.

Window System	Portable	Compatibility	Extensible	Public Domain Specifications
Common Windows	Yes	Yes	Yes	Yes
CUEForms	Yes	No	Yes	No
Symbolics Flavors	No	No	Yes	No
CLUE/CLX	Yes	Yes	Yes	Yes

Window System	Network Support	Ease of Use	Flexibility	Monolithic/ Multiple Screens
Common Windows	No	High	Medium	Monolithic
CUEForms	No	High	Low	Monolithic
Symbolics Flavors	No	High	High	Monolithic
CLUE/CLX	Yes	Low	High	Multiple Screens

The X-window system (X):

- C-based implementation
- Originated from the Athena Project at MIT.
- Easily accessible
- Used widely in the technical community
- adapted to several other languages including A (CLX)
- Low level graphical services

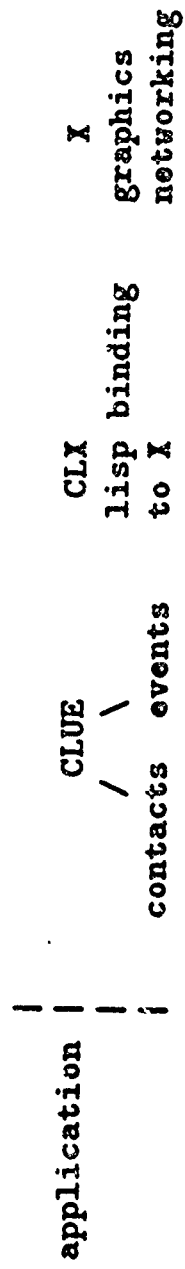
Common Lisp User Environment (CLUE):

- Developed by Texas Instruments
- Provides object-oriented (CLOSified) abstractions for CLX.
 - maps events to methods
 - hides details and bookkeeping of CLX
 - gives low level graphics capabilities
- No predefined contacts for UI components (e.g., menus, scrollbars, title bars, geometric shapes)

KUIE

- extends the notion of a contact
- makes complex constructions in CLUE primitives in KUIE
- makes extensive use of CLOS to increase flexibility and portability

The Big Picture



KUIE Components

Contacts

- predefined contact classes
- mixin classes to provide capabilities

Protocols

Events

The Purpose of the Contact Abstraction

- To simplify and raise the level of the dialog between the application and the user.
- To define a uniform framework within which many different types of user interface objects can be combined.

What a Contact Does:

- displays its contents,
- processes input events that are directed to it, and
- reports its results back to the application.

Categories of KUIE User Interface Builders:

1. The contact programmer
2. The application programmer

The Contact Class

```
(defclass contact ()
  ((parts      :type (listto contact)
               :accessor contact-parts
               :initform nil
               :initarg parts)
   (parent     :type (or null contact)
               :accessor contact-parent
               :initform nil
               :initarg parent)
   (x          :type integer
               :accessor contact-x
               :initform 0
               :initarg x)
   (y          :type integer
               :accessor contact-y
               :initform 0
               :initarg y)
   (width      :type integer
               :accessor contact-width
               :initarg width
               :initform 60)
   (height     :type integer
               :accessor contact-height
               :initarg height
               :initform 50)
   (state      :type (member '(nil organized exposed))
               :accessor contact-state
               :initform nil
               :initarg exposed)
   (stream     :type stream
               :accessor contact-stream
               :initform nil
               :initarg stream)))
```

The Stream Class

```
(defclass stream (contact))
  ((x-extent
    :type integer
    :accessor stream-x-extent
    :initform 0
    :initarg x-extent)
   (y-extent
    :type integer
    :accessor stream-y-extent
    :initform 0
    :initarg y-extent)
   (height-extent
    :type integer
    :accessor stream-height-extent
    :initform 0
    :initarg height-extent)
   (width-extent
    :type integer
    :accessor stream-width-extent
    :initform 0
    :initarg width-extent)
   (border-width
    :type integer
    :accessor stream-border-width
    :initform 0
    :initarg border-width)))
```

Manipulating Contacts

```
(make-instance contact-class-name initargs)
```

```
(defmethod organize ((c contact)))
```

```
(defmethod present ((c contact)))
```

```
(defmethod unpresent ((c contact)))
```

Predefined Contacts (continued)

```
(defclass box (contact)
```

```
  (line-width      :type integer
                   :accessor box-line-width
                   :initform 1
                   :initarg line-width))
```

```
(defclass filled-box (box))
```

```
(defclass label (contact)
```

```
  (string          :type string
                   :accessor label-string
                   :initform ""
                   :initarg string))
```

```
  (font            :type string
                   :accessor label-font
                   :initform nil
                   :initarg font))
```

Predefined Contacts (continued)

```
(defclass line (contact)
  (point1 :type integer
           :accessor line-point1
           :initform 0
           :initarg point1)
  (point2 :type integer
           :accessor line-point2
           :initform 0
           :initarg point2)
  (width :type integer
          :accessor line-width
          :initform 3
          :initarg width))

(defclass arrow (line))
```

Predefined Contacts (continued)

```
(defclass circle (contact)
  (radius :type integer
           :accessor circle-radius
           :initform 10
           :initarg radius))

(defclass filled-circle (circle))

(defclass DAG (contact))

(defclass text-window (contact))

(defclass lisp-listener (contact))
```


Predefined Contacts

```
(defclass icon (contact)
  ((bitmap :accessor icon-bitmap
           :initform nil
           :initarg bitmap)))

(defclass pop-up-menu (contact)
  ((selection-list :type list
                   :accessor pop-up-menu-selection-list
                   :initform nil
                   :initarg selection-list)
   (result :type t
            :accessor pop-up-menu-result
            :initform nil
            :initarg result)))
```

Handling User Input

User input is handled using “mixin” classes and through the generation of special event objects.

```
(defclass sensitive ()  
  (active      :type boolean  
               :accessor sensitive-active  
               :initform t  
               :initarg active))
```

```
(defclass moveable (sensitive))
```

```
(defclass event () ())
```

Handling User Input (continued)

The following is a partial list of pre-defined event classes exist and are generated as a result of the corresponding mouse operations:

```
(defclass button (event) ())
(defclass button-down (event) ())
(defclass button-up (event) ())
(defclass left-button (button) ())
(defclass left-button-up (left-button button-up) ())
(defclass left-button-down (left-button button-down) ())
(defclass middle-button (button))
(defclass middle-button-up (middle-button button-up) ())
(defclass middle-button-down (middle-button button-down) ())
(defclass right-button (button))
(defclass right-button-up (right-button (button-up)) ())
(defclass right-button-down (right-button button-down) ())
```

Handling User Input (continued)

```
(defmethod handle-event ((self contact)(event event)(stream stream))  
  (add-method #'handle-event  
    (make-instance 'method  
      :lambda-list '(contact event stream)  
      :specializers (list (find-class 'my-contact)  
                          (find-class 'left-button)  
                          '(eql ,*current-stream*))  
      :function #'(lambda (contact event stream) ...))))
```

Many more Classes Are under Development

- More sophisticated menus
(e.g., accepting-values menus, more support for menus with complex items)
- Symbolics style pointer/documentation capabilities
- Scrolling
- More presentation-types
- Display-lists

Reusing Software Developments

Allen Goldberg *
Kestrel Institute
3260 Hillview Ave.
Palo Alto, CA 94304

August 3, 1989

1 Introduction

We are addressing the issue of *reusability* in the context in which software design is achieved by a transformational development of a formal specification of the problem into an efficient implementation. This paper explores how a specification derived as a modification of an existing design can be realized by *replaying* the transformational derivation of the original and modifying it as required by changes made to the specification.

The basic underlying assumption of this work is that design activities can be formalized. Furthermore it can be done so that individual design decisions can be expressed as *transformations* whose application derive an implementation from a specification. This assumption is at the foundation of the KBSA model. The model does not assume that transformations can always be applied automatically. The goal is that *key* implementation decisions are to be made by the user and the more commonplace by the automated assistant. Indeed as the technology matures, more of the decision making will be automated.

In the KBSA model, the software maintenance problem is finessed by replay. That is, maintenance is performed by modifying the specification and then re-implementing the modified specification. The problem is to perform this re-implementation with little additional work. This means automating as many of the manually-applied steps of the original development as is possible. Replay is not restricted to the maintenance

*Supported by RADC contract F30602-88-C-0127, and NSF Grant DMC-8617759. Views and conclusions contained within this report are the author's and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

phase. It also supports evolutionary development in which a prototype is developed and then incremental enhancements are made and implemented with the aid of the replay system.

Although this paper is about replay, the efficacy of the transformational approach to non-trivial design problems is itself a key issue. Without having a transformational system with some power and generality at hand, investigations into replay are severely restricted.

Unlike design problems in other domains, the transformational approach applied to software development has been extensively studied [Dij76, Mee87, PS83, Pep83]. Powerful, generic techniques such as data refinement, finite differencing, loop combining, inversion, algorithm design, etc. have been developed. The field also benefits from related work in compiler optimization, software specification, theorem proving, and programming language theory and practice.

Our previous work on the Performance Assistant as well as other work ongoing at Kestrel has led to the development of a transformational system, called *KIDS* which provides a basis from which we can approach the problem of *design reusability*. Using the system we have been able to carry out derivations that carry non-trivial examples through many semantic levels and apply a wide range of design and optimization techniques. For example in one derivation we derive from a high-level specification of a topological sort a LISP implementation which is as efficient as any hand-coded version [L B88]. The derivation is over 40 steps long where a step involves such diverse activity as inverting maps, computing containment relations among set-theoretic data structures, simplifying expressions, combining loops, and selecting data structures. Experience with this system has been the basis of the theoretical and implementation work on replay.

In this paper we report progress in the following areas:

- An approach to representing design history was developed. The approach is to structure the derivation system using the notion of tactics, and record derivation histories as an execution trace of the application of tactics. One key idea is that tactics are compositional: higher level tactics are constructed from more rudimentary using defined control primitives. This is similar to the approach used in LCF[GMW79, Mil85] and NuPRL[BC85, Con86].
- An approach to the *correspondence problem* is described [Mos85b, Mos85a]. The correspondence problem addresses how during replay a correspondence between components of the old and new design (program) is established. Our approach uses a combination of name association, structural properties, associating components to one another by descriptions of objects defined in the transformations themselves.
- An implementation of a rudimentary derivation management and replay mechanism for *KIDS* is described. Using the system we were able to perform a

number of interesting rederivations.

2 Approach

2.1 Recording Derivations

2.1.1 Representation of the Design

The first important technical problem faced in this work is the representation of the design history to be used as the basis for the replay mechanism. The process of development has been described as starting with a specification and applying a linear sequence transformations to yield an implementation. However just recording the linear sequence of development steps is inadequate. An analogy with mathematical proofs is revealing. Formally a mathematical proof is also just a linear sequence of formulas obtained by applying inference rules. This is an appropriate view for providing a simple meta-theory (e.g. to prove the soundness of the system), but for little else. Just as a mathematical proof has structure (lemmas, case analysis, formation of induction hypothesis, reformulation, etc.) and is constructed and explained in terms of that structure, a similar, but formal, structure must be devised for transformational developments.

What kinds of structures do we observe in software developments that must be formalized? We survey a few here to motivate our solution.

- One common structure is the virtual machine model. Here the specification is expressed in terms of an abstract language and then mapped in phases to successively lower-level virtual machine or language levels. Compilers are often constructed along this paradigm. The source language is at this highest abstract level. In the first phase this may be mapped to retargetable intermediate code, at a lower abstract level, and then in a second phase to assembly language. Traditional compilers rarely involve more than two phases. Boyle's [BM84] Lisp to Fortran transformation system goes through 7 phases, each mapping to a different virtual machine level. The REFINE compiler which also is based on the transformational model operates in 5 phases.
- A second common structure is that of stepwise problem decomposition. A problem is decomposed into components and the implementation of each of the components proceeds independently.
- A third is the exhaustive application of a set rewriting rules. This is typical of routine simplification steps or steps that rewrite the program into a normal form. It is common to apply this strategy in conjunction with the first. Each

phase mapping between language levels is the exhaustive application of a set of rewriting rules.

- A fourth structure is that of case analysis. This is of course used to express strategies which are conditioned on the form of the specification or other information about the specification supplied to the system.

This is a representative but not exhaustive list of high-level development steps found in systems that formally map specifications into implementations. Observe that these high-level development steps are compositions of more elemental steps and that they are expressible in terms of common control structures found in ordinary programming such as conditional, sequential composition, parallel composition and iteration. For example, a problem decomposition step is the sequential composition of a step which divides the problem in sub-problems, and a step consisting of the parallel composition of steps that solve the subproblems. Parallel decomposition does not impose a temporal order on development steps when no logical dependency exists.

This suggests a straight-forward approach to the problem of structuring derivations. The development system is constructed from a set of primitive operators, using composition mechanisms such as the ones described above. The resulting composite operators are called *tactics*.

This is the approach taken in LCF and NuPrl, systems aimed at the construction of mathematical proofs, not programs. It is also the approach taken by Wile [Wil83]. The recorded derivation is simply a trace of the execution of the tactics. This is a direct implementation of the popular notion of process programming [Ost87].

A different approach can be based on AI-style planning theory [Fic85]. Here the description of the development step is given in terms of a *goal*—a declaratively stated postcondition that describes properties of the intended result of the step. For example, a step which transforms code into a normal form would be expressed by a declarative description of the form to be achieved. In addition to a goal structure, there are *methods*, which are operations that may be used to achieve a goal. It becomes the task of the system to synthesize a meta-program of methods whose result achieves the goal. The planning approach is a weak method because synthesizing plans is a difficult problem, and because declarative specification of post-conditions is often unwieldy.

In addition to making the intuitive structure of the development explicit, the structure must support manual as well as automatic development steps and do so in a way that supports gradual automation. The tactic approach does this. A step within a tactic may be to query the user for a tactic to apply. The approach also encourages a bottom-up methodology for gradual automation in which the fundamental transformational primitives are defined, and higher-level tactics constructed from them.

2.1.2 An Elementary Tactic Language

This section describes an elementary tactic language sufficient to illustrate the interaction of the replay mechanism and the tactic language. A full tactic language is under development.

The tactic language is a control language. The computation responsible for transforming programs lies within *primitive tactics* written in some other language, which in our case is REFINE. Primitive tactics are represented by REFINE procedures which are called by the tactic language interpreter. The form of a primitive tactic is:

$$\textit{procedure-name} (\textit{parameter-list}) \text{ [returns } \textit{identifier-list} \text{]}$$

The *identifier-list*, and *parameter-list* are each lists, separated by commas, of an identifier followed by a colon followed by a type expression. The procedure is called with the actual parameters specified in the parameter list. The procedure transforms the program which is stored in a global knowledge base as a side-effect. It returns a list of values which are then bound to the variables appearing in identifier list following the keyword *returns*. These variables are called *tactic variables*. It also returns an indication of whether the tactic succeeded or failed.

The tactic variables appearing in the identifier list must be declared in a containing tactic called an *abstraction tactic*. An abstraction tactic allows the construction of a tactic with a name, formal parameters, local variables and a body. These tactics have the form:

$$\begin{aligned} &\textit{tactic-name} (\textit{parameter-list}) = \\ &\text{let } \textit{identifier-list} \text{ in } \textit{tactic} \text{ returns } \textit{identifier-list} \end{aligned}$$

An abstraction tactic is invoked the same way as a primitive tactic. The formal parameters are bound to the actual values, the local tactic variables are allocated and the tactic following the keyword *in* is executed. The tactic fails if the tactic following the keyword *in* fails.

Primitive tactics are composed using control primitives. The most elementary is *sequential composition*. This is simply denoted as $\textit{tactic}_1; \textit{tactic}_2; \dots; \textit{tactic}_n$. It represents the tactic which executes each tactic sequentially. This tactic fails if any of its sub-tactics fail.

The parallel execution of tactics is denoted $\textit{tactic}_1 || \textit{tactic}_2 || \dots || \textit{tactic}_n$. It represents the tactic which executes each tactic once in any order or conceptually at least, in parallel. Parallel composition is used when there is no logical dependence among the

tactics, and so no temporal order on their execution should be specified. This tactic fails if any of its sub-tactics fail.

The conditional tactic has the form

if condition then tactic elseif condition ... else tactic

The condition must be a function call which returns a boolean value. The tactic fails if the sub-tactic that executes fails.

The syntax *tactic₁?tactic₂* denotes a tactic which executes *tactic₁*; if this fails it executes *tactic₂*. This is a useful exception handling mechanism.

Finally there is a repetition tactic.

while condition do tactic

Example. This is a tactic that will exhaustively find and combine all pairs of loops that may be merged within a program part *p*, which is passed as a parameter.

```
Combine-Loops(p: program-part) =
    let Loop-1 : program-part,
        Loop-2 : program-part,
        Combined-Loop : program-part
    in
    while exists-combinable-loops(p)
        (Find-Combinable-Loops(p) returns Loop1, Loop-2;
         Merge-Loops(Loop-1, Loop-2) returns Combined-Loop;
         Simplify(Combined-Loop))
```

A tactic such as *Combine-Loops* may be incorporated into another tactic or may be invoked directly by the user.

We can now define a *derivation history* as a trace of the tactics invoked, either manually or as part of the execution of some other tactic, together with the values of the actual parameters passed to them.

2.2 The Correspondence Problem

In this section we turn to the problem of replaying derivation histories on a modified specification. The replay mechanism will attempt to apply the same tactics to the

altered specification, except for conditional or repetitive tactics. If a conditional tactic is invoked the condition will be evaluated and the then or else part will be executed depending on the outcome of the test, not on which part was executed in the original derivation. Similarly the number of iterations of a loop is determined by evaluation of the condition on the loop.

The values passed as parameters to the tactics must be determined. Intuitively a correspondence must be established between components of each of the designs playing the same role. This is called the correspondence problem. Our method for establishing a correspondence is heuristic. It relies on three mechanisms:

Name Correspondence. The definition of the same identifier name within the program establishes a correspondence.

Structure Correspondence. Code appearing in the same position within the abstract syntax tree correspond.

Parameter Correspondence. The execution of a tactic may cause a tactic variable to be bound to some code. Code bound to the same variable corresponds.

Parameter correspondence is a powerful notion, because it captures a semantic correspondence. Often when a tactic is applied it creates a code segment. Suppose that the tactic is replayed a new code segment is created. With respect to the semantics encoded in the tactic both code segments play the same role and a correspondence is established. For example, a divide-and-conquer algorithm design tactic will generate identifiable code components such as code for the base case; code for dividing the problem into subproblems, etc. Parameter correspondence would identify, say, code for the base case in each derivation as corresponding.

The replay mechanism maintains a relation called the *correspondence relation*. It is a binary relation between code segments of the program as it is transformed from a specification into an implementation. When replay begins an initial correspondence relation is computed between the two specifications. This initial relation only uses the name correspondence heuristic. It identifies as corresponding code used in the definition of variables with the same name. When a tactic is applied the correspondence relation is augmented by identifying as corresponding the code segments bound to the same tactic variable. This implements the parameter correspondence heuristic. Never is a correspondence removed from the correspondence relation. However as the derivation proceeds and code segments are replaced, correspondences about the replaced segments will never be used.

If a tactic introduces a new variable into the program, when the tactic is reapplied a correspondence between variable names are made, regardless of whether they are given the same name (often these names are computer-generated).

The correspondence relation is used in replay as follows. When a tactic is invoked during replay its actual parameters must be determined. It could be that the actual

parameter is a tactic variable with a value that was assigned by the returns clause of a previously executed tactic. In this case the value is used. If the parameter is not of a type representing a code segment (for example it may be an integer representing a resource bound on an inference step), then the value from the initial derivation is used. Otherwise the correspondence relation is used. If the actual parameter of the original derivation is paired with a corresponding code segment by the correspondence relation, then the corresponding code segment is used in replay. Otherwise starting with the actual parameter in the original specification traverse up the abstract syntax tree until a code segment within the correspondence relation is found or the root is reached. Starting with the corresponding code in the modified development the corresponding path down the tree is traced to the corresponding actual parameter. By corresponding path we mean that we follow in reverse direction the labeled path that took us up the tree in the original program. An example is given in figure 1. Suppose we are trying to find a correspondence for program segment *B* in the original program. There is no established correspondence for *B* in the correspondence relation. We traverse up the tree to *A* which has a correspondence. (*A* is an if expression and the edge we just followed came from the abstract syntax tree representing the then part of the expression.) *A'* is the corresponding program part in the modified program. We reach *B'* by following the corresponding labeled path. If the paths do not correspond then replay fails on that step and manual intervention is necessary. This heuristic of using path correspondence implements the structure heuristic. It recognizes that designed artifacts have component structure and substructure. In other words, components are recursively divided into subcomponents, and this parts hierarchy can be used to find corresponding components.

2.3 An Initial Implementation

In our current implementation, we have not implemented a tactic language so that each tactic is primitive. This means that parameter correspondence cannot be used, since derivation structuring information is not present. However the implementation follows the described mechanism in all other respects. We have successfully used the replay mechanism on a number of examples. The results are described in the next section.

Experience using our system has suggested many features that would make a replay system user-friendly.

Viewing. Currently the system displays a window showing all the derivation steps. The user can mouse on any step and display the program as it appears prior to the execution of the step. The user may initiate a new derivation path from that step and the resulting tree of derivations is displayed. A desired feature is the ability to have more selected views, especially when the tactic language is implemented. For example we may wish to see an "executive" view that only shows the top-level development steps. A user may wish to explode a derivation

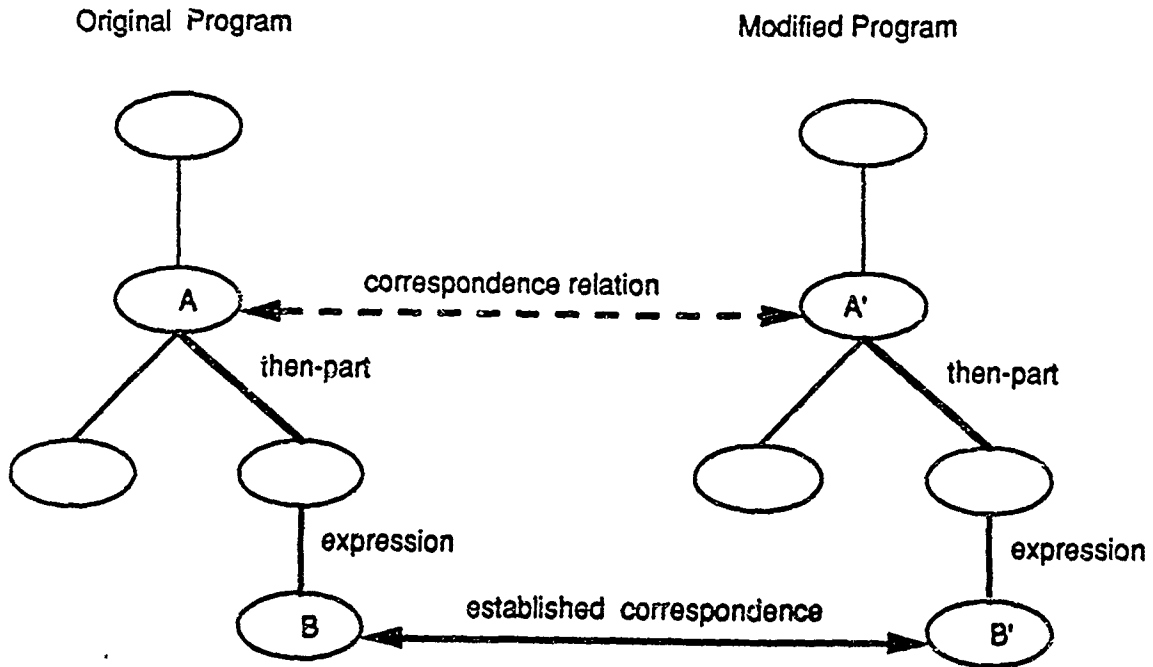


Figure 1: Establishing a Correspondence

step to see its sub-tactics. A user may wish to only see tactics that succeeded; or tactics relevant to a specified part of the program.

Editing. Prior to replay the user may wish to make edit changes to the derivation, anticipating where replay may fail. For example a sequence of transformations that applied to some program part may be abstracted and reapplied to a newly introduced object. Or the user may wish to edit the derivation and reapply it to the same specification to quickly generate a new implementation.

Debugging. Replay is the reexecution of a "process program." Thus we can imagine a set of debugging tools that are entered at breakpoints or when the replay mechanism fails. The debugger will allow tactic variables to be examined or changed, as well as give access to the top-level environment.

3 Results

We have used the replay mechanism on a simple example of computing basic statistics such as the mean, variance, and frequency. Figure 2 shows the initial specification. An explanation of the operators appearing in the program can be found in [L B88]. Figure 3 shows the development just prior to data structure selection. Each of the high-level operators such as *reduce* have been refined into loops, and these loops have

Rational Interactive Development System											
Configure	Focus	Replay?	Replay	FD	Fold	Fuse	Parallelize	Rename	Stapify	Tactics	Unfold
Abstract	Analyze	Compile	Condition								
<pre> function STATISTICS (S: seq(integer)) range(S) = (0 .. 99) return(STAT): tuple(integer, real, real, seq(integer, integer)) true) = leto (SUM = reduce('0, S), N = size(S), FREQ = (lamba (X X in (0 .. 99)) return(RESULT true) stat(filter ((lamba (Y true) return(RESULT true) Y = X), S))), SUM-OF-SQUARES = reduce ('0, image((lamba (X true) return(RESULT true) X = X), S))) 04, SUM / N, (N = SUM-OF-SQUARES - SUM * SUM) / (N * N), FREQ) </pre>						<pre> (Derivation History) Initial state @focus initialize (STATISTICS) </pre>					

Figure 2: The Initial Specification

been fused together so that a single pass is made over the input, and so no intermediate expressions are required. The efficiency of the computation of the map *freq* has been speeded up asymptotically by iterator inversion. Data structure selection will choose an array implementation for *freq* and the input sequence.

Next we modify the program by changing the definition of *freq* to yield histogram data, in which ranges of data values are counted, and by the inclusion of the computation of the maximum value. Figure 4 shows the modified program. Figure 5 shows the result of replay. Even though the definition of *freq* was changed the original development was successfully applied. The other development steps, that were independent of the change, were also replayed. Finally Figure 6, show additional development steps needed to incorporate the computation of the maximum value into the main loop of the program.

A second, more involved example is based on a scheduling problem in which precedence constrained jobs are scheduled on a uni-processor system (only one job may be scheduled at a time). In [L B88] we outlined this complex derivation which requires over 40 steps. We modified the specification to solve the problem of multi-processing scheduling and was able to successfully replay all of the steps of the derivation.

Rational Interactive Development System											
Configure	Focus	Redisplay?	Replay	FD	Fold	Fuzz	Parallelize	Remove	Simplify	Tactics	Unfold
Abstract	Analyze	Compile	Condition								
<pre> function STATISTICS (S: seq(Integer)) range(S) = (0 .. 99) return(STATS tuple(Integer, real, real, seq(Integer, Integer)) true) = for= (I-V-2, SUM = 0, N = 0, FREQ = (1 to 100 (X X in (0 .. 99) return(RESULT true) 0), SUM-OF-SQUARES = 0) I-V-2 over S, SUM <- SUM + I-V-2, N <- 1 + N, FREQ(I-V-2) <- 1 + FREQ(I-V-2), SUM-OF-SQUARES <- SUM-OF-SQUARES + I-V-2 * I-V-2 return(OK, SUM / N, (N * SUM-OF-SQUARES - SUM * SUM) / (N * N), FREQ) </pre>						<pre> (Deviation History) Initial state Focus Initialize (STATISTICS) Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Simplify: context-dependent, forward 0, backward 4 --- Simplify: context-dependent, forward 2, backward 3 --- Simplify: context-independent-fast --- Fuse Horizontally --- Fuse Horizontally --- Fuse Horizontally --- Simplify: context-independent-fast --- </pre>					

Figure 3: The Implementation of the Original Specification

Rational Interactive Development System											
Configure	Focus	Redisplay?	Replay	FD	Fold	Fuzz	Parallelize	Remove	Simplify	Tactics	Unfold
Abstract	Analyze	Compile	Condition								
<pre> function NEW-STATISTICS (S: seq(Integer)) range(S) = (0 .. 99) return(STATS tuple(Integer, real, real, seq(Integer, Integer)) true) = let= (SUM = reduce('0, 0), N = size(S), FREQ = (1 to 100 (X X in (0 .. 99) return(RESULT true) size(Filter ((1 to 100 (Y true) return(RESULT true) Y div 10 = X), 0))), SUM-OF-SQUARES = reduce ('0, 1 to 100 ((1 to 100 (X true) return(RESULT true) X = X), 0))), AVERAGE = reduce('0, 0)) OK, SUM / N, (N * SUM-OF-SQUARES - SUM * SUM) / (N * N), FREQ, AVERAGE) </pre>						<pre> (Deviation History) Initial state ... ○ Focus Initialize (STATISTICS) Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Simplify: context-dependent, forward 0, backward 4 --- Simplify: context-dependent, forward 2, backward 3 --- Simplify: context-independent-fast --- Fuse Horizontally --- Fuse Horizontally --- Simplify: context-independent-fast --- ○ Focus Initialize (NEW-STATISTICS) </pre>					

Figure 4: The Modified Specification

Kestrel Interactive Development System											
Configure	Focus	Redisplay?	Replay	FD	Fold	Fuse	Parallelize	Renew	Simplify	Tactics	Unfold
Abstract	Analyze	Compile	Condition								
<pre> function NEW-STATISTICS (S: seq(integer)) range(S) = (0 .. 99) return(STATS tuple(integer, real, real, seq(integer, integer)) true) = let (P=0 for (I-V=0, SUM = 0, N = 0, FREQ = (lambda (X X in (0 .. 9)) return(RESULT true) 0), X, SUM-OF-SQUARES = 0) I-V=0 over S, SUM = SUM + I-V=0, N = N + 1, X = I-V=0 div 10, FREQ(X) = 1 + FREQ(X), SUM-OF-SQUARES = SUM-OF-SQUARES + I-V=0 * I-V=0 return (SUM, N, FREQ, SUM-OF-SQUARES), P=0, P=0.1, SUM = P=0.1, N = P=0.2, P=0.3, P=0.2, FREQ = P=0.1, SUM-OF-SQUARES = P=0.2, MAXIMUM = reduce('MAX, 0)) OK, SUM / N, (N = SUM-OF-SQUARES - SUM * SUM) </pre>						<pre> (Derivation History) Initial state ... O Focus Initialize (STATISTICS) Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Simplify: context-dependent, forward 0, backward 4 --- Simplify: context-dependent, forward 2, backward 5 --- Simplify: context-independent-fast --- Fuse Horizontally --- Fuse Horizontally --- Simplify: context-independent-fast --- O Focus Initialize (NEW-STATISTICS) Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Simplify: context-dependent, forward 0, backward 4 --- Simplify: context-dependent, forward 2, backward 5 --- Simplify: context-independent-fast --- Fuse Horizontally --- Fuse Horizontally --- O Fuse Horizontally --- </pre>					

Figure 5: The Modified Specification after Replay

Kestrel Interactive Development System											
Configure	Focus	Redisplay?	Replay	FD	Fold	Fuse	Parallelize	Renew	Simplify	Tactics	Unfold
Abstract	Analyze	Compile	Condition								
<pre> function NEW-STATISTICS (S: seq(integer)) range(S) = (0 .. 99) return(STATS tuple(integer, real, real, seq(integer, integer)) true) = for (I-V=0, SUM = 0, N = 0, FREQ = (lambda (X X in (0 .. 9)) return(RESULT true) 0), X, SUM-OF-SQUARES = 0, ACCUM=1 = first(S)) I-V=0 over S, SUM = SUM + I-V=0, N = N + 1, X = I-V=0 div 10, FREQ(X) = 1 + FREQ(X), SUM-OF-SQUARES = SUM-OF-SQUARES + I-V=0 * I-V=0, ACCUM=1 = SUM(ACCUM=1, I-V=0) return OK, SUM / N, (N = SUM-OF-SQUARES - SUM * SUM) / (N * N), FREQ, ACCUM=1) </pre>						<pre> (Derivation History) Initial state ... O Focus Initialize (STATISTICS) Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Simplify: context-dependent, forward 0, backward 4 --- Simplify: context-dependent, forward 2, backward 5 --- Simplify: context-independent-fast --- Fuse Horizontally --- Fuse Horizontally --- Fuse Horizontally --- Simplify: context-independent-fast --- O Focus Initialize (NEW-STATISTICS) Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Tactics: Compute by Iteration --- Simplify: context-dependent, forward 0, backward 4 --- Simplify: context-dependent, forward 2, backward 5 --- Simplify: context-independent-fast --- Fuse Horizontally --- Fuse Horizontally --- Fuse Horizontally --- Unfold using rewrite rule REDUCTION-OF-TO-FOR-LOOP reduce('MAX, 0) Fuse Horizontally --- Simplify: context-independent-fast --- </pre>					

Figure 6: The Final Implementation of the Modified Program

4 Related Work

The literature on software reuse is very extensive, but most of it deals with component reuse, i.e. the reuse of subroutines. A recent collection of papers, edited by Biggerstaff and Perlis [Big89a, Big89b] emphasizes *generative systems*, such as ours which offer design reuse and the promise greater productivity improvements in the long run. Many of the existing transformational systems are described in the collection. This is an excellent survey of the field. See also [Mos87] for a perspective on the reuse of design plans.

There is also an extensive Artificial Intelligence literature on analogy and machine learning. Representative of work of this kind is [Car86].

Because of the existence of a large existing base of software there is work on recovery of design knowledge from code. In [Big88] he emphasizes the existence of semantic clues in documentation and variable names that will aid in design recovery. We have adopted in our use of name correspondence this idea. Examples of work on design recovery can be found in [Wil87, SJ85, LS86].

Our tactic language is similar to [Wil83, Mil85]. A richer more theoretical approach is being pursued by [Sin85, HdGJ*86] using the Deva language.

Closer to the spirit of the work reported here is work done at Rutgers University. Their work is couched in a transformational framework. Two domains are addressed: circuit designs [MB87] and heuristic search algorithms [MF89b, MF89a].

5 Conclusions

We see our initial results as rather impressive. A simple mechanism was quite effective on the two problems sets we gave it. We have not yet stressed the system by giving it a program with major modifications and to see how the replay mechanism fails. However we will be quite satisfied with a system that does very well on closely related problems. Interactive development will not progress very far if the debugging loop, in which a user is making many small changes to their specification and recompiling is time consuming. Our goal is to make the debugging loop fast.

Acknowledgements. I would like to thank Greg Fisher and Tom Pressburger for useful discussions and for, along with Limei Gilham, implementing the replay system.

References

- [BC85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113-136, January 1985.
- [Big88] Ted J. Biggerstaff. *Design Recovery for Maintenance and Reuse*. Technical Report STP-378-88, MCC Corporation, November 1988.
- [Big89a] Ted J. Biggerstaff, editor. *Software Reusability, Vol. 1: Concepts and Models*. ACM Press, New York, 1989.
- [Big89b] Ted J. Biggerstaff, editor. *Software Reusability, Vol. 2: Applications and Experience*. ACM Press, New York, 1989.
- [BM84] J. M. Boyle and M. N. Muralidharan. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, SE-10(5):574-588, September 1984.
- [Car86] J. Carbonell. Derivational analogy: a theory of reconstructive problem solving and expertise acquisition. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 371-392, Morgan Kaufmann, Los Altos, CA., 1986.
- [Con86] Robert L. Constable. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, New York, 1986.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [Fic85] Stephen F. Fickas. Automating the transformational development of software. *IEEE Transactions on Software Engineering*, SE-11(11):1268-1278, November 1985.
- [GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Springer-Verlag, Berlin, 1979. Lecture Notes in Computer Science, Vol. 78.
- [HdGJ*86] Fatima Ali Hussain, Philippe de Groote, René Jacquard, Stefan Jähnichen, Thanh Tung Nguyen, Michel Sintzoff, and Matthias Weber. *Esprit Project ToolUse - Requirements and Feasibility Studies for a Development Language*. Technical Report GMD 214, Gesellschaft für Mathematik und Datenverarbeitung mbH, July 1986.
- [L B88] L. Blaine, A. Goldberg, T. Pressburger, X. Qian, T. Roberts, and S. Westfold. *Progress on the KBSA Performance Estimation Assistant*. Technical Report KES.U.88.11, Kestrel Institute, May 1988.

- [LS86] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41-49, May 1986.
- [MB87] J. Mostow and M. Barley. *Automated Reuse of Design Plans*. Technical Report ML-TR-14, Rutgers University, May 1987.
- [Mee87] L.G.L.T. Meertens. *Program Specification and Transformation (Proceedings of the IFIP TC2/WG 2.1 Working Conference)*. North-Holland, Amsterdam, 1987.
- [MF89a] Jack Mostow and Greg Fisher. Replaying transformational derivations of heuristic search algorithms in DIOGENES. In *Proceedings of the DARPA Case-Based Reasoning Workshop*, Pensicola, FL, May 1989. Available as Rutgers AI/Design Project Working Paper Number 113-3.
- [MF89b] Jack Mostow and Greg Fisher. Replaying transformational derivations of heuristic search algorithms in DIOGENES. In *Proceedings of the AAAI 1989 Spring Symposium on AI and Software Engineering*, Palo Alto, CA, March 1989. Available as Rutgers AI/Design Project Working Paper Number 113-1.
- [Mil85] R. Milner. The use of machines to assist in rigorous proof. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 77-87, Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [Mos85a] J. Mostow. Some requirements for effective replay of derivations. In *Proceedings of the Third International Machine Learning Workshop*, pages 129-132, Rutgers University, Skytop, PA, June 1985.
- [Mos85b] J. Mostow. Toward better models of the design process. *AI Magazine*, 6(1):44-57, Spring 1985.
- [Mos87] J. Mostow. *Design by Derivational Analogy: Issues in the Automated Replay of Design Plans*. Technical Report ML-TR-22, Rutgers University, March 1987.
- [Ost87] Leon Osterweil. Software processes are software too. In *9th International Conference on Software Engineering*, pages 2-13, Monterey, CA, March 30-April 2, 1987.
- [Pep83] P. Pepper, editor. *Program Transformation and Programming Environments*. Springer-Verlag, New York, 1983.
- [PS83] Helmut Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199-236, September 1983.
- [Sin85] M. Sintzoff. *Desiderata for a Design Calculus*. Technical Report, RM 85-13, Université Catholique de Louvain, June 1985.

- [SJ85] E. Soloway and W. L. Johnson. PROUST: knowledge-based program understanding. *IEEE Transactions on Software Engineering*, SE-11(3):267-275, March 1985.
- [Wil83] David S. Wile. Program developments: formal explanations of implementations. *Communications of the ACM*, 26(11):902-911, November 1983.
- [Wil87] Linda M. Wills. *Automated Program Recognition*. Technical Report MIT-AI-904, MIT AI Laboratory, February 1987.

Building Evolution Transformation Libraries

Lewis Johnson and Martin Feather
USC / Information Sciences Institute

Lewis Johnson is a Research Assistant Professor of Computer Science at USC and project leader of the ARIES project at ISI. He has a Ph.D. in Computer Science from Yale University.

Martin Feather is a Research Scientist at ISI. He has a Ph.D. in Computer Science from the University of Edinburgh.

Abstract

A major focus of the Knowledge-Based Specification Assistant effort was formalized evolution of specifications. Evolution was accomplished by means of evolution transformations, which are meaning-changing transformations applied to formal specifications. A sizable library of evolution transformations was developed for our specification language, Gist. This paper assesses the results of our previous work on evolution transformations. It then describes our current efforts to build a versatile, usable evolution transformation library. We have identified important dimensions along which to describe transformation functionality, so that one can assess the coverage of a library along each dimension. Potential applicability of the formal evolution paradigm to other environments will be assessed, in terms of the ability of other environments to explicitly represent information along these dimensions.

1. Introduction

One major advance embodied in the KBSA vision as it is currently conceived [3] as compared to the vision articulated in 1983 [6] is in the treatment of specification evolution as a formal process. The original KBSA report anticipated that specifications would evolve, but did not describe the mechanism for such evolution. Research in the Knowledge-Based Specification Assistant [9] attempted to make specification evolution formal and machine-mediated. To support this, we constructed a library of transformations which could be employed to carry out the specification development process. This library contains primarily so-called "evolution transformations", i.e., transformations whose purpose is to elaborate and change specifications in specific ways. In addition, it contains meaning preserving transformations, used to paraphrase, simplify, and reorganize specifications. Thus they differ from the conventional "correctness-preserving" transformations, which are applied to derive efficient implementations from specifications. Our thesis was it was possible to develop specifications incrementally using transformations drawn from our library, resulting in a formal, understandable, and replayable specification process.

Our exploration of the space of evolution transformations was example-driven. We concentrated on two problems, a patient monitoring system and an air traffic control system, and

worked out development scenarios by hand to discover what transformations were necessary. We then implemented general-purpose versions of those transformations, which could be applied to achieve those developments mechanically. The result of this exploration was a sizable library containing around 100 transformations, of a wide variety of types. This library is significantly more extensive than similar libraries developed by Balzer [1] and Fickas [5]. Although other researchers have studied evolution steps similar to those captured by our transformations [14, 12], they have not developed transformations to enact these steps. Because of the relatively advanced state of our work, we are now at a point where we can consider the challenge of building a transformation library that is extensive enough and powerful enough to apply to a wide range of specifications.

In order to build such a library, a number of concerns must be addressed. First, there will be many more transformations in our new system, hence we must strive to make it easy to write them (our experience in the old system was that transformations were fairly complex to write, and hence difficult to understand and debug). To do this, we need new representation in which the effects of the transformations can be expressed more directly, and with a minimum of unnecessary duplication. Second, we need to understand better how transformations are structured, so as to achieve better reuse of components and increase overall uniformity. Third, we needed a better characterization of the effects of transformations, both so that potential users will be able to find the right transformation, and so that we can assess the coverage of our library. The results of these analyses will be presented in this paper. We believe that this analysis will help readers to better understand the key concepts of evolution transformations, and see how these might apply to other programming or specification environments.

2. Transformations in KBSpecA

We will first summarize the status of the evolution transformation library at the close of the Specification Assistant contract. These results have also been reported elsewhere [9].

In the Specification Assistant, evolution transformations were represented in such a way as to support interactive application and replay. As described in [10], they were represented declaratively using a combination of informal and formal descriptions. The informal descriptions are outlined in this other paper; here we will take a closer look at the formal aspects.

Each transformation applied to a set of *inputs*, each of a particular type. Possible types were categories of syntactic constructs, such as expression or predicate, and types defined in terms of these, namely disjunctions, specializations, or sets of instances, e.g., a set of relation declarations. The ability to define types in this manner came from use of our AP5 language [2].

Transformations typically had a set of *applicability conditions*, preconditions which had to be satisfied for the transformation to be applicable. We endeavored to make each transformation's set of applicability conditions complete, so that if its members were all satisfied, it was guaranteed that the transformation would successfully apply. By using a set of applicability conditions, we sought to make a distinction between possibly several reasons why a transformation might not apply. This was used by an interactive help facility to notify the user which precondition failed, and which input arguments were at fault, so that they could be adjusted appropriately. It was also employed in a transformation replay facility, to determine when previously applied transformations were no longer applicable.

Finally, each transformation had a *method*. The method was an imperative program written in either our special-purpose transformation definition language Paddle [16], or Lisp. Once a transformation's preconditions were satisfied, the method would be executed, and the result would be displayed.

We divided our set of transformations into the following categories, to facilitate users' selection of the the appropriate transformation through menus:

- Adding commands, which add a new construct into the specification,
- Replacement commands, which replace a construct with a new one,
- Reorganizing commands, which restructure the specification without changing its meaning,
- Behavior changing commands, which modify the behavior described by existing specification components,
- Data flow modifying commands, which change the flow of data through the system, while retaining the behavior of the system as a whole,
- Terminology elaboration commands, which elaborate some part of the specification terminology, often by adding or changing an existing declaration,
- Unfolding commands, which replace uses of a construct with equivalent but lower-level constructs,
- Implementation / approximate unfolding commands, which replace uses of a construct with a nearly equivalent construct which is closer to implementation, and
- Abstracting commands, which make a specification more abstract by discarding detail.

3. Current Advances

We have since been restructuring and reimplementing our transformation library. Our aim is to simplify the individual transformations, express their structure and function more declaratively, and make them more uniform and compatible. This is intended to increase the coverage, flexibility and extensibility of our library. In addition, we anticipate that the ideas as embodied in this improved library will be more readily transferred to other languages.

3.1. Representation Changes

Our first improvement was to change the representation of the specifications which the transformations evolve. This was done in order to isolate the transformations from the syntactic diversity inherent in the specification language, and to increase the compatibility between our representation and KBRA's presentation framework [13]. To make this improvement we changed the Specification Assistant from manipulating Gist parse trees - the obvious representation of the specification language's many syntactic constructs - to manipulating a more abstract "internal" representation. In this, specifications are represented as objects in an attribute-value notation, a form which is much more amenable to analysis and manipulation. We are finding that the transformations operating upon this internal form are considerably simpler than their equivalents operating upon the old parse tree representations. We employ translators to convert between the old surface syntax for Gist and the internal grammar. Thus the surface syntax becomes merely a presentation of the underlying system description, one of many possible presentations in the spirit of the KBRA framework.

Next, we have changed the representation of the transformations themselves. As alluded to in our accompanying paper in this volume [10], we now express the transformations in Gist. Thus our transformations are Gist programs that modify Gist specifications. Advantages stem from the increased uniformity - tools that operate upon Gist (e.g., for explanation and analysis) can now be applied to the transformations - and the shift from viewing specifications as being merely parse trees to being databases of inter-related information that can be queried and updated. This approach makes use of an enhanced version of our Popart-DB facility [11] for automatically generating database models from grammars. The approach of describing operators as assertions and deletions from a database is common to most AI planners, such as Grapple [7].

Finally, we have extended our representation of transformations to include declarations of not only the inputs to, but also the outputs from each transformation. This is an important step toward improving our characterization of the effects of each transformation, it enhances understandability and reasoning about transformations (e.g., makes it easier to automate the application of transformations as substeps to achieving a larger transformation).

Representation of inputs and outputs to transformations helps to make explicit the data flow view of Gist programs. Data flow presentations are an important requirements formalism, so a proper integration of data flow and Gist will be necessary as part of the new ARIES requirements / specification assistant. However, in the process we have encountered a major incompatibility between the data flow view as it is conventionally used Gist's database modeling view. In the data flow view, processes interact with the environment only through fixed input and output ports. In the database modeling view, facts about the environment are represented as a database of assertions which processes can freely access and modify. Naive characterizations of the data flow inherent in these data accesses and assertions are either much too complex or not meaningful. One naive solution would be to treat the global database as a single data store that is input by every process and output by every process in some modified state. This solution is inadequate because it tells us nothing about what actually changed in the database. Another naive solution is to treat the tuples of the database as individual data items which are separately input and output from the process. This view typically gives us far too many data flow links, so again we get a poor understanding of what the process is doing. These problems manifest themselves in our transformations because we find that although we have identified the major objects that are input and output from a process, we have not characterized the side effects can result in the specification apart from the manipulation of the input and output objects. What is required, then, is a way of segmenting database accesses into meaningful units, such that a small number of such units enter and leave the process. Possible ways of segmenting the data is to group relations into complex objects, or to treat all tuples of a relation as a single data object. Kevin Benner, formerly of RADC and currently at ISI, is trying to develop an effective characterization of data flow that is compatible with the assertional style of specification.

3.2. New Observations on Transformation Structure

3.2.1. Structure-adding transformations

As a result of these advances in representation, we can make new observations about evolution transformations. First, we now see a uniformity to the structure-adding transformations (which comprise some 60% of the transformation library we have accumulated so far). Each such transformation:

- instantiates a template,
- inserts it into the specification, and
- propagates any necessary further changes through the specification.

The purpose of this functionality clearly overlaps with that of cliché-based systems such as KBEmacs [15], PROUST [8], or KBRA. Instantiating a cliché in these system involves adding

structure. The difference is in having made the above distinction between instantiation, insertion and propagation. Cliche libraries state the constraints that must be satisfied in order to incorporate each template into the system description. They either rely upon the user to modify the surrounding text to satisfy the constraints, or they employ constraint propagation techniques to make the required further changes. Our approach instead has been incorporating the propagation phase as part of the transformation. The advantage of our approach is that we can incorporate specialized propagation tactics if we choose, rather than relying upon some general-purpose constraint propagation technique. The disadvantage is that we do not make the constraints themselves explicit, nor do we take advantage of general-purpose constraint propagation mechanisms when they would suffice.

Another advantage of our approach is that it makes possible the declarative expression of the effects of transformations which would be hard to represent in the cliche approach. For example, our "Build Declaration From Reference" constructs a declaration from some use of an undeclared concept. Its effect can be characterized as completing (or improving the completeness of) an incomplete specification by addition of the appropriate declaration. The actual form of the declaration is heavily dependent on, and derived from, the use. Thus in case there is no single structure that could be expressed as a cliche and instantiated uniformly in every situation of use.

The remainder of our evolution transformations, those that do something other than adding structure, has no obvious analogy within the cliche approach. An interesting sub-category is that of transformations to re-organize the specification while leaving its meaning unchanged, these are applied to:

- reorder the specification components for better presentation,
- rewrite specification components into an equivalent form using different language constructs, or
- eliminate redundancies and make explicit some otherwise implied features of the specification.

3.2.2. Categorizing the effects of transformations

As outlined earlier, we divided our library of specifications into a number of categories to assist in user selection. This categorization showed some signs of inadequacy, in that its classes were not completely independent, and membership of a class conveyed only a vague notion of what the transformation did. We now have a understanding of how this categorization could be improved.

The most obvious distinction between transformations - whether or not they "preserve correctness" (i.e., leave the functional behavior denoted by the specification unchanged) - is but one of many useful distinctions. There are properties other than functional behavior which are

equally important for designing systems; some such properties that we have identified are:

- the structural organization of the specification, i.e., the organization of components into modules and the scope of their definition,
- the entity-relationship model defined in the specification,
- the flow of data between system components,
- the definition-use structure of the specification,
- the calling hierarchy in the specification (its control flow), and
- the functional behaviors.

We believe that a characterization of our transformations along this improved set of dimensions will help the following:

- assisting the user to select the appropriate transformation(s),
- replaying transformations - one important use of replay is in merging the effects of separately considered evolutions, as described in [4], and
- filling the library of transformations - these dimensions can be used as the basis for estimating the extent to which our transformations cover the space of possible changes to specifications.

We believe that systems can be described along each dimension as a network where the nodes are system components or behavior states, and the meaning of the arcs depends upon the dimension being considered. From this viewpoint, the structure of all basic meaning-changing transformations becomes similar to that of the cliché-adding transformations: they make a change along some dimension, and then propagate further changes that are needed to integrate the initial change into the specification. The change performed by the cliché-adding transformations is to add a new node to the network; others delete nodes, replace nodes, and reroute arcs.

We find that the capabilities of AP5 - a database model which supports associative retrieval and can be extended with updatable derived relations - are crucial to representing these dimensions.

3.3. Building a More Complete Library

Given the above analysis, we can now assess what will be required to construct a useable transformation library. First, we must verify that all of the significant dimensions of system descriptions have been identified. Our previous example-driven approach to transformation development has uncovered a number of dimensions, and as we broaden our system representation to include more non-functional requirements, new dimensions will likely emerge.

Along each dimension, the nodes and arcs may fall into different categories, the significant categories must be identified. For example, the entity-relationship view in Gist treats types, relations, and events as different categories of nodes, the relevant arcs are specialization-of and parameter-of.¹ Then, we must ensure that at least some examples of each kind of network change are supported, i.e., adding and deleting nodes and arcs, and rerouting arcs. It may be necessary to construct specialized transformations according to the type of arc (e.g., Add Specialization) or the type of node (e.g., Add Relation).

As we indicated above, there may be multiple valid ways of integrating a change into a specification. In such cases, the alternative methods must be identified and formalized. The user can then have the option of choosing which method to apply. Transformations which arbitrarily choose a particular integration method are unlikely to get broad user acceptance. Generally, we wish to reduce the apparent complexity of transformations through this more uniform view of their effects and operation. Even large transformations that perform several changes should be simple to comprehend as a set of major changes, together with propagated effects.

We are already well on our way to providing this basic foundation for a transformation library. It is our hope that the library will achieve coverage by providing a set of compatible "building-block" transformations which users will easily be able to compose to build more transformations.

4. Applicable Results and Future Challenges

It is a straightforward matter to retarget our approach to evolving Gist specifications to many other languages. The issue at stake is whether the same dimensions of description that we have identified can be made explicit in programs in another language. Certainly other specification languages which are based on the entity-relationship approach or the data flow approach are likely candidates. Most programming languages are susceptible to formal evolution as well. The problem with evolving program codes, however, is that the necessary information for effective evolution is not present. For example, Lisp programs are not strongly typed, this means that it is impossible to propagate changes to the entity-relationship view of a Lisp program.

The main technical prerequisite to our evolution approach is a knowledge representation framework that supports retrieval and updates of the relevant program features in an abstract form. We rely heavily upon our AP5 and Popart-DB tools, other frameworks such as Refine can provide similar capabilities. Without suitable abstractions, transformations are difficult to write and understand, and are unlikely to be versatile enough to support the construction of a good library.

¹•Event• is the general category consisting of procedures and demons. Specialization-of is the subtype relation, but generalized to apply to relations and events as well.

In the mean time, we see some serious technical challenges ahead before evolution technology can be considered mature. First, the structure of our transformations is still not made adequately explicit in the form of the transformations. The instantiation, modification, and propagation steps are all bundled into the transformation methods. The constraints underlying the propagation activity are not represented. We will be directing effort in the future to overcoming these remaining representational problems. It will then be possible for our system to take a more active role in interactive planning of specification changes. Failed preconditions on transformations could then trigger a search of the transformation library for transformations that could make the preconditions true. The system could provide suggestions at each stage as to what transformations would be appropriate to perform.

Another major challenge is making effective use of evolution transformations in a multi-person project. In such a project, one developer's change may result in changes to a part of the system being developed by someone else. Two developers may attempt to elaborate the same specification in incompatible ways. Thus activity coordination will be required in order to avoid misuse of transformations. We believe that some of our previous work in studying how elaboration steps can interfere with each other will give us insights into this problem.

References

1. Balzer, R. Automated Enhancement of Knowledge Representations. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, AAAI, August 18-23, 1985, pp. 203-207.
2. Cohen, D. *AP5 Manual*. USC-Information Sciences Institute, 1985. Draft.
3. Elefante, D. What is KBSA? Proceedings of the Computers in Aerospace VII Conference, 1989.
4. Feather, M.S. Detecting interference when merging specification evolutions. Accepted for the 5th International Workshop on Software Specification and Design, May 1989.
5. Fickas, S. Automating the Specification Process. Tech. Rept. CIS-TR-87-05, Department of Computer and Information Science, University of Oregon, 1987.
6. Green, C., D. Luckham, R. Balzer, T. Cheatham, C. Rich. Report on a Knowledge-Based Software Assistant. Tech. Rept. RADC-TR-83-195, Rome Air Development Center, August, 1983.
7. Huff, K.E., and Lesser, V.R. The GRAPPLE Plan Formalism. Tech. Rept. 87-08, U. Mass. Department of Computer and Information Science, April, 1987.
8. Johnson, W.L.. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann, 1986.
9. The KBSA Project. Knowledge-Based Specification Assistant: Final Report.
10. Johnson, W.L., and Myers, J. Relating Formal and Informal Descriptions. Proceedings of the 4th Knowledge-Based Software Assistant Conference, 1989. to appear.
11. Johnson, W.L., and Yue, K. An Integrated Specification Development Framework. Tech. Rept. RS-88-215, USC / Information Sciences Institute, 1988.
12. Johnson, P. Structural Evolution in Exploratory Software Development. Proceedings of the AAAI Spring Symposium on Software Engineering, 1989.
13. Sanders Associates. Knowledge-Based Requirements Assistant - Interim technical report. Software Systems Engineering Directorate, March, 1986.
14. Narayanaswamy, K. Static Analysis-Based Program Evolution Support in the Common Lisp Framework. Proceedings of the 10th International Software Engineering Conf., 1988.
15. Waters, R.C. KBEmacs. A Step Toward the Programmer's Apprentice. Tech. Rept. 753, MIT Artificial Intelligence Laboratory, May, 1985.
16. Wile, D. *POPART. Producer of Parsers and Related Tools. System Builders' Manual*. USC Information Sciences Institute, 1981.

COORDINATORS AND COORDINATOR GENERATORS

W. G. Morris
Senior Engineer
Software Options, Inc.
22 Hilliard St.
Cambridge, MA 02138
(617) 497-5054
`chip%soi@harvard.harvard.edu`

Biography 'Chip' Morris received his Ph.D. in Mathematics from the University of Wisconsin, Madison in 1982. He has worked at Software Options since 1985, leading the implementation of a highly optimizing code generator for procedural languages. His other interests include software engineering and management, computer algebra, and combinatorics.

Abstract A *coordinator* is a reactive facility that manages communication and coordination among people and automated tools in a software engineering environment. A *coordinator generator* translates an abstract protocol into a coordinator. In this paper we give some requirements for the formalism that describes an abstract protocol, including an analysis of its graphical component (sketches) and its program components (refinements and interface pragmatics). We also discuss the problems posed by long-term execution of coordinators, the demands placed on database connections, and the challenges posed to programming languages used to refine sketches. We illustrate these issues with examples of existing coordinators and some potential applications for coordinator technology.

1 Introduction

Coordination is usually the last thing to be added to a software engineering environment. First we design the languages and build compilers, editors and debuggers. Then we invent some simple means for a single programmer to lash together the tools at the terminal. Eventually we may consider what happens when more than one programmer changes a document; along comes version control. When a project gets big enough, lasts a long enough time, or must comply with a funding agency's documentation standards, we begin to think about how managers, designers and programmers should communicate to get the project done soon and well.

What software engineering environments need, right from the start, is a facility to manage communication and coordination among the participants in the software life cycle: a *coordinator*. A coordinator should be *reactive* in Harel's sense[2], something that continuously responds to and stimulates its environment. Its actions and reactions are computer-mediated and embody the protocols for communication in a programming environment.

Different environments, of course, use different rules and customs for communication. Because each coordinator must have detailed knowledge about the protocols it manages, no single coordinator will serve all projects. This suggests that we must be able to easily build customized coordinators. Just as the difficulty in hand-writing a new parser for each new language led to the development of parser generators, so we propose the development of *coordinator generators*.

A protocol is, informally, a set of procedures, rules and customs that govern the practice of some domain of activity. We use "protocol" in this informal sense, but we also wish to think of a protocol as an abstract, mathematical object which may have a formal representation. In the latter sense we sometimes speak of a *protocol abstraction*, expressed in some *formalism*.

In this paper we first outline some desirable features of coordinators. Next we discuss protocol abstractions and coordinator generators. Finally, we summarize some of the technical challenges posed by coordinators and suggest wider applications of the technology.

2 Coordinators

For our purposes a *coordinator* is a reactive facility that manages communication and coordination among people and automated tools in a computer-mediated environment. While it is useful to refer to a coordinator as a single object, it will generally *not* be a single program or process. Rather, it will be a distributed collection of processes and associated databases that function together as an active microcosm of the protocol they represent.

A coordinator is a multi-user facility. It must respond to simultaneous messages from many sources and model concurrent tasks with a rich interaction among the people and tools that it mediates. It should alert a user to changes in relevant status while concealing irrelevant details. Much of its utility will lie in the extent to which it eliminates unnecessary message traffic among participants in a protocol.

Users should have a clear model of their role in the protocols in which they are involved. They should be able to ask questions like

- Which activities do I take part in?
- What should I do next in activity X?
- What has happened so far in activity Y?

- What is the status of the report I wrote last week?
- Who is waiting for results from me?

A coordinator should present views appropriate to the interests and expertise of the user. Managers may want to see PERT or Gantt charts. Programmers may want personal to-do lists, calendars of meetings, and status reports on software modules or bug reports. All of these reports may be regarded as aspects of the current state and the *history* of a coordination protocol.

Figure 1: Trouble Report Protocol

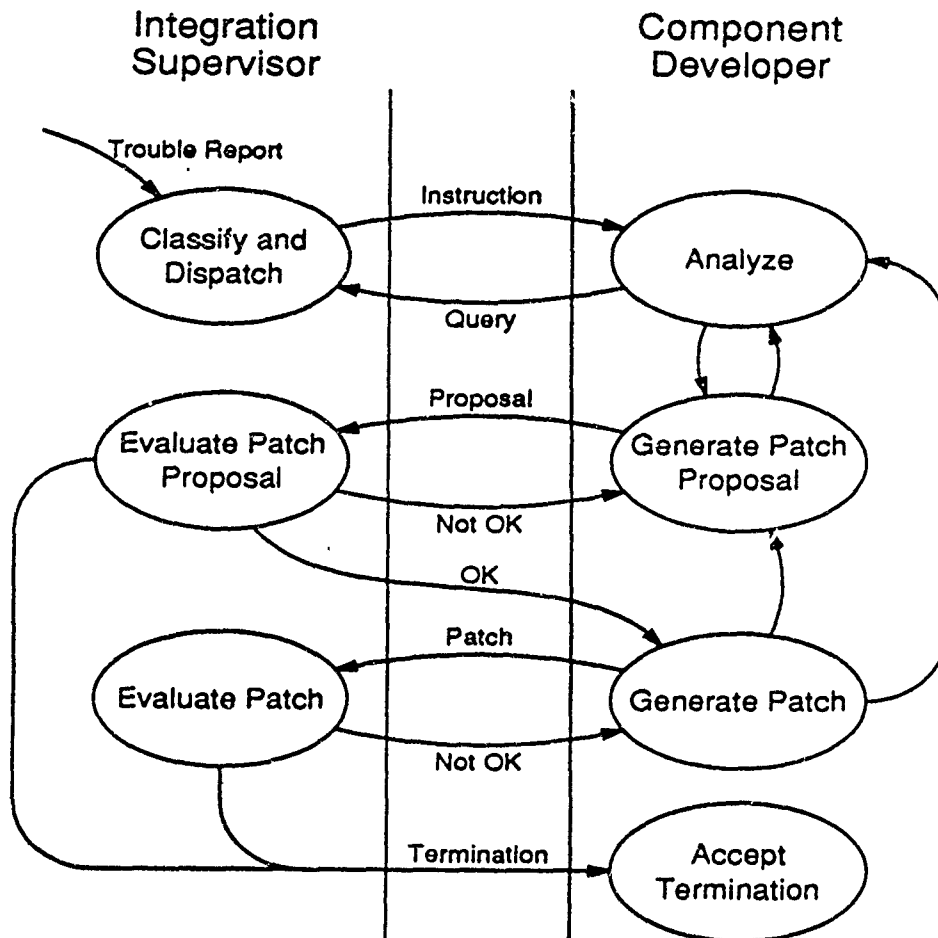


Figure 1 describes communication between two organizations involved in responding to software trouble reports. The one whose states are shown on the left (labeled *Integration Supervisor*) is responsible for coordinating the evolution of the system. The other represents the developer of a typical system component. Each oval node describes

the current protocol state of the organization within whose border it lies. Arcs show state transitions, and those between organizations typically represent the transmission of messages whose nature is given by arc labels.

A graphical representation can clarify complex protocols, such as the one used by MONSTR, a coordinator created to aid the processing of software trouble reports [4]. Winograd [12] describes another example of a coordinator based on a state diagram describing conversations. His work has evolved into a network tool for maintaining structured conversations, largely replacing traditional electronic mail functions.

One of the most vexing aspects of software development is that plans and methodologies, not to mention personnel and resources, change while the project is in progress. Any coordinator that depended on a fixed protocol must either be very specialized (such as Winograd's Coordinator or MONSTR) or be doomed to be discarded when the cost of working around its obsolete requirements grows too great. The alternative, of course, is to allow flexibility in the specification of the protocol and the ability to *cut over*, i.e., incrementally to change the protocol and incorporate the changes into an executing coordinator. We consider the ability to manage cutover gracefully to be one of the primary requirements for implementing coordinators.

In practice, protocols vary greatly in the degree of control they exert over their participants. Similarly, a coordinator may merely track activities or it may control details like user permissions and file access. Since a salient advantage of a coordinator is that it shields users from exposure to unnecessary information and choice, what may appear to be a restriction actually increases productivity. Experience with MONSTR suggests that "guided communication" and central access to information about the state of the protocol were the major sources of its success (Knoke [9]).

It is important to understand that many of the methods currently used to mitigate coordination problems are not coordinators. The UNIX *make* facility is *not* a coordinator. Traditional project planning tools, PERT charts for example, are not coordinators. Neither is reactive, nor do they deal well with multiple users. Electronic mail is *not* a coordinator, lacking structure or much ability to inquire about the state of the "protocol". Version control systems such as RCS are *not* coordinators. The "integration module" that ties together a set of single-user CASE tools is *not* a coordinator, although the most sophisticated such environments seem to be evolving coordination facilities that may, in time, gracefully handle many users.

3 Coordinator Generators

To live up to their potential, coordinators must contain considerable information about the domain of activity being coordinated. The situation is analogous to parsing a programming language; soon enough it becomes apparent that writing a parser for each language and language variant is a task best delegated to computers themselves. Thus, just as parser generators were developed to mechanically build parsers for large and useful classes of languages, so we propose to construct a *coordinator generator* that can

build coordinators for large and useful classes of activity domains.

In order to mechanically generate a coordinator we first need a formalism in which to specify coordinators. This formalism should allow us to express the essential features of a protocol without the complicating details of implementation. However, a coordinator does not execute in isolation; it must be able to communicate with people, databases, and other processes across many computers over indefinite spans of time. Thus we will need to augment the abstract description of a protocol with pragmatic details about its execution environment.

In Section 3.1 we discuss the requirements for a protocol formalism and briefly describe our approach. We reflect on the challenges these requirements pose for the execution environment in Section 3.3. Finally we turn in Section 3.4 to some details concerning the coordinator generator itself.

3.1 Formalisms

3.1.1 Requirements

Before considering any particular formalism, let us review our requirements for a formalism.

Formal Syntax and Semantics Just as parser generators required a means to specify grammars, so coordinators require a means to describe protocols. When writing down a formal version of a protocol we must make precise our informal, intuitive understanding of its workings. This involves building a *protocol abstraction* and expressing it in some formalism analogous to, say, a BNF grammar.

A well-defined syntax and semantics is, of course, necessary for mechanical translation, but we wish to emphasize that it may not be possible to reduce *all* of the aspects of a protocol to a single formalism. Nor is it desirable to try. Many aspects of a coordinator, such as details of the user interface or low-level process communications, are best handled by a programming language. But the high level aspects of coordination should be directly expressible in a form particularly suited to the purpose.

Concurrency and Iteration All interesting cases of coordination involve concurrency, so its expression should be natural in any formalism for protocols. Iteration, too, should be natural. Often a protocol will consist of a "conversation" between two agents in which messages are exchanged over and over until some criterion is satisfied.

Re-Use There are many situations that occur repeatedly in protocols, the conversation mentioned above being one example. To support modularity and re-use a formalism should allow fragments, or perhaps entire protocols, to be used as parts of new protocols. One method of particular utility is composition, making the "output" of one protocol the

"input" of another. The product of a conversation could be the input of another. The results of a test run could be data for a design review.

Accessibility Not only programmers and software engineers will be reading protocols. Managers especially, and even customers, will need to understand and even themselves write large portions of a protocol. It is therefore essential that the formalism be in large part accessible to non-technical personnel. This would suggest that a coordinator formalism should have a substantial visual component.

Accessibility also suggests that the formalism should be amenable to presentation in different views, each appropriate to its audience. The manager should be able to see a high level view involving all of the persons under his/her authority. But programmers should be able to see just those tasks relevant to them or be able to track documents under their responsibility.

History A formalism should allow the protocol designer to specify what constitutes the *history* of activities in a protocol. When it is useful to describe a sequence of events as belonging to a particular activity, the formalism should allow it. Users need to be able to ask "What has happened to document X" or "Review progress to date on activity Y."

Analysis Besides semi-mechanical coordinator generation a formalism should support effective analysis. Among the questions an analyzer should be able to address are: Are there states of a particular protocol that can never be reached? Are there constraints on the protocol, such as limits on time or resources, that can be expressed formally and checked?

Cutover One of the most subtle and difficult criteria to satisfy is the ability to modify a protocol during its execution. The formalism must support this in a natural way by enabling one easily to identify the parts of an executing model that are affected by a change in the protocol abstraction. Generally, a "small" change in the specification should produce a corresponding "small" change in the execution. Moreover, the formalism should support the kind of analysis needed to determine when cutover is feasible, or what conditions must be imposed to make it feasible. We expect that many, probably most, of the situations requiring cutover will be expressible as incremental change and propagated to the executing coordinator.

Scale It is essential that we be able to model very large protocols efficiently. While this appears to be a question of implementation, in fact the choice of formalism may have a profound effect. (A small change in a language's grammar may make it considerably more difficult to parse.) All of the desirable features such as reuseability and support for cutover are even more important in large examples.

3.1.2 Alternative Formalisms

In some sense the most general specification of protocols is as programs, and Osterweil has suggested treating them as such [10]. We agree that there are many aspects of a coordinator that require the generality of a programming language, so we must allow the possibility of combining programs with some other formalism to specify fully a protocol. Some researchers tackle coordinator problems using inferential, or "rule-based" techniques, for example Huff and Lesser [6], and Kaiser [8]. However, despite the appeal of inferential and pure programming methods, our decision to emphasize interaction and communication suggests that we explore the consequences of combining visual methods with programs.

Among visual formalisms two are of particular interest, Harel's statecharts [2] and hierarchical colored Petri nets [7].¹ Both have a well-defined syntax and semantics, support concurrency and iteration, allow hierarchy and alternative views, and are relatively accessible to non-technical readers. While we regard statecharts as a reasonable alternative, we favor hierarchical colored Petri nets (h-CPNs) for two reasons. First, it is much easier to see how to compose Petri nets than statecharts. Second, statecharts separate the visual representation of control from that of data-flow. While this is an advantage in some realms, our experience indicates that for coordination it is more natural to combine data and control flow as is most commonly done with h-CPNs. We will discuss the use of hierarchical colored Petri nets when we examine sketches, below.

3.2 Protocol Abstractions

In this section we describe one method of formalizing protocols. It is based on work in progress at Software Options, and should be taken only as an indication of the direction of our effort. We expect the details to change as we draw closer to a complete formal description of coordination protocols.

We propose to think of a *protocol abstraction* as having two parts, a *sketch* and *refinements*. A sketch is a diagrammatic rendering of a protocol based on a well-defined visual formalism, hierarchical colored Petri nets. Refinements are program fragments that provide details of the actions described by the sketch while preserving the sketch's semantics.

3.2.1 Sketches

Colored Petri nets were invented by Jensen as an extension of predicate/transition nets[1], which were themselves a higher-order version of the original net formalism of Petri [11]. We do not describe Petri nets in detail here, but give an informal presentation of their role in sketches in an example, below.

Huber, Jensen, and Shapiro [5] further elaborated colored Petri nets by added hierarchy in conjunction with their work on design/CPN, a product of Meta Software, Inc. A

¹We also wish to mention Holt's diplans[3] with which we have too little familiarity to comment here.

principal motive for all these extensions is the well-known problem of scaling of Petri nets to large systems. Colored Petri nets (CPNs) allow the designer to make use of a system's symmetries to simplify their representation as nets while adding flexibility.

In colored Petri nets the tokens have attributes called *colors*. By using colors one can give tokens individual identity, thus allowing them to represent data flowing among the places as well as control elements. This is very natural in systems where the flow of control depends on flow of data, say when a document moves from one person to another. One place in the net may represent the first person reading the document, and the subsequent place the second.

Figure 2 shows a colored Petri net for a simple edit/review protocol. Ovals are *places* that represent a state of the protocol. Although "state" implies a static condition, in a protocol a state often suggests that some activities are taking place which are simply irreducible with respect to the protocol.

Each place has a *color-set* (written in slant type) that describes the kinds of tokens that may appear in that place. Thus the place **Ready To Review** may contain only *reviewers*. The arcs bear expressions (in this case simply variables) whose types are constrained to be a color-set, and only tokens having a color from that color-set may flow along the arc.

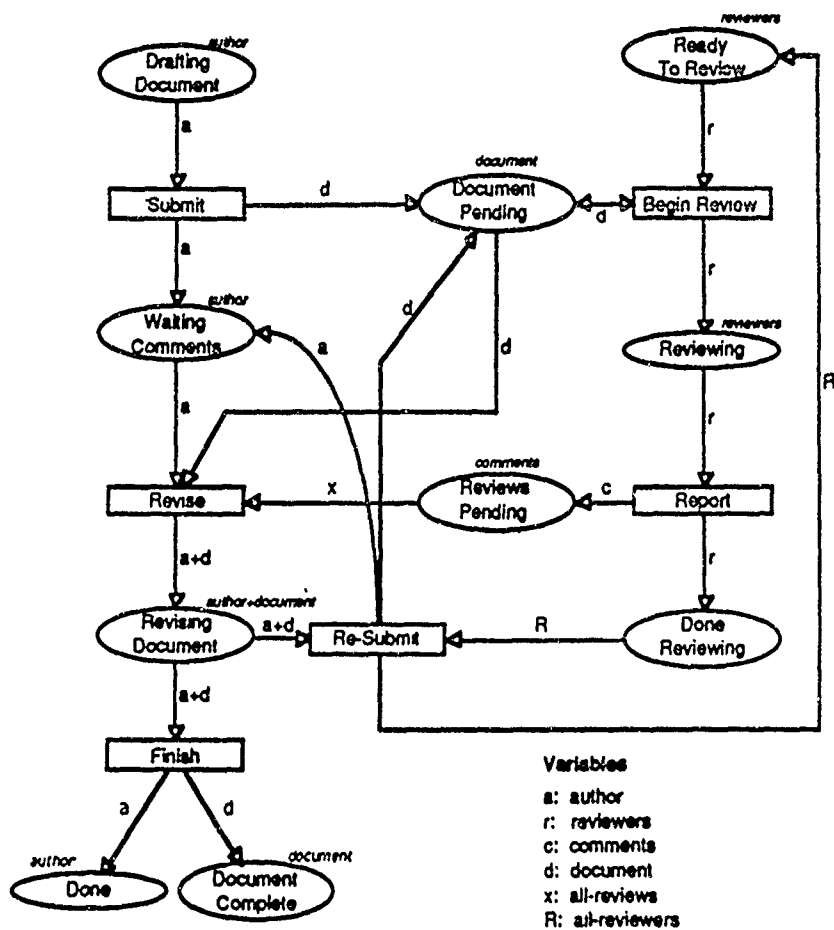
The boxes in Figure 2 are *transitions*. Places with arrows into a transition are *input places* for that transition, and places with arrows out of transitions are its *output places*. A *binding* of an arc label has a meaning analogous to that in programming languages: substitution of an admissible value (a color) for each variable in the expression.

A transition is *enabled* if there is a binding of its input arcs and sufficient tokens in its input places so that tokens can flow along the input arcs according to the colors given by the values of arc expressions. For example, if the place **Ready To Review** contains at least one reviewer token, and **Document Pending** contains a document token, then the transition **Begin Review** is enabled. When it *occurs*, it removes one document token and one reviewer token from their respective places. It then places the document token right back in the **Document Pending** place and the reviewer token in the **Reviewing** place. The two headed arrow between **Document Pending** and **Begin Review** is shorthand for one input arrow and one output arrow bearing the same arc expression.

A distribution of tokens on the places is called a *marking*. We begin "execution" of the net with an *initial marking* consisting of three reviewer tokens in **Ready To Review** and one author token in **Drafting Document**. At first only the **Submit** transition is enabled. It occurs, placing the author token into **Waiting Comments** and the document into **Document Pending**. Now **Begin Review** is enabled in three possible ways, depending on which reviewer is bound to *r*.

The execution of the net continues, which transitions occurring when they are enabled and tokens moving about the net. Execution ceases when no transitions are enabled. At any point during execution we can inspect the "state" of the protocol by seeing where the tokens are. In subsequent paragraphs we discuss how net execution can be a useful model of a coordination protocol.

Figure 2: Edit/Review Protocol



Queries The sketch in Figure 2 shows how a coordinator might support queries about the state of the system. For example, we can establish by inspection that there is always only one author token present on the net. The question "What is the author doing now?" then has three possible answers depending on the location of the author token in one of the three places it can reside. Hierarchical CPNs also allow one to *prove* facts about the net (such as "there is only one author token at any moment") or determine what distributions of tokens are possible.

The author might be inclined to ask about the location of his/her document, and similar reasoning shows that the question is well-formed and has three answers: "Pending," "Under Revision" or "Complete." Once again, the fact that it is provable that there is at most one document token on the net allows us to refer to *the* document. The net is designed so that initially, while the author token occupies **Drafting Document**, no document is yet in existence.

Another natural query that should be available to each agent involved in the protocol is "What can I do next?" This question can be answered in terms of which transitions are enabled. If the **Begin Review** transition is enabled, then every person whose reviewer token is in **Ready To Review** has "Begin Review" on his/her task queue.

Decisions A person often has more than one task available, that is, his/her token occupies a place for which more than one transition is enabled. This situation is known as *conflict*, and it may be used to represent a decision. In Figure 2 if the author and document tokens are in **Revising Document** then two transitions, **Re-Submit** and **Finish** are enabled. The net itself does not determine which should occur, so we may use this situation to model decision by outside agents, e.g., the author.

Another way in which decisions may be modelled is when more than one binding enables a transition. This happens when **Document Pending** holds the document and there is more than one reviewer in **Ready To Review**. Any one of the reviewer tokens may be bound to *r*, but which one is not determined by the net. One can imagine this being represented in an executing coordinator by presenting a menu item to each of the reviewers asking them if they wish to "Begin Review."

Conversion to a Coordinator The coordinator that results from Figure 2 would reside in a library much like an executable object. A user, presumably someone who wants a paper reviewed or perhaps a manager, would invoke an instance of this coordinator. It would prompt for the identity of the author and the number and identity of the reviewers. (Although the sketch specifies three reviewers, it could be modified slightly to allow *n* users, where *n* is bound at invocation or during execution.) It would then begin with the initial marking (the author in **Drafting Document** and the reviewers in **Ready To Review**, presenting status information as required by the participants. When the author is done with the draft, he/she has some standard way of indicating this to the coordinator (perhaps a menu choice, depending on the user interface), and the coordinator would proceed with the protocol.

3.3 Execution Environment

We have already hinted at aspects of the execution environment, and in this section we address three issues presented by coordinators: long-term execution, connections with existing databases, and the user interface.

Long-Term Execution Activities in a large software project may persist over very long time periods, so a coordinator must be kept in a reliably persistent form. Should the system crash, it must be possible to recover information about the state of the protocol quickly. We are assisted by the fact that a coordinator's work is not particularly CPU intensive. Each "transaction" will tend to be brief and involve relatively small parts of the total data structure. The load imposed by each transaction should therefore be small. We have already implemented such a "very long execution" mechanism and intend to use it as our coordinator execution platform.

Database Connections A coordinator will generally require substantial amounts of information from databases that already exist on the host system. User identities, access privileges, document libraries and code repositories are a few examples that will arise even in the simplest protocols. This may require dealing with databases with heterogeneous facilities. Some may be simple text files, others will have customized interfaces (such as version control systems), and still others will have sophisticated query languages.

There are two basic approaches to this problem: strong and weak coupling. A strongly coupled approach essentially abandons the goal of adapting to an existing heterogeneous set of tools and opts for customizing each part to work closely with the others. However, the utility of a coordinator generator in dealing with a wide variety of protocols suggests that we try to attack the problems presented by weak coupling and allow ourselves access to existing databases and associated tools. We call this the *database connection* problem.

Because a coordinator must be reactive, it must be sensitive to changes in a database that affect its protocol. In effect, all of the relevant databases must support stimulus/response rules for reacting to events. Since few indigenous databases are likely to have this facility, a coordinator environment will have to provide some special interface to achieve the same effect.

Our solution lies in the language of the programs that form a part of sketches. In addition to providing stimulus/response rules, the language should encapsulate database primitives in a uniform interface, helping make coordination models portable from one environment to another. The language might also provide a number of abstractions, such as constructs that permit the statement of object relationships using first-order logic, the retrieval of sets of objects through inferential queries, and description of atomic transactions together with associated rules for maintaining database consistency, and the modular expression of stimulus/response rules for reacting to model events. By providing these facilities we make very limited demands on existing databases.

User Interface A idea of a coordinator does not require a sophisticated user interface for many interactions. Intending a coordinator to be a tool of wide applicability, we favor a weak coupling of the user interface to the central executing model. At the very least there should be a notification mechanism to alert participating users to the change in a protocol's status and a corresponding means for the user to send messages back to the coordinator.

3.4 Generators

A coordinator generator translates coordinator specifications into executable realizations of their original protocol abstractions. Such a realization, a coordinator in our terminology, incorporates the protocol in a reactive, ongoing model. In general, the implementation of such a model should be distributed, not embodied in some central process, and it should tolerate failures of the underlying computing machinery, since the activities being modeled may last for months or years. Moreover, the state of an executing protocol realization must be accessible and even modifiable, to facilitate protocol cutover.

We have implemented a mechanism for the long-term execution of programs that we propose to use in realizing coordinators. The coordinator generator will essentially compile a sketch into a program specially suited to this mechanism. For very long executions the states of programs are no longer *implicit* objects like the volatile states of programs produced by usual compilation. Instead they are explicit objects having a machine-independent representation and residing in a special database. Translation of a sketch breaks it into *transition programs*, which are so named because when an event that is to cause a transition between protocol states occurs, it activates one of these program fragments. The fragment fetches the proper state object, carries out the appropriate (usually very brief) action, saves the resulting state object, and returns to dormancy.

Using transition programs takes full advantage of the transaction-like nature of interaction with a coordinator. Even if the host system crashes during a transition program, almost the entire state of the protocol will be unaffected. We can rely on the integrity of the database used to store program state rather than inventing new mechanisms.

4 Summary

At Software Options we are in the process of implementing a prototype coordinator generator that can support the kind of protocols typically found in a software engineering environment. To build a particular coordinator we begin with a sketch that describes the protocol in a high-level visual formalism, in our case hierarchical colored Petri nets. We augment the sketch with protocol refinements, programs that detail some of the high-level operations of the sketch. The result is a protocol abstraction that embodies the protocol and permits analysis of its formal properties, but has little detail about its implementation.

In order to produce a running coordinator, we supplement the protocol abstraction with interface pragmatics that detail the database connections, user interface, and process

communication for the host system. The result is a specification that is transformed by the coordinator generator into a very long executing program.

4.1 Technical Challenges

Coordinators and coordinator generators present many technical challenges that we must address even in a first implementation. While we do not mean to suggest that we will provide complete solutions to these problems, we think that coordinators, besides being of value in their own right, will serve as an important platform for research. For example, protocol abstractions are a challenging application of visual formalisms, and our requirements of accessibility and formal rigor will help clarify the value of hierarchical colored Petri nets.

Coordinators also challenge traditional approaches to programming languages. The programs that make up protocol refinements should be as high-level as possible, using locutions natural to the domain being modelled. It should support polymorphism in types and procedures and extensible syntax, so that non-programming experts can understand the details of a protocol abstraction.

The need to manage persistent data also imposes new requirements on a programming language. Because a coordinator must fit into an existing environment of heterogeneous databases and standards for inter-process communication, the language in which we specify coordinators must directly address the database connection problem. This also suggests extending a general-purpose language to encompass database primitives and other features discussed in Section 3.3.

We have already partly addressed the problem of long-term execution of programs at Software Options by implementing low-level mechanisms for storing program state and altering it via brief transition programs. These mechanisms also give skeletal support for cutover, the change in an executing coordinator caused by an incremental change in its protocol abstraction.

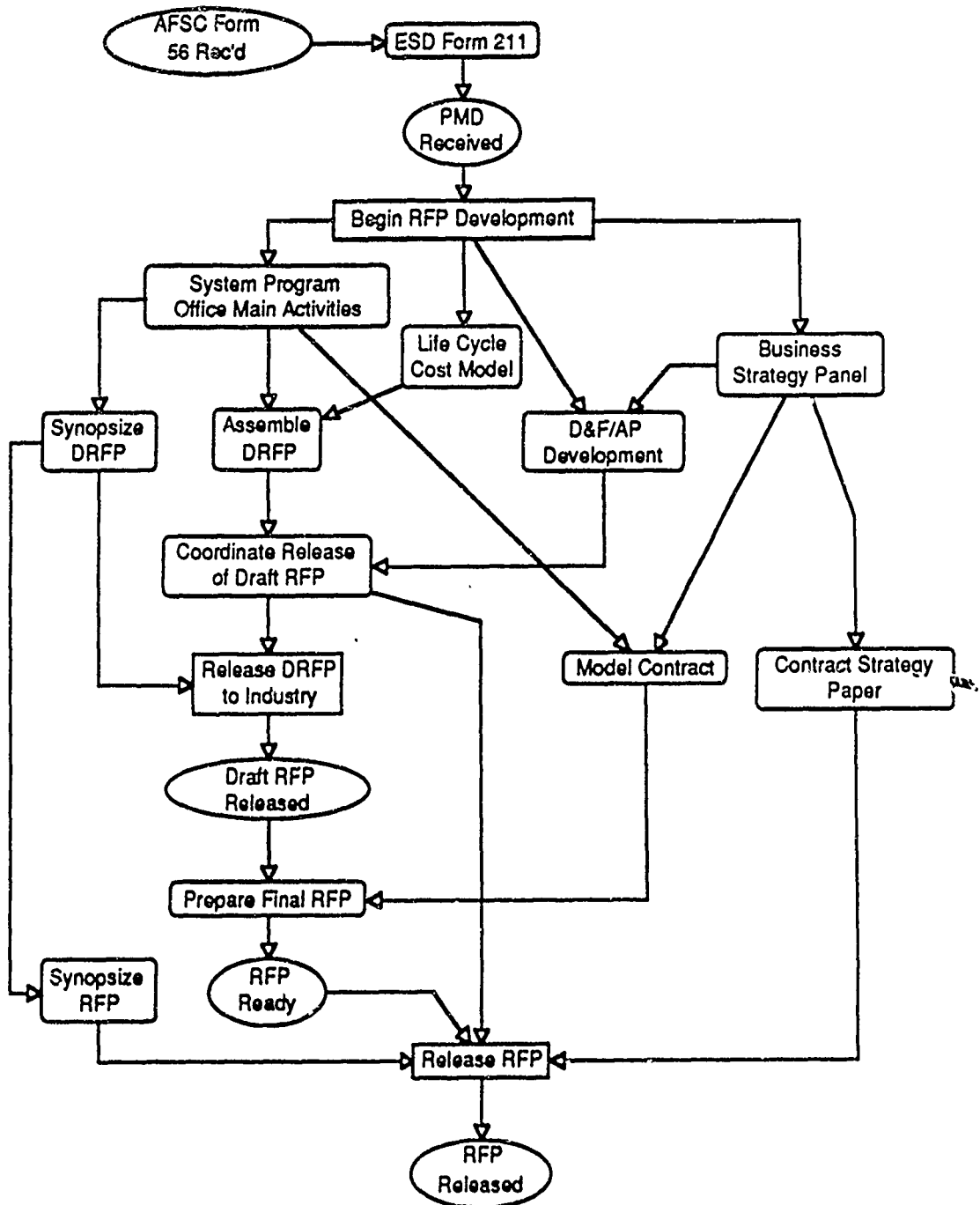
4.2 Coordinators in Other Domains

Although we have discussed coordinators in the context of software engineering environments, it is apparent that our means of expressing abstract protocols does not limit us to this domain. As part of ongoing work for the Air Force, we have been pursuing an example of coordination during the process used by the military services for systems acquisition. Figure 3 shows the top-level view of a sketch describing the process leading up to issuing a Request for Proposals, based on a 1982 document published by the Air Force Electronic Systems Division.

Our experience with this example shows the flexibility and potential utility of coordination technology for a wide range of endeavors. It involves a large number of individuals, organizations and documents with essentially asymmetrical interactions. The process takes place over eighteen-months and has numerous variations and exceptions

that must be handled in "real time". In practice, it would also be likely to undergo change during its lifetime both in structure and in the identities of its participants.

Figure 3: Protocol for Issuing a Request for Proposals



References

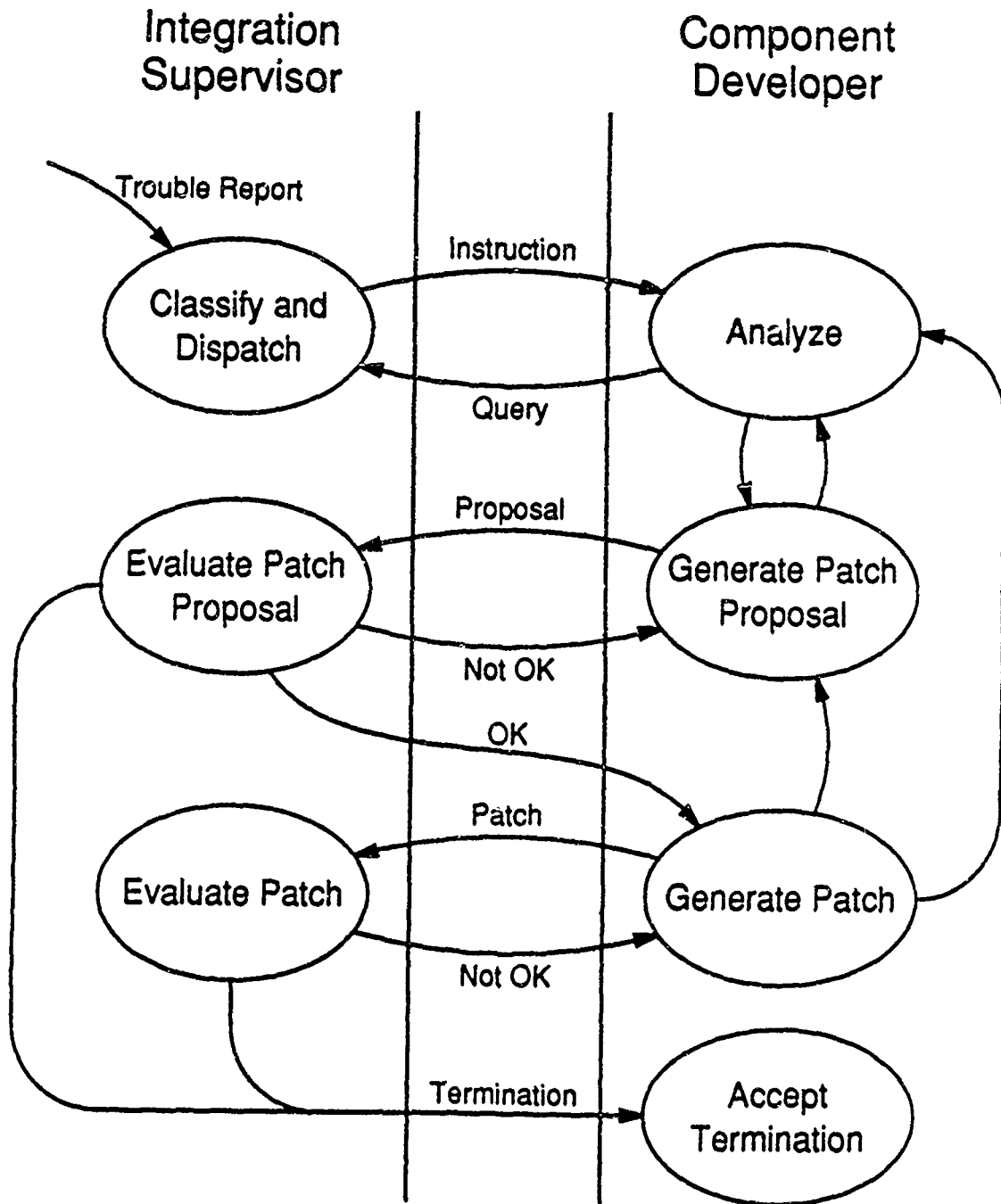
- [1] H.J. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science*, 13:109–136, 1981.
- [2] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [3] Anatol W. Holt. Diplans: a new language for the study and implementation of co-ordination. *ACM Transactions on Office Information Systems*, 6(2):109–125, April 1989.
- [4] Anatol W. Holt and Paul M. Cashman. Designing systems to support cooperative activity: an example from software maintenance mangement. In *Fifth Internation Computer Software & Applications Conference*, pages 184–191, IEEE, November 1981.
- [5] Peter Huber, Kurt Jensen, and Robert M. Shapiro. Hierarchies in coloured Petri nets. In *Tenth International Conference on Application and Theory of Petri Nets*, June 1989. (Submitted).
- [6] Karen E. Huff and Victor R. Lesser. A plan-based intelligent assistant that supports the software development process. *SIGPLAN Notices*, 24(2):97–106, February 1989.
- [7] Kurt Jensen. Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties (Lecture Notes in Computer Science No. 254)*, pages 248–299, Springer-Verlag, Berlin, 1986.
- [8] Gail E. Kaiser. Rule-based modeling of the software development process. In *Proceedings of the Fourth International Software Process Workshop, Devon, UK*, pages 36–38, ACM Sigsoft, May 1988.
- [9] Kathleen Knobe. Early experience with monstr: a software maintenance management tool. In *Proceedings of 23rd IEEE Computer Sociery International Conference*, pages 214–218, IEEE, September 1981.
- [10] Leon Osterweil. Software processes are software too! In *Ninth International Conference on Software Engineering*, pages 2–13, IEEE, Monterey, CA, April 1987.
- [11] Wolfgang Reisig. *Petri Nets. EATCS Monographs on Theoretical Computer Science*, Springer Verlag, 1985.

- [12] Terry Winograd. A language perspective on the design of cooperative work. In Irene Greif, editor, *Computer-Supported Cooperative Work: A Book of Readings*, chapter 23, pages 623–653, Morgan Kaufman Publishers, Inc., San Mateo, California, 1988.

Coordinator

- Enactment of a formal protocol
- Dynamic, persistent microcosm
- Reactive — can track *and* affect activities
- Provides role-specific perspectives
- Shows history and status
-
- Evolves as protocol changes

Trouble Report Protocol



After: Holt & Cashman, 1981

Experimental Applications

- Software Project — Design Phase
 - Frequent changes in protocol
 - Iteration and feedback
- Monitoring trouble reports
 - NSW procedures encoded in MONSTR
 - Many similar data objects
 - Multiple perspectives
 - Protocols to handle exceptions
- Acquisition management
 - RFP development (AFSC/ESD)
 - Concurrency
 - *Ad hoc* roles
 - Complex organizational structure
 - Multiple perspectives

Technical Challenges

- Visual formalism
 - Well-defined syntax and semantics
 - Usable by non-technical personnel
 - Editing and reporting
- Flexible (extensible) specification language
 - Support high-level database operations
 - Adapt to particular coordination setting
- Strong database connections
 - History of protocol
 - Indigenous data bases
 - Stimulus/response model
- Long-term computations
- Managing cutover

THE RLF LIBRARIAN: A REUSABILITY LIBRARIAN BASED ON COOPERATING KNOWLEDGE-BASED SYSTEMS

Raymond C. McDowell
Keith A. Cassell

Unisys Defense Systems
Paoli Research Center
PO Box 517
Paoli, PA 19301-0517

(215) 648-7529

ray@prc.unisys.com

Raymond McDowell is a member of the research staff at the Software Technology Laboratory of the Unisys Paoli Research Center. He is interested in the application of artificial intelligence to software engineering. Mr. McDowell received the BS and MS degrees in computer science from the Massachusetts Institute of Technology.

Keith Cassell was a research scientist in the Knowledge Systems Center of the Unisys Paoli Research Center. His primary interests are knowledge representation and software reusability. Mr. Cassell received his BS in biology and MS in computer science from the University of Texas at Austin. He may currently be reached at Lockheed, O/96-20 B/30E, 2100 E. St. Elmo, Austin, TX 78744, (512) 448-5716, cassell@stc.lockheed.com.

Abstract

Knowledge-based techniques provide tremendous leverage for managing and using repositories of reusable software. The Unisys Reusability Library Framework demonstrates this by providing an intelligent Librarian application that operates on a model of a software domain. The Librarian uses structured inheritance networks to model the library domain and organize the repository contents. The resulting structure of the domain model allows convenient repository browsing and component retrieval and provides a foundation for a variety of powerful tools. Rule-based inferencers complement this structure by making heuristic knowledge and advice available to the user. This makes the domain knowledge and repository organization more accessible to the user. The RLF uses a hybridization of these two knowledge representation paradigms, taking advantage of the strengths of each to provide intelligent assistance in the use and management of a software repository.

1. Introduction

Software reusability environments evolved to enable programmers to take advantage of the large assortment of previously written code when producing new systems. For reuse to be practical, it must be possible to locate previously written code and make whatever modifications are necessary in less time than it would take to write the code from scratch. At first glance, this would seem a simple organizational task; however, there is thus far no consensus as to what the best organization is. There is no standard hierarchy of components, and there is no agreed upon set of software attributes that serves as an index into these components. Furthermore, attributes that are relevant for one software domain (e.g., statistical packages) may not be relevant to others (e.g., speech recognition systems).

To overcome this problem, it is necessary to devise a way of storing knowledge about software components that is versatile enough to represent the myriad domains of software while still providing fast and accurate access to those components. The Reusability Library Framework (RLF) project[†] has incorporated ideas from knowledge representation research to devise a flexible and powerful modeling system. This paper discusses the knowledge representation paradigms used and the benefits of their application to software reuse.

2. The Reusability Library Framework

The RLF project applies knowledge-based techniques to software reuse, constructing an intelligent Librarian application that operates on models of software domains [Solderitsch89]. At the core of the RLF Librarian is the AdaKNET subsystem, a

[†]The Reusability Library Framework project began under the STARS Foundations program with a contract administered by the Naval Research Laboratory (contract number N00014-88-C-2052).

structured inheritance network knowledge representation system that provides a taxonomy of the reuse domain. This is supplemented by the AdaTAU subsystem, a rule-based inferencing system that provides heuristic guidance on the insertion and retrieval of parts from the repository.

2.1. Domain Modeling

We use AdaKNET and AdaTAU to provide a knowledge-based framework for modeling complex domains. For a reusability library, domain modeling involves grouping software into meaningful classes based on their attributes, and appropriately classifying software based on these attributes. To be maximally efficient, all listed attributes of a software unit should be relevant, and there must be a sufficient number of attributes to adequately differentiate between different classes. We anticipate rapid change in the kinds of software being developed over the following years. Thus, it is important to have a representation scheme capable of evolving with its domain.

The components of the RLF Librarian provide such a scheme. In the following two subsections, we discuss the rationale behind our choice of knowledge representation systems for modeling the software stored in reusability libraries.

2.2. AdaKNET: A Structured Inheritance Network Modeling System

Structured inheritance networks are a common way of representing knowledge in artificial intelligence applications. They provide a well-understood formal model that has served as the basis of many expert system applications. AdaKNET is based on KNET [Searls89], a Unisys proprietary system, and on KL-ONE [Brachman85].

Structured inheritance networks are directed graphs whose nodes represent objects and classes of objects, and whose edges represent relationships that exist between these objects. Taken as a whole, the network describes a domain, allowing applications using the network to exhibit behavior that seems intelligent or knowledgeable about the domain. A sample AdaKNET network is shown in Figure 1. The nodes of the network, depicted as ovals, are called concepts; in a librarian application, concepts can be used to represent such things as software units, categories of software units, algorithms, data values, data structures, and designs. AdaKNET networks have two primary types of links: specialization links and aggregation links. The specialization links, depicted as wide arrows, are used to indicate that the class of objects represented by one concept is a subset of the class of objects represented by another concept. The narrow arrows with a name and range are aggregation links, which show the attributes or component parts of a concept.

The specialization links of an AdaKNET network form a hierarchy of object classes in the modeled domain. One concept *specializes* another if the first concept represents a subset of the category described by the second concept. In Figure 1, the concept "Stack-Operation" is a specialization of the concept "Data-Structure-Manipulator", indicating that a stack operation is a specific kind of data structure manipulator. A concept may directly specialize more than one concept; for instance, "Pop-Directory-Stack" specializes both "Operating-System-Operation" and "Pop".

For a reuse library, the AdaKNET specialization hierarchy provides a software taxonomy that facilitates the location of units in the repository. To locate needed code, one starts at a category that is general and moves through more specific categories until the desired code is found. It is expected that inexperienced library

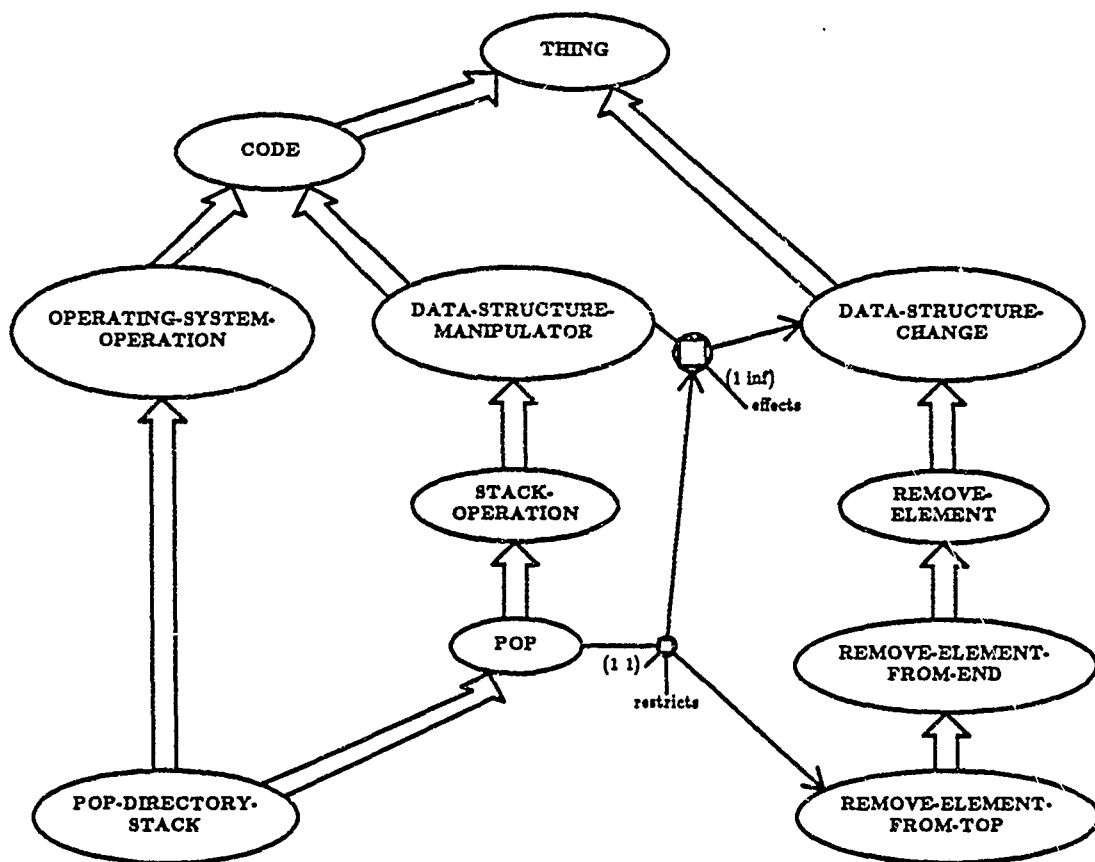


Figure 1: A Small AdaKNET Network

users will spend much of their time searching for applicable software by browsing the specialization hierarchy in this manner. A concept may have multiple "parents" in the specialization hierarchy; this enhances browsing by allowing that concept to be located through more than one search path. If a library user is looking for a routine that will pop a directory from a stack, he could come to "Pop-Directory-Stack" either from "Stack-Operation" or from "Operating-System-Operation". Such flexibility is particularly important for the naive user.

Aggregation links, also called *roles*, model the structure and attributes of objects. Roles have three properties: a name, which identifies the component or attribute modeled by the role, a type, which indicates the kind of thing that the component is or the kind of value that the attribute has, and a range, which specifies the number of these components or attributes the object may have. "Data-Structure-Manipulator" in Figure 1 has a single role which indicates that it has one or more "effects", each of which is a "Data-Structure-Change." The roles of a concept model necessary conditions; for an object to be included in the class denoted by a concept, it must satisfy the conditions imposed by all of the roles of that concept.

AdaKNET's aggregation links are useful to the Librarian in several ways. A concept's roles characterize the concept by describing the attributes and components of the objects in the class that the concept models. Thus the user need not rely solely on descriptive concept names in navigating through the taxonomy; the roles can also be used to clarify the meaning of concepts and to distinguish between similar concepts. Notice also that roles can be used to represent partial information; the range may be as broad or narrow as desired, and the type may be as general or specific as desired. This is not only useful for modeling categories of software, but also for representing specific software units whose characteristics may be variable, such as Ada generic units and software which may be automatically generated. Finally, the aggregation links enable the Librarian to support a query interface to the repository in addition to the hierarchical search approach described above. This allows the user to ask to see the set of software units that have certain characteristics or the concept representing the most general class of software with those characteristics. The combination of querying and browsing lets the user rapidly focus on items of interest by placing him at a relevant

concept in the taxonomy and allowing him to examine the various software units that meet his stated requirements.

In AdaKNET, a concept *inherits* the roles of the concepts it specializes; that is, each role defined at the parent concept is also a role of the child. Since a "Stack-Operation" is a kind of "Data-Structure-Manipulator", it has one or more "effects", each of which is a "Data-Structure-Change". When a concept directly specializes more than one other concept, it will inherit attributes from all of its parents. Inheritance allows economy of representation by associating a characteristic with the most general concept having that characteristic; all specializations of that concept will automatically have that characteristic associated with them. To define a new concept, one must only specify that concept's parents and the information which distinguishes it from its parents. Such distinguishing information may be new roles introduced at the specialization or further restrictions on inherited roles (e.g. narrowing the range or specializing the type). The "Pop" concept in Figure 1 restricts the inherited "effects" role by narrowing the range to indicate exactly one effect and narrowing the type to be "Remove-Element-From-Top".

The AdaKNET structured inheritance network subsystem provides a flexible and powerful means for modeling the library domain and organizing the repository contents. The specialization hierarchy provides a taxonomy for informal browsing, while its aggregation links support queries which can quickly focus the user on software units or categories that meet his needs. Among the benefits that an AdaKNET model provides for the Librarian are multiple access paths, efficient representation, and reasoning with partial information. In addition, it provides a foundation for intelligent assistance via the inferencing capabilities of AdaTAU.

2.3. AdaTAU: A Rule-Based Inferencing System

Like AdaKNET, AdaTAU is based on a Unisys-proprietary system called TAU (for Think, Act, Update) [Matuszek88]. AdaTAU is a forward-chaining rule-based system that processes two kinds of information: rules and facts. The state of an AdaTAU inference session is described by the current collections of rules and facts.

All facts are stored in fact bases that are accessible for the purpose of checking the applicability of rules (and therefore the addition of new facts). In its simplest form, AdaTAU processes a collection of rules and facts by firing all applicable rules until no new facts can be added to the fact base.

AdaTAU has a more complex, distributed mode of action as well. In distributed AdaTAU, there are a number of distinct rule and fact base pairs known as inferencers. A focusing mechanism determines which of these inferencers are active and may also transport facts between them, thus providing a mechanism for partitioning global knowledge into manageable subunits.

The current version of the Librarian uses distributed AdaTAU to provide advice for a user by associating inferencers with AdaKNET concepts. When a user requests advice at some position in the library, an inferencer there initiates a rule-based dialogue. If the user's answers indicate that another concept is more likely to contain the information that the user wants, control is passed to the inferencers housed at that other concept (which may represent either a specific component or a class of components). This pattern of rule-based interaction followed by a change of position in the network continues until the user requests a halt or until no further advice is available, leaving the user positioned at the most applicable concept.

Rule-based advice greatly enhances the ability of the Librarian. Any static organization of a library, including one based on a structured inheritance network, will be better-suited for some classes of users than for others; however, a good Librarian must serve a variety of people with different needs and abilities. By using AdaTAU to capture knowledge about a user's competence, we provide a means of effectively aiding both advanced and novice users. Similarly, we can provide rule bases to supply advice based on knowledge specific to a particular task (e.g. development or quality assurance).

In addition, the inferencers capture knowledge that is poorly suited to or obscured by AdaKNET. One such use of AdaTAU is to slice through complicated portions of a network, revealing a relationship several links removed from the concept currently being examined. More importantly, AdaTAU enables the use of heuristic knowledge such as weighting the relative importance of attributes, and it facilitates reasoning when the user can not provide otherwise needed information.

By adding heuristic knowledge to the Librarian, AdaTAU removes some of the reasoning pressure from the library user. It elicits information from the user according to his needs and abilities, and uses this information to concentrate on the most relevant parts of the underlying network. In this way, AdaTAU magnifies the effectiveness of the Librarian by making the domain knowledge and library organization captured by AdaKNET more accessible to the user.

3. Future Enhancements

This paper indicates some of the strengths that knowledge-based techniques can bring to reusability library technology. For the future, we intend to enhance the Librarian with features such as a graphics-oriented user interface and an underlying

database management system. More interesting, however, are the additions we intend to make that exploit the underlying knowledge representation of the Librarian.

There is a rich heritage of artificial intelligence research that we can exploit, particularly for structured inheritance networks. One problem we intend to address is that of software configuration [Falacara88], that is, ensuring that all auxiliary required units are made available together with the main software unit. This corresponds closely to the problem of hardware configuration, to which Unisys has already successfully applied structured inheritance network technology [Searls89].

We also intend to further exploit classification technology based on structured inheritance networks. Applications using this technology take a description of an item and use it to insert the item into the appropriate part of a network. Significant work has already been done in both automatic classification [Schmolze83] and in interactive classification [Finin86]. By applying this work to the RLF, we may be able to further automate portions of the component installation and retrieval tasks.

Similarly, machine learning research may help automate network construction. By starting with basic descriptions and applying conceptual clustering techniques [Fisher85], it may be possible to semi-automatically reorganize the knowledge base for maximal efficiency.

We would like the Librarian to be more than a passive tool. A library user should be able to easily find items in the library; however, we feel that a good tool should do more than that. Among the enhancements we are considering are facilities to keep track of the use of the system (as advocated by Jones [Jones86]). By collecting usage statistics, we can determine the most heavily used portions of a library and adjust its

contents and organization accordingly.

4. Conclusions

Much work has been done in artificial intelligence on ways of representing large bodies of knowledge using structured inheritance networks and rule-based systems. The RLF draws on this work to provide an effective environment for modeling a software library.

The structure supplied by the AdaKNET framework provides most of the RLF's power. By having concepts associated with their attributes and strictly enforcing inheritance conditions, AdaKNET provides a structured environment that allows not only convenient browsing and retrieval, but also more advanced classification and library reorganization tools. The AdaKNET aggregation hierarchy provides leverage for software configuration as well.

The RLF's rule-base system, AdaTAU, provides complementary functionality. It is well-suited for capturing heuristics and other poorly structured information not readily expressed by AdaKNET and further allows the Librarian to cater its interaction to different groups of users.

We use a hybrid of these two systems to model a software domain. By using domain-specific libraries, the Librarian can take advantage of specialized knowledge about a domain to assist the user in his selection and retrieval of software units, utilizing each knowledge-based subsystem to its best advantage. We believe that our hybrid approach can be expanded upon, ultimately resulting in the construction of a truly intelligent librarian application.

References

- [Brachman85] R. J. Brachman and J. Schmolze, "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, 9(2) (Spring 1985), pp. 171-216.
- [Falacara88] G. Falacara, M. Angevine, S. Bailey, and J. Laird, "A Tool For Ada Run-Time Tailoring," *Proceedings AdaExpo '88*, Oct. 1988.
- [Finin86] T. Finin, "Interactive Classification: a Technique for Acquiring and Maintaining Knowledge Bases," *Proceedings of the IEEE*, 74(10) (1986), pp. 1414-1421.
- [Fisher85] D. Fisher and P. Langley, "Approaches to Conceptual Clustering," *Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence*, Los Angeles, CA, 1985, pp. 691-697.
- [Jones86] W. Jones, "On the Applied Use of Human Memory Models: the Memory Extender Personal Filing System," *International Journal of Man-Machine Studies*, 25(2) (1986), pp. 191-228.
- [Matuszek88] P. Matuszek, J. Clark, J. Sable, D. Corpron, and D. Searls, "KSTAMP: A Knowledge-Based System for the Maintenance of Postal Equipment," *Proceedings of the Third US Postal Service Advanced Technology Conference*, 1988, pp. 421-435.
- [Schmolze83] J. Schmolze and T. Lipkis, "Classification in the KL-ONE Knowledge Representation System," *Proceedings Int. Joint Conf. on Artificial Intelligence*, Karlsruhe, West Germany, 1983, pp. 330-332.
- [Searls89] D. B. Searls and L. M. Norton, "Logic-Based Configuration with a Semantic Network," *Journal of Logic Programming (in press)*, 1989.
- [Solderitsch89] J. Solderitsch, K. Wallnau, and J. Thalhamer, "Constructing Domain-Specific Ada Reuse Libraries," *Proceedings of Seventh Annual National Conference on Ada Technology*, March 1989, pp. 419-433.

The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems

**Raymond C. McDowell
Keith A. Cassell**

UNISYS

Defense Systems
PAO d6164_vul_890811-1

Outline

- Statement of problem
- Overview of other approaches
- Analysis of RLF (knowledge-based) approach
 - Use of structured inheritance networks
 - Use of rule-based inferencers
 - Future directions

UNISYS

Defense Systems
PAO d6164_vu1_890811-2

Software Reuse

- Objective: increase programmer productivity
 - “Advantages of theft over honest toil” (Standish)
 - Eliminate redundant development
 - Components are well tested, designed, documented
- Requirement: less time to locate and modify than to write

UNISYS

Defense Systems
PAO 06164_vu1_890811-3

A Small Collection of Benchmarks

a0000091	a0000098	nppca2	p0000005	p0000012
a0000092	a0000099	nrpca2	p0000006	p0000013
a0000093	addsa2	nulla2	p0000007	prpca2
a0000094	drpca2	p0000001	p0000008	wheta2
a0000095	l0000001	p0000002	p0000009	wheta3
a0000096	l0000002	p0000003	p0000010	
a0000097	l0000003	p0000004	p0000011	

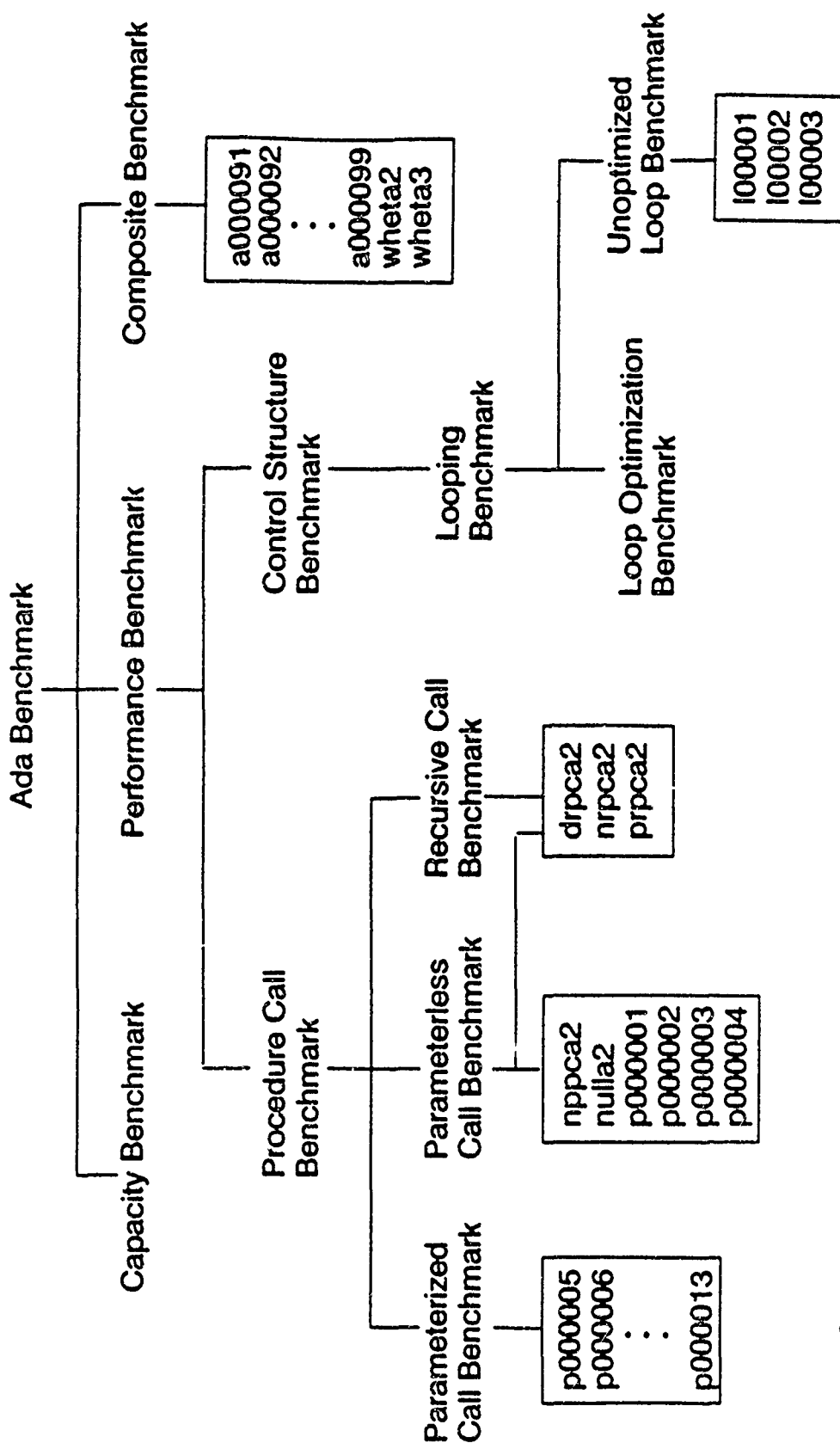
UNISYS

Defense Systems
PAO d6164_vu1_890811-4

Common Schemes for Organizing Repositories

- Hierarchical model
 - Convenient browsing
 - Limited descriptive content
- Keyword or attribute search
 - Rapid component identification
 - Requires domain/repository expertise
 - Model not extensible

A Benchmark Hierarchy



Defense Systems
PAO 06164_vu1_890811-6

UNISYS

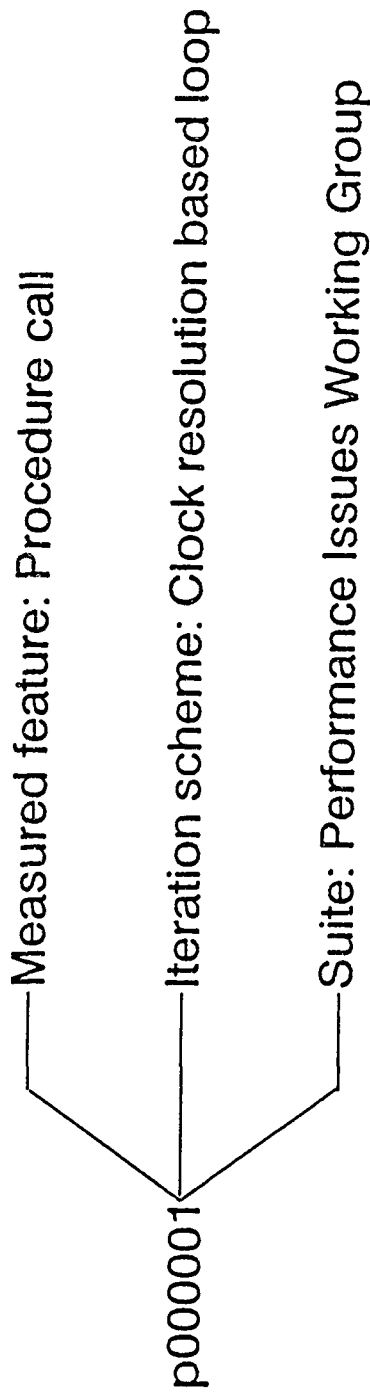
Benchmark Attributes and Values

<u>Measured Feature</u>	<u>Iteration Scheme</u>	<u>Suite</u>
System capacity	Clock resolution based loop	Ada Compiler Evaluation Criteria
Control feature	User-supplied number of iterations	Performance Issues Working Group
Indexed components	Fixed loop within test	•
Operators and expression evaluation	Redundancy without loop	•
References to local and non-local objects		•
Procedure calls		•
		•
		•

UNISYS

Defense Systems
PAO d6164_vu1_890811-7

A Benchmark Description



UNISYS

Defense Systems
PAO d6164_vu1_890811-8

The RLF Approach

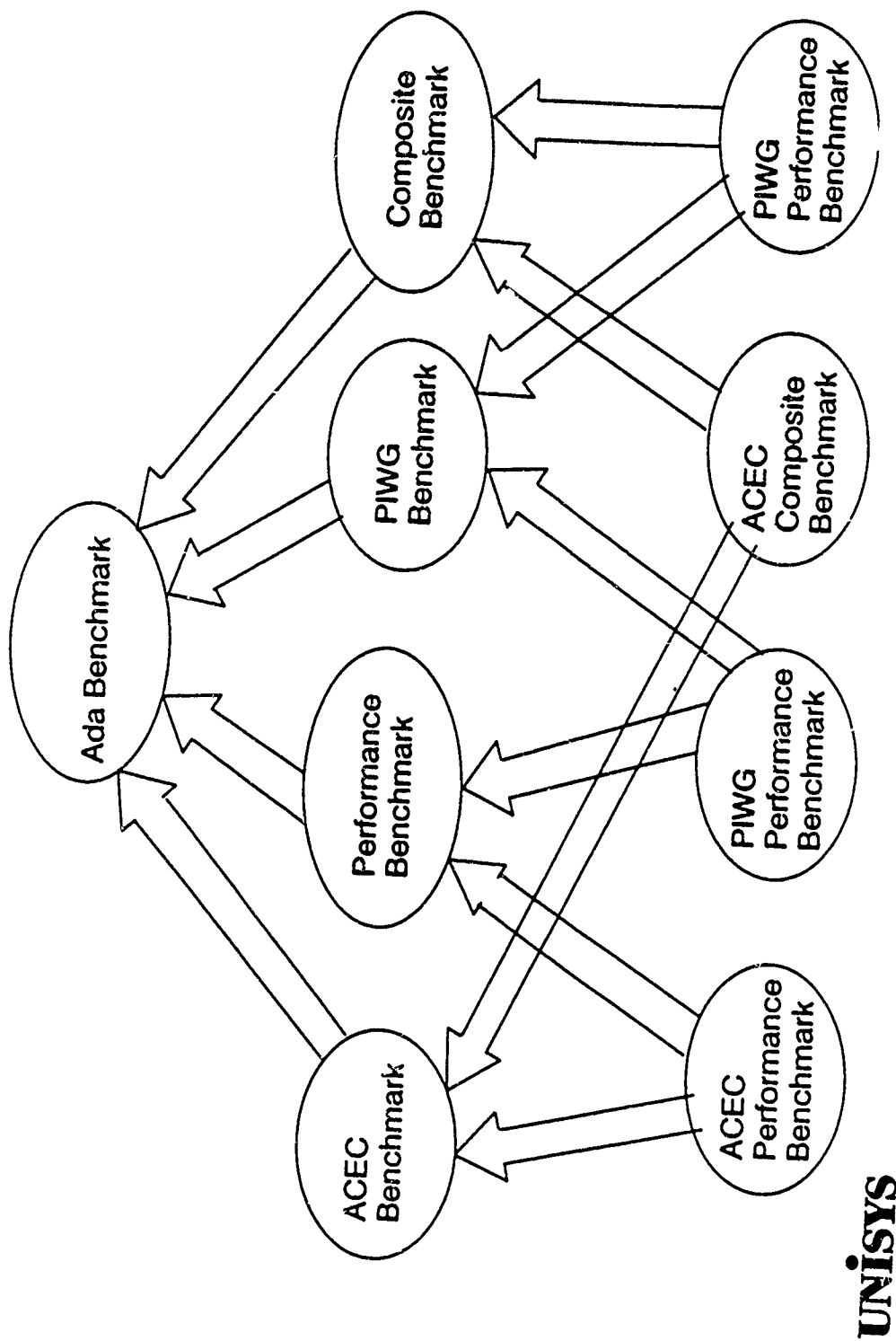
- Model domain and organize repository using structured inheritance networks
 - Allows convenient component location
 - Provides foundation for many powerful tools
- Capture heuristic knowledge using rule-based inferencers
 - Makes domain knowledge and repository organization more accessible to user

Benefits of Structured Inheritance Networks

- Specialization
 - Taxonomy for browsing
 - Multiple access paths
- Aggregation
 - Description of categories and components
 - Representation of partial information
 - Query interface
- Inheritance
 - Economy of representation
 - Guaranteed consistency

UNISYS

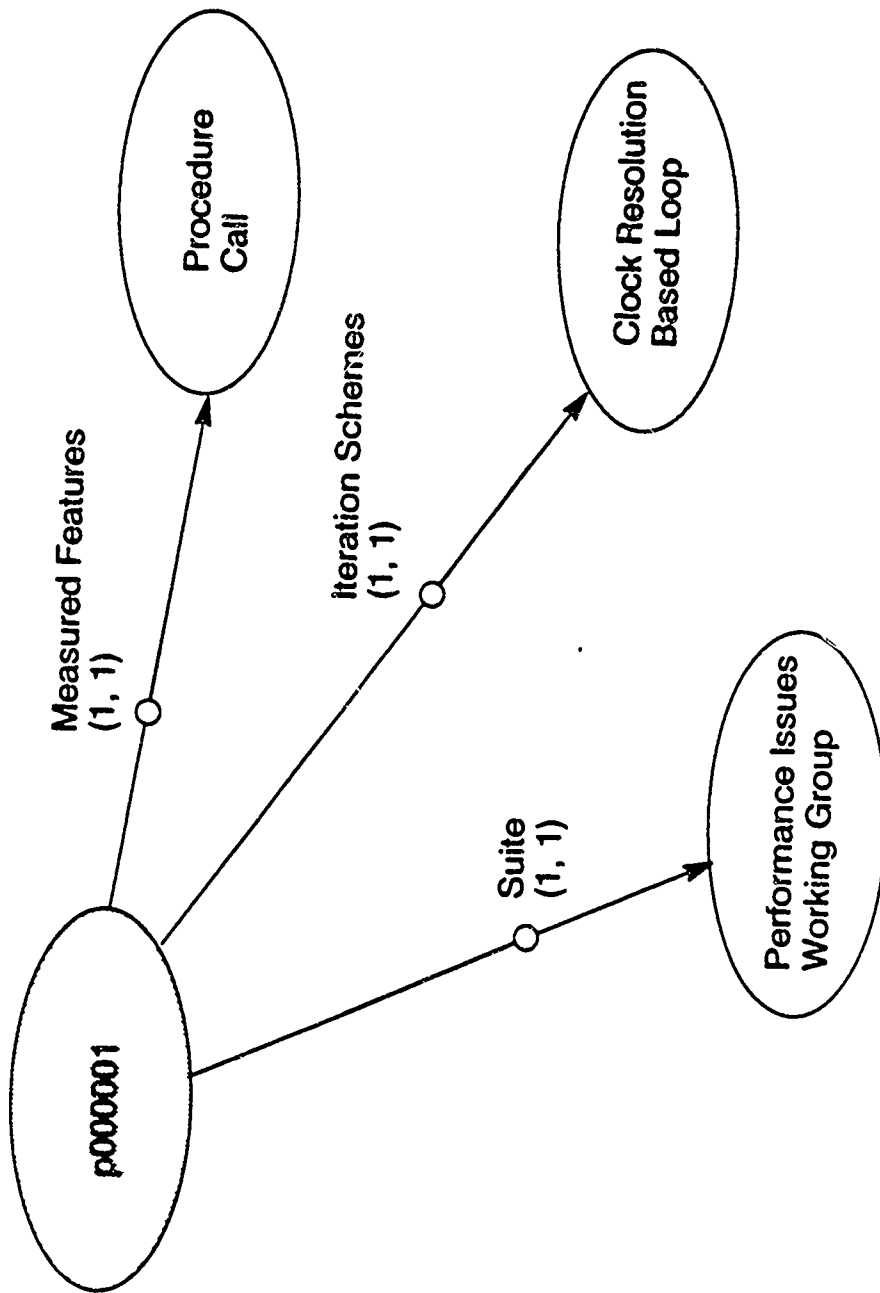
A Specialization Hierarchy for Benchmarks



UNISYS

Defense Systems
PAO d6164_vu1_800811-11

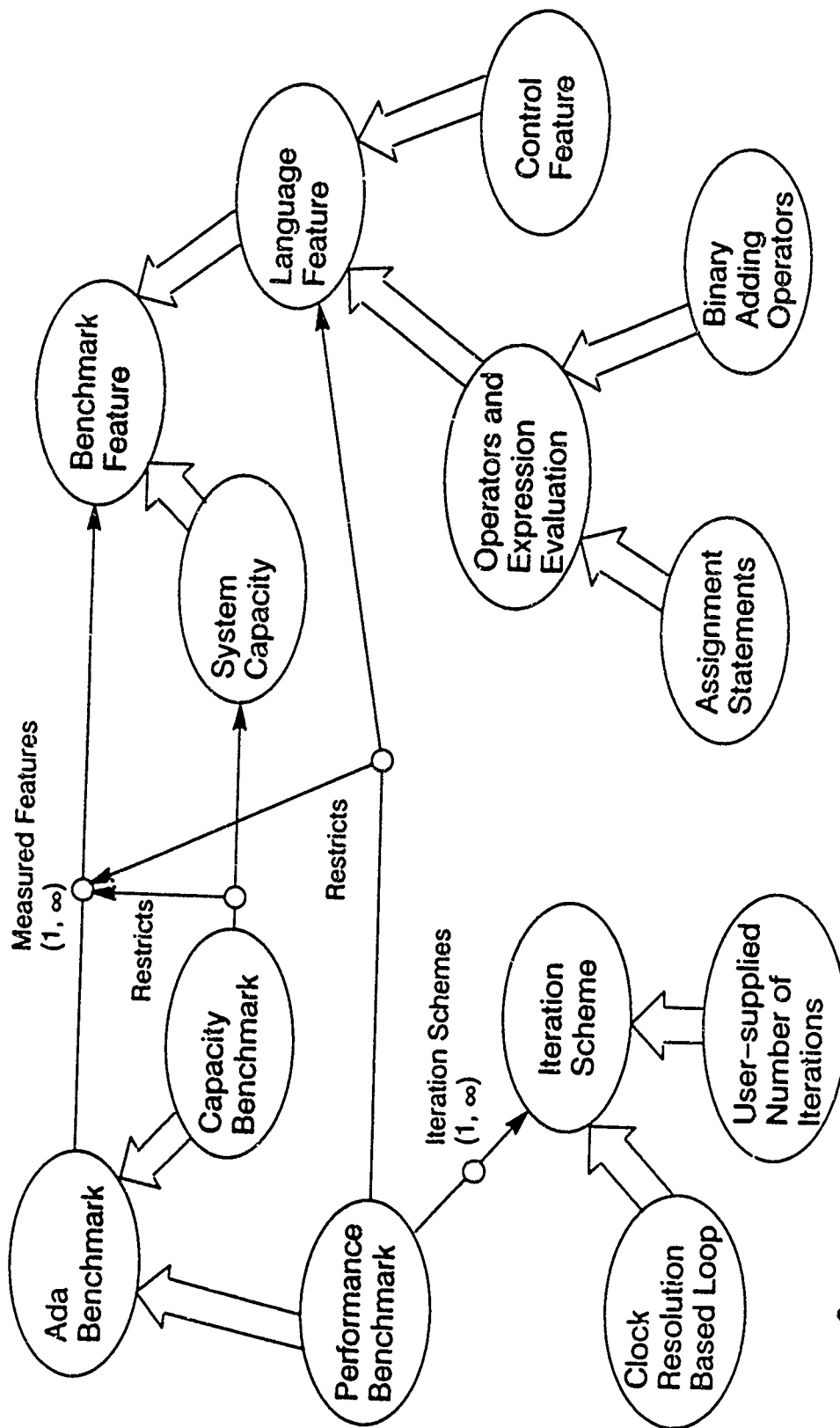
An Aggregation Description of a Benchmark



UNISYS

Defense Systems
PAO 06164_vul_890611-13

A Benchmark Structured Inheritance Network



UNISYS

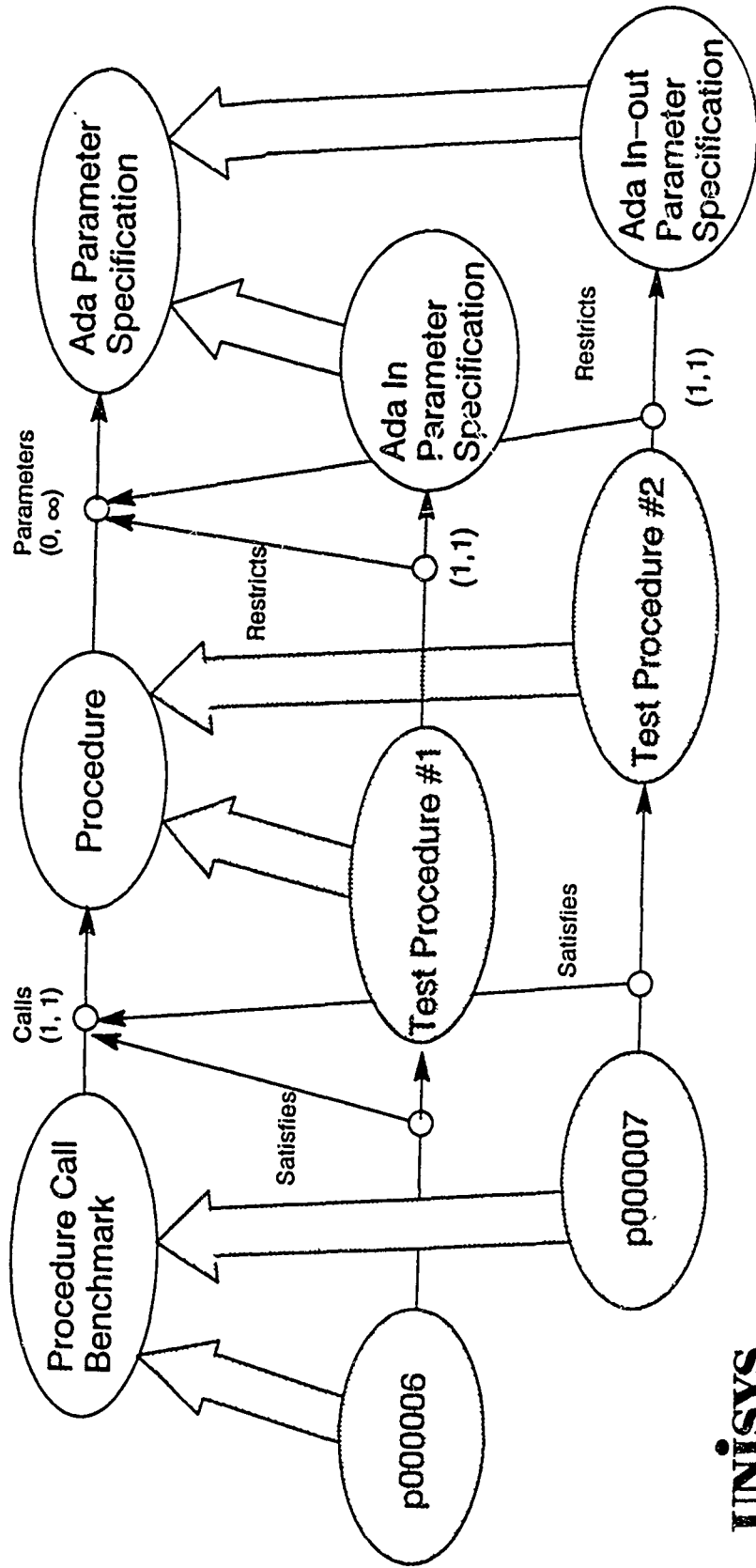
Benefits of Rule-Based Inference

- Tailor interaction to user expertise and user objective
- Highlight relevant information in network
- Provide unstructured knowledge

UNISYS

Defense Systems
PAO 06164_vu1_890811-15

Two Benchmarks with Remote Differences



Defense Systems
PAO d6164_vu1_890811-16

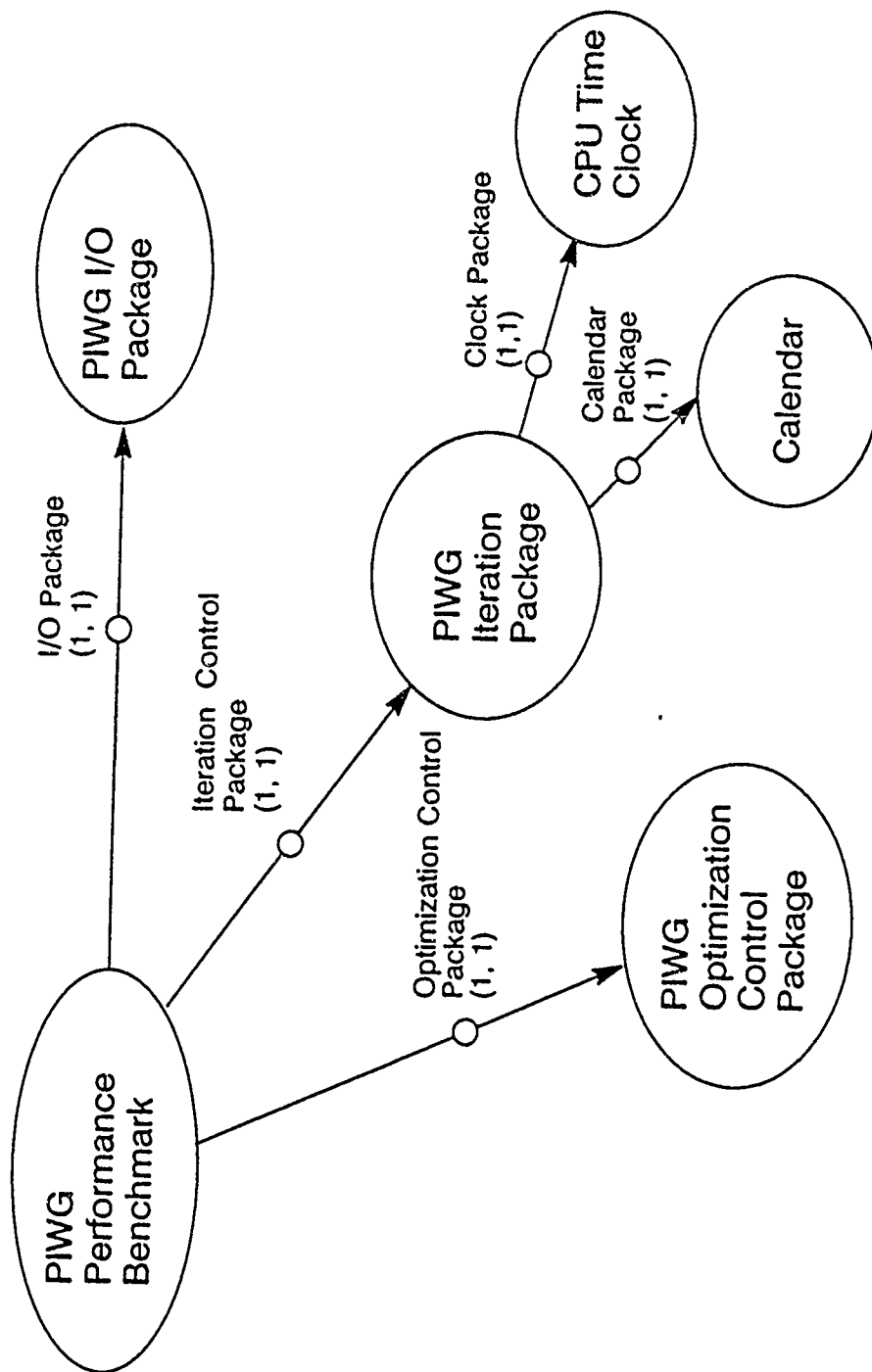
UNISYS

Future Enhancements

- Software configuration
 - Extract all subcomponents with chosen component
 - Support alternative implementations of components
- Classification
 - Automate component retrieval
 - Automate component insertion
- Repository maintenance
 - Monitor component usage
 - Partially automate taxonomy reorganization

UNISYS

Component Dependencies



UNISYS

Defense Systems
PAO d6164_vul_880811-18

Conclusions

- Knowledge representation systems are well-suited for repository organization
 - Structured inheritance networks provide combined advantages of other schemes without their disadvantages
 - Structured inheritance networks support advanced tools for repository use and management
 - Rule-based inferencers supplement structured inheritance networks and enhance the usability of the Librarian

UNISYS

Defense Systems
PAO d6164_vu1_890811-19

A SOFTWARE ASSISTANT FOR AUTOMATIC TEST EQUIPMENT ENGINEERING

David R. Harris
Sanders Associates, Inc
95 Canal St.
Nashua, NH 03061
(603) 885-9182

Mr. Harris joined Sanders Associates, Inc. in 1979 and has worked on intelligent engineering assistant programs for the past 6 years. Currently he is co-principal investigator on the Requirements/Specification Facet for KBSA. Prior to this assignment he helped develop the Test Engineer's Assistant, served as principal investigator for KBRA (The Requirements Assistant for KBSA), and has contributed to several automatic programming efforts within the Lockheed Corporation. His interests include knowledge representation, presentation architectures, requirements engineering, and intelligent automated support for engineering problem solving.

Abstract The Test Engineer's Assistant is a knowledge-based assistant - a machine junior partner to experienced engineers - for solving the problems of automatic test equipment configuration. In this paper, we consider a scenario of engineer-assistant interaction, we look at representation issues for domain artifacts, process knowledge, and requirements to code transformations, and finally we look at this assistant from the perspective of the Knowledge-Based Software Assistant paradigm.

1 Introduction

The applicability of Knowledge-Based Software Assistant (KBSA) [Green et al] technology to specific application domains is an important issue. We need to understand how KBSA general purpose capabilities match or improve on expectations and practices (manual and/or automated) of system/software engineers in specific domains. Codification of reusable domain knowledge, development of special purpose presentations, and considerations of domain-specific entry points for re-engineering all will help with technology transfer.

The Test Engineer's Assistant We can investigate these issues through the Test Engineer's Assistant (Tess), a knowledge-based assistant for the important area of automatic test equipment engineering. Automatic Test Equipment engineering is a very promising area for automation since engineers perform analysis mostly off-line with little machine assistance and often delayed feedback on the impact of engineering decisions. The Tess prototype system supplies machine assistance in several important ways which will be described in this paper. Tess is written in Socle [Harris-86], a Sanders developed knowledge representation and reasoning system, and runs on Symbolics 36xx computers. Although Tess development at Sanders preceded KBRA [Czuchry, Harris] development (KBRA is the KBSA requirements assistant), information on Tess has not been widely distributed.

and a retrospective report will help bring several technology transfer issues into focus.

The Application Domain Tess mediates the configuration - hardware/software selection and setup - of multi-purpose automatic test equipment (ATE) components to meet the testing needs of individual electronic countermeasures (ECM) systems. These countermeasures systems protect aircraft by either warning a pilot of a potential threat or emitting signals which actively mislead threatening radar systems. Testing ensures that ECM systems will detect and respond appropriately. Broadly, test set configuration plays a role in the rapid reprogramming of suites of systems that must be quickly modified to meet changing threat situations.

Highly skilled test engineers determine the testing needs for each countermeasures system under test (SUT). Unique configurations may be required for particular aircraft installation, antenna positioning, or mission.

Figure 1 below displays a high level view of this process as currently performed. In SUT partitioning, a requirements analysis task, engineers analyze the system to be tested along with the capabilities and limitations of state-of-the-art test equipment in order to determine configuration requirements. Later, engineers determine test order and specify the ATE hardware setup and the software configuration - parameterization of reusable software modules - needed for each test to be performed. In some cases additional SUT-specific software may need to be developed. (SUT-specific software development was not selected

as a task to be covered by the Tess prototype.) Finally mission data tables (MDT's) containing encoded parameterization data are created. This data tailors reusable software modules during ATE operation.

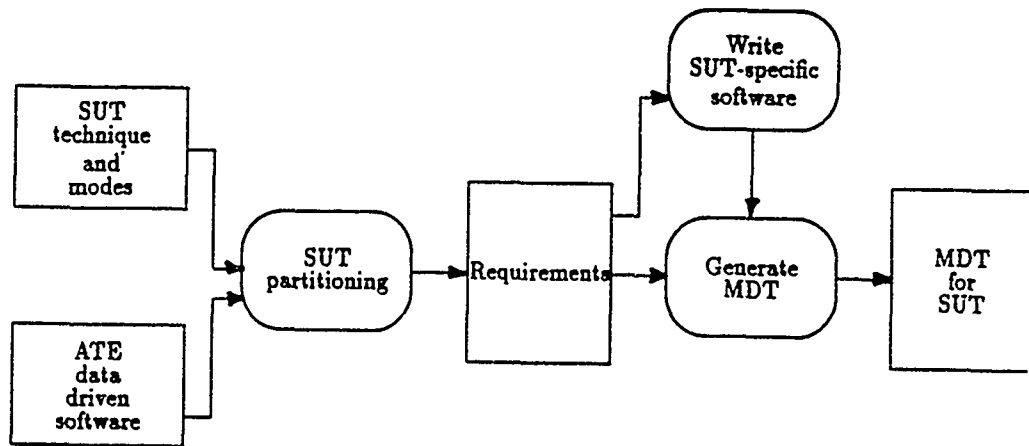


Figure 1: Test Configuration Process

Tess/KBSA Common Themes Tess formalizes engineering decision making and automatically derives implementations from high level specifications. Importantly, Tess plays the role of a junior partner to an experienced test engineer. As Tess designers, we were very concerned with the division of labor between the engineer and the junior partner. The engineer always controls the development and can deviate from standard practice when creative solutions to problems are required. Meanwhile, Tess makes extensive use of encoded knowledge of the testing domain in order to propagate ramifications of engineering decisions, check for consistency, and remind the engineer of available options and typical

practices. The codification of this knowledge was the most important segment of Tess development.

Tess includes a program generator which can be compared to transformational approaches envisioned for KBSA at large. This program generator transforms high level specifications into mission data tables.

This prototype leverages off of much of the same technology that supports KBRA, but covers the entire software life-cycle while providing only a portion of the broadly applicable services provided by KBRA.

2 A Scenario of Engineer-Tess Interaction

A brief scenario points out key Tess features and sets the stage for more detailed descriptions of the underlying inference capabilities of Tess. The screen images, showing work in progress, should give the reader a sense of actual Tess use.

2.1 Process Model

First, note that in the upper left hand corner of figure 2, a "Road Map" presents an intentional view of the process. That is to say, it shows a plan for completing the configuration task and indicates where the engineer is working within that plan.

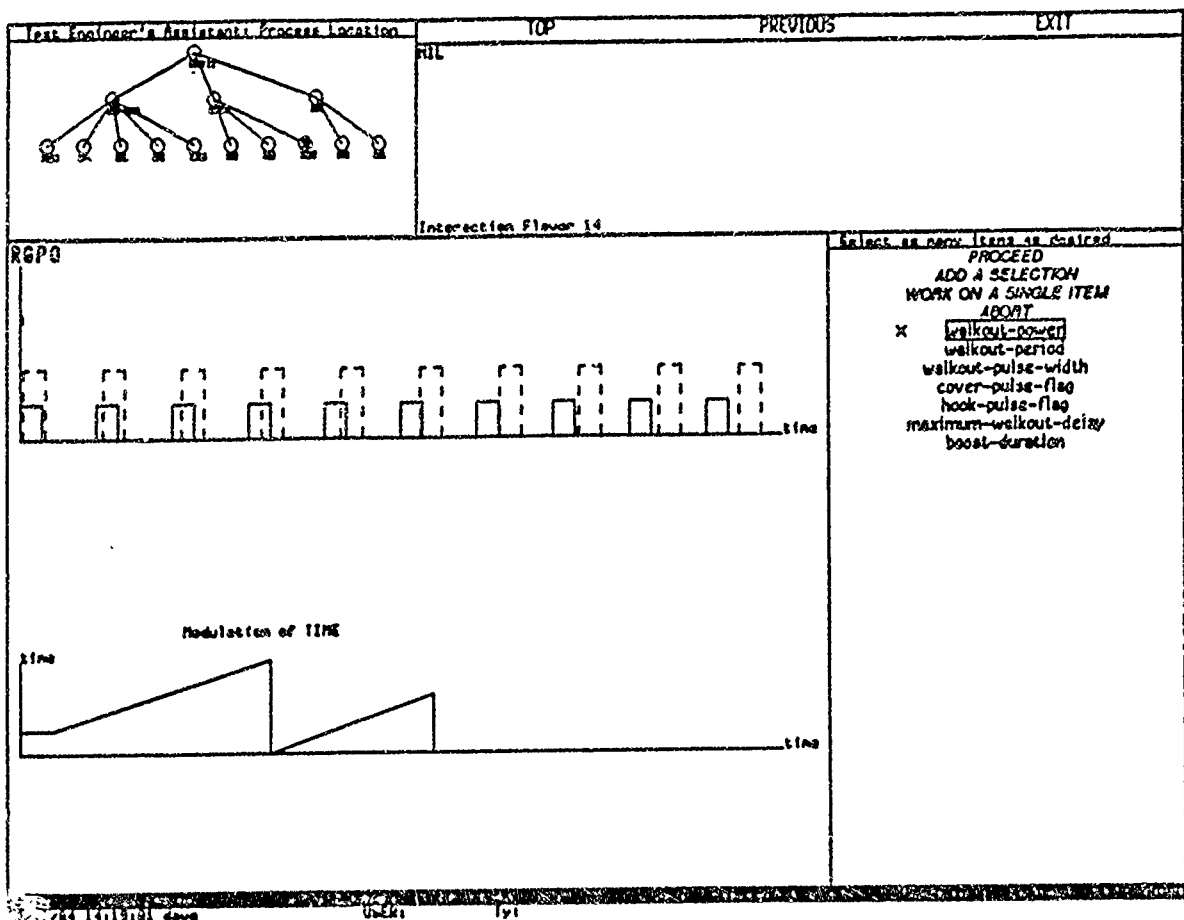


Figure 2: Requirements Acquisition Using Tess

The engineer can move immediately to any point on the map. For example, in order to work on definition of a test, the engineer has used the mouse pointing device to select "TEST" from the options available in the road map display. This action shifts the focus of attention to a test definition presentation. Note that the circle near "TEST" is darkened indicating this focus.

2.2 Dealing With Standard Engineering Artifacts

The lower pane of figure 2 contains a graphical reminder of engineering knowledge. In this example, the engineer studies a range gate pull off (RGPO) electronic countermeasures technique in order to determine testing requirements. The top graph shows a dotted pulse-train response superimposed over a fixed repetition interval radar pulse train. The dotted pulse repeats with an increasing time delay. This delay is plotted as a function of time in the bottom graph. The engineer works with this presentation to select parameters that are to be measured (i.e., parameters are selected from the mouse-sensitive items at the right of the screen) and defines the pass-fail criteria for each parameter. Tess supports this activity by supplying defaults and automatically deriving dependent parameters. For example, the walkout period is the sum of the initial dwell time plus the time of the walkout itself. The displays, default values, and relationships require knowledge of signal primitives, signal processing capabilities, and ECM techniques.

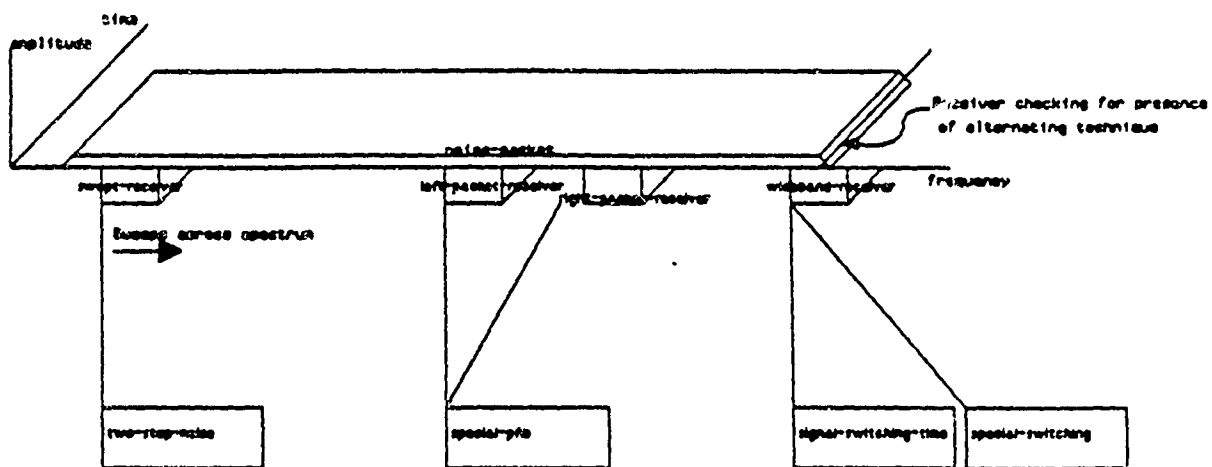


Figure 3: Measurement Specification

Specification evolution continues through engineer interactions with other displays. For example, when the engineer works on the measurement subsystem, Tess displays diagrams such as figure 3 above. Using this diagram, the engineer can indicate an intention to focus on the characteristics of a measurement receiver. Before any receiver entries are made by the engineer, Tess fills in values. Default values are provided (a default identifier for the receiver is set to 1), choices are listed (bandwidth might be set to 1 khz, 500 khz etc) and propagated values are recorded (in order to meet a 3000 usec processing time constraint, the receiver can only be stepped through 5 positions.) If the engineer adds the start and end frequencies, Tess derives the step size (i.e., $\text{step-size} = (\text{end} - \text{start}) / \text{number-of-steps}$).

2.3 Trade-off analysis

Later in the session, the engineer decides that 8 steps are required in order to adequately test the response characteristics. This conflicts with the processing time restriction. Tess informs the engineer of the conflict and identifies the options for retraction (i.e., the new 8 steps or the 3000 usec maximum time previously asserted). Assuming the engineer retracts the 3000 usec, Tess automatically updates values for the processing time and for the step size.

Tess initiative relies on the underlying constraint propagation technology that also underlies KBRA. The constraint inference engine captures dependencies and supports engineer-

ing problem solving by providing engineers with immediate feedback of ramifications of decisions and by allowing engineers to review dependencies and change their minds.

Subsequent to the development in the scenario, the engineer will continue to work flexibly until the test is completely defined. When the entire configuration is specified, Tess automatically generates executable MDT code which will run on the ATE.

3 Knowledge Realms

Requirements, specifications, and implementations are all stored in a central repository of knowledge-based objects. Frame inheritance, default reasoning, and constraint propagation maintain consistency and derive new facts as specifications are acquired. The knowledge base consists of a taxonomy of concepts managed by test engineers. When appropriate, default values (e.g., confidence levels, tolerance values for measurements, test names, test name formats) and domain formulas (e.g., physical relationships between parameters, interval algebra) are encoded.

Figure 4 indicates the top nodes in a taxonomy of Tess object types. Whenever possible, representations are test-set independent. In this way we guarantee the applicability of much of the Tess knowledge base generically for many countermeasures engineering problems. This reflects our vision for an on-line community memory of engineering practice.

APPLICATION ARTIFACTS	TRANSFORMATIONS	PROCESS	DATA STRUCTURES
Device	Viewpoint	SUT Partitioning	Table
Technique	Table-builder	Mission Data Reprogramg.	
Signal Primitive	Bitpack	Test Definition	
Test	Pointer		
Measurement			
Parameter			
Evaluation			

Figure 4: Taxonomy of Object Types in Tess

Test-set independent knowledge has been developed for countermeasures techniques, technique parameters, signal processing primitives (such as pulse trains, walkoffs, bandpasses, amplitude modulations), antennas, test name formats, receivers, interval arithmetic, and units used for dimensional analysis. A review of the representational approach for several of these object types will be informative.

3.1 Application Artifacts

3.1.1 Why is a technique more than a signal?

Figure 5 summarizes the internal description of the RGPO technique. Each bubble is labeled with an object type and a high level supertype (either technique, signal primitive, parameter, or interval). Reusable formulas are attached to these object types. For example, in interval objects min-pt, max-pt, center, and width are constrained by a midpoint formula. Arrows in the figure represent roles of an object type. For example, the figure contains an arrow which indicates that the transmit-signal role of RGPO is played by objects of type TIME-WALK-OFF.

Such technique representation presented us with some unusual challenges. Techniques can be organized along the following dimensions:

1. Graphical display.

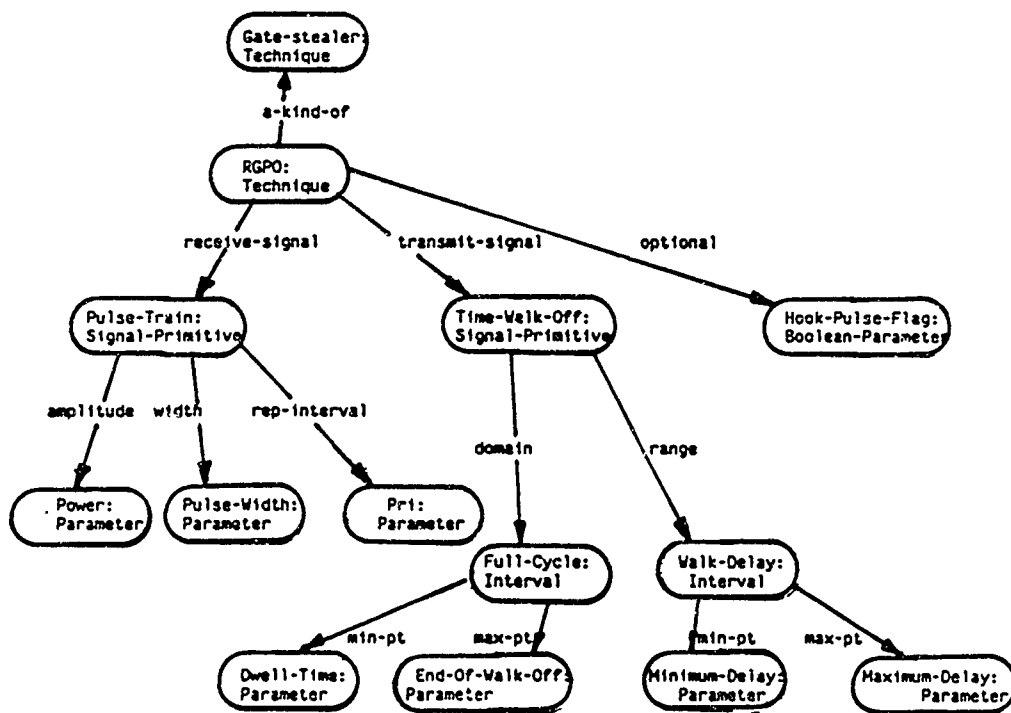


Figure 5: Technique Definition

2. Purpose of the technique

3. Identification of hardware and hardware parameters used to generate the signal.

4. Important parameter abstractions

Each of these dimensions are needed to support retrieval of information in flexible ways during specification evolution.

A simple mathematical representation is insufficient for two reasons. First, techniques are often best described by the relationships between the receive signal and the response signal. For example, the RGPO technique can best be understood by considering the walk-off characteristics of the generated pulse in response to the received pulse as baseline. Second, engineers think of transmit and receive signals as composed of more primitive signal cliches such as bandpass, pulse-train, and walk-off. These cliches are particularly useful in providing engineers with building blocks for description of new techniques. Tess supports this domain modeling activity. With Tess support, the engineer can describe new techniques through composition of Tess-presented cliches. The new techniques can be used in SUT partitioning or saved for use in other configuration problems.

In addition reusable formulas form another point of departure from straight forward representation. Formulas which interrelate parameters in a technique definition (e.g., center frequency = average of upper and lower) and formulas which relate parameters of two

different techniques (e.g., bandwidth of a noise technique is the total spectral coverage of a packet switching technique [i.e. noise packets are alternated in frequency to gain broader coverage]) are attached to object types and inherited by instances of the techniques. [Harris-88] contains a general discussion on the impact of formulaic knowledge on domain modeling.

3.1.2 Test Specification Knowledge

In Tess, a configuration specification is a collection of test object instances. Importantly, this approach to test specification provides for a fine-grained representation allowing required and optional parts to be specified in any order without forcing completion too early - completion at any level is not required before proceeding to the next. The encoded test object types include many examples of test, measurement setup, parameter, evaluation, and device.

MDT's are highly condensed tables which encode the specification information. Tess users do not interact with MDT descriptions. Tess helps engineers focus on the specification without concern for MDT representation.

Internally, Tess operates on MDT's at two levels - a knowledge level and an implementation level. At the knowledge level an MDT is a collection of table instances which profit from inheritance and default reasoning just like other knowledge-based objects. Important

attributes of these objects include table type, number of words, and octal word entries. Implementation level MDT's are generated from table objects through a straight forward process which resolves pointer information and stores words in arrays.

Importantly, MDT's represent an entry point for re-engineering associated with SUT testing needs. In typical practice test modules remain intact and basic table structures are stable. If ECM characteristics change dramatically, MDT's may need to be redefined to support more sophisticated processing. This redefinition occurs through modification of data layout language programs and support for MDT redefinition is an area of future growth for Tess.

3.2 Process Knowledge

The process knowledge realm contains information about what is required to complete a task and the likely order for achieving tasks. Stereotypical work practices are displayed to remind the engineer of where he or she is in a formalized process. Tess process knowledge includes heuristics such as SUT-specific work (e.g., establish fields for unique test names to be displayed to an ATE operator) should be completed prior to work on individual tests. Figure 6 displays each task with a subtask decomposition below it.

In the current prototype this process is hardwired in. In the future we will provide some automatic planning capabilities. We are currently using the DEVISER [Vere] system to

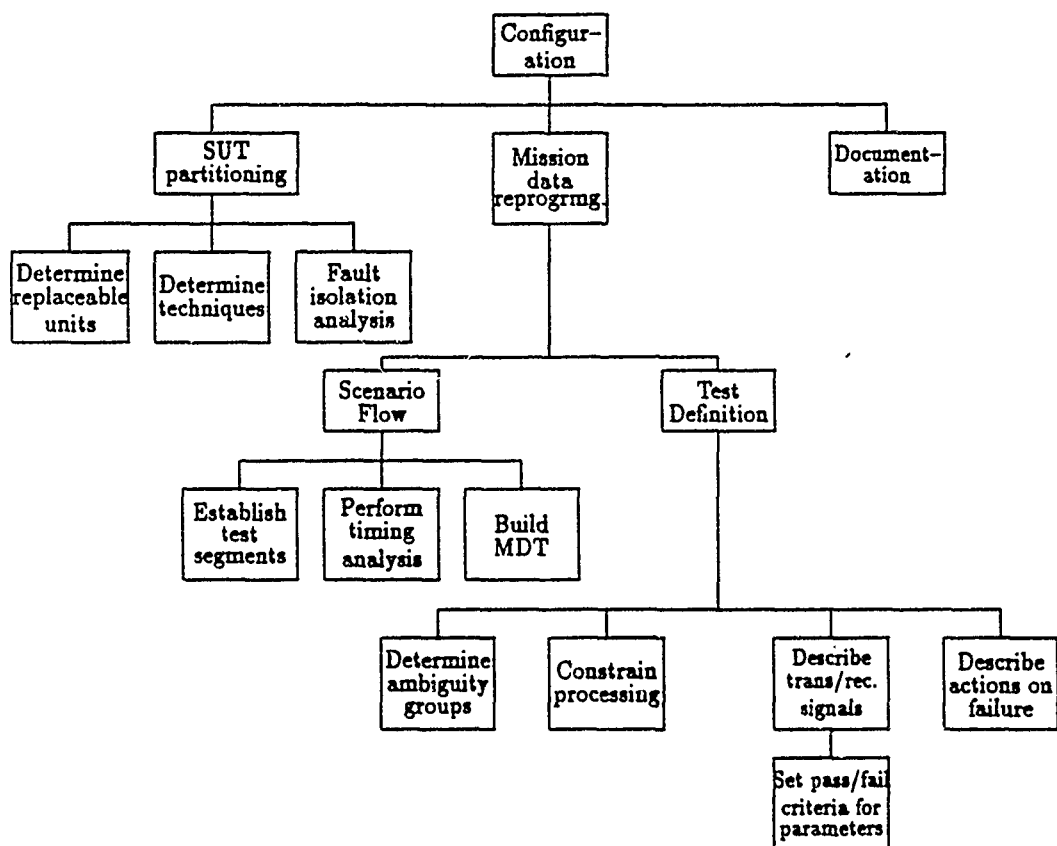


Figure 6: Process Model

explore AI planning techniques for the general requirements acquisition and analysis domain. Plans in this domain cover general problem solving strategies such as brainstorming, decomposition, selecting from alternative designs (the work reported on in [Potts, Bruns] is very relevant here), and negotiation to reconcile conflicts in work products. We are encouraged by the University of Massachusetts research on planning in the software engineering domain [Huff, Lesser] and by the high level editing commands of the KBSA specification assistant [Johnson, et al] and we believe that much of this work will be applicable to requirements engineering as well. For further development in the ATE domain, we anticipate expressing test engineering process knowledge in planning formalisms and relating it to the general problem solving strategies under development for general requirements acquisition

4 Transformations

There are two forms of transformations used in Tess. Viewpoints take requirements to specifications, and Data Layout Language (DLL) programs take specifications to MDT implementations.

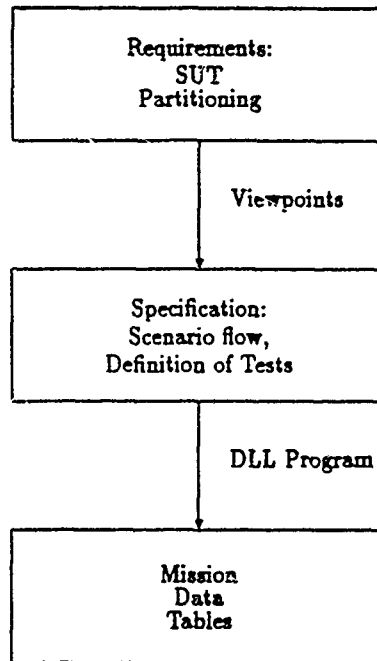


Figure 7: Transformations from requirements to code

4.1 Viewpoints

The relationship between requirement and specification is captured as the relationship between techniques and tests. While a test may be designed specifically for a single technique, it is often the case that technique to test relations are many-to-one on both directions. One test may measure characteristics of many techniques. A sequence of tests, sharing parameter measurements, may be required to fully test a single technique. A characterization of this is to think of each test imposes a viewpoint on the technique. That is to say, a test looks for specific features of a signal and may ignore important technique nuances entirely. The viewpoint defines precisely the parameters of the signal that are measurable by the test. For example, an amplitude-modulation test will look at the amplitude modulation of a packet switching technique while ignoring the interplay between the individual packets of the technique.

Viewpoint transformations represent this relationship. Each viewpoint is a function whose domain is the technique and whose range is the test. Both technique and test contain parameter lists which are related through viewpoint-specific constraint formulas. As illustrated in figure 8, a packet-noise-for-barrage-noise viewpoint equates center frequency of the spectral interval of the transmit signal to the set up frequency for the sensor of the measurement setup. Viewpoints are much like overlays in the Plan Calculus [Rich] of the Programmer's Apprentice. Overlays map between instances of plans - language

independent descriptions of programs - and are used to construct the refinement tree for a particular program. In comparison, viewpoints construct trees which are very shallow and stop at the specification level.

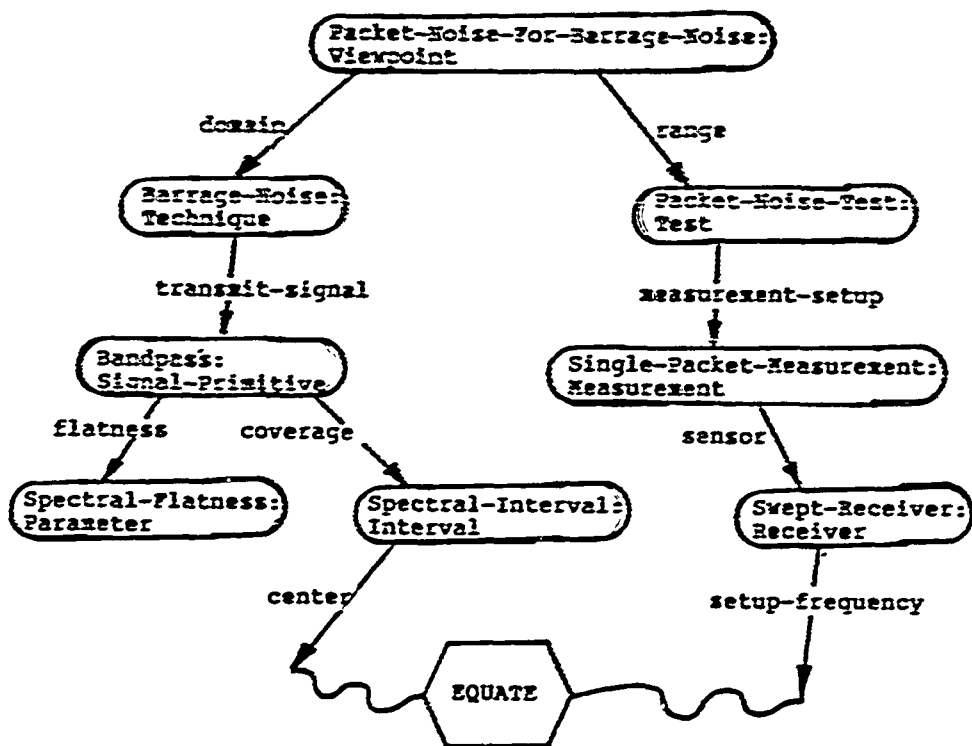


Figure 8: Viewpoint Example

Engineers take the initiative in viewpoint selection. While many heuristics are possible, Tess initiative would not substantially increase productivity and would likely be viewed as limiting the engineers ability to construct creative solutions.

The output of the viewpoint transformations is a collection of test instances. The specification is complete when the engineer and Tess fill in all required attributes of these instances.

4.2 Data Layout Language

The Data Layout Language (DLL) is a language for design of highly condensed variable length data structures such as MDT tables. Such data structures typically appear in embedded systems work. Declarative expressions for table word filling (bitpacked words, integers, multi-word records, etc.), locators (table number, word number, bit positions), and retrieval of filler values from the specification constitute the vocabulary of the language.

In Tess, a DLL program takes specifications and transforms them into the target MDT. The transformations operate on the representation of the specification and use knowledge of MDT data structures. Execution of this description results in building of knowledge-based table and record objects.

DLL is modeled on the Design Procedure Language (DPL) [Batali, Hartheimer] developed at MIT for designing large scale integrated circuits. DLL data table entries are analogous to DPL circuit component artifacts.

DLL addresses the need for engineering support at the MDT level. Declarative DLL table layout descriptions are Lisp code which when invoked retrieve data and generate the tables. Hence we have a merging of two functions which are conventionally separate. The vehicle for data base description and the vehicle for data base generation are one in the same. Any changes in MDT design automatically change the generator. A number of freedoms are provided for writing a DLL program. For example, locators are evaluated at table

building time and actual locations of words within tables or fields within words can be ignored when constructing tables.

Descriptions of DLL constructs Table definitions are shown in figure 9. Each includes a name, an optional list of local bindings, and a list of locator filler pairs. Locators indicate the starting word position in the table and the number of words that are being described. If the number is not provided, it is calculated from the filler. A wildcard argument, *, indicates that the filler should occupy the next available word.

Record definitions are expressed similarly and are often invoked by the FOR-EACH-OF-THE macro which builds a record for each entry in a list (e.g., TEST-RECORDS are built for each entry in TESTS when a SEQUENCE table is built).

Fillers are bitpacked words, integer words, extra precision multiple words, or entire records made up of several words. In addition a special pointer word holds the place for an array index which is computed when an implementation MDT is constructed. Fillers are computed by macros, such as BITPACK, which take locator filler pairs as arguments and return an octal number with encoded specification information stored in the appropriate fields - indicated by the starting bit and number of bits found in the locator. For example,

(bitpack (0 8) (>> bandwidth) (8 8) (>> center-frequency))

will return the octal number obtained by placing a receiver bandwidth in the lower 8 bits

(deftable sequence (&aux (tests (>> configuration tests)))

- 1 (bitpack (0 8) (>> test-id-words) (8 8) 0)
- 2 (integer (>> version-number))
- * (for-each-of-the tests (test-record)))

(defrecord test-record (&aux (id (>> test-id)))

- wd1 (characters *current-item*)
- wd2 (double-index-pointer (test-descriptor-block *current-item*))

(deftable test-descriptor-block (&aux (type 'fixed))

- wd3 (bitpack (st4 ln5) (>> retry-count) (st5 ln6) (>> failure-action))
- wd4 (pointer (pass-fail-criteria (>> parameters-to-measure domain)))
- wd5 (pointer (hardware-set-up (>> measurement))))

Figure 9: Interpretation written in DLL

and the center frequency in the upper 8 bits.

Tess assumes that specification properties are stored on frames, but the retrieval facilities are separable from the Data Layout Language and could be written to retrieve data from other knowledge base formats or directly from the engineer. This can be accomplished by redefining >>, a macro whose arguments point to a database location.

5 Conclusion

Support for ATE configuration requires knowledge-based approaches. Tess combines an inference mechanism and encoded knowledge of application artifacts, process, and transformations. Throughout an ATE configuration, Tess acts as a junior partner drawing on reusable descriptions of application artifacts for consistency checking and propagation

The Tess interface allows a user to work interactively with graphical displays, bring up graphical displays which serve as reminders of the engineering practice, review work that has been done, obtain explanations for facts, and be reminded of where in the configuration process he or she is now concentrating.

We have described many of the significant components including technique description, test components, process knowledge, viewpoint transformations, and the Data Layout Language.

Examining Tess as a domain-specific knowledge-based software assistant, suggests several concerns for KBSA technology transfer.

System level issues strongly interact with software development issues. Performance requirements for Tess-generated software are very dependent on the rapidly changing hardware capabilities of the ATE world. Significant knowledge-based assistance requires that we address system level knowledge within the KBSA paradigm.

Extensive domain modeling is required. Domain description that is roughly 5 times the size of the knowledge base in the Tess prototype or roughly 100 times the size of the Air Traffic Control knowledge base in KBRA will be required.

Presentations need to show the interconnections between system parameters at the level of domain knowledge. In KBRA, users establish non-functional properties through neutral spread-sheets. We need to do better. The technique and test diagrams which help Tess users visualize important data modeling and performance relationships indicate the form of engineering visualization that is required.

Process knowledge is important. Even in the limited Tess process formalism, this encoded knowledge is helpful. Important questions will arise as we try to place domain-specific process models into richer domain independent formalisms.

Acknowledgments Important contributions to the work described in this paper were made by Dr. Charles Rich of MIT's AI Lab and Sanders colleagues J. Terry Ginn and Lynne Higbie. I would also like to thank Richard Goller for many helpful suggestions on the paper.

6 References

1. Batali, Hartheimer, "The Design Procedure Language Manual", MIT/AI Memo 598, 1980.
2. Czuchry, Harris, "KBRA, A New Paradigm for Requirements Engineering", IEEE Expert, Nov. 1988.
3. Green, Luckham, Balzer, Cheatham, Rich, "Report on a Knowledge-based Software Assistant" RADC-TR-83-195, 1983.
4. Harris, D. "A Hybrid Structured-Object and Constraint Representation Language", Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, 1986.
5. Harris, D. "Issues on the Knowledge-based Requirements Assistant: Reusable Formulas", Proceedings of the 3rd Annual KBSA Conference, Utica, NY, 1988.
6. Huff, Lesser, "A Plan-based Intelligent Assistant That Supports the Process of Programming", Univ. of Mass at Amherst TR87-09, Sept 1987.
7. Johnson, Cohen, Feather, Kogan, Meyers, Yue, Balzer, "The Knowledge-Based Specification Assistant" RADC-TR-88-313, Final Technical Report, 1989.
8. Potts, Bruns, "Recording the Reasons for Design Decisions", Proceedings 10th Int. Conf. on Software Engineering, p418-427, 1987

9. Rich, "A Formal Representation of Plans in the Programmer's Apprentice", Proceedings 7th Int. Joint Conf. on Artificial Intelligence, Vancouver, Canada 1981.
10. Vere, "Planning in Time: Windows and Durations for Activities and Goals", IEEE Transactions on Pattern Analysis and Machine Intelligence, v5, p246-267, 1983.

KBSA FOR AUTOMATED SOFTWARE ANALYSIS, TEST GENERATION AND MANAGEMENT*

Gordon B. Kotik, President
Lawrence Z. Markosian, Vice President for Applications Development
Reasoning Systems, Inc.
3260 Hillview Avenue
Palo Alto, CA 94304

Tel.: (415) 494-6201
Internet: kotik@reasoning.com, zaven@reasoning.com

Author Biographies. Gordon Kotik and Lawrence Markosian are founders of Reasoning Systems, Inc. Before becoming President Mr. Kotik was Vice President for Product Development at Reasoning. Before the formation of Reasoning Mr. Kotik was one of the principal architects of the CHI knowledge-based programming environment at the Kestrel Institute. Mr. Kotik has supervised projects involving development and maintenance of submarine tracking software.

Prior to the formation of Reasoning, Mr. Markosian applied AI technology to DoD problems including data fusion, tactical air battle management and control system reconfiguration at Systems Control Technology. Previously Mr. Markosian was a Research Associate at Stanford University. Markosian is a member of the Association for Symbolic Logic, the American Association for Artificial Intelligence and the Association for Computing Machinery.

Abstract. We describe a KBSA-based approach to software analysis, management and test generation that incorporates several technologies: *object-oriented databases* and parsers for capturing and managing software; *pattern languages* for writing program templates and querying and analyzing a database of software; and *transformation rules* for automatically generating test cases based on the analysis results. We present a program transformation system, REFINE^{TM**}, that incorporates these and other technologies in an open environment for software test generation, and we demonstrate how REFINE is used for managing the software testing process. Finally, we present examples of how our approach is being applied to automatic test generation.

1 Overview

The literature describes a number of approaches to software analysis and test generation [1]. We describe how to automate such approaches by building a collection of software tools that represent and process software as network structures stored in a database. This

* Copyright (c) 1989 by Reasoning Systems, Inc. Scott Burson implemented the C Language System and the application to boundary value test generation discussed in this paper.

** REFINE is a trademark of Reasoning Systems, Inc.

representation captures the abstract structure of the software, which is a network of interacting pieces, and abstracts away from the details of the character string form of a program. Related objects of discourse in software analysis and testing include specifications, test suites, bug reports, related software (such as test drivers), and project plans, most of which also exhibit a network structure that can be captured with an appropriate database representation. Capturing the abstract structure of the software in a database facilitates building tools that automate analysis and testing.

Typical analysis and testing tasks might include the following:

- Analysis: If I make a certain change to the program, which existing tests must be modified or deleted and what new tests must be added? From the results of integration testing, which modules are most likely to contain bugs?
- Testing: Create test cases directly from programs.

Software-related objects are usually represented as groups of text files in a variety of languages, both formal and informal. The links in the network (e.g., T is the test suite for program P) are largely informal and implicit ("Most of the short files with extension DAT in subdirectory HARRY and written prior to 11/88 contain integration tests for the TRANSACT system"). The programmer typically analyzes and modifies these objects using file-based tools—text editors, string searches, etc.

Hence, software testers must map their conceptual universe, a complex *network of structured objects*, onto a *hierarchical file system* and use file-based utilities to pursue the conceptual links. The structural mismatch between the conceptual model and the file representation makes it extremely time-consuming to answer even simple questions like "who last modified this function" or "where are the test cases for this program".

Overall, the problems are similar to what would be encountered trying to use a bitmap editor like MacPaint as a CAD tool—an array of bits holds all the information, but the information cannot readily be extracted by automation tools.

In the maintenance phase of the software lifecycle, the problems are compounded. The engineers who created the original mapping of the project onto a file system might have

left the company. With them went the rationale and the details of the mapping, as well as key information about the individual programs, test suites, etc. Bug fixes start to introduce more bugs than they fix. It becomes successively more difficult to maintain the consistency among programs, their test cases and documentation.

Our approach to software analysis, test generation and management is to use an object-oriented database representation for programs and other software engineering objects and to build tools to query, analyze and transform this database.

The approach described here builds on the experience of many previous systems and is closely related to a number of current efforts. Many Lisp programming environments have been built around a representation for Lisp programs as list structures in virtual memory. The prime example is the Interlisp programming environment developed at Xerox PARC [2]. Interlisp provided many analysis tools for software, including Masterscope, an interactive query system for program analysis.

Language-based environments introduced the idea of building general programming environments that could be customized to a particular programming language [3].

We illustrate our approach with examples of automatic generation of test suites.

In Section 2 we review alternative methods for representing software. Section 3 presents REFINE, an enabling technology for software analysis and testing that uses an object-oriented database for representing software and transformation rules for modifying software in the database. In Section 4 we examine examples of analysis, mutation generation, test management and test generation for C software. Section 5 summarizes our results.

2 Alternatives For Software Representation

One basic reason why analysis and testing of software using file-oriented tools is expensive and unreliable is that source text is a poor data structure for representing programs and complex data. As a result, compilers have to parse and analyze the source

text to create a better representation, namely abstract syntax trees, symbol tables, and other data structures that capture the abstract relationships among program units. The compiler then performs further analysis (e.g., type checking) and transformation (e.g., code generation) on the internal representation. The style of program representation used by compilers is an improvement over source files, but it still has drawbacks because the representation is not designed for use by tools other than the compiler.

Some of the problems of compiler data structures (lack of persistence, browsers and query languages) may be solved by relational databases. However, relational databases are unsuitable for program manipulation because of (a) the limits of the relational data model and (b) their inefficiency in executing the graph traversal algorithms that occur so frequently in program manipulation [4].

Object-Oriented Databases

Object-oriented databases are an emerging technology that is expected to play an important role in engineering database applications. They preserve some of the advantages of relational databases (persistence, browsers, and, to some extent, query languages). They offer much more powerful data models that directly support network manipulation with features such as multi-valued slots, inheritance and constraint maintenance. Furthermore, their efficiency characteristics are better-suited than those of relational databases to fast manipulation of complex graph structures.

A practical system for software analysis and testing needs the following extensions to object-oriented databases:

- The data definition and query languages should support the mathematical abstractions used in high-level program representation (set notation, first order logic, tree comparison).
- Tools must be provided for parsing source files into the database and printing the database back to source files.
- A syntactic pattern-matching capability for describing programs in terms of

templates should be provided.

- The query language should support a rule-based programming paradigm for specifying transformations of software and software-related objects.

The next section describes how the REFINE™ knowledge-based software development system retains the benefits of object-oriented databases while providing the extensions listed above. Thus we show how REFINE can serve as a technological foundation for a new generation of software analysis and testing tools.

3 REFINE: An enabling technology for software analysis and testing

REFINE [5] is an interactive software development system that integrates three key tools to provide a basis for software analysis and testing:

- a wide-spectrum, very-high-level specification language;
- an object-oriented database that provides the necessary abstractions listed in Section 2;
- a language processing system that accepts definitions of programming languages and produces syntax tools (parsers, printers and pattern matchers)

Programs are converted between source file and the object-oriented database using the parsers and printers created by the language processing system. Thus the database is fully integrated with conventional file-based systems and tools. The REFINE object system and other high-level data types in the specification language (sets, sequences, maps, etc.) support a data model for software objects that is very close to the standard conceptual view of annotated abstract syntax trees. Tools that analyze and transform software in the database are written in the REFINE specification language, which provides mechanisms for template-based program description and rule-based program

transformation.

3.1 The REFINE specification language

The REFINE specification language (also called REFINE) is a very-high-level, wide-spectrum language that supports a variety of specification techniques including set-theoretic data types, first order logic, rules, object-oriented and procedural programming. The specification language is used as the query/update language for the database. The compiler for the specification language is implemented as a rule-based program transformation system. The current version of the compiler generates Common Lisp. The compiler and most of the rest of REFINE are written in REFINE.

3.2 Object-oriented database

The REFINE database provides persistent storage of objects created using the object-oriented part of the REFINE specification language. It includes mechanisms for version control, multiple users with concurrency control, computed attributes, and constraint maintenance. The database is used to manage networks of software objects including specifications, code, documents and test cases. It is also used as a repository for application-specific data.

3.3 Interactive graphics

REFINE contains an interactive graphics package that allows quick development of customized end-user interfaces. The graphics system is built on the X 11 window system and supports multiple viewports onto surfaces, graphical editing and layout of object-oriented database structures using different styles of icons and links, and mouse-sensitive text.

The REFINE graphics package has been used to build end-user interfaces for specifying communication protocols and CASE tools.

3.4 REFINE Language processing system

The REFINE language processing system takes as input a description of a language in the form of a grammar. It produces a parser, printer, pattern-matcher and mouse-sensitive text browser for the language. The language processing system is an extension of LALR(1) parser generator technology. Grammars are written using a high-level syntax description language that includes

- regular right-part operators
- precedence tables
- semantic actions of productions
- a mechanism for specifying lexical analyzers.

In addition, REFINE provides a capability for defining *program templates*. These templates can be used—

- in pattern matching, to test whether an existing program is an instance of a template, and
- in pattern instantiation, to build a new program that is an instance of the template.

Use of templates in program analysis and transformation applications makes the application code clearer and much shorter—frequently an order of magnitude shorter than the hand-coded equivalent. Examples are provided in the next section.

The language processing system has been used to build software management tools for a number of languages, including REFINE itself, C, SQL, IBM JCL, COBOL, Ada, SDL and NATURAL. These tools have been used for applications including automated software maintenance, re-engineering, code generation, software test generation and test management.

4 Examples

We now turn to examples of using an object-oriented database and associated language-based tools in software analysis and testing. In addition to the examples discussed in this section, the approach has been applied to path generation and analysis for specifications of communication protocols [7].

4.1 Analysis Examples

The use of an abstract syntax-based representation of software for analysis is fairly common in modern approaches to software engineering. Most software analysis tools build such a model as a preliminary to the actual analysis phase. Perry summarizes some of the important uses of abstract syntax tree analysis in software development in [6]. The analysis capabilities and applications described here can be regarded as extensions of these common analyses in that they take advantage of the novel features of the database model of software, namely:

- an open, user-extensible, object-oriented representation of abstract syntax
- use of high-level data types and query operations
- integration of abstract syntax trees with other software objects in the database such as documentation, test cases, and bug reports

The goal is to allow a wider class of analyses, including queries that reference diverse software attributes (not just the abstract syntax), and to make it much easier to specify routine analyses such as cross-reference listings.

We illustrate database queries for program analysis with an example taken from software testing—assume all the functions in a module have been run through the test suite and that a large number of functions failed. We would like to pinpoint the incorrect function or functions that caused the failures. Suppose we know, in addition, that the suspect functions were probably maintained by Bill. In modern programming systems, finding a function is made easier by the existence of structuring mechanisms such as hierarchical file structure, modules, and class hierarchies in object-oriented programming systems.

However, when the volume of programs grows very large, these mechanisms are not good enough—they are based either on file-search commands, fixed indexings of the software (libraries), or interactive browsers. They fail to let the user bring to bear all the available information about the hypothetical defective code. For example, the user may know such things as:

- the probable author or authors of the code,
- data structures that are probably used in the code and
- approximately when the code was written.

The query language for the object-oriented database allows all available information to be used in general searches through the software database. For example, REFINER stores the abstract syntax trees for the user's specifications annotated with information such as—

- when the program was last modified, and by whom;
- what diagnostics were issued by the compiler;
- what other programs it uses and is used by;
- what test suites are available for it;
- the complete testing history;
- what documents describe the program.

Since the representation of programs is open and extensible, other relationships can be easily added. For example, a user can add an attribute that stores, for each program, the name of the person responsible for maintaining that program. This attribute could then be used in queries over the software database, as in the following example.

In a REFINER-like system, you could write a query like the one below to find a set of candidate functions responsible for the test failures:

```
{ f | f in application-functions &
  maintainer(f) = 'Bill &
  failed-test-suite(f) &
  ~ exists(g) (g in application-functions & f calls g &
    not failed-test-suite(g)) }
```

The above query finds all functions in the application that are maintained by Bill, failed the test suite and do not call any other function in the application that failed the test suite.

To make a facility like this more accessible to programmers, REFINe provides a graphical user interface toolkit that can be used for specifying queries to replace the above notation. The notation above could then be used as an accelerator for experienced users, or for cases when the graphical interface was not powerful enough.

4.2 Test management examples

We outline RETESTS, a test management system developed in REFINe and used internally at Reasoning Systems for managing testing of new software releases. RETESTS is a prototype general-purpose testing system designed to support testing of applications written in REFINe. RETESTS provides the following capabilities:

- a methodology for structuring test cases;
- a method of running collections of test cases, recording the results, and resuming from abnormal events such as runtime errors and infinite loops; and
- report generation for summarizing the results of testing.

RETESTS has been used at Reasoning Systems since 1985. A typical use of the system is to develop a battery of test cases (called a "test suite") for an application and to run the entire test suite both on a regular basis and also before events such as system releases.

Structuring tests. RETESTS models testing-related information (test cases, reports, etc.) using REFINe object classes and attributes. RETESTS also provides a grammar for its object classes so that test cases can be specified and viewed textually. Object classes include—

```
test-item:  a test case and related information
test-result: the result of running a test-item
test-harness: a collection of related test-item
test-suite: a collection of test-harness
```


`system-version:` information on the software version under test
`test-report:` a summary of the results of executing a test-suite.

The lowest level of test case information is a single test case, which tests a specific behavior of the application. For example, a test case might say that a function called quick-sort, when applied to the sequence [3, 2, 1] should return the sequence [1, 2, 3]. When RETESTS runs an individual test case, it builds an object that represents the result of executing that test case. This object stores information such as whether the test case succeeded, and if not, how it failed (incorrect result, stack overflow, etc).

The next level structure is `test-harness`, which groups together a sequence of related test cases. This is provided both for convenience in grouping related test cases and to allow factoring common parts of test cases.

Finally, the `test-suite` for a complete application is a sequence of test harnesses together with the information common to the harnesses such as the directory in which reports should be placed.

Running tests and reporting results. RETESTS allows the user to specify the function to be used to conduct tests, the environment in which the tests are to be run and the predicate used to evaluate the success or failure of a test.

```
test-harness Test-Arithmetic-Functions
  preamble Load-Sort-Functions
  program-for-test Parse-Compile-Execute
  comparison-predicate Boolean-Equal
  test-items
    a-test-item
      test-datum "sort-1([1, 1, 0]) = [0, 1, 1]"
      correct-outcome True
    a-test-item
      test-datum "abs(3.14159265 / 180.0 - .01745329)
                  < .00000001"
      correct-outcome True
```

In this example of a test harness, the function `Load-Sort-Functions` is first executed; this is a user-defined REFINE function that presumably loads the functions to be tested, including `sort-1` into the environment. The predefined REFINE function `Parse-Compile-Execute` is used to conduct the tests, and `Boolean-Equal` is used to evaluate the result of the tests.

Reports are written to the directory specified as the `REPORTS-DIRECTORY` attribute of the test suite containing this test harness.

4.3 Transformation example: Generating test cases automatically from software

This example focuses on perhaps the most novel capability of the REFINE system—specifying and automatically executing transformation rules that perform complex modifications to software. This is the heart of providing automation for software testing activities including automatic derivation of test cases. Many of the analysis activities discussed earlier are performed with the goal of determining where or how subsequent modifications to the software should be made, or what tests should be generated. The examples apply to C software and make use of the C Language System, a customization of the REFINE system for representing C programs.

The literature includes a number of approaches to generating test cases. We illustrate the program analysis and transformation approach to test generation by applying it to generating “boundary value” test cases for functions. The purpose of boundary value testing is to determine whether the function works correctly for extremal values of its arguments. Typical boundary values for a function whose argument is an integer are the smallest integer representable on the machine, -2, -1, 0, 1, 2 and the largest representable integer.

The example transformation rules look at the data type of each argument of a function and generate a test driver that calls that function with boundary values appropriate to the data type. For example, suppose we have a file containing the following C definition of the

factorial function:

```
int
fact(n)
    int n;
{   if (n < 2) return 1;
    else return n * fact(n-1);}
```

Then the following test driver function is generated automatically:

```
test_fact ( )
{   {int n;
    int intsl [] = {-2147483648, -2, -1, 0, 1, 2,
                    2147483647 };
    for ( il = 0;
        il < sizeof intsl / sizeof *intsl;
        ++ il )
        { n = intsl [il]; fact(n)}}
```

Several patterns and transformation rules are used to generate the test driver from the function. For example, the function `Test-Values-Declaration` shown below generates a C declaration based on the data type of the parameter. The function is specified using several assertions, each of which is used to compute the value of the function for a specified data type (only the assertions for integer and floating point data types are shown).

```
function Test-Values-Declaration(param-type, values-var):
    Declaration
```

computed-using

```
Test-Values-Declaration('int', values-var) =
    'int @values-var[] =
        {-2147483648, -2, -1, 0, 1, 2, 2147483647};',
```

```

Test-Values-Declaration('float', values-var) =
  'float @values-var[] =
    { -3.4028232e+38, -1.1754944e-38, 0.0,
      1.1754944e-38, 3.4028232e+38};'

```

Other functions are used to generate the test loop and the complete testing function shown above.

5 Summary and conclusion

We observed that software analysis and testing requires two broad categories of activities: *analyzing* and *transforming* programs and related objects. We have described an approach to software analysis and testing based on

- an object-oriented database representation for software lifecycle objects and
- automated transformation of the objects represented in this database.

We have found that the object-oriented database representation more closely approximates the conceptual model held by developers than is possible with a text file-based system. Analysis and transformation can be significantly automated by tools that take advantage of the database representation. We have described REFINE, an environment for program representation and transformation that provides the underlying framework needed for software analysis and testing. We have described a tool, RETESTS, built in this framework, that is used to manage testing REFINE applications. We have further illustrated our approach with an example of automatic test data generation.

The ability to support automation for test generation and management for large software systems by using rule-based transformation is a key innovation of our approach that distinguishes it from systems that focus only on automation of program analysis. The features required to support this transformational technique require substantial extensions to an object-oriented database system to support efficient representation of programs and the ability to convert easily between the text and database representations.

The transformational approach to software development was first developed for synthesizing code from high-level specifications, but its range of applicability now appears to be much larger. It may well be that this technology will make its first significant impact on software engineering practices in the areas of analysis and testing rather than code synthesis.

References

- [1] *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*. IEEE Computer Society Press, Washington, D.C., 1988
- [2] Teitelman, W. and Masinter, L. "The Interlisp Programming Environment," *Computer*, 14 (4), pp 25-34
- [3] Reps, T. *Generating Language-Based Environments*, MIT Press, Cambridge, MA, 1984
- [4] Linton, M. "Implementing Relational Views of Programs," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium*, The Association for Computing Machinery, NY, 1984
- [5] *The REFINE User's Guide*. Reasoning Systems, Palo Alto, CA, 1985
- [6] Perry, D. "Software Interconnection Models," *Ninth International Conference on Software Engineering*, IEEE Computer Society Press, 1987
- [7] Markosian, L.Z. and Riemenschneider, R.A. "Automatic generation of conformance tests using REFINE", Reasoning Systems, Inc., 1986

SYSTEM INTEGRATION TECHNOLOGY

Laszlo A. Belady

Vice President & Program Director

Microelectronics and Computer Technology Corporation (MCC)

The greatest opportunity in software for the U.S. and so far the greatest investment in talent and experience, lies in the ever increasing integration of computer applications. There is a definite trend to interconnect islands of applications through communication lines within an enterprise as well as among different ones.

The computerization of a modern enterprise, therefore, becomes a single large application comprising business development, manufacturing and marketing. These large applications are created by cleverly interconnecting the raw material which is a set of hardware and software boxes. This set may include hardware pieces such as workstations, main-frame and minicomputers and time-proven software packages such as payroll, numerical control, for manufacturing and decision-making software. The remarkable thing about this is that the software is the glue which holds these boxes together and also brings the dynamic to the system by controlling the traffic as well as the data flow among the different nodes.

Due to their very nature, these future applications systems are all customized and cannot possibly be developed as standard packages serving more than one enterprise - they must be built individually. The team that builds these systems must consist not only of experts in computer hardware and software but also of experts in the particular business (or domain) for which the system is being built.

As a result of this trend, the software business will probably split into two major categories: 1) the component business developing sometimes quite large software packages to be marketed worldwide in different large applications systems and 2) the system integration software using the components. While the former business will become international and very competitive, the U.S. has a definite built-in advantage for developing the latter that should be exploited and emphasized. Education, on-the-job training and research should all support this model. Computer science education should be the basis for superb craftsmanship of software componentry, while the future engineers of integrated applications must become system engineers who are as familiar with hardware as they are with software. At the same time, research must focus on the computer-aiding of the system development process, which includes interdisciplinary cooperative work.

In support of some of the above, the major objective of the research efforts in MCC's Software Technology Program is to computer aid the complex, software intensive, system integration and design process.

Gaia: An Object-Oriented Framework for an Ada Environment

**Don Vines and Tim King
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, MN 55418**

**For Paper See: Proceedings of the Third International IEEE Conference on AdaTM
Applications and Environment, The Sheraton-Wayfarer Inn and Conference Center,
Manchester, New Hampshire, May 23-25, 1988, Pages 81-90.**

KBSA'S REQUIREMENTS ASSISTANT AND AEROSPACE INDUSTRY NEEDS

Douglas A. Abbott, Ph.D.
McDonnell Douglas Corporation
MCAIR Dept 052
P.O. Box 516
St Louis, MO 63166-0516

(314) 895-1405

Dr. Abbott holds a Ph.D. in meteorology from Florida State University. He has done extensive work on predictive modeling of atmospheric flows using techniques such as numerical solution of the Navier Stokes equations and advanced statistical methods. Recent interests include identification and evaluation of high productivity software engineering environments. Prior to joining McDonnell Douglas, Dr. Abbott served 24 years in the U.S. Air Force, retiring in 1988 in the grade of Colonel.

Upon receipt of the Knowledge-Based Software Assistant (KBSA) Requirements Assistant (RA) software, developed by Sanders (Lockheed) Associates under Phase I of the KBSA proof of concept deliveries, we formulated a plan to evaluate RA's technology transition potential, within the aerospace industry. Our plan began with software familiarization and culminated with building a workplace prototype. When we encountered problems with some planned tasks, we chose not to attempt a workplace prototype. RA's view of the evolving system differed from ours and we were not successful in reconciling these views. In fairness to RA, we point out that we treated it as a finished product even though the developer recommended a different approach. However, our approach was useful for comparing RA to Computer-Aided Software Engineering (CASE) products. RA closely parallels the structured analysis of requirements function within CASE tools. RA's departure from CASE tools lies in use of artificial intelligence (AI) and knowledge-base support to aid structured analysis. This paper enumerates the difficulties encountered while using this early version of RA and suggests directions for continued research, focusing on aerospace industry needs. Our principal finding is a conviction that the research goal, of supporting software development outside the constraints of a formal method, should be reassessed. We think the only merit of this goal is that no method equates to an ability to support any method, a flexibility which has both positive and negative qualities. However, our primary concern is that adherence to this goal competes with (rather than augments) the more fundamental goal of developing effective AI support. On another tack, RA assumes that the user is the system developer. This assumption overlooks the process of contracting for systems and the associated need for formal requirements documents. A more likely scenario for the nineties is early user involvement through simulation/prototyping rather than user development. Under this scenario, RA needs the capability to scan a requirements document into the notepad and analyze it to produce user oriented simulations.

1. INTRODUCTION: The goal of the Rome Air Development Center (RADC) KBSA project is to demonstrate proof of concept for a new software engineering paradigm capable of increasing the productivity of software engineers by an order of magnitude. The KBSA paradigm is being developed in three phases, each phase building incrementally on the lessons of the earlier phases. Phase I has been completed and Phase II is just commencing.

Phase I software consists of five loosely coupled software components, or facets. These are the RA, the Specification Assistant, the Framework, the Performance Assistant, and the Program Management Assistant.[3] Discussion, here, will be limited to RA, the only facet we have evaluated to date.

To appreciate the support provided by the RA facet of KBSA, it is useful to consider comparable features found within CASE tools which typically follow the "Waterfall" software development paradigm. RA corresponds closely to the Structured Analysis of Requirements phase of CASE tool support. RA research focuses on potential applications of AI and knowledge base capabilities to support the task of structured analysis of requirements. The concept is for the RA software to support an engineer, whose primary field is not software, independent of the formal methodologies typically imposed within CASE tool structured analysis implementations.[2,8]

Before presenting our evaluation procedures and results, we would like to address some factors which shaped this report. First, user documentation, in a readily usable form, does not exist. While this impeded our evaluation of RA, we believe that we gained sufficient familiarity with the software to avoid

erroneous conclusions. Since RA is proof of concept research software generated under a level of effort contract, we did not attempt to rigorously document weaknesses in the RA user interface. Understandably, a polished user interface was not high among research priorities. However, we do present some concerns in this area which may impinge on the philosophy of the development. Finally, we would like to say that it was not our intent to criticize this research. We believe candor, offered in a constructive spirit, provides the most effective form of communication. We hope our inputs contribute to shaping the implementation of the KBSA paradigm to better meet user needs.

2. RA CONCEPTS SUMMARIZED: The underlying philosophy of RA is that the supported engineer will not be constrained to work within a specified methodology. Rather, RA will support individual work preferences. System requirements are entered using any of several different presentation formats. All formats contribute to a single knowledge-based data base, and system development conducted within any presentation format will be reflected in the other formats. As system requirements take shape, a "smart" software assistant critiques new entries and brings inconsistencies to the developer's attention (an active role) or intervenes upon request (a passive role) to perform assigned tasks. The presentation formats, coupled with a brief description of their respective purposes, follow.[5]

- Intelligent Notepad: An on-line notebook for impromptu ideas. It interprets requirement entries and creates system objects or failing interpretation, captures the information.

- Context Diagram: A standard, graphical representation of the system with its environment interfaces.

- System Function Diagram: A hierarchal, graphical view of system level functionality.

- Internal Interface Diagram: A graphical display of the system configuration (hardware and software) interfaces.

- Functional Decomposition Diagram: A hierarchal graphical breakdown of system functionality.

- Data Flow Diagram: A graphical model of flow of data through the system.

- State Transition Diagram: A graphical display of system states and the events which trigger changes of states.

- Activation Tables: A graphical display of the active system functions, by state.

- Calculator/Spreadsheet: The calculator is a tool for making entries into the spreadsheet. The spreadsheet provides a matrix view of the non-functional properties or constraints for each system function.

- Requirements Document: A formal requirements document based upon information entered via other presentation formats.

3. EVALUATION PROCEDURES: We attended the familiarization training provided by the RA developer, Sanders (Lockheed) Associates. RA training consisted of demonstrations of RA functionality within the air traffic control (ATC) domain, intertwined with discussions of research goals and the philosophy of RA goals. Rather than accepting delivery of the original RA software, developed for the Symbolics computer under the now

obsolete 6.1 operating system (OS), we chose to wait for delivery of the Symbolics Genera 7-2 OS RA version and we used this software in our evaluation.

Our initial step was to build an extremely simple system from the ATC environment in accordance with the walk-through example provided within the RA documentation.[4] However, when we expanded this example using development inputs from the multiple RA views, RA began identifying inputs as contradictory and asking us to retract activities or data assignments. We were not able to complete our envisioned system.

We, next, worked with a statement of requirements for an automobile cruise control (CC) system. This CC problem was selected as it constituted a easily understood system with sufficient complexity to test RA's software engineering support capabilities. We first attempted to load the CC requirement problem statement into the intelligent notepad, but RA became hopelessly confused after entry of about five sentences and requested a retraction with each additional input. The complex sentence structure of the problem statement appeared to be a likely cause of this problem, so we converted the CC problem statement into concise statements which paralleled the ATC inputs of our first example. We were able to go further, but we encountered the same difficulties which arose during our attempt to expand the ATC example. Finally, we started over and attempted to develop the system totally within the graphical views without using the intelligent notepad. This effort encountered the same type of difficulties.

We had planned to move from the CC example into two workplace prototypes. Candidates in scientific applications and business applications were being considered. However, following our experiences with the initial simple development tasks, we concluded that RA has not reached a sufficient state of development to warrant such tests. We hasten to point out that Sanders Associates, the RA developer, did not misrepresent the state of RA software. They recommended loading new systems directly into the knowledge base. We chose not to do this as we felt we could gain the best view of research directions by treating RA as a finished tool and evaluating it from the engineer's view point.

4. THE EVALUATION EXPERIENCE: We believe our problems in developing simple models using RA stem, at least in part, from "over zealous behavior" on the part of RA's assistant feature. RA checks entries in the intelligent notepad and assigns objects. In a sense, RA develops a view of the evolving system. However, RA's concept may not be consistent with our system view. Like any good assistant, RA needs an ability to discern when initiative is called for and when it would be better to get more information from the person in charge. For example, when RA works with the intelligent notepad, we would like to participate in naming objects so that we can understand the system and share the RA view. One method would be to require the engineer's approval for object assignment. In addition, we would like RA to use our naming conventions (rather than the corresponding RA designation) when identifying conflicts. In this way, we would

be able to respond without flipping back and forth through the pull down menus when RA asks for a retraction to eliminate a perceived conflict.

When our evaluation experiences arrived at a point of conflict, RA asked us to make a choice--either retract an assignment or proceed with the knowledge that a conflict exists. When we directed RA to accept the choice and proceed, our path had entered a tunnel which narrowed as we proceeded, had no light at the end, and no option to turn back. When we took this course, we were without exception on a path leading to a "cold boot" with subsequent total restart. If such options are to be offered; RA needs the ability to "undo" N steps or to save a "system version" prior to entering the tunnel, i.e. a provision to backtrack from the tunnel and then take an alternative path. Once more, a better understanding of RA's view of the system and an effective means of reconciling this view with our own is needed to effectively use an option to over-ride (or defer) the conflict and proceed with development.

We believe that the divergence of RA's view and the engineer's view of an emerging system would be reduced if RA understood key words corresponding to the various views of the knowledge-base. For example, "context," "external," and similar words would help RA relate notepad entries to the context diagram; "state," "transition," "control," and similar words would point to the state transition diagram; etc. In addition, the engineer needs to understand the significance of such word usage. Whatever the source, more attention is needed to some vehicle for common understandings between the engineer and RA.

Current support to such understandings appears to be inadequate. Perhaps, the easiest way to achieve such understandings would be by introducing a formal method for use of RA.

5. COMMENTS AND CONCLUSIONS: We believe rapid advances within the CASE product market, results achieved after the RA contract had terminated, produced a level of expectation on our part which RA had no way of meeting. However, future RA research can certainly benefit from close monitoring of the CASE industry. An example is the use of bubble charts in structured analysis. The industry has produced "defacto standards" for labeling, decomposition, meanings of geometrical shapes, and related features which provide rapid insight into functional support features. [6,7,8,9,10] Today, most CASE tools still implement the waterfall software engineering paradigm, but early applications of knowledge-based support are emerging.[10] In addition, we are seeing some examples of graphical, compilable specifications leading to incremental prototyping capabilities. In effect, such products implement the spiral model paradigm, a close relative of the KBSA paradigm.[1] In the CASE market, we observe that the gap between available CASE products and the KBSA paradigm is systematically closing. While this is positive, in the sense of technology transition, it challenges KBSA research efforts to look to future needs, to address the high-risk, high-productivity fringes so as to remain relevant. CASE tools provide important guideposts pointing to leading edge technology issues for KBSA research to address.

Earlier, we mentioned the problem of maintaining consistent views of an emerging system in the respective "minds" of RA and the engineer. We have some suggestions (beyond those already provided) which may help. Currently, if we introduce a name into the intelligent notepad which is matched in the reuseable knowledge base, we obtain an automatic import for reuse. We think RA should suggest import and let us make the decision whether or not to proceed. Furthermore, if we give approval for import, then reconciliation becomes an issue. In case of conflict, should the imported system be held constant (because much of the productivity potential within reuse lies in the fact that the reuse component is tested and proven) or should the reused component be adjusted to the new system and retested? The engineer needs to make such decisions and once made, these decisions must guide RA's future suggestions, e.g. conflict resolutions, determining the system is ready to pass to the next KBSA facet, etc.

Several other areas exist in which we would like to interact with RA. One is spread sheet organization. We would like to participate in the assignment of row and column labels. We would also like to be able to pop-up a data dictionary with a list and brief description of each assigned object and of currently defined types (perhaps current pop-up descriptions were intended to evolve to meet this need). It would be nice to be able to choose a standard editor for notepad entry or editing (perhaps using a tabulator driven indentation convention in lieu of the implemented convention), but this is not a high priority compared to the primary research goals, even though current notepad

editing procedures constitute a somewhat painful process. Finally, we would like a pop-up list of remaining tasks which in the view of RA would lead to a complete system, a system ready to move to the next KBSA facet.

Next, we would like to focus on the concept of allowing the engineer to work according to personal preference while receiving support from RA. This is one of the fundamental concepts underlying RA. We agree with this concept only to the extent that if one can work in any manner, then RA can support any formal method. We do not foresee our engineers working outside the bounds of formal methods. In effect, software engineers have been working in an "art form" mode since the inception of computer programming, but studies conducted over the past ten years consistently support the conclusion that too much intellectual freedom is detrimental to good design and to high productivity.[10] In particular, the process of breaking very large projects into well defined parallel tasks which merge smoothly upon completion, leads naturally to invocation of disciplined methods. We suggest that our views are not unique; nearly all users of the KBSA paradigm will be employing disciplined methods. This is equally true within all three principal software development categories; business applications, scientific applications, and embedded systems applications; which exist within today's aerospace industry. While we gain the ability to use any formal method, we lose built-in, imposed discipline. Such discipline can be very beneficial when managing development of large systems.

Let us now look at the same issue from a different perspective. We stated that we think RA will be employed supporting a formal method when developing large systems. This implies that the goal of total flexibility for the engineer is not a high priority. However, it appears that pursuit of this goal may retard (or at least complicate) progress toward the more important goal of defining effective roles for AI within the assistant. The key question is: Would research progress on AI goals be realized more rapidly if a formal method were defined and implemented? It is not necessary to adhere to an existing formal method, for all such methods are today, somewhat immature. KBSA research may be able to contribute to advancing the methods science, as well. Certainly, effective use of AI and knowledge-based support will require today's formal methods to evolve. We view AI support goals as primary, for without AI and knowledge-base support, RA becomes nothing more than a standard CASE tool.

Addressing another problem, we think that a methods viewpoint may be the best way to introduce the perspective needed to maintain, or reconcile, consistent views of an emerging system between the engineer and RA. Subsequent research could then provide additional flexibility, step by step, until RA was totally divorced from the method, if this remains the goal. However, we think a more fruitful path might be to adopt and improve a formal method which permeates all KBSA facets. The method would focus on AI and knowledge-based support, with a view toward addressing the problem of scale. We think the future trends within our industry applications will be toward larger more complex software constructed by multiple teams and thence

assembled to perform assigned functions. If these concepts are valid, perhaps we need to adjust the KBSA goals for RA with regard to methods implementation.

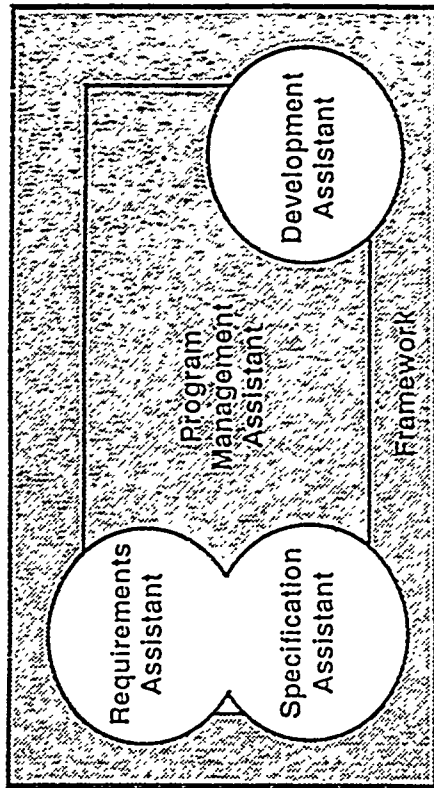
Finally, we would like to address prerequisites for effective industry evaluation of new software concepts, such as RA. Formal user documentation and a short training course are necessary, but not sufficient conditions for opening the full power of the software and the underlying concepts to prospective evaluators. The final ingredient is a tutorial. This tutorial must employ a sufficiently complex example to fully exercise software capabilities while providing a relatively simple problem domain. The tutorial should lead the user step by step down the path (perhaps one path of many possible paths) from problem statement to the point where RA would hand off to another KBSA facet. Such a tutorial may go a long way toward eliminating the difficulties we had in maintaining a view of the evolving system which was consistent with the view RA held.

We have made no mention of several problems which tend to hinder the technology transfer goals of the consortium. The existence of a "fuzzy boundary" between RA and SA has been addressed (at least conceptually) by combining RA and SA into a single software facet in the Phase II KBSA contract. This should facilitate progress using simulation and incremental prototypes to evolve the requirements and specifications to the final system. Tighter integration with the KBSA framework and migration to an open hardware environment with a POSIX operating system would also contribute to technology transfer; we assume

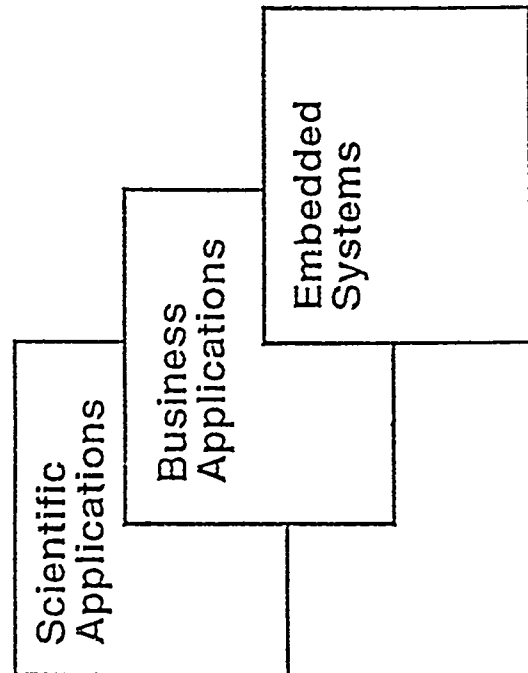
the KBSA prototype contract will address such issues. Overall, we think that Phase I software has laid a solid foundation, and we look forward to the next iteration, the Phase II software, which will provide an integrated requirements and specification development process.

BIBLIOGRAPHY

1. Boehm, B.W.; 1986: Spiral Model of Software Development and Enhancement, ACM Software Engineering Notes, VOL 11, NO 4.
2. Czuchry, A. & D. Harris; 1988: KBRA: A New Paradigm for Requirements Engineering. IEEE Expert, Winter 88. pp 21-35.
3. Green, C., D. Luckhan, R. Baizer, T. Cheatham, & R. Rich, 1983: Report on a Knowledge-Based Software Assistant. RADC-TR-83-195; Rome Air Development Center, AFSC, Griffiss AFB, NY. 71 pp.
4. Harris, D.; 1987: A Session Using KBRA. Sanders Associates, Nashua NH. 4 pp.
5. Harris, D. & A. Czuchry; 1988: The Knowledge-Based Requirements Assistant Final Technical Report, Volume I. Sanders Associates, Nashua NH. 60 pp.
6. Martin, J.; 1989: Modify Your Methods to Take Advantage of I-CASE Tools. PC Week, Vol 6, No 6, pp 36.
7. Mason, J.; 1989: Despite Its Drawbacks, Few Can Afford to Overlook CASE. PC Week, Vol 6, No 6, pp 89.
8. Pressman, R.S.; 1987: Software Engineering, A Practitioner's Approach. McGraw-Hill, New York. 567 pp.
9. Pressman, R.S.; 1988: Making Software Engineering Happen. Prentice Hall, Englewood Cliffs NJ 07632. 258 pp.
10. Yourdon, E.; 1988: Managing the Structured Technologies. Prentice Hall, Englewood Cliffs, NJ. 267 pp.



KBSA'S REQUIREMENTS ASSISTANT and AEROSPACE INDUSTRY NEEDS



EVALUATION PLAN

- ☒ "CANNED" ATC DEMO
- ☐ EXPANDED ATC DEMO
- ☐ CRUISE CONTROL SYSTEM
- ☐ WORKPLACE PROTOTYPE
 - Scientific Application
 - Business Application
 - Embedded Systems

THE EVALUATION EXPERIENCE

- CURRENT PROBLEM AREAS
 - Overzealous Assistant
 - Synergistic Relations Not Clear
 - Notepad Vocabulary Does Not Include "External," "Context," "States," ...
 - Need Quick Reference Support: Types, Objects, ...
 - Need "Undo" Support

THE EVALUATION EXPERIENCE

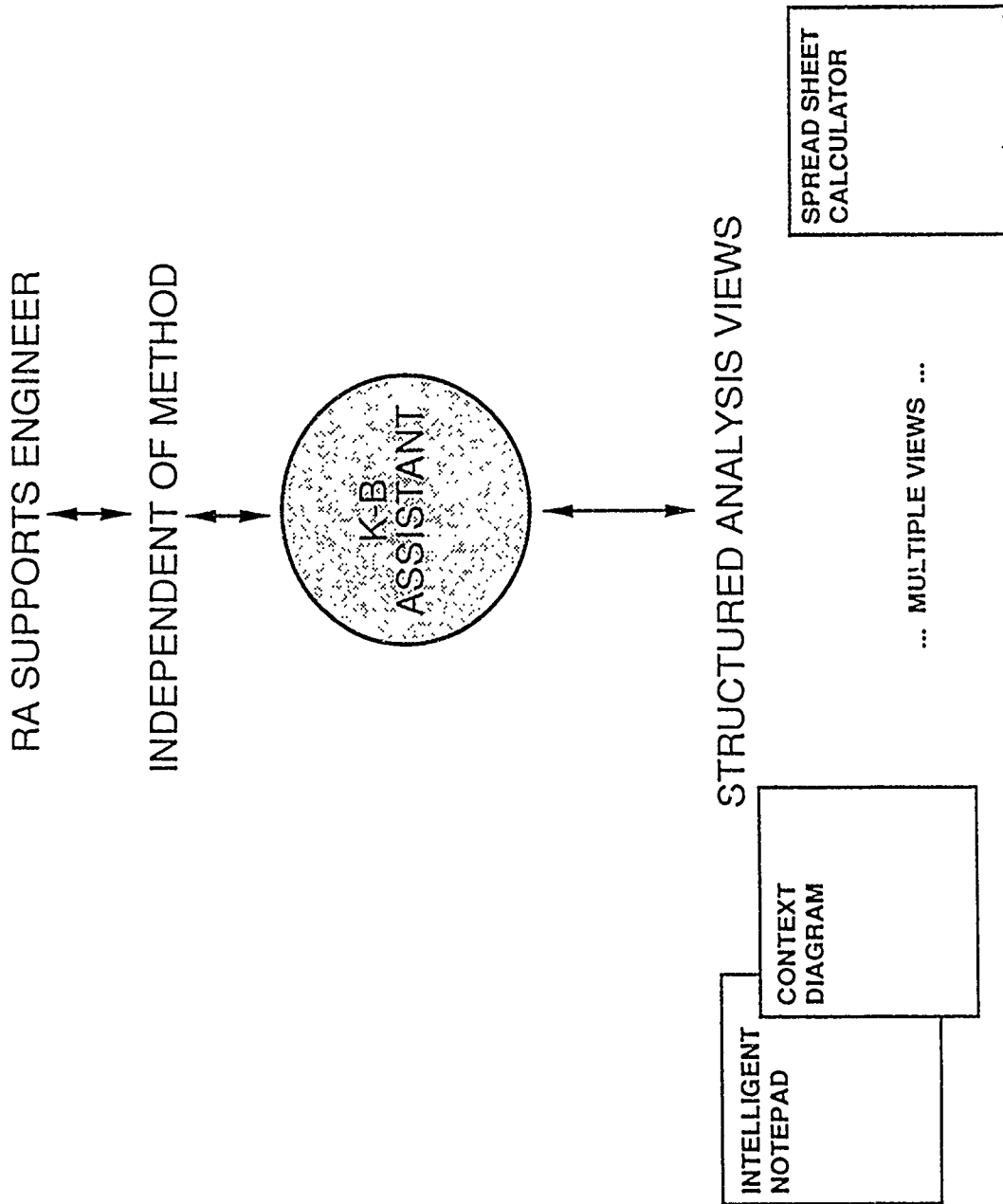
- **MINOR CONCERNS**
 - **Standards for Structured Analysis**
 - **Notepad Inputs Cumbersome**
 - **Hardware / Framework / ...**

COMMENTS AND CONCLUSIONS

Some Ideas

- CASE TOOL INTERFACE?
- K-B ASSISTANCE
 - Reuse Reconciliation
 - Spread Sheet Tailoring
 - Pop-up Unfinished Tasks
 - Pop-up Assigned Objects and Types
- METHOD AS A RESEARCH PATH?

COMMENTS AND CONCLUSIONS



KNOWLEDGE-BASED SPECIFICATION, ANALYSIS AND SYNTHESIS OF COMMUNICATION PROTOCOLS*

Lawrence Z. Markosian
Vice President for Applications Development
Reasoning Systems, Inc.
3260 Hillview Avenue
Palo Alto, CA 94304

Tel.: (415) 494-6201
Internet: zaven@reasoning.com

Author Biography. Lawrence Markosian is a founder of Reasoning Systems, Inc. Previously Mr. Markosian applied AI approaches to DoD problems at Systems Control Technology. As a Research Associate at Stanford University, he specialized in intelligent computer-assisted instruction systems. Markosian is a member of the Association for Symbolic Logic, the American Association for Artificial Intelligence and the Association for Computing Machinery.

Abstract. We present a knowledge-based workbench for communication protocol development. The workbench uses a graphical interface for entering protocol specifications, and a protocol description language that provides an equivalent textual specification. Protocols entered either graphically or textually are captured in an object-oriented database where they can be analyzed. Analysis knowledge is represented in the workbench using a very high level, general purpose specification language augmented by the protocol language. Once a protocol specification has been analyzed and validated, a knowledge-based compiler transforms it into executable C code. Protocol specification, analysis and synthesis are performed using the REFINE^{TM**} knowledge-based software development environment.

Introduction. Communications engineers use a range of techniques for specifying communications protocols. These techniques may employ

- graphical representations,
- event/transition tables,
- event scenarios,
- time lines,
- message format diagrams, etc.

* Copyright (C) 1989 by Reasoning Systems, Inc. Scott Burson of Reasoning Systems implemented the communication protocol specification workbench described in this paper. Scottie Brooks incorporated the application as a project in the REFINE Fundamentals Training Course, which is presented monthly at Reasoning Systems.

** REFINE is a trademark of Reasoning Systems, Inc.

Finite state automata (FSA) are a mathematical model that underlies many of the specification techniques. We present a knowledge-based workbench for protocol specification with the following capabilities:

- specification of communication protocols using FSAs,
- validation of the specifications,
- simulation of the specified system and
- automatic derivation of C code to implement the specifications.

The communication workbench is built in REFINE[1], a knowledge-based environment for software specification and synthesis. Several regional Bell operating companies are using REFINE and the protocol workbench for developing intelligent network applications, particularly service creation and testing. This paper emphasizes the knowledge-based aspects of the workbench, which made it easy to develop and highly extensible. The knowledge-based approach in developing it, as well as the domain-specific knowledge that it contains, distinguish it from tools with similar end-user capabilities written in a 3rd generation language such as C or FORTRAN.

Specification. Finite state automata are discussed in [2]. We provide a partial REFINE object-oriented definition of finite state automata and related objects below.

```
% Definition of the FSA object class
var FSA : object-class subtype-of universe

% Definition of the attributes (slots) of an FSA
var name : map(FSA, SYMBOL) = emptymap
var FSA-States : map(FSA, set(STATE)) = emptymap
var FSA-Initial-States : map(FSA, set(STATE)) = emptymap
var FSA-Final-States : map(FSA, set(STATE)) = emptymap
var FSA-Transitions : map(FSA, set(TRANSITION)) = emptymap
var FSA-Events : map(FSA, set(EVENT))
  computed-using FSA-Events(the-fsa) =
    {transition-event(tr) | (tr) tr in FSA-Transitions(the-fsa)}
```

```

% Definition of the STATE object class
var STATE : object-class subtype-of universe

% Definition of the TRANSITION object class
var TRANSITION : object-class subtype of universe

% Attributes of TRANSITION
var Transition-From : map(TRANSITION, STATE) = emptymap
var Transition-To : map(TRANSITION, STATE) = emptymap
var Transition-Event : : map(TRANSITION, EVENT) = emptymap

% Definition of the EVENT object class
var EVENT : object-class subtype of universe

```

The above specification is applicable to FSAs in general. However, in communications applications, most events represent the transmission or reception of a message. Thus we refine the notion of an event to capture this idea by introducing the following event attributes:

```

% Attributes of EVENT
var Event-Type : map(EVENT, symbol) = emptymap
var Event-Message : map(EVENT, symbol) = emptymap

```

The Event-Type of an event must be one of the following symbols: TRANSMIT, RECEIVE or SILENT. The Event-Message (i.e., the message transmitted or received) is also represented as a symbol. An extension of the workbench might represent message as an object class refine the concept appropriately (for example, by adding attributes for header and body).

Silent events are state-changing events that are not explicitly associated with the transmission or reception of a message at the communication level being modeled by the protocol. An example of a silent event might be a timeout.

Messages to be transmitted and received by a connection establishment protocol might

include REQUEST-CONNECTION, CONNECTION-CONFIRMED, and CONNECTION-REFUSED.

These extensions to the original FSA definition give rise to the notion of a *communicating finite state automaton*. There is an extensive literature on the use of communicating finite state automata in specifying communication protocols [3].

Once the domain model for communicating finite state machines has been specified, we turn to the definition of a language specifically for describing these objects. Such a language is convenient to have for several reasons:

- it will sometimes be desirable for an end-user to enter a textual, rather than graphical, description of a communication protocol;
- the protocol development workbench will need to print descriptions of protocols—for example, to describe parts of a protocol for which there is no appropriate graphical representation; and
- the application-specific syntax can be used to simplify the specification of the protocol development workbench itself, in a way that will be made clear below.

Below is the REFINE definition of several of the grammar productions for the FSA language.

```
FSA ::= [ "fsa" name
        { [ "states" "(" FSA-States * "," " )" ]
          [ "transitions" "(" FSA-Transitions * "," " )" ] }
      ],
```

```
TRANSITION ::= [ "from" Transition-From "to" Transition-To
                  Transition-Event ],
```

```
EVENT ::= [ ":" Event-Type { [ Event-Message ] } ]
```


For each object class (FSA, TRANSITION and EVENT) the grammar specifies a corresponding surface syntax. The syntax includes keywords, delimiters, and information about where the values of the attributes appear. REFINE generates a parser from the grammar that will be able to read examples of FSAs such as the following:

```
fsa Connection-Establishment
  states (Unconnected, Waiting-for-Confirmation, Connected)
  transitions (
    from Unconnected to Waiting-for-Confirmation:
      transmit REQUEST-CONNECTION,
    from Waiting-for-Confirmation to Connected:
      receive CONFIRM-CONNECTION )
```

REFINE also automatically generates a *printer* for the FSA language, so that instances of FSAs represented in the knowledge base can be pretty-printed in a format similar to the above example.

Analysis. As developers of the protocol development workbench, we would like to be able to express knowledge about protocols (for example, what constitutes a valid protocol) in a succinct way, certainly in a way that eliminates programming details. It would be even better if we could express this knowledge in a way that minimizes the need to understand the underlying knowledge-base representation of the relevant object classes. For example, one validation criterion for a protocol is that it have no “duplicate” transitions—i.e., no two distinct transitions from the same state to the same successor state under the same event type and symbol. In REFINE, this constraint can be expressed declaratively as follows:

```
~ exists(tr1, tr2) (
  tr1 = 'from @state-1 to @state-2 : @type @msg' &
  tr2 = 'from @state-1 to @state-2 : @type @msg' &
  tr1 ≠ tr2)
```

The single quotes enclose a *pattern* that is used to match against transitions. Within the pattern we have used the syntax that we defined earlier for transitions and events. The

variables preceded by @ are *pattern variables*. When REFINE generates a parser for a user-defined language, it also automatically generates a pattern matcher that allows the user to write patterns as in the above example.

Why use patterns? Compare the above example with an equivalent constraint written without patterns:

```
~ exists(tr1, tr2, ev1, ev2)
    ev1 = Transition-Event(tr1) &
    ev2 = Transition-Event(tr2) &
    Event-Type(ev1) = Event-Type(ev2) &
    Event-Message(ev1) = Event-Message(ev2) &
    Transition-From(tr1) = Transition-From(tr2) &
    Transition-To(tr1) = Transition-To(tr2) &
    tr1 ≠ tr2 )
```

This form is still declarative—it is a specification with no procedural content. However, not only is this form of the constraint twice as long, but understanding it requires understanding the details of the knowledge base representation of transitions and events. Thus the first form achieves *two* levels of abstraction over an equivalent 3rd generation language program: (1) it is purely declarative *specification* and (2) it uses a *domain-specific language* to simplify the logic- and set-theoretic specification. The REFINE compiler generates comparable code for both forms of the constraint.

The above validation criterion is “local”—it represents a constraint on a single protocol. However, a communication network has many interacting protocols, which are replicated at many nodes on the network. Each protocol may meet “local” validation criteria but the network as a whole may fail to meet global requirements. *Freedom from deadlock* is an example of a global requirement—the network should have no deadlock states. The deadlock states in a network’s state space is the set of

- network states that are
- not final states
- have no transitions out and

- have all communication channels empty.

After extending the communications domain model to include networks, channels and related objects, this definition of a network's deadlock states is stated in REFINE as follows:

```
function Deadlock-States (ntwk : NETWORK) :
set (NETWORK-STATE) =
{ ntwk-st | (ntwk-st)                                % The set of all
  ntwk-st in Network-States (ntwk) & % network states that are
  ~ ntwk-st in Network-Final-States (ntwk) & % NOT final states,
  Transitions-Out (ntwk-st) = { } & % have NO transitions out,
  for-all (ch) (ch in Network-State-Channels (ntwk-st) =>
    Channel-Contents (ch) = []) }
% and have all channels empty
```

The above function specification is defined purely declaratively and constitutes virtually a line-for-line formalization of the preceding informally stated definition of deadlock. The specification is declarative because no details of the procedural implementation are provided. For example, we have specified that the value to be returned is a *set* but we have given no details on how the set is to be implemented (e.g., as an array, hash table, linked list, etc.). Nor have we provided any algorithm for searching the state space for deadlock states. The REFINE program synthesis system is able to generate an executable implementation automatically from this and related definitions [4]. Other global validation criteria are easily specified, automatically implemented, and executed in REFINE, including freedom from unspecified message receptions, lack of "dead" code, etc.

The network model is being extended to support intelligent network management, which requires not only static analysis but simulation, scenario generation and execution, dynamic monitoring and performance evaluation.

Synthesis. Once a protocol has been fully validated, a communications engineer may wish to program a C implementation based on the validated specification. The protocol

workbench that we have developed uses REFINE's program synthesis capability to do this automatically. As an example, we provide part of the C implementation generated for the simple connection establishment protocol given above. The code includes—

- a data declaration for the global variable holding the pointer to the current state function,
- function prototypes for the state functions (required by ANSI C),
- the state function for the first state ("Unconnected"),
- stubs for the (undefined) functions that select one of the (possibly several) enabled transitions in a state (e.g., `unconnected_trans_selector`) and
- exception-handling code.

```
void *current_state ( ); void unconnected_state_function ( );
void waiting_for_confirmation_state_function ( );
void connected_state_function ( );
void unconnected_state_function ( )
{int trans = unconnected_trans_selector ( );
  if ( trans == tr1
      pipe_write( "REQUEST-CONNECTION" );
      current_state =
          waiting_for_confirmation_state_function;
      else error ( "Unknown transition %d in state %s\n",
                  trans, "Unconnected" ); }
...

```

The application-specific C code generator for our examples of protocol specifications constitutes less than two pages of REFINE code. The C code generator makes use of Reasoning Systems' C Language System (CLS) [5] which contains a domain model, parser, printer, pattern matcher and other tools for manipulating C software.

Graphical interface. FSAs have a natural graphical representation that can form the basis for an interactive, graphical specification environment. REFINE contains a user interface toolkit that can be used in building such an environment. A principal component

of this toolkit is a graphics editor that allows placing and connecting icons on surfaces and viewing surfaces through "viewports". In addition, REFINE supplies customizable menus and a mouse-handling capability. There is a programmatic interface to all these capabilities so that the displays and interaction can be highly customized. The REFINE user interface is built on X Windows for portability.

The first step in building the graphical interface for the communication protocol workbench in REFINE was to extend the FSA domain model to include attributes that associate FSAs, STATES, EVENTS and TRANSITIONS with appropriate graphical objects provided by the REFINE graphics package—SURFACE, LINK and ICON. An FSA is associated with a SURFACE on which its diagram will be drawn, a TRANSITION is associated with a LINK, and a STATE with an ICON. Different icon types (diamond, circle, square) were used to denote initial, intermediate and final states.

The next step was to customize the REFINE mouse handler functions to capture the desired interaction. Figure 1 shows a sample screen dump.

Summary and conclusion. The communications protocol workbench is a spin-off of KBSA technology. Communications knowledge represented in the system includes—

- local and global validity criteria for communication protocols and
- programming knowledge.

The *validity criteria for protocols* are stated purely declaratively in either the basic REFINE specification language or in a language developed in REFINE for describing FSAs.

The *programming knowledge* includes both general programming knowledge, used to generate executable implementations of the validity criteria, as well as knowledge specific to the communications domain that is used to generate C implementations of protocols from very high level specifications. The general programming knowledge represented in the REFINE program synthesis system is what makes it easy to add new capabilities, such as new validation criteria or automatic test generation. These new capabilities can be described purely declaratively, and the tedious programming details can be left to the

REFINE program synthesis system. The general programming knowledge distinguishes the REFINE development approach from the use of a 3rd generation language such as C or FORTRAN in building applications with similar end-user capabilities.

The workbench has an interactive graphical user interface that admits use by a communications engineer with no knowledge of REFINE. In addition, the workbench uses a domain-specific language both for textually describing protocols entered graphically and for making it easier to extend the workbench's capabilities.

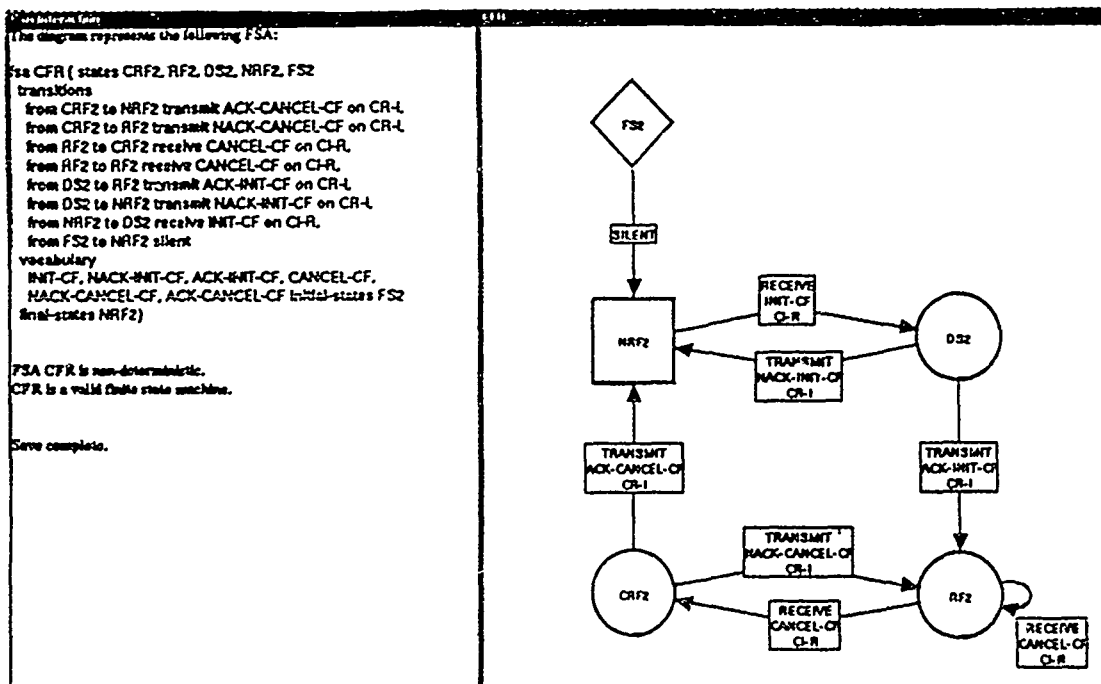


Figure 1: Screen dump from the protocol workbench. The window on the right shows a specification of a protocol named CFR that has been entered graphically. The top part of the window on the left shows an equivalent textual specification generated automatically from the knowledge base using a language defined for protocols that differs slightly from the example in the text of this paper. Below the textual specification is the result of an analysis step—the protocol is a valid finite machine that is non-deterministic. The non-determinism arises from the fact that the protocol has two transitions out of state RF2 that have the same event type (RECEIVE) and event symbol (CANCEL-CF). Non-determinism is expressed in REFINE using patterns and the protocol language as follows:

```
exists(tr1, tr2) (
  tr1 = 'from @state-1 to @state-2 : @type @msg' &
  tr2 = 'from @state-1 to @state-3 : @type @msg' &
  state-2 ≠ state-3)
```

References

- [1] *REFINE User's Guide*. Reasoning Systems, Inc., Palo Alto, CA, 1985
- [2] Aho, A.V., and Ullman, J.D. *Principles of Compiler Design*. Addison-Wesley, Reading, MA, 1977
- [3] Sunshine, C. (ed) *Protocol Specification, Testing and Verification*. North-Holland, Amsterdam, 1982
- [4] Smith, D.R., Kotik, G.B. and Westfold, S.J. "Research on Knowledge-Based Software Environments at Kestrel Institute", *IEEE Transactions on Software Engineering* v. SE-11, No. 11, November, 1985
- [5] *CLS Data Sheet*. Reasoning Systems, Inc., Palo Alto, CA, 1988

KNOWLEDGE-BASED SOFTWARE REVERSE-ENGINEERING FOR RE-ENGINEERING AND REUSE.

Mr. Philip H. Newcomb
Principal Investigator
Software Reverse Engineering
Boeing Advanced Technology Center
Boeing Computer Services
P.O. Box 24346, MS 7L-64
Seattle, Washington 98124-0346

(206) 865-3431
philn@atc.boeing.com

BIOGRAPHY

Mr. Newcomb is a specialist in the application of artificial intelligence to software engineering in Boeing's Advanced Technology Center. His responsibilities include the investigating of reverse engineering for scientific, engineering, business, parallel, and embedded computing systems.

ABSTRACT

Software reverse engineering is concerned with computer-aided extraction of specifications and design from existing software systems. Software re-engineering is concerned with the transformation of existing systems to achieve conformance with prevailing programming standards, for language conversion, or to support rehosting to advanced computing languages or architectures. Software reuse is concerned with identification of reusable software for adaptation to, or use within new computing applications. A coordinated effort to reverse-engineering, re-engineering and reuse software may lead to dramatic reductions in software maintenance costs and reduce the cost and risks of future development by enabling present and future CASE technologies to be applied to existing software systems. This paper describes several problems in software reverse-engineering which may be amenable to solution by knowledge-based technology.

Introduction

A software intensive corporation, a corporation which is fundamentally dependent upon software for the generation of goods and services, may spend several hundred million dollars a year and employ tens of thousands of programmers to maintain a vast inventory of business, scientific, manufacturing, engineering and embedded mission-critical software systems. It has been estimated that these maintenance expenditures average 80% or more of software dollars and will rise above 90% during the 1990s. These spiralling software maintenance costs are shrinking the percentage of dollars available for new software development and hence pose a fundamental threat to the productivity and innovation of American industry.

Computer Aided Software Engineering (CASE) tools, advanced programming languages and development techniques have been shown to demonstrate productivity improvements from 100% to 1000% with dramatic improvement in system reliability and quality. Heretofore, however, CASE tools have had very little effect upon the software maintenance problem. CASE has been viewed primarily as a means to reduce cost and risk, and improve reliability and quality during the development of new software. The impact of CASE since its maturation as a technology during the early and mid 1980s has been small relative to the size of the overall maintenance problem. Today, less than 4% of software is maintained with CASE tools. If a means were available for existing software to be made compatible with available advanced software engineering tools and programming techniques, enormous savings might be achieved in software maintenance expenditures.

A new technology is emerging which may reduce software maintenance costs by improving maintenance programming practices by enabling existing software systems to be migrated into CASE tools and advanced programming environments. Reverse-engineering and re-engineering of software for reuse (SR3) technology is concerned with the methods, tools, and techniques that reverse-engineer and re-engineer existing software to enable programmers to more easily maintain it and reuse it in future software development projects. If a workable approach to SR3 can be demonstrated and fielded, its expected value is roughly equivalent to the salvage value of a company's existing software systems plus the savings in software maintenance costs over the expected life of the systems. The dollar value of the software for a typical large corporation, which has 500 hundred million to a billion lines of software, may range from \$12.5 Billion to \$50 Billion (based upon a crude estimate of \$25 to \$50 to develop and maintain a line of code). Maintenance costs for this software easily exceeds \$500 Million per year. With a modest estimated 50% reduction in maintenance costs over a ten year period and 25% salvage rate of existing software, a conservative estimate of the expected present value of potential savings from SR3 technology may approach \$5 Billion for a typical large company.

Industry-wide there are at least a 100 billion lines of software written in 3rd generation computing languages, such as COBOL, FORTRAN, and Jovial, C and PL/1. By the method above the dollar value of this software ranges from \$2.5 to \$5 trillion and its annual maintenance costs most likely exceed \$50 Billion per year. The industry-wide potential savings in software costs over a ten year period from a proven SR3 technology could exceed \$1 Trillion.

Scope of the Maintenance Problem

Software maintenance is not just the single most costly computing expenditure facing industry, it is also the single most serious impediment to business modernization. To remain competitive businesses must rapidly assimilate new computing intensive technologies into their existing computing infrastructures to provide better products and services. The rate at which such modernization can take place limits the ability of businesses to react to new opportunities and to adapt to changing market conditions.

Without adequate mechanisms for managing and controlling the complexity of software, the process of developing and maintaining systems leads inexorably to exponential explosions in system complexity and costs. As a business changes, so must the computing systems which model and automate functions of the business organization. As maintenance is performed it tends to corrupt and degrade system structure and documentation. Maintained systems have been observed to experience cycles of exponential growth followed by a build-up of change requests until the maintained system reaches a critical size beyond which it cannot grow without significant restructuring or redevelopment. Around a certain critical size which varies with application and language type growth slows or becomes negative. As pressure for change continues to build up, there is a reduction in quality and further increase in system complexity. Eventually the systems must be re-engineered or replaced.

Widescale replacement of software systems does not seem to be feasible, even with modern application generators. A typical large business possesses hundreds of software systems that constitute a tremendous business investment. Enormous sums have been expended to capture and transfer into a machine executable form the functions of the business enterprise. We have succeeded in the last thirty years in locking up in obsolete programming languages much of our knowledge about the way our businesses operate and our complex information systems function. For most of these systems, their replacement requires replication or extraction of the original requirements, the knowledge of what the system was intended to do, before the system can be upgraded or replaced.

Reverse Engineering for Re-engineering and Reuse.

A coordinated effort to reverse-engineering, re-engineering and reuse software may lead to dramatic reductions in software maintenance costs and reduce the cost and risks of future development by enabling present and future CASE technologies to be applied to existing software systems. Software

reverse-engineering is concerned with extracting design documentation and high-level specifications from source code, and storing this information in data-dictionaries and CASE tools. Once the code and design are captured by CASE tools, software maintenance is performed more reliably and economically than by manual means. Software reverse-engineering may be done alone or in conjunction with software re-engineering. Software re-engineering is concerned with the restructuring, transforming and instrumenting transformation of existing existing source code of systems to achieve conformance with prevailing programming standards, for language conversion, or to support rehosting to advanced computing languages or architectures. Software reuse is concerned with the identification of reusable software for adaptation to, or use within new computing applications. A class of tools closely related to software reverse-engineering and re-engineering tools are software maintenance engineering tools. Software maintenance engineering tools are smart-editing and debugging tools and environments that help maintenance programmers analyze, debug, and test source code. Reverse-engineering, re-engineering, reuse and maintenance engineering tools salvage an organization's investment in existing software by enabling it to more effectively manage and control its information resources, by enabling maintenance programmers to improve their ability to understand and change software, and by enabling developers to more easily identify and reuse existing software. Dramatic reductions in software maintenance costs and reduction in the cost and risks of future development will result if a complex set of methodological and functional problems are solved that lead to a comprehensive technology for reverse-engineering, re-engineering and reuse of software large businesses.

State-of-the-Art

Recently, CASE vendors begun investigating transition pathways for reliably and economically migrate existing software into their CASE tools. While reverse-engineering has been applied successfully on a small scale, tools and techniques that reliably and cost-effectively reverse-engineer major systems or very large collections of programs are still not yet available. Conventional technology has failed to formulate a satisfactory approach to many significant and outstanding problems reverse-engineering and its related disciplines.

SR3 Problems Amenable to a Knowledge-Based Solutions

This paper describes several problems, which challenge conventional approaches, whose solution, if possible, would contribute incremental advancement towards an overall solution of the reverse-engineer for re-engineering and reuse problem. The results of the investigation of these problems would provide guidance and direction to an overall program to solve the problem of upgrading existing software to achieve compatibility with modern computing architectures and software engineering methods.

Problem 1: Reverse-Engineering and Re-engineering of COBOL Business Systems

Despite optimistic vendor claims, reverse-engineering of COBOL into Computer Aided Software Engineering (CASE) tools that complement a business system software development life cycle is still a largely unsolved problem. COBOL is the single most widely used business application development language. It is the standard for most commercial/business application development. Successful reverse-engineering of COBOL will provide the highest return on investment. Almost all commercially available reverse-engineering and re-engineering tools have been developed specifically for COBOL. While vendor products are maturing quickly, the two most important elements of the overall reverse-engineering and re-engineering problem, reverse-engineering of process models and data models from COBOL systems, are still unsolved, particularly for very large systems, by any available tool.

The benefits of reverse-engineering of business systems is quite large. COBOL maintenance constitutes the single largest computing cost of most corporations and in the government sector. Much maintenance costs are attributable in one fashion or another to poorly structured and poorly understood software and uncontrolled definition and usage of data.

The data modeling problem is especially hard and complex because of the problem of resolving data

definition and usage across very large systems consisting of hundreds of individual programs in which no standardization or convention was imposed upon the selection of record element and local and common variable names. In very large systems an alias rate of 30 data-names to single common data element is not uncommon, even after the application of tools which attempt to automatically reduce data naming redundancy. Most of these tools require manual inspection of code to resolve such data usage conflicts. Analysts must visually inspect the program data and procedure sections to resolve the way in which data is used. Most of these tools do not possess mechanisms for locating where data elements are used within a program space or for categorizing the kinds of programming idioms which operate upon data elements. Such mechanisms could significantly reduce the complexity of the task of performing data definition and usage resolution, a task which must be performed for a common data dictionary to be defined for a company. A few of tools are providing mechanisms for automating changing of data name in the procedure sections of programs from a common data dictionary, but little help is available to analysts to resolve how data elements should be renamed to clarify their usage and achieve a common data dictionary definition.

Three forms of re-engineering tools are becoming available. COBOL restructuring tools are available from several vendors which automate the transformation of spaghetti code into well-structured code. Often what results from such restructuring is somewhat suspect. In many test cases the code does not function correctly after it is restructured. Maintenance programmers frequently reject restructuring tools because the tools make the code unfamiliar to them. Database re-engineering tools are available from a few sources which will convert an IMS database into a Relational database. Such tools will read the record structures described in the data section of program, convert the data definition through several stages into an Entity Relationship (ER) model, and then convert the ER model into the relational tables of a Relational Database and transform the program to run with the Relational Database. A number of vendor products are beginning to automatically produce design artifacts such as structure charts, data flow diagrams, data dictionaries and ERA models from parsed COBOL source code and Job Control Language. These products are useful for documenting the design of existing COBOL systems.

Knowledge-based techniques can provide assistance in the reverse-engineering of business systems by providing techniques for parsing code to generate abstract syntax trees of the COBOL programs which make up a large system. A composite forest of abstract syntax trees (ASTs) of all the programs which make up a system constitutes a machine readable and manipulable formal structured representation of an entire system. By means of powerful pattern-matching and transformational rules available in some knowledge-based tools the individual programs of a large system can be altered by means of correctness preserving transformations and improved code can be regenerate from the modified abstract syntax trees. The substitution of data names consistent with the data dictionary is accomplished by means of the straight-forward transformation of the program's AST and the regeneration of the corrected programs of the system. A knowledge-based approach can be applied to process modeling as well as to data modeling problems.

An important data modeling problem is the extracuon of consolidated data dictionaries from large systems. This can be accomplished by developing an intelligent assistant to analyze the data usage of individual programs and data flow dependencies between programs and automate the process of resolving the usage and definition of data.

Whatever solution is found eventually is not likely have a single vendor source. It is likely to be some carefully selected composite of several vendor tools, internally developed management practices, and procedures and methods for methodically upgrading and testing the software to be reverse-engineering and re-engineering. The solution must be cognizant of improvements in software engineering practices and languages and compatible with major trends in repositories and systems applications architectures

Problem 2: Mapping Heterogeneous Systems Into Homogeneous Systems.

Th mapping of a heterogeneous system into a homogeneous system is concerned with the identification and development of tools and methods that would facilitate re-engineering of large heterogeneous

systems, systems written in many 3rd generation languages, into homogeneous systems, written in one or two advanced (4th or 5th generation) languages. To accomplish this the AS IS model, consisting of a data and process model of all the programs and data bases of the system, must be extracted by analysis, and those of the systems components which are reusable must be mapped onto a TO BE model of the desired system and rewritten in the advanced language.

Many very large applications are hybrid systems which perform business and engineering or scientific functions. Such systems are implemented in several programming languages, such as JCL, COBOL, MARK IV, and FORTRAN. Nearly all conventional vendor-supplied reverse-engineering products are made for systems written in a single language, namely COBOL. These products cannot be used for reverse-engineering or re-engineer of such large and complex heterogeneous systems.

A knowledge-based approach to re-engineering of heterogeneous systems into homogeneous systems may be possible. The generic parser generators and program rewriting tools available through some knowledge-based systems enable the relatively cheap and reliable creation of parsers and domain models for many computer languages. On a workstation equipped with sufficient main storage, secondary storage, and virtual memory, the parsers and domain models of the languages in which the system's programs are written can be made to be co-resident and non-conflicting so that the programs of a very large heterogeneous system can be read into a single large AS IS model that will be resident in virtual memory. Ideally, both the AS IS system model as well as the TO BE system models would permanently reside in a persistent knowledge base and all analyses and re-engineering transformations would take place against these model.

One step in the re-engineering of the heterogeneous system involves re-engineering the data model of the system, as discussed in problem 1. Re-engineering also involves extraction of reusable program logic from the process model of the system. The program logic which will be preserved in the TO BE system model is a purified representation of the business rules. Extraction of business rules requires the analysis of program logic to formulate condition tables which capture and depict the conditions under which the computational processes occur in a program. Automatic formulation of condition tables of a programs logic can be accomplished with knowledge-based tools which can analyze for dependencies between data-flow and control-flow in a program and generate data structures which represent these relationships in the canonical formalism of a condition table. Once these condition tables are extracted from the code, there are tools which will translate condition tables into SQL. If a systematic approach is taken to construct maximally uniform parsers and domain models, it may be feasible to employ common data and control structure analysis techniques to extract the condition tables for many languages.

Problem 3: Extraction and Interchange of CASE Design Information

Industry has not yet created a common standard for CASE design information interchange. Such a standard would define the composition of software engineering products and the interrelationships between software engineering processes. The absence of such a standard restricts freedom in the use, application, and interaction of CASE tools. It imposes conversion costs and forces overcommitment to a single vendor's tool set. It is essential that interfaces standards be defined that will permit the interchange of design information between CASE products and tools. Design interchange mechanisms must be developed that enable flexible integration of sets of plug-compatible tools with the broadest and highest quality functionality.

Many CASE vendors have created closed systems that do not support the import and export of design information to and from of their tools. With more than fifteen national and international CASE standards organizations considering CASE standards, it is unlikely that agreement on CASE design interchange format will be reached any time soon.

Knowledge-based tools may prove to be very useful as a lingua franca between CASE tools and non-standard CASE design interchange formats. Using parsers, readers, writers, and transformations

applied to the domain models of various tool's published import/export formats, interchange of information between the tools could be accomplished relatively easily.

If a sufficiently rich CASE design interchange format existed, knowledge-based tools could be routinely employed along with conventional tools. As software engineering evolves, tools for software reverse-engineering, re-engineering, and reuse, formal verification, etc. will augment the traditional CASE tools for design, analysis, code, and so on. The formal knowledge-based models of the KBSE facets could prove of great benefit in establishing the foundation of a CASE design interchange format standard which can permit the flexible interchange of many kinds of software engineering information between knowledge-based and conventional software engineering tools.

As an example, while many conventional CASE tools possess such useful capabilities such as syntax-directed editors or action diagrams, they do not automatically layout design artifacts such as structure charts or data flow diagrams. Generation of such presentations has already been accomplished by many knowledge-based tools for many kinds of CASE design artifacts. The layouts generated by auto-layout capabilities could be made available to conventional CASE tools through a standard CASE design interchange format.

Provided that CASE tool interfaces with their data bases were simple, clean and well defined they might be replaced by a knowledge-base. Such a substitution would permit the flexible integration of knowledge-based tools within a vendor's tool set.

Problem 4: Modernization of 3rd Generation Procedural Languages

Most software is expressed in 3rd generation languages that were created long before computer science matured as a discipline. The most widely used computer languages such as COBOL and FORTRAN are very inelegant, low-level languages when compared to many of the rule-based, non-procedural languages used by the artificial intelligence community. It is well-known that computational formalisms such as procedural, rule-based, logic, and object-oriented programming are ultimately computationally equivalent. These formalisms differ significantly, however, in the way in which they represent a computation and hence in the amount of effort required to express a programming problem and maintain a programming solution. Through its development of more and more advanced computing languages computer science has made continuous progress towards the objective of achieving higher and higher levels of functionality from software with less specification. In general, the non-procedural programming forms provide significantly more compact (up to a factor of 10 or higher) representation of programming problems than do their procedural cousins. Unfortunately, the cultural inertia of 3rd generation software has precluded the wide-scale use of many of very promising advances in computer languages.

Re-engineering of 3rd generation languages into more advanced 4th and 5th generation languages could produce significant reductions in the sizes of the computer programs, and facilitate the use of sophisticated programming environments and CASE environments, such as the DoD's Ada Programming Support Environment (APSE). Like-to-like translations between somewhat comparable computer languages (e.g. the translation of scientific FORTRAN software into ADA), could be accomplished relatively straightforward with knowledge-based re-engineering tools and attain great leverage from such well-funded government programs as STARS and SEI. Conversions between procedural and rule-based languages is much harder because it requires deep sensitivity to subtle differences between the semantics of different programming formalisms.

There are many incremental levels of attainment achievable in the translation of 3rd generation software into higher level languages. High quality conversions between two procedural software languages such as FORTRAN and ADA will probably prove feasible with knowledge-based parsers and language-pair specific transformation libraries. A much more difficult problem is the extraction of high quality reusable non-procedural software specifications from procedural code.

Problem 5: Reverse-engineering into the Knowledge Based Software Assistant

Many commercially available CASE tools and methods religiously implement some variation on the waterfall method of the software development life-cycle (SDLC). This software development method has created by the the mandate of government and industrial software development standards. The CASE tools that adhere to these standards implement assumptions about design methods and development techniques which, it can be argued, have become antiquated because of new advances in computer science, particularly in the knowledge-engineering, rapid prototyping, and automatic programming fields.

The Rome Air Development Center's Knowledge-Based Software Assistant itself is a government sponsored research effort in knowledge-based software engineering that anticipates the emergence of a new knowledge-based SDLC which will be manifested within a mature KBSA within a few years. These new methods and standards of the KBSA may well revolutionize software development and lead to the far more rapid development of much higher quality software than is attainable by means of current CASE tools.

Methods and techniques that could reverse-engineer existing software into a form compatible with the advanced software development tools and knowledge-bases of KBSA facets (or a collection of comparable commercial CASE tools) could set the stage for the eventual transition from conventional to knowledge-based software development methods.

As software is gradually transitioned into current generation CASE tools and into higher level languages, and as KBSA concepts, facets and underlying technology migrate into industry, the difficulty of the task of reverse-engineering and re-engineering of existing software systems for reuse and maintenance within the KBSA will decline.

Conclusion

Reverse-engineering, re-engineering and reuse have recently become centers of focus at major software engineering conferences and forums. There have been a flood of vendor product announcements, but for the most part, the products have fallen short of vendor claims. Reverse-engineering and re-engineering to migrate software into CASE has an extraordinarily high potential return on investment to end-users in maintenance savings. The application of CASE tools to existing software systems provides the basis for a dramatic expansion of the CASE market that has spurred CASE vendors to race to develop reverse-engineering and re-engineering tools and capabilities, particularly for the huge world of COBOL applications. But, there are a number of hard problems and complex issues in software reverse-engineering, re-engineering and reuse, obscured behind the flurry and excitement of quick and easy fixes, that will require careful study and a well thought out approach. It is not surprisingly, given the many years of research that have gone into knowledge-based technology, particularly into language systems and program transformation systems, that these types of tools have such great leverage potential on reverse-engineering for re-engineering, and reuse problems.

``Standards, Enabling or crippling?''

Position Statement

Joshua Glasser

Honeywell Systems & Research Center

Overview

The emerging X standard, once in place, will provide a common platform of low level graphics and windowing services. This will in itself be of great benefit. But, such low level services, while vital, are not in themselves all that useful to the KBSA developer. Our needs lay in higher level graphic and user interface services. However, no standards for these higher level services have emerged. A major factor for the nonemergence of these higher level standards has been the inability to share prototype systems between developers. Thus, the true value of having a standard such as X: With X in place we will have a common ground for experimentation and refinement at these higher levels.

The X Window System, or X for short, is a network transparent window system. X allows you to run multiple applications simultaneously in windows, generating text and graphics in monochrome or color on a bitmap display. Network transparency means you can use a display on one machine and run application programs on other machines scattered throughout a network, and have the applications interact with you using the full graphics facilities of the system, with the identical user interface as when you run the applications on your local machine. X is designed to permit applications to be device independent; that is, applications need not be rewritten, recompiled, or even relinked to work with new display hardware.

X provides facilities for generating multi-font text and two dimensional graphics (such as points, lines, arcs, and polygons) in a hierarchy of rectangular windows. Every window can be thought of as a "virtual screen", and can in turn contain windows within it (called subwindows), to arbitrary depth. Windows can overlap each other, like stacks of papers on a desk, and can be moved, resized, and restacked dynamically. Windows are designed to be inexpensive resources; applications using several hundred subwindows are common. For example, windows are often used to implement individual user interface components such as scroll bars, menus, buttons, and so forth.

Although you can think of yourself as a client of the system, in network terms the application programs you run are called clients, utilizing the network services of the window system. A program running on the machine with your display provides these services, and so is called the X server. The server acts as an intermediary between you and the applications, handling output from the clients to the display, and forwarding input (typed on the keyboard or entered with a tablet or mouse) from you on to the appropriate clients for processing.

Clients and servers use a form of interprocess communication to exchange information. The syntax and semantics of the elements of this conversation are defined by a communication protocol. This protocol is the foundation of the X Window System. Clients use the protocol to send requests to the server to create and manipulate windows, to generate text and graphics, to control input from the user, and to communicate with other clients. The server uses the protocol to send information back to the client in response to

various requests, and to forward keyboard and other input generated by the user on to the appropriate clients.

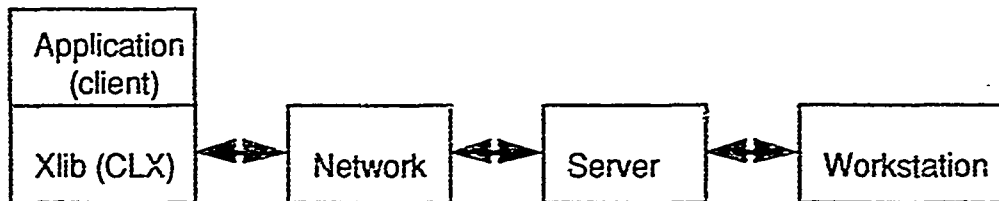
Since a network round-trip is a very expensive operation relative to basic request execution, the protocol is primarily asynchronous in nature, and data can be in transit in both directions (client to server and server to client) simultaneously. After generating a request, a client typically does not wait for the server to execute the request before generating a new request. Instead, the client continues immediately, generating a stream of requests which are eventually received by the server and executed. The server does not acknowledge receipt of a request, and in most cases does not acknowledge execution of a request. (This is possible because the underlying transport being used is reliable.)

The protocol has been designed explicitly to minimize the need to query the window system for information. Clients should not depend on the server to obtain information which the clients initially supplied. In addition, clients do not poll for user input by sending requests to the server. Instead, clients use requests to register interest in various events, and event notifications are sent asynchronously by the server. Asynchronous operation may be one of the most significant differences between X and other window systems you may be familiar with.

When client and server reside on the same machine, communication between them often will be implemented using shared memory, for highest performance. When client and server reside on different machines, communication can take place over any network transport layer providing reliable, in order delivery of data in both directions (usually called a reliable duplex byte stream). In particular, TCP (in the Internet protocol family) and DECnet streams are two commonly used transport layers. To support distributed computing in a heterogeneous environment, the communication protocol is designed to be independent of operating system, programming language, and processor hardware. Thus, it is possible to simultaneously run applications written in multiple languages under multiple operating systems on multiple hardware architectures, all sharing a single display.

Although X is fundamentally defined by a network protocol, most application programmers do not want to think about bits, bytes, and message formats. Instead, an interface library is used. This library provides a familiar procedural interface, masking the details of the protocol encoding and transport interactions, and automatically handling the buffering of requests for efficient transport to the server, much as the C standard I/O library buffers output to minimize system calls. The library also provides various utility functions which are not directly related to the protocol but which are nevertheless important in building applications. The exact interface for this library will differ for each programming language. The library for the C programming language is called Xlib. The library for the Lisp language is called CLX.

The figure below shows a block diagram of a complete X environment. Each X server controls one or more screens, a keyboard, and pointing device (typically a mouse) with one or more buttons on it. There may be many X servers; typically there is one per workstation on your network. Applications can run on any machine, even those without X servers. An application might communicate with multiple servers simultaneously (for example, to support computer conferencing between individuals in different locations). Multiple applications can be active at once on a single server.



Many facilities that are built into other window systems are provided by client libraries in X. You will not find specifications of things like menus, scroll bars, and dialog boxes, or the interpretation of particular key and button sequences in X. To the extent possible, the protocol and Xlib have been designed to avoid mandating such policy decisions. That is, the protocol and Xlib should be viewed as a construction kit, providing a rich set of mechanisms, with which a wide variety of user interface policies can be implemented. {cite:X Window System: C Library and Protocol Reference}

There have been many toolkits implemented on top of Xlib which provide higher-level graphics libraries for things such as menus, scroll bars, dialog boxes, etc..

The standard one of these is the MIT X Consortium's Xt tool kit for C., which is distributed along with X by MIT.

TI's CLUE is their attempt at a toolkit, to lie on top of CLX, that provides a conceptual model similar to that of Xt, capturing the intrinsics of Xt and merging them with CLOS to make these services available in lisp.

(It has been noted that the current release of CLUE has some problems in that it does not yet provide everything that Xt does. However, TI is in the process of extending CLUE and should address this immediate problem future releases of CLUE software.)

Our (Honeywell's) KUIE package is a toolkit, consisting of a set of object classes and methods that may be used to construct application user interfaces, designed to support KBSA activities.

Those KBSA activities demand network transparency- the ability to have an interactive program run on a remote computer yet control a UI on a local workstation. And to provide this, the KUIE system must be implemented atop a portable, networked window system, or provide such a system itself.

X achieves the main requirement of a networked window system. Furthermore, the number of platforms supporting X and the scope of the X standard (OSF, ANSI/ISO) means that X, today, already delivers a level portability that most other window systems may never achieve.

Since we are implementing KUIE on top of CLUE/CLX, and CLUE/CLX delivers not only X, but those higher level capabilities of the Xt toolkit, and on a CLOS platform besides, KUIE benefits by having to concern itself with a minimum of low level details.

Thus, we have very much brought into the X standard. X is out there, its being widely used, it is fairly mature, it works well, and its free.

We feel that the X standard, and the common ground for experimentation and development that X would provide, is (will be) very much an enabling contribution to the success of the KBSA program.

References ----- 1) X Window System: C Library and Protocol Reference Robert W. Scheifler, James Gettys, Ron Newman Digital Press, Bedford MA, 1988

``Standards, Enabling or crippling?"

Position Statement

Aaron Larson

Honeywell Systems & Research Center

Overview

Below I present an overview of CLOS, some comments on standardization in general and a brief statement about standardization in the KBSA.

Common Lisp Objects System (CLOS) overview.

CLOS defines an object oriented language embedded within Common Lisp. The ANSI X3J13 committee has formally adopted the CLOS specification as part of the forthcoming ANSI Common Lisp standard. CLOS is based on research and experience with a number of object oriented programming languages including SMALLTALK, FLAVORS, and LOOPS. The following (taken without permission of the authors) is an excerpt from the introduction to the specification.

The Common Lisp Object System is an object-oriented extension to Common Lisp as defined in "Common Lisp: The Language", by Guy L. Steele Jr. It is based on generic functions, multiple inheritance, declarative method combination, and a meta-object protocol. ...

The fundamental objects of the Common Lisp Object System are classes, instances, generic functions, and methods.

A "class" object determines the structure and behavior of a set of other objects, which are called its "instances". Every Common Lisp object is an "instance" of a class. The class of an object determines the set of operations that can be performed on the object.

A "generic function" is a function whose behavior depends on the classes or identities of the arguments supplied to it. A generic function object contains a set of methods, a lambda-list, a method combination type, and other information. The "methods" define the class-specific behavior and operations of the generic function; a method is said to "specialize" a generic function. When invoked, a generic function executes a subset of its methods based on the classes of its arguments.

A generic function can be used in the same ways that an ordinary function can be used in Common Lisp; in particular, a generic function can be used as an argument to FUNCALL and APPLY and can be given a global or a local name.

A "method" is an object that contains a method function, a sequence of "parameter specializers" that specify when the given method is applicable, and a sequence of "qualifiers" that is used by the "method combination" facility to distinguish among methods. Each required formal parameter of each method has an associated parameter specializer, and the method will be invoked only on arguments that satisfy its parameter specializers.

The method combination facility controls the selection of methods, the order in which they are run, and the values that are returned by the generic function. The Common Lisp Object System offers a default method combination type and provides a facility for declaring new types of method combination.

Standardization

In order for a standardization process to begin, several things must be in place. First the thing to be standardized on must be understood well enough to know along what dimensions the standardization effort should proceed. Secondly, there must be a "committed supporter". A committed supporter is a person/organization that is energized enough to start the standardization effort, find people interested in being on the standards committee and handle the initial administrative overhead associated with setting up communication paths among the members of the standards committee. Third, there must be enough other people that understand the issues involved and agree that standardization is worthwhile.

A standardization process should adhere to several policies. In order to gain widespread acceptance, the standardization process should involve the affected community as early as possible and remain open to community review. The resulting standard should remain under the control of the community, i.e. it should be "non proprietary". Since standards in software tend to embody abstract and complex concepts, the community needs to have hands on experience with the standard in order to understand the issues being standardized. The most desirable way to get hands on experience is to have an inexpensive (in dollars and time) reference platform. The standard should "be at the right level", specifically lower level details should either be hidden, or handled by referencing other standards that encapsulate them, of course if other standards are referenced it is important that the way the standards interact be described. And lastly, the standardization process itself should stress the codification of existing knowledge and practices over the development of new ideas (standards organizations are poor places to do research).

There are several effects of a standard. Standards tend to leverage effort because people have some ownership of the ideas, expect that the standard will be valid for a period of time, and a belief that the standard will leverage effort. Typically standards restrict research in the area standardized on (if you have a standard presumably you understand the problem well enough that further research is not necessary). Also, since standards are rarely complete solutions to a problem, they tend to bring researchers together during the standardization process and point out (presumably higher level) areas where further research is necessary prior to standardization. Lastly, standards are slow to change.

The CLOS standardization process was instigated by several people who had substantial knowledge of object oriented programming system development. Xerox Corp sponsored a development effort to create a freely available reference model and set up a forum where interested people could communicate. During the development of the reference model, many substantial changes were made based on the usage of a variety of users in industry, and academia. The resulting standard has been turned over to ANSI and included as part of the Common Lisp standard. To date the CLOS specification has been widely distributed and accepted by a large community, including most of the Common Lisp vendors.

Standardization in the KBSA

The decision to standardize something is a cost trade off. On one hand standardization tends to reduce redundant efforts, on the other hand it tends to slow the evolution of the thing being standardized on.

Before the KBSA can achieve a tight integration, it is necessary that a standard computational model and notation be agreed upon so that cooperative development of the information structures of the overall KBSA can begin. Although the selected computational model may not be the one used by the final KBSA, it is important that multi facet concerns begin to be addressed within a common formalism. The price paid for standardizing on the computational model is that we will undoubtedly be developing better computational models during the KBSA research, and hence will not be using the "latest and greatest" language structures in the standard. I believe this is a necessary cost of cooperative research on a project as complex as the KBSA. The complexity of the information model for the KBSA is going to require hands on experience by a number of people, and I believe that the development of the overall information model of the KBSA must begin to take precedence over the development of new computational systems.

The Information Resource Dictionary System Standard

Dr. Henry C. Lefkovits
AOG Systems Corporation
Harvard, MA

The Information Resource Dictionary System (IRDS) became an ANSI Standard in October 1988 and a Federal Information Processing Standard (FIPS) in April 1989. The IRDS is a software tool which can be used to control, describe, protect, document, and facilitate use of an installation's information resources.

The IRDS database is composed of the IRD Schema and the Information Resource Dictionary (IRD); the former contains a description of the structure of the IRD. Both use a strongly typed Entity-Relationship model. Relationships are binary, and both entities and relationships can have attributes. Attributes can be grouped to a single level.

The IRDS contains extensive facilities that allow versioning and life cycle phase control in the IRDS database. Multiple versions of an entity can exist and an audit trail is maintained. Security facilities for access control are specified.

The IRDS also contains facilities that allow an installation to fully customize the IRD Schema; hence, the IRDS can be used to model the information resources environment of interest to that installation. The IRDS can then serve as the central repository for the descriptions of all the information resources of interest.

The IRDS Services Interface proposed standard is currently undergoing public review. This standard specifies a low-level programmatic interface to the IRDS database. This interface will enable access to the IRDS by those tools used by the installation which require access to metadata.

An important aspect of the IRDS is that not only can the IRDS serve as a central repository at a single site, but facilities are specified which allow transfer of metadata from one Standard IRDS to another Standard IRDS. Information can be extracted selectively and sent to another IRDS; the format of the transfer file is the subject of a forthcoming standard entitled the IRDS Export/Import File Format.

RELATING FORMAL AND INFORMAL DESCRIPTIONS OF SYSTEMS*

Lewis Johnson and Jay J. Myers

USC Information Sciences Institute
4676 Admiralty Way Marina del Rey, CA 90292
(213)822-1511 johnson@isi.edu jmyers@isi.edu

Lewis Johnson is a Research Assistant Professor of Computer Science at USC and project leader of the ARIES project at ISI. He has a Ph.D. in Computer Science from Yale University.

Jay Myers is a member of the ARIES project at ISI working on the acquisition, explanation and simulation of formal specifications. He has a Ph.D. in Psychobiology from the California Institute of Technology.

Abstract

Communications in natural language are important for knowledge-based software assistants. Natural language understanding tools can assist the requirements acquisition process. On the other hand, translating formal system specifications into English can aid understanding and validation. We are now extending the natural language communication capabilities of our system, aiming toward automated generation of documentation. In so doing, we are breaking down the barrier between informal and formal specifications. Natural language is now viewed not only as a communication medium, but also as an intrinsic part of system representations themselves. We are adopting semi-formal representations, which intermix natural language and formal notations. Semi-formal descriptions also serve as an appropriate medium for requirements acquisition, allowing engineers freedom to delay modeling commitments.

1. Introduction

Communications in natural language have long been recognized as essential to the success of knowledge-based software assistants. For several years now, ISI has been conducting research in natural language in the context of developing

* This research was supported by the Air Force Systems Command, Rome Air Development Center, under Contract F30602-85-C-0221. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

formal specifications. While we represent our specifications in a formal language. Gist [4], we regard natural language to be an important medium for communicating requirements and specifications. Clients express their needs in natural language, and are generally unable to interpret formal specifications. System developers also benefit from natural language summaries of formal specifications, as an aid to understanding and validation. We therefore have been engaged in building tools for translating between natural language and formal descriptions of systems. In this paper we will assess the status of our natural-language-related efforts, noting where technology spinoffs are under way and where technical effort will be required to achieve further progress.

The SAFE project (Specification Acquisition From Experts) was an early natural language effort at ISI [1, 3]. The SAFE system would analyze paragraphs of natural language text describing a system to be built. From this input, it would then attempt to generate a formal specification of the system. It made use of context information to complete the informal descriptions. Among the natural language imprecisions it could resolve were: supplying missing operands for actions and relations, completing incomplete references, determining the scope of conditional statements and detecting implicitly stated associations among objects.

Working in the opposite direction, the Gist Paraphraser [13] generates natural language translations in order to make formal Gist specifications more understandable. It can be used to paraphrase individual specification constructs in English, or to output a narrative summary of an entire specification. The Gist Behavior Explainer [14] similarly generates English explanations of behavior traces output by symbolic evaluation of a specification. These tools were initially developed as part of the Tools for Specification Validation and Understanding project of RADC [2], and subsequently revised and extended in the Knowledge-Based Specification Assistant project [7]. The coverage and versatility of the

Paraphraser have been significantly extended, so that it now covers all constructs in Gist, and can generate a range of possible natural language paraphrases [9]. Figure 1-1 shows an example translation of Gist by the Paraphraser.

```
type physical-object
  with {singleton relation
        physical-object-location(location)};

type aircraft subtype of physical object
  with {procedure land []
        definition
          atomic {remove in-flight(),
                  update self.physical-object-location
                      to a ground-location}}
```

All aircraft are physical-objects. Each physical-object has one physical-object-location.

Land is a procedure of an aircraft. To perform a land, the system simultaneously (atomically) does the following. It deletes the fact that the aircraft is in-flight. It updates the physical-object-location of the aircraft to any ground-location.

Figure 1-1: Example of Paraphraser output

2. Assessment of Previous Work

We now have considerable experience in paraphrasing formal specifications. Due to the careful attention given to its robustness, efficiency and ease of use, the Gist Paraphraser has become a practical facility. It has served as an effective tool for validating specifications. By presenting a significantly different view of a specification, the Paraphraser frequently uncovers problems which are less evident in the original formal syntax. For example, it often reveals missing count restrictions on relations, such as the restriction that physical objects have only one location.

The Paraphraser also has proven to be a useful tool for training. We have made extensive use of the Paraphraser in our training courses for the Knowledge-

Based Specification Assistant, both when teaching the Gist language and when demonstrating the evolution transformations performed by the Assistant. Trainees find it helpful to be able to make arbitrary changes to a specification and to see these changes reflected in the natural language paraphrase.

While our natural language paraphrasing capabilities have continued to improve, we have not addressed the problem of interpreting natural language input since the SAFE project. We are not alone in this regard: other researchers, such as the developers of the Ψ system [5], have not pursued further research in natural language understanding. There are a number of reasons for this retrenchment.

First, natural language understanding systems have great difficulty tolerating incomplete knowledge. When a sentence contains a number of words unknown to the system, its meaning is much more difficult to discern. Current understanding systems have been successful primarily when they can exploit extensive domain-specific knowledge with restricted natural language inputs. Natural language generation systems, on the other hand, can more readily function with incomplete knowledge. For example, while the Paraphraser can utilize grammatical information about selected definitions to help it to formulate appropriate translations, it also has default methods to produce somewhat clumsy, but understandable, translations when the grammatical information is not provided.

Second, natural language understanding systems often fail to scale up because of the increasing problem of ambiguous interpretations. In the SAFE project, ambiguity was resolved by having the user choose among possible interpretations. However, this reliance upon user intervention is likely to increase as the system scales up. Furthermore, there is a risk that such choices of interpretation will be forced at the wrong time. Requirements analysts need the freedom to focus on

restricted aspects of the problem at certain times, and to postpone modeling decisions until they become important. A naive design for a natural language understanding system would force the user to choose an interpretation whenever an ambiguity arises, to allow the system to parse the input. However, making choices of interpretation, e.g., deciding which natural language statements describe events, states and objects, involves making modeling decisions. An effective natural language processing system must tolerate ambiguity and delay interpretation until enough information becomes available to resolve the ambiguity, or until the analyst is ready to make the necessary modeling choices.

3. Application: The Documentation Problem

Of the natural language processing tools that we have developed, the Paraphaser is the one that is closest to practical applicability. We have recently undertaken to put it to the test in our own work, in order to generate documentation and on-line help for our evolution transformations (described in these proceedings [8]).

The benefits of documentation to people trying to understand systems have been well established through empirical studies. Schneiderman's results [11] emphasize the importance of providing the right kind of documentary information, i.e., data structures vs. control flow; whereas the choice of medium used to present the documentation, i.e., text vs. graphics, was not important. Soloway's work [12] shows that documentation is often consulted on an "as needed" basis when the reader has particular questions that need answering. A good documentation system must therefore be flexible enough to provide information from a number of different points of view, so that the user's questions are likely to be answered. The notation of the documentation must be rich enough to accommodate these different points of view. Natural language text is the only medium with the versatility to capture the wide range of documentary information that is needed.

Documentation is particularly important for the evolution transformations used to elaborate a specification. Because evolution transformations are a new concept, they are unfamiliar to potential users. We need to be able to provide a variety of types of information about these transformations and to present this information at various levels of detail. The user may wish to understand the function of a transformation in general terms, or to receive a detailed account of what effects it will have on the specification. Users need to be able to examine how a transformation works and to determine what substeps are involved (these substeps may be useful to the user even if the transformation as a whole is not). Each transformation applies to a series of arguments; the user needs to know what each argument is, what its type is, how it will be used and how it is input to the system.

To meet these needs, we developed an on-line help facility for the evolution transformations in the Specification Assistant. Figure 3-1 shows the on-line help for the command **Parameterize**. It provides a help string describing the command, lists the inputs required, and indicates which preconditions must be met. The system is also able to provide interactive help when transformations are applied. That is, when the user is trying to apply a **Parameterize** transformation, the system will step the user through the process, prompting for each input with messages such as "Please mark the the term to be parameterized (a parameterizable declaration). Mark with meta-Mouse-L". These messages are constructed by examining the attributes of the parameters and assembling text strings. In addition, we automatically generate similar off-line documentation for inclusion in our reports and manuals.

In order to support both on-line help and documentation, we developed a representation for the transformations that combined formal descriptions and explanatory natural language text. Each form defined a nested collection of objects and attributes, where some of the attributes were formal and some were

Plan: PARAMETERIZE Add a parameter to a declaration, and attempt to generalize references to use new parameter. Inputs: DECL: Term To be parameterized, a Parameterizeable construct; chosen by (Meta-LeftMouse) NEW-ROLE: New parameter, a role; chosen by (User typein) Precondition: The new parameter type must be declared, and there cannot be outstanding static analysis errors.
--

Figure 3-1: Documentation for the Parameterize command

not. The following is a breakdown of the common attributes and their values.

- name - symbol
- help-string - text
- display-name - text
- inputs
 - name - symbol
 - class - type
 - display-name - text
- precondition
 - form - Lisp code
 - failure-message - Lisp code
 - display-string - text
- method - Lisp or Paddle code

Although a useful start, this representation had a number of shortcomings.

- The preconditions and methods were written in Lisp code; they therefore could not be paraphrased in English. Instead a corresponding message string had to be entered by hand.
- The definitions often contained multiple text messages which were closely related. In particular, for the preconditions we had to enter both a display string describing the precondition, and a piece of Lisp code for producing the error message to display if the precondition

fails. The error messages potentially could have been derived from the precondition descriptions.

- Crucial information for explaining the transformations was missing. In our Specification Assistant training course, we made extensive use of examples in explaining transformations, but the examples were not a part of the on-line documentation facility itself.

We are redesigning our representation for transformations to help address these problems and to allow us to provide better documentation. We have also begun to use an executable subset of Gist as our transformation language, rather than Lisp and Paddle. We expect to be able to use the Paraphraser to describe the parts of the transformations written in Gist. We will then make less use of text strings, and at the same time generate a wider range of documentation. We also are including pedagogical examples in the description of the transformations. This will allow us to automatically generate the kinds of example-oriented documentation that we have found to be useful in our training work.

At the time of this writing we have converted about half of our transformations to use Gist. In the meantime we have moved to a new version of Gist, and the Paraphraser has not yet been fully converted over to use it. However, by comparing the Gist text and the natural language descriptions, and by running examples through the Paraphraser in the old Gist, we can get a sense now of how useful the Paraphraser will be in generating documentation.

Figure 3-2 shows the new representation for *Parameterize*^{*} which now explicitly includes demonstration examples. By comparing the Gist expressions and the corresponding natural language text, we can make the following observations. First, some of the information provided in documentary strings is

^{*} Note that we have made some minor syntactic changes to Gist, to make it somewhat closer to REFINE and related languages. We use ":" to mean "of type", "." for attribute retrieval and "|" to mean "such that".

Transformation: Parameterize

Concept description: "Add a parameter to a declaration, and modify references to include new parameter."

Parameters:

Form: decl : concept-declaration
Display Name: "Term to be parameterized"
Form: new-parameter : parameter
Display Name: "Parameter to be added"
Form: expression : expression-tree
default dummy-actual(new-parameter)
Display Name: "The new actual"
Concept Description: "An expression to be inserted into references to the concept, to compute the new actual"
Notes: "The expression may freely use the names of existing formal parameters to refer to their corresponding actual values"

Precondition:

declared(name(object-type(new-parameter)),
 'type-declaration,
 decl)

Failure Message: format(nil,
 "~A is not declared",
 name(object-type(new-parameter)))

Concept Description: "The type of the new parameter must be declared"

Method:

steps [add-parameter-to-signature[new-parameter, decl];
 add-actual-computed-wrt-existing-actuals
 [decl, expression]]

Example Spec:

```
{type aircraft;  
  relation controlled(ac : aircraft);  
  relation in-flight(ac : aircraft);  
  invariant foo for-all ac : aircraft |  
    in-flight(ac) => controlled(ac)}
```

Example Invocation:

```
parameterize  
  [declaration-of('controlled, 'relation-declaration),  
   gist-template("c : controller"),  
   declaration-of('controller, 'type-declaration),  
   gist-template("ac.assigned-controller")]
```

Figure 3-2: New definition of Parameterize

simply absent from the Gist definitions. For example, one of the arguments to `Parameterize`, the expression which will serve as the new parameter, carries with it a range of documentation:

- Display Name - the phrase to use when referring to the argument
- Concept Description - a description of the function of the argument
- Notes - supplementary information about properties of the parameter

None of these is reflected in the Gist representation. Furthermore, it would be inappropriate to extend the formal representation solely for the purpose of generating this additional text. The use of the Paraphraser therefore will not make documentary strings unnecessary.

The Paraphraser will allow us to generate text which we could not previously provide as documentation. In particular, we can paraphrase the parameters, method and preconditions of a transformation as well as the example specification and invocation. For example, the Paraphraser could generate the following description of the steps comprising the `Parameterize` transformation:

`Parameterize` is a transformation. To perform a `parameterize` a involving a `decl`, a `new-parameter` and an expression, the Assistant does the following. It invokes `add-parameter-to-signature` with the `new-parameter` and the `decl`. It invokes `add-actual-computed-wrt-existing-actuals` with the `decl` and the expression.

A major concern when generating documentation is to produce text that is meaningful to the user. In some cases, the surface description by the Paraphraser is inadequate. Consider the following paraphrase of the precondition of `Parameterize`:

There is a precondition that the `DECLARED` relation must hold between the name of the object-type of the `new-parameter`, 'type-declaration, and `decl`.

Without additional information it is not possible to get the output that we would really like:

There is a precondition that the object-type of the `new-parameter`

must be declared as a type-declaration in the scope of decl.

This manually generated text relies upon information that is absent from the formal representation. It indicates the significance of each parameter: that one defines the scope in which the type must be declared, while another indicates that the declaration should be a type declaration, as opposed to a relation or event declaration. It would not be appropriate to extend the formal representation solely for the purpose of generating better descriptions of the parameters, such as "in the scope of decl". Rather, we would like to explore ways of supplying advice to the Paraphraser in the form of short text strings that it can insert into the text when composing a paraphrase. The presumption is that these advice strings will appear in a fixed form, whereas the surrounding text, which structures the paraphrase, will vary.

We conclude from these examples that the Paraphraser can assist in the documentation process, but only if we supplement our formal descriptions of transformations with textual phrases. We have thus adopted what we call a *semi-formal notation*, i.e., one where formalism and text are intermixed, and where the text is categorized to indicate its semantic role. Given such a semi-formal framework, we believe the Paraphraser can be used to generate new text descriptions as needed.

4. Using Semi-Formal Notations

Our interest in semi-formal representations initially arose in connection with plans to integrate requirements acquisition and specification development. We believe that semi-formal notations potentially have a pervasive role in the development and maintenance of software. Since code and documentation are both products of software development projects, there are advantages to developing them both together. Concurrent maintenance of code and documentation is important in order to keep the documentation up to date.

Semi-formal notations help make this possible.

We make one further conjecture: that not only should the representation be semi-formal throughout, but the representation should gradually evolve, starting from primarily informal notations and becoming progressively more formal. When specification components are initially introduced, they will be largely informal, consisting of natural language descriptions of what the finished component will do. The first step toward formality is to categorize and interlink these descriptions forming a kind of hypertext. Some of the informal descriptions will then be incrementally supplemented and supplanted with increasingly formal descriptions (e.g., components hierarchies, data flow structures, specification language constructs). The representation never becomes exclusively formal, because the ability to generate high-quality documentation would be lost.

One reason that the gradual formalization approach is attractive is that it allows the developer to avoid premature modeling commitments. Initially writing specifications entirely in a formal notation forces modeling commitments from the very beginning: one needs to determine which concepts to model as types, which to model as relations or events, and indeed which concepts to model at all. However, the appropriateness of a domain model is determined by whether it allows the specifier to describe the components of the system and the entities with which the system will interact. This results in a Catch-22 situation: one cannot formalize the domain without formalizing the system's requirements in that domain, and one cannot formalize those requirements without stating them in terms of domain concepts. Incremental formalization breaks this impasse.

The following example illustrates this point. Aircraft have many properties, including location, heading, speed, maneuverability, make, model, transponder-id and number of passengers. Until we know what properties are relevant to a

particular air traffic control system. we do not know which to incorporate into our model. We might think that make and model are unimportant. but if we wish to distinguish military, commercial and civilian aircraft. knowing the make and model would be useful. If we choose not to model certain aspects of aircraft. it is still important to retain informal text describing those features that we choose to omit. This is particularly important if we are working from a natural language requirements document or Statement of Work; then it is crucial to be able to account for how each natural language statement is reflected in the specification.

This accountability is another important feature of an incremental approach to formalization. We expect many specification concepts to be ultimately grounded in the initial natural language statements. By taking a principled step-wise approach to the evolution of formal specification concepts from their informal beginnings, we can produce a record of the evolution steps linking together the inputs and outputs of each step. Such a record will allow a developer to trace the acquisition of a formal specification concept in order to better understand and debug the concept or perhaps to choose a different path in order to formalize its informal precursors.

We intend to adopt this gradual formalization approach in the ARIES (Acquisition of Requirements Integrated with Evolution of Specifications) system. It is fully consistent with the approach to informality taken in the Knowledge-Based Requirements Assistant [6, 10]. We view the use of informality as a specification freedom, one which is gradually removed. The semi-formal framework has the potential to extend the power of the "catch as catch can" natural language interpretation employed by the Requirements Assistant, to go beyond acquisition of requirements to formalization. Since all natural language will appear within the context of a structure with formal semantics, the context resolution problem is substantially resolved. Furthermore, when the system is

unable to interpret natural language. it will carry along the natural language until formal interpretation becomes possible or the developer chooses to supply an interpretation by hand. We thus foresee a means of providing the kind of support for informality envisioned in SAFE. in the absence of significant advances in natural language understanding.

References

1. Balzer, R., N. Goldman, and D. Wile. "Informality in program specifications". *IEEE Transactions on Software Engineering SE-4*, 2 (March 1978), 94-103.
2. Balzer, R., Cohen, D., and Swartout, W. Tools for Specification Validation and Understanding. Tech. Rept. RADC-TR-292, Rome Air Development Center, December, 1983.
3. Balzer, R., Goldman, N., and Wile, D. On the Use of Programming Knowledge to Understand Informal Process Descriptions. Proceedings of the Pattern-Directed Inference Systems Workshop, 1978.
4. Goldman, N., Wile, D., Feather, M., and Johnson, W.L. Gist Language Description. USC Information Sciences Institute, 1989.
5. Green, C. The Design of the PSI Program Synthesis System. Proceedings of the 2nd International Conference on Software Engineering, 1976.
6. Czuchry, A.J. and Harris, D.R. "KBRA: A New Paradigm for Requirements Engineering". *IEEE Expert* 3, 4 (Winter 1988), 21-35.
7. The KBSA Project. Knowledge-Based Specification Assistant: Final Report. USC Information Sciences Institute, September, 1988.
8. Johnson, W.L., and Feather, M. Evolution Transformations: Results and New Directions. Proceedings of the 4th Annual Knowledge-Based Software Assistant Conference, 1989.
9. Myers, J.J., and Johnson, W.L. Towards Specification Explanation: Issues and Lessons. Proceedings of the 3rd Annual Knowledge-Based Software Assistant Conference. 1988.
10. Sanders Associates. The Knowledge-Based Requirements Assistant - Final Technical Report. Software Systems Engineering Directorate, February, 1988.
11. Schneiderman, B. "Control Flow and Data Structure Documentation: Two Experiments". *Communications of the ACM* 25, 1 (January 1982).
12. Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. "Designing Documentation to Compensate for Delocalized Plans". *Communications of the ACM* 31, 11 (November 1988).
13. Swartout, W. GIST English Generator. Proceedings of the National Conference on Artificial Intelligence, AAAI, 1982.

14. Swartout, W. The Gist Behavior Explainer. Proceedings of the National Conference on Artificial Intelligence. AAAI. Washington, D.C., 1983. (Also available as ISI/RR-83-3).

Relating Formal And Informal Descriptions of Systems

W. Lewis Johnson
Jay J. Myers

USC / Information Sciences Institute

Overview

Communication via informal descriptions is important for KBSA's

- for communication with clients
- to aid in understanding and validation
- to aid in training

Two key approaches have been explored:

- informal \leftrightarrow formal translation
- tools for structuring informal descriptions (e.g., hypertext)

This talk will assess current support for informal descriptions in KBSA

- technical spinoffs
- places where future work can achieve high payoff
- focus of ARIES work to achieve payoff

Informal-Formal Translation

- The SAFE project analyzed natural language text to generate specifications. Also Ψ system, KBRA system.
- The Gist Paraphraser generates English from Gist specifications. KBRA also generates English.

Status:

- Natural language input has not passed the toy example stage.
- Powerful, broad-coverage output capabilities now exist.
- Demonstrated utility in validation, training, generation of 2167 documents

Example Paraphrase

Gist text:

```
type physical-object
  with {
    singleton relation
      physical-object-location(location)};

type aircraft subtype of physical-object
  with {
    procedure land[]
      definition
        atomic {
          remove in-flight();
          update self.physical-object-location
            to a ground-location}}
```

Paraphrase:

All aircraft are physical-objects. Each physical-object has one physical-object-location.

Land is a procedure of an aircraft. To perform a land, the system simultaneously (atomically) does the following. It deletes the fact that the aircraft is in-flight. It updates the physical-object-location of the aircraft to any ground-location.

Problems with Natural Language Input

- intolerance of incomplete knowledge
- context-dependent
- can force premature modeling and design decisions

An Example

From BARTCC requirements:

"Up to 2 student positions which have all OCP controller capabilities."

Issues to resolve:

- What is a position? -- a computer console, in a particular operational mode.
- Are there 2 student positions, or 2 positions with all capabilities?
- Do we really mean all capabilities?

Another Approach: Hypertext

Existing systems:

- gIBIS
- IDE

Hypertext objects are used to describe:

- domain concepts and requirements
- design decisions
- design components
- rationales

Arcs link decisions to arguments and rationales, decisions to component specifications

Hypertext supports *knowledge identification*, not *knowledge representation*.

IDE

Technology

Students

Instructional Context

Domain Knowledge

Principles

Rationale

Principle: Advanced Organizer

R

Principle: Causality supports inference

Doc: Fault Predictions

Prepare for troubleshooting

R

Doc: Cause-effect relations

Students should be able to predict the effects of each of the 7 processes being nonfunctional.

Hierarchical knowledge structures are reconstructable in top-down order.

Knowledge of the functionality of an artifact is often used in reasoning about that artifact.

Instruction

Instructional Rules

Pedagogical Rules

Strategy Rules

Tactical Rules

Pedagogical Rules

(Teach BX) =>

{ (Teach "XeroGraphic Processes")

{ Teach "Seven Steps" }

{ Teach "Cause-Effect" }

}

}

Lessons from Hypertext Experiments

- Knowledge identification is a crucial part of requirements analysis, and must be supported.
- Hypertext is a suitable medium for knowledge identification.
- Structured hypertext more valuable than unstructured hypertext.
- Tools ready for application

Outstanding challenges:

- Need to combine informal and formal descriptions.
- Need to support incremental formalization.

ARIES Goals

- Tight integration of hypertext and formal descriptions
- Reusable knowledge base resulting from both knowledge representation and knowledge identification
- Incremental formalization of semi-formal hypertext
- Regeneration of informal text from formalized descriptions

Accomplishments So Far

- Partial integration of hypertext and formalisms
- application on editing transformations
- application on ARIES metamodel
- beginning application on ATC domain knowledge and requirements

Our Model of Hypertext

Basic building block: hyperstrings - strings containing objects

Object attributes can take hyperstrings as values

Attributes currently supporting hyperstrings:

- display-name
- concept-description
- notes

Objects thus can have multiple hyperstring attributes -- including names!

Coverage will gradually be extended to most object attributes

Mixed model for transformations

- `display-name` - hyperstring
- `parameter` - formalized, plus:
 - `display-name` - hyperstring
 - `concept-description` - hyperstring
 - `notes` - hyperstring
- `precondition` - formalized, plus:
 - `concept-description` - hyperstring
 - `failure-message` - template
- `method` - formalized
- `effects` - formalized
- `example-spec` - formalized
- `example-invocation` - formalized

Transformation: Parameterize

Concept description: "Add a new *parameter* to a *concept-declaration*, and modify references to include references to the *added parameter*."

Parameters:

Form: `decl : concept-declaration`
Display Name: "*concept-declaration* to be parameterized"
Form: `param-name : symbol`
Display Name: "Name of the *added parameter*"
Form: `param-type : type-declaration`
Display Name: "type of the *added parameter*"
Form: `expression : expression-tree`
default `dummy-actual(new-parameter)`
Display Name: "The new *actual*"
Concept Description: "An *expression* to be inserted into references to the *concept-declaration*, to compute the new *actual*"
Notes: "The *expression* may use the names of existing formal *parameters* freely to refer to their corresponding *actual* values"

Outputs:

Form: `added-parameter : parameter`
Display Name: "*added parameter*"

Precondition: `not exists x:parameter | (parameter-of(decl, x) and name(x, param-name))`

Failure Message: "A *parameter* named [*param-name*] already exists"

Concept Description: "A *parameter* of the same name must not exist"

Method:

```
steps [add-parameter-to-signature[new-parameter, decl]
      yielding added-parameter;
      add-actual-computed-wrt-existing-actuals[decl,
      expression]]
```

Example Spec:

```
{type aircraft;
 relation controlled(ac : aircraft);
 relation in-flight(ac : aircraft);
 invariant foo for-all ac : aircraft |
   in-flight(ac) => controlled(ac)}
```

Example Invocation:

```
parameterize[declaration-of('controlled,
                             'relation-declaration),
             gist-template("c : controller"),
             declaration-of('controller, 'type-declaration),
             gist-template("ac.assigned-controller")]
```

Lessons Learned So Far

Hypertext and objectbases can be merged

Adjustable thickness is preferable

Hypertext useful even on "formalized" objects, for documentation

Conjectures to Verify

Incremental formalization can be effective approach to natural-language input

Hyperstring templates can improve natural-language generation capability

KBSA Technology Transfer: An Industrial Perspective

Chunka Mui and Michael DeBellis

Andersen Consulting
Center for Strategic Technology Research
100 South Wacker Drive
Chicago, Illinois 60606

(312) 507-6318, (312) 507-6530
Chunka@andersen.com, DeBellis@andersen.com

Biography

Chunka Mui is an Associate Scientist in the Software Engineering Lab (SEL) of Andersen Consulting's Center for Strategic Technology Research. He is the project leader for Andersen Consulting's KBSA-related efforts. Michael DeBellis is a Research Associate in the Software Engineering Lab.

Abstract

In the project plan proposed in their 1983 KBSA report, Green et al. predicted that the Phase 1 KBSA facet research efforts would produce beneficial technological fallout. This would be in the form of intermediate products that might have immediate application for a broad segment of the software community. In order to facilitate the transfer of such intermediate results, RADC established the KBSA Technology Transfer Consortium in 1988. We have worked as an alpha test site within the consortium since its inception to evaluate the Phase 1 results and transfer relevant concepts and tools into our systems development practice. The results of our evaluation have been mixed. The consortium itself has been very effective in establishing a channel for information exchange between ourselves and the KBSA developers. However, the current candidates for technology transfer have less immediate application than we had hoped. Some reasons for the lack of significant technology transfer include the inadequate attention paid thus far to (1) achieving significant relevance to industrial practice, (2) tackling certain important life-cycle areas, and (3) validating and communicating current research results. We will expand on these problems and offer some suggestions for dealing with them. We will also suggest ways of facilitating future technology transfer. We assume that the reader is familiar with the KBSA program.

1. Introduction

The original project plan proposed in the 1983 KBSA report[1] outlined a 15 year, three stage KBSA development effort. The overall plan adopted by RADC called for the simultaneous development of individual KBSA facets through parallel research efforts. Each facet development effort was structured to produce intermediate products in the short-term (3 to 5 years) and mid-term (5 - 10 years). It was hoped that these intermediate products would have immediate practical application.

In 1988 (roughly at the short-term milestone), RADC established the KBSA Technology Transfer Consortium to facilitate the transition of Phase I KBSA technology.[2] The consortium provides a formal relationship between KBSA developers, potential users, and RADC designed to support the technology transfer process. This relationship is designed to enhance the low-bandwidth, bilateral relationships which previously existed between some consortium members and to foster communication that previously did not exist.

In this paper, we describe the results of our KBSA technology transfer activities over the last year. In section 2 we analyze the contributions and shortcomings of the facet projects from the standpoint of technology transfer to industry. In section 3 we discuss problems which relate to industrial software which have not been adequately addressed by KBSA. In section 4 we present some suggestions for integration and standardization which we believe will facilitate future technology transfer. Section 5 presents our conclusions.

2. Facet Analysis

The 1983 KBSA report predicted that Phase I would reveal the sparkling facets of an emerging gem. We feel, rather, that it identified a promising vein. Bits of sparkle can be seen. But it is not clear whether they are small, isolated specks or the tips of a still-concealed precious diamond. In this section, we briefly summarize our analysis of the individual facets. We highlight some of the concepts that we found valuable and some which prove to be less worthwhile. In cases where we experimented with the actual software prototypes, we will offer some reflections from that perspective. For the sake of brevity, we assume that the reader is familiar with each of the facets. Later sections will provide a more holistic analysis of the Phase 1 research results.

2.1. Knowledge Based Requirements Assistant

The Knowledge Based Requirements Assistant (KBRA) [3] provides insights into addressing early stages of systems development. Two important concepts in KBRA are the presentation based architecture and

the "catch as catch can" approach to capturing and formalizing requirements. The presentation-based architecture provides the user with a powerful method for reviewing the evolving requirements. It is a promising approach for general editing and manipulation of requirements throughout the various facets.

The "catch as catch can" approach to capturing requirements in the KBRA has both positive and negative points. This approach conforms with our belief that the requirements gathering process cannot be completely well-structured or predefined. No tool can have the complete domain or process knowledge necessary to turn requirements gathering into a "cookbook" process. The tool must therefore provide a "safety net" which allows the analyst to input information that the tool does not understand.

The weakness of the "catch as catch can" approach is that it provides too little structure. The KBRA lacks the facilities to provide feedback to the user on the consistency and completeness of the requirements. It also provides little assistance to help guide the analyst through the requirements analysis process. This lack of support results from the absence of either a formal requirements gathering process model or a formal requirements language.

The input mechanisms developed in KBRA included a valuable experiment in the use of natural language processing (NLP) techniques for requirements gathering. The Intelligent Notepad utilizes a case-frame NLP scheme to interpret natural language requirements. We concluded from our own experiences and from the results of the KBRA that natural language input is not a scaleable approach. Natural language is ambiguous. There is a great deal of variance in usage and terminology between individual speakers and across various domains. An NLP requirements tool that has to work robustly on industrial size problems would require an unrealistically complete knowledge-base of both application and requirement analysis domain concepts.

2.2. Specification Assistant

The Specification Assistant[4] made clear progress towards the KBSA vision of systems development. It

provides the user with tools to develop and reuse specifications which are represented using the Gist[5] specification language. Gist allows for the description of a wide range of specification information, ranging from abstract descriptions of the domain to compileable descriptions of program behavior. Elaboration of the Gist specification is supported through user-directed transformations known as high level editing commands (HLEC). The HLEC's represent a step towards the goal of machine-assisted formal implementation. The HLEC's may form the basis for developing higher level plans or strategies for specification development. Along the same lines, the annotations supported by the Specification Assistant might be used to provide process guidance.

The Specification Assistant provides valuable analysis tools that support specification evaluation. The static, resource, and ontological analysis tools provide straight-forward but critical consistency checking mechanisms.

The Specification Assistant demonstrates that paraphrasing is a useful and realizable technique for specification understanding. We found that the paraphraser worked well. It consistently generated natural sounding text and worked in a reasonable amount of time. On a number of occasions we found the paraphraser useful in clarifying the meaning of a Gist specification. Paraphrasing helps to alleviate the cognitive complexity of formal notations like Gist. This is important not only to support the specification developer, but also to make the specification understandable to others (e.g. end-users, managers, and maintainers) involved in the system development process.

The symbolic evaluator uses a special purpose theorem prover to generate behaviors from a high level Gist specification. This supports the analysis of Gist specifications which are too abstract to be compiled. However, in our experiments we could not successfully apply the symbolic evaluator to even small specifications. It either failed to terminate after running for several minutes or aborted.

We believe that the problems with the symbolic evaluator have more to do with the technique in general

than with the current implementation in the Specification Assistant. As with many applications which utilize theorem proving, the primary problem is constraining the theorem prover so that it does not spend most of its time generating useless information. If a specification is not set up correctly, the symbolic evaluator will generate large quantities of uninteresting behaviors. In this case "setting up correctly" means adding many constraints (which would not be a part of the normal specification) to help the theorem prover avoid generating uninteresting behaviors. While we do not doubt that advances will continue to be made in the use of theorem provers in specialized domains, we feel that this approach has not yet been proven viable for realistic problems in specification analysis.

2.3. Project Management Assistant

The Project Management Assistant[6] (PMA) explores a number of concepts needed for effective project management which are not adequately supported in current generation tools. The PMA's most significant contributions are the development of a sophisticated representation for reasoning about time and a model of the concepts involved in project management.

The ability to reason about time is an important aspect of any project management system. The PMA work extends the interval based representation of time developed by James Allen[7]. This provides a foundation for the PMA to represent concepts such as one task overlapping another, one task preceding another, etc. This general mechanism for representing time could be very useful not only for project management but for other KBSA research issues as well.

The project management model serves as the representation for abstract process models (such as Boehm's Spiral Model[8]), instantiated for a particular project. At this point in the PMA development, it seems that such process models could function as on-line documentation or passive record keeping aids. The PMA will offer significant benefits if its capabilities are expanded to the point where it interacts with project managers to provide insight on non-trivial problems and inconsistencies.

2.4. Performance Estimation Assistant

The Performance Estimation Assistant[9,10] (PEA) made valuable progress in providing guidance during the transformation from program specifications to code. It showed how performance information could be used to guide data structure selection and transformation of iteration constructs. The PEA is a part of the larger Kestrel Interactive Development System (KIDS). KIDS utilizes information provided by the PEA to assist in the application of interactive transformations to programs written in a single-assignment functional language called Performo. The information provided by the PEA is crucial for supporting design exploration and optimization of software components.

The limitations of the PEA in terms of technology transfer stem from its focus on using performance information to aid in low level transformations. In some PEA papers (e.g., [9]), there is more emphasis on the application of the low level transformations than the explanation of how the performance information is deduced. The choice of the single-assignment functional Performo language as the object of study (as opposed to a general specification language such as Gist or Refine) and the concentration on performance information from the sole standpoint of low-level transformations (as opposed to being able to take into account higher level information provided by functional and non-functional requirements) severely limits the applicability of the PEA, even for other KBSA facets. There is also a question as to whether the PEA will scale up to system design issues such as those discussed in Section 3.2.1.

2.5. Framework

The KBSA Framework[11] addressed a number of issues critical for the integration of individual facets into a single framework. It developed a distributed object protocol using metaclass programming in CLOS. This is the only example to-date of KBSA research which attempted to solve problems dealing with multiple users. The Framework project also discussed various approaches to integrating different tools and showed that for KBSA, the only feasible approach was *deep integration*, i.e. reimplementing much of the tool so that it shared the same underlying knowledge representation mechanisms. This was accomplished for the PMA and for parts of KBRA. The experience of the Framework developers in these

reimplementation projects will be extremely valuable in future integration efforts.

Perhaps one of the most important accomplishments of the framework was to point the way toward higher level compatibility standards. As the framework research points out, it will be essential for future KBSA facets to be written using similar knowledge representation and interface tools. It is not enough for the facets to all use common lisp. As long as they use different supporting tools (AP5, Loglisp, Refine, Socle, etc.) the overhead in communication and in size of the lisp image will make it impossible to have several facets cooperating in the same lisp environment. The framework project took steps towards a higher level of standardization through its use of CLOS and CLUE (Common Lisp User Environment).

3. Industrial Hesitations

Given the range of valuable concepts described above, one might expect that there would be numerous research results of immediate application to industry. This, however, is not the case. There are significant additional development and technology assimilation costs associated with technology transfer. In order to tip the "cost vs. benefit" scales towards industrial adoption, industry must be reasonably assured that KBSA technology will have a positive impact on large scale systems development. In order to achieve this level of assurance, the KBSA program needs to produce research results that are applicable to industrial-sized problems. But the current research has not paid enough attention to a number of issues necessary for industrial applicability. Additionally, inadequate attention has been paid towards the communication of the status and potential of the program results.

KBSA does not have to prove that it has solved all the software problems of industry before it will be accepted. However, current and future KBSA efforts must extend themselves towards these critical issues. Unless this is done, industry will hesitate to make the investments necessary to transition the technology. This section will discuss a number of factors that have hindered technology transfer and need to be addressed. These factors are grouped into three categories: inadequate relevance to industrial practice, inadequate attention to key life-cycle areas, and inadequate conceptual validation and communication.

3.1. Inadequate Relevance to Industrial Practice

KBSA research results have not yet achieved enough relevance to industrial practice to warrant the investments necessary for technology transfer. Three major factors in this category that need to be addressed are scale, process, and usability.

3.1.1. Dealing with Issues of Scale

Issues of scale deal with the differences between programming-in-the-small and programming-in-the-large. Programming-in-the-small deals with the development of programs by a few people, while programming-in-the-large deals with the development of large systems by large teams of engineers. Programming-in-the-large introduces much more complexity into the development process due to (1) the need for collaboration between a large number of people and (2) the need to integrate large numbers of system components. Barstow[12] has described the research issues associated with each.

Most current KBSA research is focused on programming-in-the-small, while industry is very much concerned with programming-in-the-large. In order to motivate technology transfer to industry, KBSA research needs to show substantial relevance to programming-in-the-large issues. Languages and representations must be able to describe both functional and non-functional characteristics of large systems, not just the functional characteristics of individual programs. Reasoning processes must be tractable, rather than growing combinatorially with the size of the system. Process support must extend to collaboration between project team members, not just to the development activities of an individual.

3.1.2. Not Supporting Processes Under the Lamppost

There is an old story about a man who lost his wallet across the street, but searches for it on the opposite side under a lamppost because the light is better. Curtis et al.[13] effectively relates this story to research on software process models. Researchers need to understand actual software development practice in order to develop process models which capture processes that actually control software productivity and quality. They must be careful not to rely on simplifications that miss the real software problems. This ad-

monition holds true for KBSA, which is attempting to provide substantial process support for the software life-cycle. Substantiation is needed that the proposed alternative relates to industrial practice and will have a positive effect on the development of large systems.

Current KBSA research suffers a bit from the lamppost effect. The program proposes a radically different approach to software development, but has offered no detailed process models which deal with industrial problems. Facet process models need to relate more to actual practice: how project teams actually work, how systems are actually built, what information is actually available during development. KBSA research should take advantage of empirical studies of the design process (such as [14] and [15]) to insure a focus on high leverage activities and realistic assumptions. Section 4.1 discusses the need to develop detailed process models as part of the conceptual integration of KBSA research.

3.1.3. Building Tools for the Common Man

Because of the technical problems associated with each facet, the KBSA developers have so far focused almost exclusively on technical rather than usability issues. While it could be argued that usability issues are logically subsequent to many technical issues, attention to usability must precede technology transfer.

A catch-22 situation arises around KBSA: the more powerful the tools, the more difficult it becomes to apply them. We perceive two major usability issues for the KBSA: the cognitive effort required to utilize KBSA's various formal representations and the complexities involved in mastering the extremely rich software development environment which KBSA will provide.

Rich and Waters[16] point out that an inordinate amount of effort is often required to represent and process knowledge using formal representation languages. One of the reasons for this is that current formal specification languages do not map well to the user's conceptual representations of software and the development process. As Sasso[17] and Soloway[18] demonstrate, such a mapping is essential to provide

usable tools for software development.

In order to facilitate industrial acceptance of KBSA, future software will need to pay increased attention to these usability issues and to providing friendly, intelligent interfaces. These interfaces will have to be geared towards common practitioners, not researchers. KBSA should take advantage of the vast amounts of existing research in computer-human interaction and intelligent interfaces. We present an extended discussion of how such research could be incorporated into KBSA in [19].

3.2. Addressing Key Life Cycle Areas

Another hindrance to technology transfer results from the lack of attention paid thus far in the KBSA program to a number of key software life-cycle areas. Issues such as system design, maintenance, and reengineering need to be addressed by KBSA if it is to offer a comprehensive approach to systems development.

3.2.1. System design

Systems are collections of programs, data, user interfaces, or analog devices with requirements covering functionality, persistence of data, performance, capacity, interfaces with other systems, and other environmental constraints. At this point, KBSA has not adequately addressed system level design. Little attention has been paid, for example, to the design of system architectures. The system architecture is of critical importance because (1) it governs the decomposition of the system into components, (2) it determines the component-to-component interface requirements, and (3) it has a tremendous impact on the performance and usability of the system.

Architectural design has to be dealt with at the earliest stages of the development process and will impact all later development activities. The system architecture is heavily influenced by non-functional requirements such as capacity, throughput, security, operational environment, etc. KBSA needs to support the

capture and modeling of such information during the requirements acquisition process. The system architecture governs the decomposition of a system; KBSA's system specification and code generation activities cannot be independent of architecture. System level performance is influenced not only by algorithm complexity but also by i/o between components, external data sources, and users. Performance estimation assistance must extend to the system level in order to support architectural design.

3.2.2. Maintenance

Maintenance is the process of evolving the system over time as errors are discovered or requirements change. It has been well documented that maintenance activities constitute a significant percentage of a system's application life cycle cost. Maintenance was one of the main areas of assistance targeted by the '83 KBSA report[1]. That report envisioned that "maintenance would be performed under the KBSA paradigm by altering the specification and replaying the previous development process (the series of transformations), slightly modified, rather than by attempting to patch the implementation." This maintenance strategy would be supported by activities at each stage of the life-cycle.

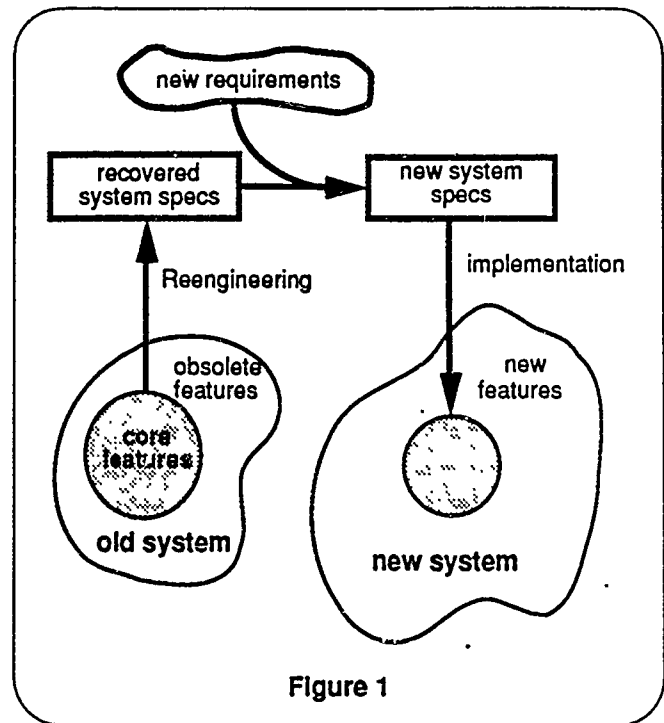
Little attention, however, has been paid thus far to maintenance. Most major issues remain to be addressed. These research issues include: appropriate models of design and implementation history; formalisms for capturing and reasoning about rationales, strategies for impact analysis and replay, and change management issues[20].

3.2.3. Reengineering

Reengineering is the process of recovering information about an existing system through examination of various system development artifacts, e.g. source code, documentation, etc. We feel that reengineering is of prime importance to KBSA for two reasons.

First, as shown by Alagappan and Kozaczynski[21], reengineering is sometimes a necessary component of the system development process. This is true when some core functionality of a new system replicates

a portion of an existing system and, due to the age of the existing system, the only accurate description of that needed functionality is embodied in the system source code. The specification of this needed functionality must then be reengineered through a machine-assisted design recovery process. Figure 1 is reproduced from [21] to illustrate this concept.



Second, reengineering may provide a method for introducing the revolutionary KBSA approach in an evolutionary manner.

Existing systems could be reengineered into KBSA representations and evolved using the new paradigm. There are a number of significant research issues associated with this approach. But some method of "easing" KBSA into place will eventually be needed if it is to find a place in industry.

3.3. Inadequate Conceptual Validation and Communication

A third category of factors which have hindered technology transfer is the lack of attention paid to the conceptual validation and communication of facet capabilities. In the course of our evaluations, we had significant problems identifying the range of each facet's capabilities and determining the generality of those capabilities. In order to communicate the validity of KBSA concepts, it is important for each facet to provide a complete description of its functionality and to facilitate hands-on experimentation with its software.

3.3.1. Complete Functional Descriptions

Many of the facet reports tend to list the various techniques or methods utilized in the prototype without

an explicit functional description of the program. Each facet report should, at least briefly, answer questions such as: Which life-cycle activities can and cannot the program support? How general are the program's capabilities?

It is often difficult to determine the generality of a piece of work. However, as McCarthy[22] suggests, this is something that the researcher should attempt to accomplish through extensive experimentation. Experimentation will help the researcher understand what compromises had to be made in order to make the program work through her examples. She will also gain an understanding of the difficulties involved in developing workable examples. These experiments should then be reported to show not only how the program works but where and how it fails to work. This will enable others to fairly evaluate the research.

3.3.2. Hands-On Experimentation

In order to facilitate hands-on experimentation, it is important to provide a detailed user's guide for the software and numerous example scenarios which will demonstrate the ideas and functionality of the software. The current facets provide relatively good user's guides. However, they are very weak in providing scenarios. We found it to be very difficult to perform hands-on experimentation with the existing facet software because of the poor quality of existing scenarios. Without a scenario one is left to try and construct realistic examples from scratch. This invariably leads to discovering undocumented bugs and going down dead ends. Where scenarios do exist in the current facets, they are either too brief (showing only a few example commands and leaving the user unconvinced that the facet can work on any realistic problem) or too sketchy (leaving details about how to select items or deal with potential bugs unstated).

4. Facilitating Technology Transfer

In addition to the issues of industrialization and usability described in the previous section, a number of other steps can be taken to facilitate future technology transfer. This section discusses the need for (1) deeper integration of the research results and (2) greater communication and cooperation within the KBSA developer community.

4.1. Integration

While some might argue for hardware and software standardization and integration, we feel that this would be premature. There are many daunting research issues facing the program--not just regarding integration but also regarding basic representation and problem solving. Rigid standardization at this time would be stifling. We do believe, however, that significant steps can be taken to begin the integration process and insure that the individual KBSA efforts will move onto convergent paths.

4.1.1. Conceptual Integration

The KBSA program must begin a conceptual integration process. At this point in time, the different facets seem to be developing in a rather unconcerted manner. Each research team has developed its own view of the overall KBSA model. There has been a tendency to focus on facet-specific problems, while ignoring important problems that overlap facets. Now is an appropriate time for the KBSA developers to work together to build an integrated model of KBSA. We realize that a single "grand unifying theory" probably cannot be built. But collaboration in this conceptual integration will serve to test ideas and to coordinate future research. Two major elements make up a comprehensive KBSA model: the information model and the process model. These are described below.

Information Model

An information model is composed of two levels. a modeling level and a knowledge level. The modeling (or meta-knowledge) level encompasses the descriptive facilities which make up a representation, e.g. objects, relations, constraints, etc. Constructive dialogue needs to begin regarding the descriptive capabilities provided in each facet (i.e. through REFINE, AP5, LogLisp, etc.). We believe that this is feasible. The languages used in each facet are already roughly equivalent in expressiveness. An in-depth analysis of the differences should be performed with the goal of choosing or designing a unified modeling language.

The knowledge level consists of the classes of information (expressed using the modeling facilities) which

describe the domains of interest, e.g. system descriptions, application domain models, and software engineering knowledge. Discussions should begin to conceptually integrate knowledge level information that is implicitly or explicitly modeled in each facet. We expect that this knowledge will sometimes complement one another; at other times they will conflict.

Process Model

A software development process model defines a set of operations considered legitimate. More importantly, it organizes the individual operations into groups which enable, encourage, or enforce a particular sequence of execution through the entire process. The KBSA program promises freedom from the traditional waterfall process model but has yet to present a well-articulated alternative model. The individual facets approach this issue in different ways. Facets range from having no explicit process model (the KBRA), to an informal localized process model (the Specification Assistant), to providing the building blocks for a process model without actually constructing alternative models (the PMA).

While we do not believe that there is one comprehensive, all-encompassing KBSA process model waiting to be discovered, the development of detailed software process model alternatives would be of immense benefit. The question needs to be asked: Can KBSA succeed where other paradigms have failed? Can it support process models which address recognized flaws in other models such as those identified by Curtis[13]?:

- Does the process model provide mechanisms for managing the inevitable changes in requirements and for involving end-users throughout the development process?
- Does it provide insight into and improve processes that control the largest share of the variability in software development (such as the coordination of interacting agents), thereby actually boosting productivity and quality?
- Does it positively influence software development productivity and quality (and software maintainability) as the size of the effort grows to programming-in-the-large and even programming-in-the-gargantuan?

If successfully developed, such process models would benefit the KBSA program at two levels. At the research level, they can (1) guide and structure the development of KBSA tools (in terms of their functions and interfaces to the user and to each other) and (2) serve as the basis for the PMA's representation of the software project. At the technology transfer level, process models would serve to (1) communicate the added value provided by KBSA to its potential user community; (2) help the audience compare the KBSA software development process with traditional ones; and (3) suggest promising ways to transfer KBSA technology.

4.1.2. Platform Standardization

While discussions proceed on the unification of the KBSA modeling languages (at the level of AP5, RE-FINE, or LogLisp), the program should adopt platform standards at lower levels. This should greatly assist the development of integration guidelines which will ensure higher levels of software convergence in the future.

Hardware Standardization

Constant debate takes place within both research and industry over the hardware platform of choice. This debate is never-ending because the crown of hardware performance supremacy is constantly changing. We do not believe that the KBSA program should become embroiled in this debate. Several generations of hardware platforms have risen and fallen during the course of Phase I. We expect several more generations to rise and fall during Phase II. A commitment to any one hardware platform before KBSA is ready for significant technology transfer carries with it only one guarantee. the hardware platform chosen will soon be eclipsed by some other platform. Rather than hardware platform standardization, we propose instead a standardization on the principle of hardware independence. The route to hardware independence is through software standardization, which is explained in the next section.

Software Standardization

Software evolves much more slowly than hardware. Also, software standards tend to outlive a number of hardware generations. In order to develop greater hardware independence, we propose that the program actively adopt the following software standards, which are supported across multiple platforms:

- Common Lisp & CLOS (the Common Lisp Object System) should be the base language and object system.
- CLX & CLUE (the Common Lisp User Environment) should be the low level interface standard.

We believe that these choices represent significant industry trends. A move to these standards should also be relatively straight-forward for current developers. In addition to these standards, we propose the following guidelines:

- All high level languages utilized by the facets should have object level compatibility with CLOS. New languages, e.g. ARIES, should be built using the Metaclass programming facility of CLOS. Existing languages, such as REFINER, should be reimplemented on top of CLOS.
- All KBSA user interfaces should be built in an object-oriented manner through CLOS libraries built on top of CLUE. Although we have no proposals at this point, we believe that the program should raise this standard even more, to the level of user interface toolkits and "look and feel" standards.
- Facet researchers are free to adopt the hardware platform of their choice, provided that they adhere to the hardware independence principle. Maximize productivity but minimize the incorporation of hardware-specific environments into their systems. Where such couplings cannot be avoided, they should be made modular and easy to detach in case of a port.

The "CLOSification" of the KBSA facets accomplishes one small step towards eventual KBSA software convergence. It will allow different facets to have a compatible view (albeit at a low level) of their data. The construction of window system libraries on top of CLUE using CLOS is also consistent with industry and research trends. This opens up greater opportunity for software leverage. Experiments with both these approaches have been undertaken by the Framework research team[23].

4.2. Internal Technology Transfer

Much attention has been paid to the process of transferring KBSA technology from the developers to the software community at large. This topic was the focus of a panel discussion at the 2nd KBSA Conference in 1987. It led to the creation of the KBSA Technology Transfer Consortium in 1988 and it is the main theme of the 1989 KBSA Conference. The current model of technology transfer focuses on the flow of technology from individual developers to industrial alpha test sites. While we fully support these efforts and believe that they should continue (and expand), we believe that not enough attention has been paid towards technology transfer between the KBSA developers themselves. We believe that the developer community needs to be the true alpha test sites for KBSA technology, and that industrial sites should perform beta testing on software that has already been tested within the developer community.

As DiNitto[24] points out, there are certain natural impediments associated with the transfer of advanced software technology. These include:

- Acquisition of custom hardware and software.
- Culture clash between researchers and industry.
- Training people in new techniques and technologies.

These impediments to transferring KBSA technology from developers to industry are not nearly as strong when transferring technologies from one KBSA developer to another. It involves less effort for one KBSA developer to test the software of another than it does for the same test to be made by an industrial practitioner.

There would be a number of benefits to having developers serve as alpha test sites. This internal technology transfer would lead to a greater understanding by each developer of the work of the others. This would encourage greater conceptual and software integration, and would help to bridge the artificial separations caused by the facet approach. Also, the extra testing and validation that would result from developer alpha testing would mean that the products which were shipped to industrial beta sites would be more robust, reliable, and would have deeper functionality.

The Aries project is a good example of internal technology transfer. It will integrate and advance the previous research of the Specification Assistant and KBRA teams to produce a combined Requirements/Specification tool. Facet combination is an extreme example of internal technology transfer. Less extreme forms would also be valuable and should be encouraged. For example:

- **Specification and Development.** In the KBSA vision, the development assistant works on the output produced by the specification assistant. It is essential for future integration that the developers of these facets communicate so that they have conceptually compatible views of the specification/development process.
- **Project Management and the Framework.** The project management facet and the Framework must both be intimately intertwined with the other facets. It is essential for developers working on these facets to have a deep understanding of the functionality and conceptual models of the other projects (and vice versa).
- **Activity Coordination.** Work is beginning on developing activity coordination facilities for KBSA. It is essential for this project to incorporate the results of other projects, and for other projects to be aware of the expectations and formalisms required in order to address activity coordination.

5. Concluding Remarks

The software problems facing industry and government today are even greater than when the KBSA program first began. It is clear that the payoff from a workable KBSA system would be enormous. The first round of KBSA research has led to some significant results and influences on other areas of research. In this paper, we have evaluated that research from a technology transfer perspective and offered some suggestions for facilitating future technology transfer.

The suggestions that we have offered are based upon our belief that technology transfer is a multi-phase process. Innovative technology such as KBSA cannot make a one-step jump from researchers to practitioners. We believe that it is necessary both to nurture new technologies but at the same time ensure that they are headed towards industrial applicability. Timely standardization is necessary to set the stage

for eventual deep integration. Greater cooperation between the various developers will lead to the development of integrated systems which cover a broader spectrum of the software life-cycle and which can achieve synergy by leveraging off of each others capabilities. Industrialization and humanization should result in tools which are responsive to the requirements of industrial users.

Acknowledgements

Bruce Johnson, Gilles Lafue, Bill Sasso, and Gerald Williams have made substantial contributions to the ideas expressed in this paper. Vairam Alagappan, Lewis Johnson, Richard Jullig, Gordon Kotik and Aaron Larson provided valuable feedback on earlier versions of this paper.

References

- [1] Green, C., Luckham, D., Balzer, R., Cheatham, T., & Rich, C., "Report on a Knowledge-Based Software Assistant," RADC TR 83-195, Contract No. F30602-81-C-0206, Kestrel Institute, Palo Alto, CA, June, 1983.
- [2] KBSA Technology Transfer Consortium, Operational Concept Agreement. RADC, 1988.
- [3] Harris, D. & Runkel, J., "An Introduction to the Knowledge Based Requirements Assistant Capabilities," Rome Air Development Center, Contract No. F30602-85-C-0267, September, 1988.
- [4] Johnson et al., "The Knowledge-Based Specification Assistant, Final Report", Rome Air Development Center, Contract No. F30602-85-C-0221, September, 1988.
- [5] Goldman, N., Wile, D., Feather, M.S. & Johnson, W.L. "Gist Language Description", 1988.
- [6] Gilham, L., Jullig, R., Ladkin P., Polak, W., "Knowledge-Based Software Project Management", Kestrel Institute, Palo Alto, CA, November, 1986, KES.U.87.3.
- [7] J. Allen, Towards a General Theory of Action and Time, *Artificial Intelligence* 23 (2), July 1984, pp. 123-154.
- [8] Boehm, B. W. "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, pp. 61-72 (May 1988).
- [9] Blaine, L., Goldberg, A., Pressburger, T., Qian, X., Roberts, T., & Westfold, S., "Progress on the KBSA Performance Estimation Assistant", Proceedings of the 3rd KBSA Conference, 1988.
- [10] Goldberg, A., "Technical Issues for Performance Estimation", Proceedings of the 2nd Knowledge-Based Software Assistant Conference, July 9, 1987.
- [11] Huseth, S. & King, T., "A Common Framework for Knowledge-Based Programming," Proceedings of the 2nd Knowledge-Based Software Assistant Conference, 1987.
- [12] Barstow, D. "Tutorial on Knowledge-Based Programming Environments," Presented at the 9th International Conference on Software Engineering, Monterey, CA. March 30, 1987.
- [13] Curtis, B., Krasner, H., Shen, V. & Iscoe, N., "On Building Process Models Under the Lamppost", 9th International Conference on Software Engineering, Monterey California, IEEE Computer Society Press, 1987.
- [14] Boehm, B.W., *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.
- [15] Curtis, B., Krasner, H., & Iscoe, N., "A Field Study of the Software Design Process for Large Systems" Communication of the ACM Vol 31, No. 11, Nov. 1988.
- [16] Rich, C., and Waters, R., "The Programmer's Apprentice: A Research Overview," *IEEE Computer*, 21(11):10-25 (November 1988).
- [17] Sasso, W.C., "The Roles of Constructive Diagrams in Systems Development." *Journal of Information Systems Management* 6(1) (Winter, 1989).
- [18] Soloway, E., Pinto, J., Letovsky, S., Littman, D., & Lampert, R., Designing Documentation to Compensate for Delocalized Plans, *Communications of the ACM* Vol. 31, Number 1 (Nov. 1988), 1259-1267.
- [19] DeBellis, M. & Mui, C., "Developing an Intelligent Interface for KBSA," to be published in the proceedings of the Workshop for Intelligent Interfaces, IJCAI, Detroit, Mich., August, 1989.
- [20] Muncaster-Jewell, P., "Change Management Needs for Persistent Design Databases," Proceedings of the 3rd KBSA Workshop, August, 1988.
- [21] Alagappan, V. & Kozaczynski, W., "Specification Recovery in the Context of Re-engineering Very Large Information Systems" to be published in the proceedings of the IJCAI '89 Workshop on Automating Software Design, August 1989.
- [22] McCarthy, J., Presidents Message: We Need Better Standards for AI Research, *The AI Magazine*, Fall 1984, pp. 7-8.
- [23] Huseth, S., Larson, A., Glasser, J., & Thelen, K., "KBSA Framework 36-Month Technical Report," Honeywell Systems and Research Center, April, 1989.
- [24] DiNitto "Problems and Prospects for Technology Transfer in Expert Systems", Proceedings of the 2nd Knowledge-Based Software Assistant Conference, 1987.

KBSA Technology Transfer: An Industrial Perspective

Chunka Mui
Mike DeBellis

September 13, 1989

ANDERSEN
CONSULTING
ARTHUR ANDERSEN & CO.

Premises

1. Want to Transfer Current KBSA Technology
2. KBSA is Relevant for a Commercial Setting

Discussion Topics

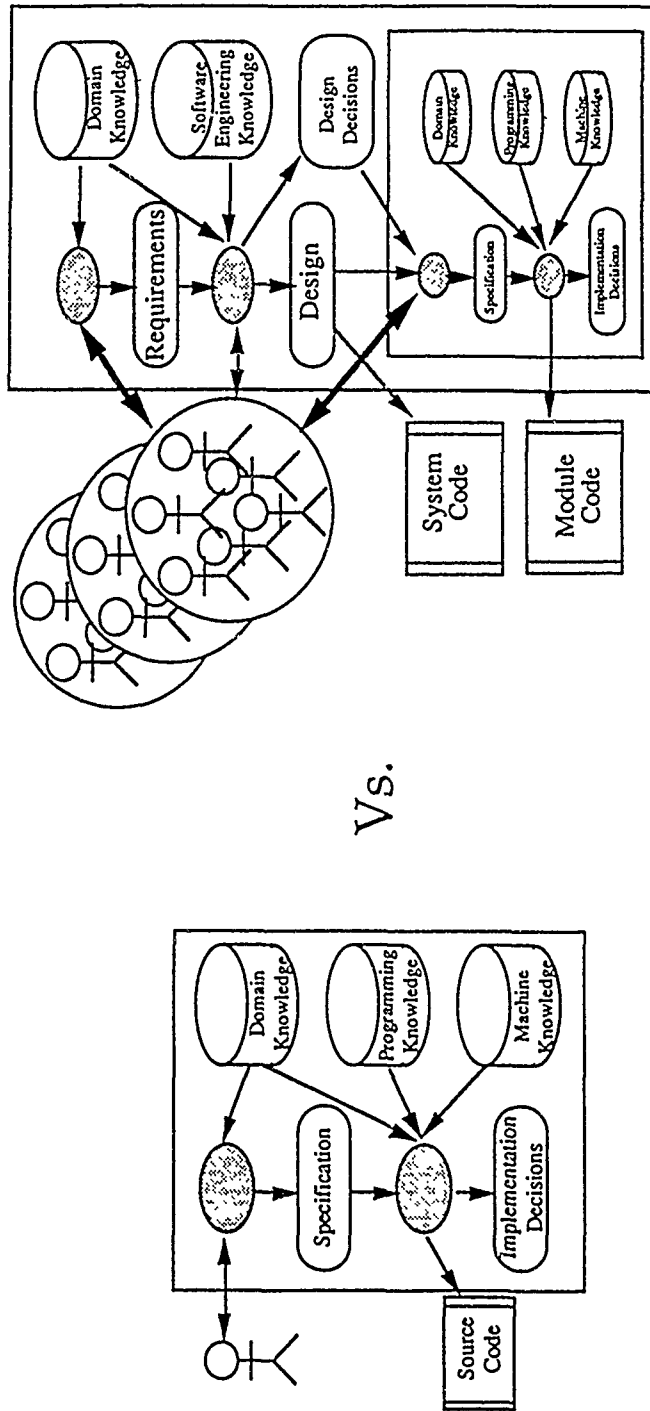
KBSA Program has yielded a range of valuable concepts but little of immediate applicability. Presentation will focus not on KBSA concepts, but instead on:

1. Industrial Hesitations to the Transferring of Current KBSA Technology
2. Methods for Facilitating Future Technology Transfer

Industrial Hesitations

1. Relevance to Industrial Practice
 - Issues of Scale
 - Processes under the Lamp Post
 - Building Tools for the Practitioner
2. Key Life Cycle Areas
 - Systems Design
 - Maintenance
 - Reengineering
3. Conceptual Validation & Communication
 - Functional Descriptions
 - Hands-On Experimentation

Issues of Scale



Vs.

Actual Practice?

Or Processes Under
the Lamp Post?

ANDERSEN
CONSULTING
ARTHUR ANDERSEN & CO.

Usability: Building Tools for the Practitioner

KBSA Research has focused almost exclusively on technical rather than usability issues.

In order to make KBSA ready for technology transfer, usability issues must be addressed:

1. Formal specification languages are difficult to use and understand.
2. KBSA will provide a rich, complex environment. A good user interface will be essential to making such an environment usable.

System Design

System design decisions play a critical part in the development of industrial software.

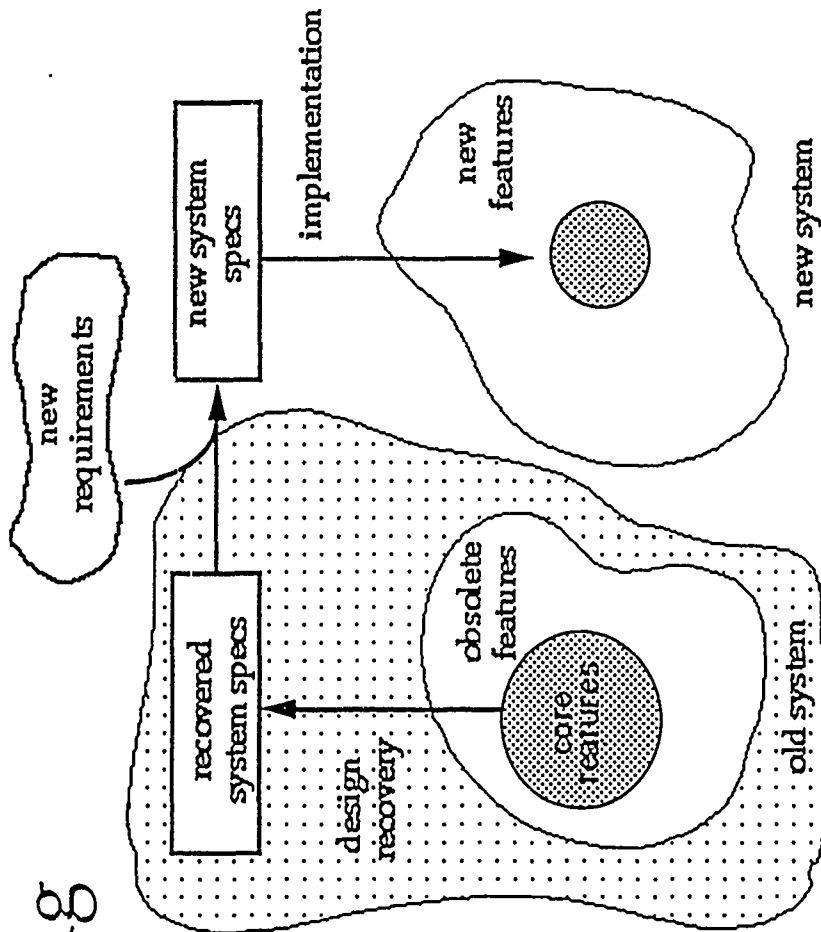
1. System architecture governs the decomposition of the system into components and determines the interface requirements for those components.
2. Performance information and non-functional requirements are very important for system design decisions.

Maintenance

"Maintenance at the Spec level" is one of the most enticing prospects offered by KBSA. However, many crucial maintenance issues have not yet been addressed:

- Models of design and implementation history
- Formalisms for capturing and reasoning about rationales
- Strategies for impact analysis and replay
- Change management

Reengineering



Complete Functional Descriptions

1. What Life Cycle Activities can or cannot the system support?
2. How general are the capabilities?
3. Where does it fail?

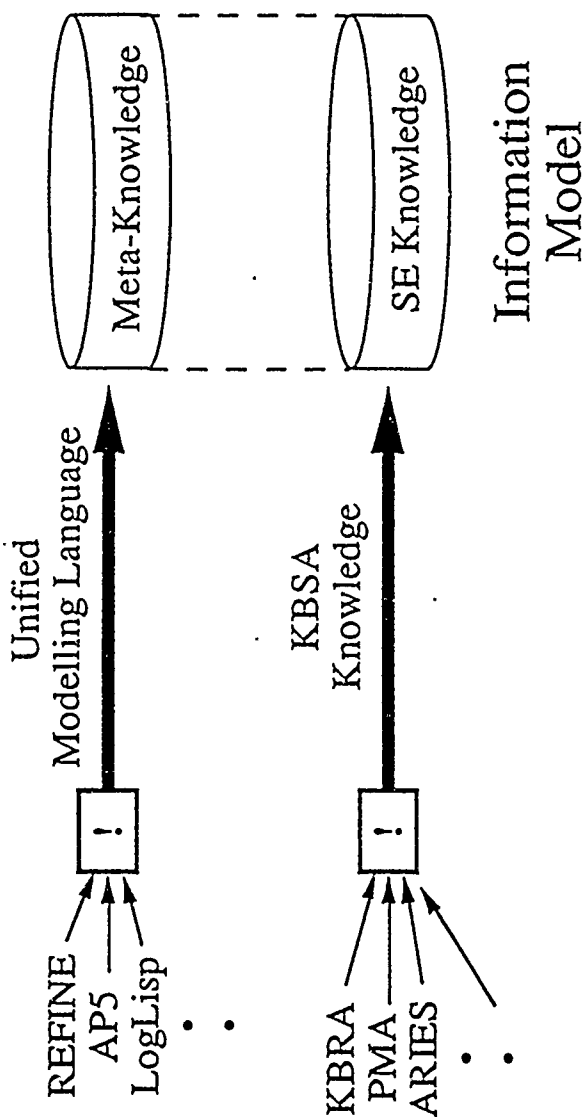
Hands-On Experimentation

1. User's Guide for Users not Developers
2. Well documented, repeatable test scenarios

Facilitating Future Technology Transfer

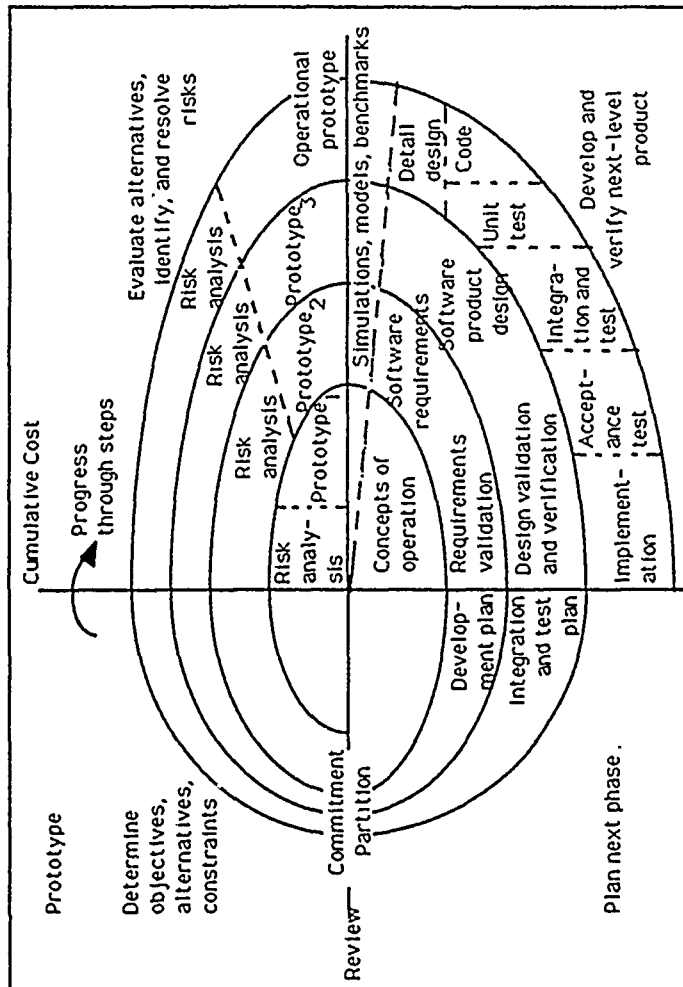
1. Conceptual Integration
 - Information Model
 - Process Model
2. Platform Standardization
 - Hardware
 - Software
3. Internal Technology Transfer

Towards a Unified Information Model

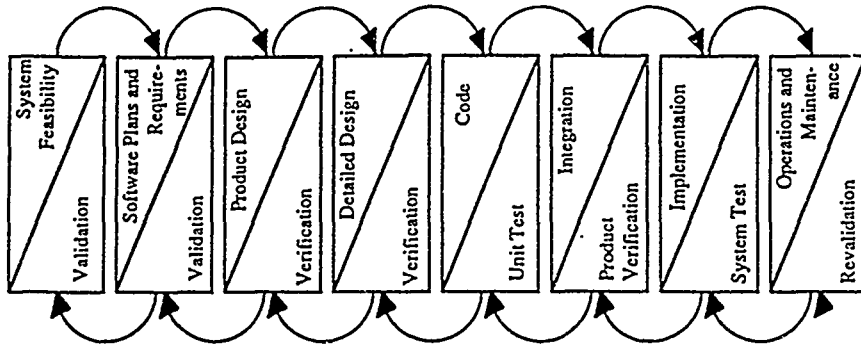


Conceptual Integration

Other Process Models



Boehm's Spiral Model



Waterfall

ANDERSEN
CONSULTING
ARTHUR ANDERSEN & CO.

Testing through Development of KBSA Supported Process Models

- Managing Change during Development
- Providing Non-Trivial Insights
- Improving Process
- Enhancing Productivity & Quality

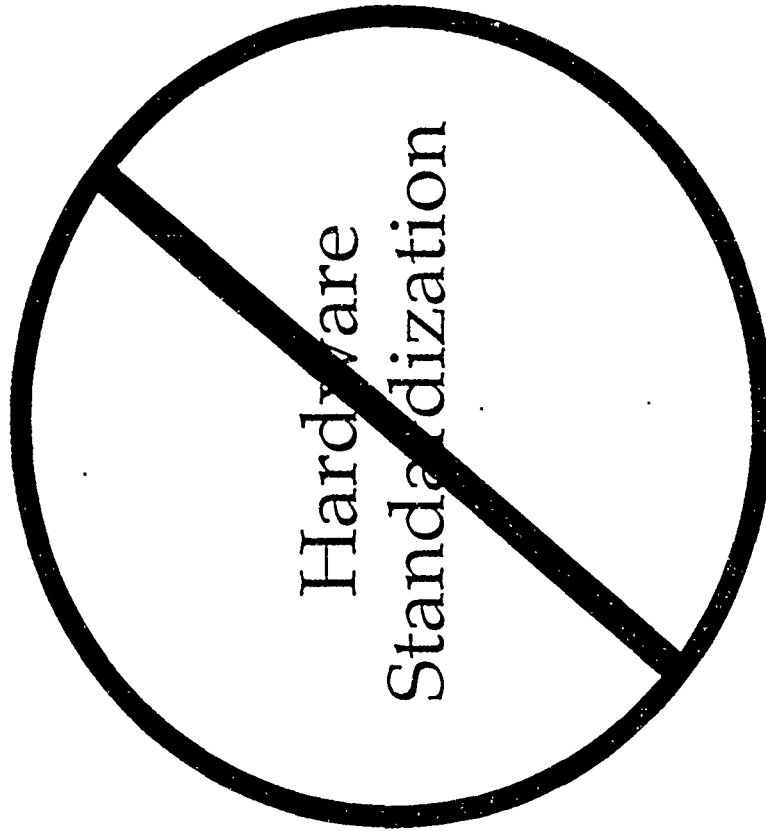
Process Models Benefit both Research & Technology Transfer

Research

- Guide & Structure Development of Tools
- Serve as basis for Project Management

Technology Transfer

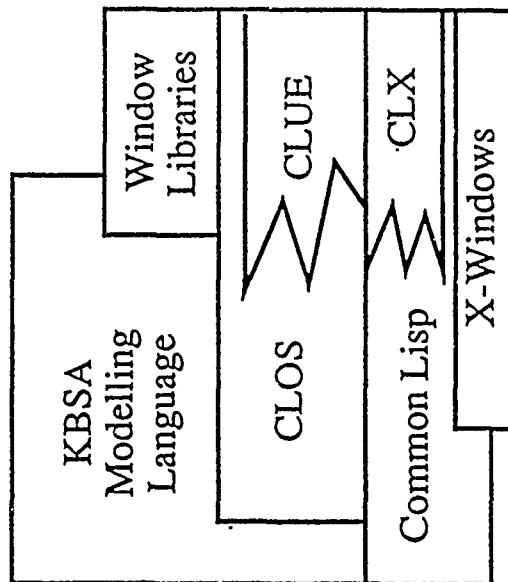
- Communicate Added Value
- Suggests Transfer Paths



Maintain Hardware Independence through Software Standardization

ANDERSEN
CONSULTING
ARTHUR ANDERSEN & CO.

Software Standardization



- Common Lisp and CLOS provide a base language and object system.
- CLX and CLUE provide a low-level interface standard.

Internal Technology Transfer

- Developer Alpha Test Sites
- Industrial Beta Test Sites

Concluding Remarks

- Benefit to industry and government from a workable KBSA system would be enormous.
- Technology transfer is a multi-phase process.
- In order to make future technology transfer feasible, it is necessary to begin addressing issues of standardization, cooperation, industrialization, and humanization in ongoing KBSA research.

KBSA FOR SOFTWARE MAINTENANCE AND RE-ENGINEERING

Gordon B. Kotik, President
Lawrence Z. Markosian, Vice President for Applications Development
Reasoning Systems, Inc.
3260 Hillview Avenue
Palo Alto, CA 94304

Tel.: (415) 494-6201
Internet: kotik@reasoning.com, zaven@reasoning.com

Author Biographies. Gordon Kotik and Lawrence Markosian are founders of Reasoning Systems, Inc. Before becoming President Mr. Kotik was Vice President for Product Development at Reasoning. Before the formation of Reasoning Mr. Kotik was one of the principal architects of the CHI knowledge-based programming environment at the Kestrel Institute. Mr. Kotik has supervised projects involving development and maintenance of submarine tracking software.

Prior to the formation of Reasoning, Mr. Markosian applied AI technology to DoD problems including data fusion, tactical air battle management and control system reconfiguration at Systems Control Technology. Previously Mr. Markosian was a Research Associate at Stanford University. He is a member of the Association for Symbolic Logic, the American Association for Artificial Intelligence and the Association for Computing Machinery.

Abstract. We describe a new approach to software maintenance and re-engineering that has demonstrated success in automating these hitherto intractable problems. The new approach incorporates several technologies: *object-oriented databases* and parsers for capturing and representing software; *pattern languages* for writing program templates and querying a database of software to find code that matches the templates; and *transformation rules* for automatically re-writing programs to meet new requirements. We present a program transformation system, REFINE™, that incorporates these and other technologies in an accessible environment for software maintenance and re-engineering. Finally, we present examples of how our approach has been successfully applied to maintaining software in a variety of languages.

1 Overview

Software systems are engineering artifacts of unprecedented complexity because of their sheer size and the number and complexity of non-local and implicit interactions among pieces. Nevertheless, most maintenance tools have been based primarily on linear text

* Copyright (c) 1989 by Reasoning Systems, Inc.

** REFINE is a trademark of Reasoning Systems, Inc.

string representations of software that do not reflect the underlying, complex software structure. We can make maintenance easier by building a collection of software tools that represent and process software as network structures stored in a database. This representation captures the abstract structure of the software, which is a network of interacting pieces, and abstracts away from the details of the character string form of a program.

Other objects of discourse in software development and maintenance include documents, test cases, bug reports and project plans, most of which also exhibit a network structure. Software maintainers' tasks can be characterized as *analyzing* and *transforming* complex networks of all these software engineering objects. The goal of these analyses and transformations is to change some properties of the program while preserving other properties. Programming environments should provide the network representation of software and related objects for use in writing tools that analyze and transform software.

Typical tasks under each category might be the following:

- Analysis: If I make a certain change to the program, what are all the other parts of the program (and associated documents, test cases, etc.) I will have to change?
- Transformation: Change the program to use the X window system instead of OldWindows.

Software development objects are usually represented as groups of text files in a variety of languages, both formal and informal. The links in the network (e.g., document D describes program P) are largely informal and implicit ("Most of the short files with extension C in subdirectory LARRY and written prior to 11/88 contain C source code for the first prototype of the FOO system"). The programmer typically analyzes and modifies these objects using file-based tools—text editors, string searches, etc.

Hence, programmers must map their conceptual universe, a complex network of structured objects, onto a hierarchical file system and interact with it at that level. The difference between the conceptual model and the file representation occupies a sizeable fraction of a programmer's time. Answering questions like "who last modified this

function” or “where is the user documentation for this program” can be extremely time-consuming. Overall, the problems are similar to what would be encountered while trying to use a bitmap editor like MacPaint as a CAD tool—an array of bits holds all the information, but the information cannot readily be extracted by automation tools.

In the maintenance phase, the problems are compounded. The engineers who created the original mapping of the project onto a file system might have left the company. With them went the rationale and the details of the mapping, as well as key information about the individual programs, documents, etc. It becomes successively more difficult to maintain the internal consistency of programs, and also the consistency of programs with their requirements, test cases, and documentation.

One approach to improving the situation is to build programming environments that support the user in mapping the network of software objects into hierarchical file systems. This is a solid, evolutionary approach that is exemplified by tools such as file-based programs for system modelling, version control, and configuration management [1, 2, 3, 4]. An alternative approach is to use an object-oriented database representation for programs and other software engineering objects and to build tools to query, analyze and transform this database.

The approach described here builds on the experience of many previous systems and is closely related to a number of current efforts. Many Lisp programming environments have been build around a representation for Lisp programs as list structures in virtual memory. The prime example is the Interlisp programming environment developed at Xerox PARC [5]. Interlisp provided many analysis tools for software, including Masterscope, an interactive query system for program analysis.

Language-based environments introduced the idea of building general programming environments that could be customized to a particular programming language [6]. Much of the effort was devoted to automating the generation of tools such as syntax analyzers, structure editors and static semantic analyzers.

The Smalltalk-80 system represented parse trees within its compiler using instances of Smalltalk objects [7]. More recently, this representation has been used in Smalltalk

compilers that perform compile-time analysis to eliminate much of the runtime interpretation usually associated with Smalltalk execution [8].

In Section 2 we look at alternative methods for representing software. Section 3 presents REFINE, an enabling technology for software maintenance and re-engineering that uses an object-oriented database for representing software and transformation rules for modifying software in the database. In Section 4 we examine actual software re-engineering and maintenance projects undertaken using REFINE. We chose to look at actual customer examples because we believe that examples of such use are much more convincing than hypothetical examples that more succinctly highlight key technical details. Section 5 summarizes our results.

2 Alternatives For Software Representation

Earlier we characterized the fundamental software maintenance activities as analyzing and transforming networks of software engineering objects. We will now focus on representing programs. We will evaluate various techniques for software representation based on how well they support automating the analysis and modification of programs.

Source Files

One basic reason why analysis and transformation of software using file-oriented tools is expensive and unreliable is that source text is a poor data structure for representing programs. Compilers parse and analyze the source text to create a better representation, namely abstract syntax trees, symbol tables, and other data structures that capture the abstract relationships among program units. The compiler then performs further analysis (e.g., type checking) and transformation (e.g., code generation) on the internal representation.

Programmers do not generally have convenient access to the internal representation created by compilers, so they are forced to deal directly with the "raw" form of the program: source files. This limits their productivity in program manipulation, particularly during maintenance where they must systematically analyze and modify large,

unfamiliar programs.

Compiler Data Structures

The style of program representation used by compilers is an improvement over source files, but it still has drawbacks because the representation is not designed for use by tools other than the compiler:

- The representation does not persist between runs of the compiler. There is no version control of data structures, not even within a single run.
- Low-level data structures are used to encode abstract types (e.g., arrays are used to encode sets, mappings and relations). The hand-coded implementation details are difficult for the user to decode.
- No interactive tools are provided for browsing or editing the data structures.
- No specialized query language is provided to support basic operations such as enumerating all the statements in a program.

Relational Databases

Some of the problems of compiler data structures may be solved by relational databases (RLDB), namely, persistence, browsers and query languages. However, relational databases are unsuitable for program manipulation (and engineering applications in general) because of (a) the limits of the relational data model and (b) their inefficiency in executing the graph traversal algorithms that occur so frequently in program manipulation [9].

Object-Oriented Databases

Object-oriented databases are an emerging technology that is expected to play an important role in engineering database applications. They preserve some of the advantages of relational databases (persistence, browsers, and, to some extent, query

languages). They offer much more powerful data models that directly support network manipulation with features such as multi-valued slots, inheritance and constraint maintenance. Furthermore, their efficiency characteristics are better-suited than those of RLDBs to fast manipulation of complex graph structures.

A practical system for software maintenance needs the following extensions to object-oriented databases:

- The data definition and query languages should support the mathematical abstractions used in high-level program representation (set notation, first order logic, tree comparison)
- Tools must be provided for parsing source files into the database and printing the database back to source files
- A syntactic pattern-matching capability for describing programs in terms of templates should be provided.
- The query language should support a rule-based programming paradigm for specifying transformations of software.

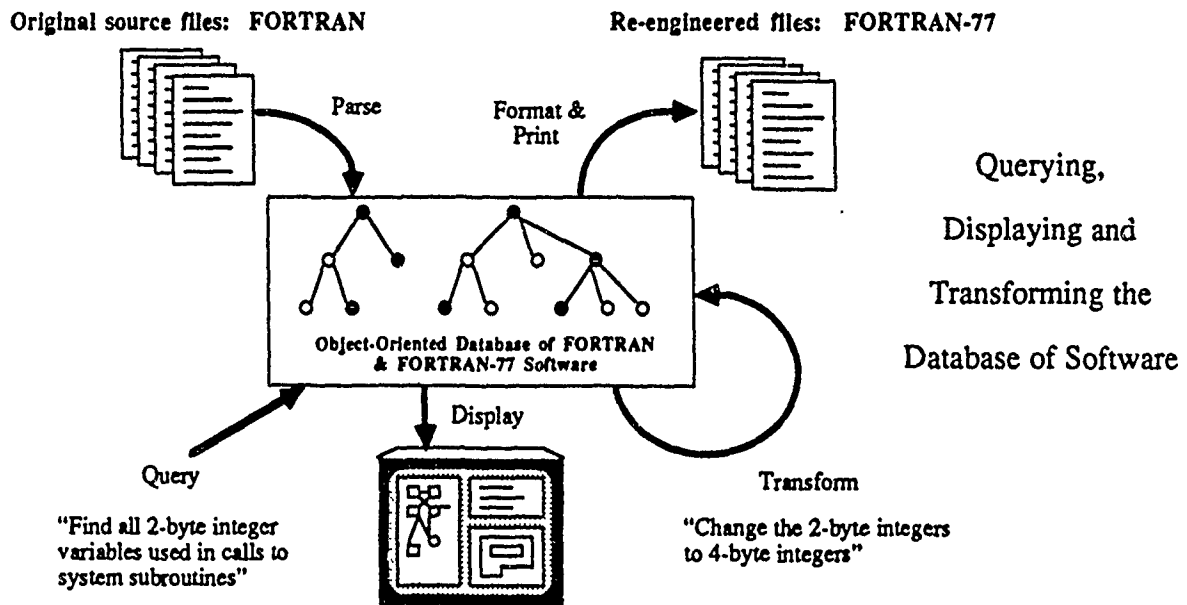
In the next section, we will describe how the REFINE knowledge-based software development system retains the benefits of object-oriented databases while providing the extensions listed above, and hence can serve as a technological foundation for a new generation of software maintenance tools.

3 REFINE: An enabling technology for software maintenance and re-engineering

REFINE [10] is an interactive software development system that integrates three key tools to provide a basis for software manipulation:

- a wide-spectrum, very-high-level specification language
- an object-oriented database that provides the necessary abstractions described in the preceding section
- a language processing system that accepts definitions of programming languages and produces syntax tools (parsers and printers)

The figure below depicts how these tools can be used in software maintenance.



Programs are converted between source file and the object-oriented database using the parsers and printers created by the language processing system. Thus the database is fully integrated with conventional file-based systems and tools. The REFINe object system and other high-level data types in the specification language (sets, sequences, maps, etc.) support a data model for software objects that is very close to the standard conceptual view of annotated abstract syntax trees. Tools that analyze and transform software in the database are written in the REFINe specification language, which provides mechanisms for template-based program description and rule-based program transformation.

3.1 The REFINE specification language

The REFINE specification language (also called REFINE) is a very-high-level, wide-spectrum language that supports a variety of specification techniques including set theory, first order logic, rules, object-oriented and procedural programming. The specification language is used as the query/update language for the database. The compiler for the specification language is implemented as a rule-based program transformation system; the current version generates Common Lisp and a code generator for C is under development. The compiler and most of the rest of REFINE are written in REFINE.

3.2 Object-oriented database

The REFINE database provides persistent storage of objects created using the object-oriented part of the REFINE specification language. It includes mechanisms for version control, multiple users with concurrency control, computed attributes, and constraint maintenance. The database is used to manage networks of software objects including specifications, code, documents and test cases. It is also used as a repository for application-specific data.

3.3 Language processing system

The REFINE language processing system takes as input a description of a language in the form of a grammar and produces a parser, printer and pattern-matcher for the language. The language processing system is an extension of LALR(1) parser generator technology. Grammars are written using a high-level syntax description language that includes

- regular right-part operators
- precedence tables

- semantic actions of productions
- a mechanism for specifying lexical analyzers.

REFINE provides a template capability whereby templates for programs in a language can be written in an extension of the language that includes wildcards. These templates can be used—

- in pattern matching, to test whether an existing program is an instance of a template, and
- in pattern instantiation, to build a new program that is an instance of the template.

Use of templates in program analysis and transformation applications makes the application code clearer and much shorter—frequently an order of magnitude shorter than the hand-coded equivalent.

The language processing system has been used to build software management tools for a number of languages, including REFINE itself, COBOL, JCL, C, SQL, Ada, and NATURAL. These tools have in turn been used for applications including automated software maintenance, re-engineering, code generation and program verification. Users have created domain-specific languages and tools for domains such as documentation, testing, project management and bug reporting.

4 Examples

We now turn to examples of using an object-oriented database and associated tools in software maintenance, drawn from our customers' and our own experience in using REFINE.

4.1 Analysis Examples

The use of an abstract syntax-based representation of software for analysis is fairly common in modern approaches to software engineering. Most software analysis tools

build such a model as a preliminary to the actual analysis phase. Perry summarizes some of the important uses of abstract syntax trees analysis in software development in [11]. The analysis capabilities and applications described here can be regarded as extensions of these common analyses in that they take advantage of the novel features of the database model of software, namely:

- object-oriented representation of abstract syntax
- use of high-level mathematical data types and query operations
- integration of abstract syntax trees with other software objects in the database such as documentation, test cases, and bug reports

The goal is to allow a wider class of analyses, including very general software queries, and to make it much easier to specify routine analyses such as cross-reference listings.

We illustrate database queries for program analysis with an example taken from software reuse. In order to reuse a piece of software, you first have to be able to find it. In modern programming systems, this task is made easier by the existence of structuring mechanisms such as hierarchical file structure, modules, and class hierarchies in object-oriented programming systems. However, when the volume of programs grows very large, these mechanisms are not good enough—they are based either on file-search commands, fixed indexings of the software (libraries), or interactive browsers. They fail to let the user bring to bear all the available information about the hypothetical extant reusable code. For example, the user may know such things as:

- the probable author or authors of the code
- strings that are probably used in the code
- data structures that are probably used in the code
- approximately when the code was written

Also, the bulk of software maintenance involves programs that were created before modern software structuring techniques were available. Therefore techniques are needed that do not depend on program structuring methodologies but that can take advantage of them where they have been used.

If the software is stored in an object-oriented database that has the properties discussed earlier, then the query language for the database serves as a tool that allows all available information to be used in general searches through the software database. For example, REFINE stores the abstract syntax trees for the user's specifications together with information such as:

- when the program was last modified, and by whom
- what diagnostics were issued by the compiler
- what other programs it uses and is used by
- what documents describe the program

Since the representation of programs is extensible, other relationships can be easily added. For example, it is easy to add an attribute that stores, for each program, the name of the person responsible for maintaining that program. This attribute could then be used in queries over the software database.

To see how this query capability might be used, imagine that you are a programmer maintaining a very large communications system that has been maintained by many other programmers over its lifecycle. You need to use a function that will take two test message sequences, and compute the maximal common subsequence of the two sequences. You have reason to believe that such a function exists. The program's file organization is undisciplined and there is no well-defined "library of test message sequence functions" to scan.

If you were using a REFINE-like system, you could write a query like the one below to find a set of candidate functions:

```
{f | f = `function @@ (a1:test-msg-seq, a2: test-msg-seq):test-msg-seq
      begin .. end'}
```

The above set expression can be read as “find the set of all functions that take two inputs of type test-msg-seq and return a test-msg-seq”. The expression that f is equated to is a syntactic pattern (i.e., a program template) for such a function; it would be written in the syntax of the language used to write the system (e.g., FORTRAN). The “@@” that occurs where the name of the function would go is a wild card; it means that the name of the function is not relevant to the query. Similarly, the “..” in the begin-end is a wild card that allows any sequence of steps as the body of the function.

If you had some ideas as to when the function had been written and by whom, you might write a modified query such as:

```
{f | f = `function @@ (a1:test-msg-seq, a2:test-msg-seq): test-msg-seq
      begin .. end'
  & author(f) in {joe, brenda, kelly}
  & in-time-interval?(date-written(f), <"1/1/72", "1/1/78">) }
```

To make a facility like this more accessible to programmers, we could provide a graphical or form-based interface for specifying queries to replace the above logic-and-set-theory notation. The notation above might then be used as an accelerator for experienced users, or for cases when the graphical interface was not powerful enough.

In developing REFINE, we have used the capability of querying a database of software extensively. It has been useful not only for code reuse, but for answering many other sorts of questions such as:

- What has Joe been working on lately?
- What are all the functions that have changed since the last release?

- What are all the functions that elicited a particular diagnostic from the compiler?

The ability to quickly answer questions involving many different properties is one of the most useful characteristics of databases in general; the need for answering such questions about software during development and maintenance is a strong argument for the type of software representation we advocate.

4.2 Program Transformation Examples

These examples focus on perhaps the most novel capability of the REFINE system—specifying and automatically executing transformation rules that perform complex modifications to software. This is the heart of providing automation for software maintenance activities. Many of the analysis activities discussed earlier are performed with the goal of determining where or how subsequent modifications to the software should be made.

4.2.1 Porting C Applications To A Microcomputer

One common instance of non-portability in programs written in “portable” languages is the syntax of identifiers. For example, in the C language, newer implementations for engineering workstations usually allow identifiers to be more than 8 characters long, but implementations for smaller machines often require identifiers to be 8 characters or less. This means that porting a C program to run on a small machine may require renaming all identifiers longer than 8 characters to identifiers shorter than 8 characters.

Here we have a good example of a transformation that is simple to describe and formalize, complicated to implement using a text-based approach, but easy to implement using a transformation-rule based approach. In fact, a bare-bones REFINE program that performs this program transformation correctly is about 20 lines long and easy to understand.

It is instructive to examine some of the complications that arise in performing this

transformation using a text-based approach, because they are typical of problems that arise in performing non-trivial program manipulations on text strings.

- Lexical analysis: the text-based approach will require performing the equivalent of lexical analysis to determine which character string in the input file defines tokens in the C language—including details such as parsing comments and string constants correctly.
- Identifying identifiers: the text-based approach will require knowing which tokens are identifiers, so that we don't inadvertently rename keywords in the language.
- Avoiding conflicts: the text-based approach will require keeping track of which identifiers (both original and shortened) have been encountered so far, so that we don't create a name conflict arising from C's scoping rules.

Getting all the details correct using this approach will take time and experimentation. The program will duplicate many of the analyses performed by a C compiler during parsing—breaking the input into tokens, performing lexical scope analysis, etc. On the other hand, the REFINE program that performs the same task has none of these problems—the C program stored in the database embodies the results of syntactic and static semantic analysis.

Here is the REFINE rule that forms the heart of the required program transformation:

```
rule rename-long-identifier (id)           % The input to the rule is an object "id"
  identifier(id)                           % If id is an instance of the class "identifier"
  & length(name(id)) > *max-id-length*    % and id's name is longer than the limit
  -->                                     % then
    new-name = make-new-name(id)          % generate a new name called "new-name"
    & name(id) = new-name                 % rename the identifier id to new-name, and
    & (ref in identifier-references(id)   % for each occurrence of the id in the program
      --> name(ref) = new-name)           % rename it too to preserve consistency.
```

The REFINE code that applies this transformation to an entire C program is:

```
preorder-transform(my-program, ['rename-long-identifier])
```

which can be read as “traverse the tree rooted at the object `my-program`, applying the rule `rename-long-identifier` at each object in the tree”.

This approach can be used to solve related problems that arise in software maintenance, such as the problem of merging two large programs written in a language that does not support any scoping mechanism for functions.

4.2.2 Porting SQL Database Applications

This example has features that occur in a variety of software porting applications.

Suppose you are a software vendor producing application programs that use an SQL-based relational database. Such programs are often written in C with embedded calls to SQL. While there are standards for SQL, the *embedding* of SQL in C (or other languages, for that matter) is far from standardized—each database vendor defines its own embedding, and they are substantially incompatible. How can a database application written for a specific database be ported to work with a different relational database?

When we encountered an instance of this problem, we evaluated a few different approaches including:

- manual recoding of the application
- restructuring the application to use macros and libraries that encapsulate the differences between the database systems
- creating the new version from the original using program transformations

The first approach was feasible, but the application was already quite large and was projected to grow to one million lines of code. Since embedded SQL was used pervasively, the first approach would lead to a large manual recoding effort.

The second approach initially seemed promising, but on closer analysis it was fraught with problems. The two most serious problems were:

- Encapsulating functionality in libraries of functions and/or macros presupposes the ability to pass the variable part of the behavior as parameters to functions or macros, but the parameterization mechanisms in C did not provide the necessary features.
- A particular operation may have a variety of special case translations that will result in much greater efficiency but can only be used under special circumstances that depend on the context of the operation.

We decided to prototype the third approach—a program transformation system for automatically porting programs different to vendors' SQL implementations. Our prototype focussed on the most difficult part of the process, namely, the translation of application programs involving *dynamic queries*—queries that are generated at runtime.

Dynamic queries allow an application to obtain arbitrary SQL queries from the user and then pass them to the DBMS. Dynamic queries must allocate space to store the table that results from the query. Because the details of the query are not known at compile time, the shape of the table—the number and types of its columns—is also unknown at compile time. The application must include C code to interpret a descriptor of the table's shape that is returned (along with the a pointer to the table) by the database, thus allowing the

application to destructure the rows it extracts from the table. Other details that must be handled at runtime include interpreting error conditions. The protocol for performing dynamic queries and interpreting the results differs greatly among SQL vendors.

The prototype program transformation system was built in RE/FINE in one week, and resulted in the following tools:

- Two extensions of a RE/FINE grammar for the C language — one each for the two vendors' extensions of C to allow embedded SQL. From each grammar, RE/FINE automatically produced a parser, printer, and pattern-matcher.
- A set of about 10 transformation rules that implement a translation strategy for porting application code involving dynamic queries.
- A file translator that parses input files, applies the transformation rules automatically, and prints the resulting program in an output file.

One of the transformation rules is shown below.

This experience demonstrates that real-world software porting problems can be automated using commercially available program transformation systems.

The overall strategy for translating C files containing XX SQL into C files containing YY SQL calls for—

- maintaining, in the YY (translated) version, a data area that simulates the XX data area, together with a YY data area that will actually receive the results of the YY SQL calls;
- including, in the YY version, code that converts YY data format to XX data area so that the original C code that uses this data can be preserved, assuring correct functionality; and
- adding optimizations as our understanding of performance increases.

What are the incompatibilities in SQL data area declarations that require translation? In XX embedded SQL, it is not necessary explicitly to include a C declaration of an SQL data area variable. In YY embedded SQL, an explicit C declaration is necessary. Because the translation strategy calls for maintaining a simulated XX data area, the translator will have to introduce a declaration for this variable. Also, the YY version of SQL requires *two* SQLDA variables, one for BIND and one for SELECT. Thus a total of *three* new variables must be declared.

Below we show a transformation rule used to introduce the new variable declarations. Translation rules are applied *after* the source files have been parsed; they are applied to the abstract syntax representation of the source code.

The rule looks for an embedded SQL data area declaration statement in the C source code. Such a statement has the form "EXEC SQL INCLUDE SQLDA". (The syntax of this embedded SQL statement is the same in both XX and YY embedded SQL. Translation is needed not for this SQL statement but for the related C code.) When one is found, the three new C variables are declared immediately following it:

- one in the declaration

```
xx_sqlda_type xx_sqlda
```

to hold the simulated XX-formatted data, and

- the other two combined into one declaration

```
SQLDA * yy_bind_sqlda, * yy_select_sqlda
```

to hold the BIND and SELECT results.

Note that the rule also adds comments indicating that a change was made and what the introduced variables will be used for.

```
Rule Add-YY-SQLDA-Declarations(s) % The input to the rule is an object "s"
    embedded-sql-statement?(s) % If s is an embedded SQL statement
& sql-form(s) = 'INCLUDE SQLDA' % of the form "INCLUDE SQLDA"
& parent-expr(s) = the-file % and s is from the file "the-file"
& file-definitions(the-file) =
    [$preceding-defs, s, $succeeding-defs]
    % and s is one of the declarations in the file
& ds = make-SQLDA-declarations-for-name("SQLDA")
    % and ds is a sequence of two new variable
    % declarations based on the string "SQLDA"
    % (one to simulate the XX SQL data area
    % and one to be the actual YY data area)-generated
    % by calling a function that creates unique variable
    % names-
& ds = [new-XX-decl, YY-decl]

--> % then do the following:

file-definitions(the-file) =
    [$preceding-defs, s, $ds, $succeeding-defs]
    % include the new variable declarations among
    % the file's original declarations and
& doc-strings(new-XX-decl) = % add appropriate documentation lines for both
    % new variable declarations.
    ["/* CHANGE: added new variable below, used to simulate the XX */
/* Descriptor Area named SQLDA */"]
& doc-strings(YY-decl) =
    ["/* CHANGE: added below two new variables that will hold the YY */
/* BIND and SELECT Descriptor Areas corresponding to SQLDA */"]
```

After introducing the new variable declarations, the translator must translate C references to the old variable (SQLDA) to references to the new variable name. When the original C/SQL source file was first parsed, C identifier references to SQLDA were detected. Also, while generating the new variable declarations, `make-SQLDA-declarations-for-name` maintained a global mapping of old variable names to new names. Here is the rule that translates C variable references:

```
Rule translate-references-to-XX-SQLDAs (r)
    identifier-ref (r)           % if r is an reference to an identifier
    & name(r) = "SQLDA"         % and its name is SQLDA
    -->                          % then
    name(r) = *new_XX_DA_name*("SQLDA")
                                % change the name of the identifier reference to be
                                % the name of the new variable introduced earlier
```

Several more translation rules are needed to handle data areas—for example, to translate SQL references to the old variable names, and to add code to translate among the different DA formats.

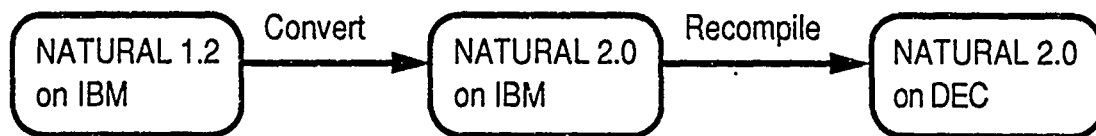
4.2.4 Converting between versions of a programming language

Two of the advantages of high-level languages over assembler and machine languages are portability and ease of integration. In theory, porting a program written in a high-level language to a different machine is easy if there exists a compiler for the same language on the target machine—one simply recompiles the source code. Also, in theory, high-level languages make it easy to combine programs using mechanisms such as external subroutines.

In practice, of course, it's not so simple. One reason is that most popular high-level languages (e.g., FORTRAN, Pascal, COBOL) have a maddening variety of dialects and versions that at least partially (and sometimes completely) remove the advantages mentioned above. Programs written in one dialect cannot be compiled with a compiler written for another dialect, and programs written in different dialects cannot necessarily call each others' routines. These incompatibilities between dialects introduce the need to convert programs among dialects in order to combine them and run them on different machines. Program transformation is often the most effective technology for automating

these conversion processes, because the necessary changes are structural (rather than textual) in nature, and must be made pervasively across large bodies of application source code.

An example of such a conversion task arises in connection with the language NATURAL™. NATURAL is a COBOL-like language used primarily for MIS applications. There are several versions of NATURAL on different machines with varying degrees of compatibility. In particular, there is a recent version (NATURAL 2.0) that runs almost identically on IBM mainframes and DEC equipment. The older and most widely used version for IBM mainframes is NATURAL 1.2, which is substantially incompatible with NATURAL 2.0. Thus customers must convert NATURAL 1.2 applications to NATURAL 2.0 and then recompile them in order to run them on DEC equipment.



A fully automated NATURAL 1.2 → 2.0 converter is currently under development using REFINe. Before choosing REFINe, the customer investigated several strategies for conversion tools, including text-based approaches and approaches using YACC and C. The customer and Reasoning Systems jointly developed a prototype converter using REFINe in two weeks. The prototype included parser/printers for subsets of both NATURAL 1.2 and NATURAL 2.0, and transformation rules that handled several key incompatibilities between the two language versions. The prototype was able to completely convert several examples.

Below is an example conversion rule used in the NATURAL 1.2 → 2.0 prototype converter. This rule makes a conversion that is necessary because NATURAL 1.2 allows variables to be defined anywhere within a program, whereas NATURAL 2.0 requires that all variable definitions occur within a single "data definition" clause at the

beginning of the program.

```
rule hoist-variable-definition (node)
  variable(node)
  & var-fmt = variable-format(node)
  & *data-def-clause* = `DEFINE DATA LOCAL $old-defs END-DEFINE'
  —>
  variable-format(node) = undefined
  & new-var = make-variable(var-name, var-fmt)
  & *data-def-clause* =
    `DEFINE DATA LOCAL $old-defs, @new-var END-DEFINE'
```

In this rule, the input, *node*, is first tested to see if it is an inline variable declaration. If so, then *var-fmt* is set to the format of the inline declaration. Then (on the right hand side of the arrow), a new variable declaration *new-var* is created and added to the "data definition" clause **data-def-clause**. Also, the variable format of *node* is erased, effectively deleting *node* from the program.

5 Summary and conclusion

We observed that software maintenance and re-engineering requires two broad categories of activities: *analyzing* and *transforming* programs and related objects. We have described an approach to software maintenance based on

- an object-oriented database representation for software lifecycle objects and
- automated transformation of the objects represented in this database.

We have found that the object-oriented database representation more closely approximates the conceptual model held by developers than is possible with a text file-based system. Analysis and transformation can be significantly automated by tools that take advantage of the database representation. We have described REFINE, an environment for program representation and transformation that provides the tools needed for software maintenance and re-engineering. We have illustrated the approach with examples taken from actual experience in re-engineering customers' code in C, SQL and NATURAL, and we have illustrated how the maintenance of REFINE itself has been automated using the same tools.

The ability to support automation in modifying large software systems by using rule-based program transformation is a key innovation of our approach that distinguishes it from systems that focus only on automation of program analysis. The features required to support this transformational technique require substantial extensions to an object-oriented database system to support efficient representation of programs and the ability to convert easily between the text and database representations.

The transformational approach to software development was first developed for synthesizing code from high-level specifications, but its range of applicability now appears to be much larger. It may well be that this technology will make its first significant impact on software engineering practices in the areas of maintenance and re-engineering, particularly in MIS applications, rather than in software development.

Acknowledgements

The authors would like to thank Cordell Green, Scottie Brooks, Scott Burson and Peter Ladkin for their many ideas and insights.

References

- [1] Feldman, S. "MAKE - a program for maintaining computer programs," *Software - Practice and Experience*, 9 (1979), pp 255-265
- [2] Rochkind, M. "The source code control system," *IEEE Transactions on Software Engineering*, SE-1 (1975), pp 364-370
- [3] Leblang, D. and Chase, R. "Parallel Software Configuration Management in a Network Computing Environment," *IEEE Software*, November 1987
- [4] Tichy, W. "RCS - A System for Version Control," *Software - Practice and Experience*, 15 (7), pp 637-645
- [5] Teitelman, W. and Masinter, L. "The Interlisp Programming Environment," *Computer*, 14 (4), pp 25-34
- [6] Reps, T. *Generating Language-Based Environments*, MIT Press, Cambridge, MA, 1984
- [7] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983
- [8] Johnson, R., Graver, J., and Zuraski, L. "TS: An Optimizing Compiler for Smalltalk," *Proceedings of OOPSLA-88*, September, 1988
- [9] Linton, M. "Implementing Relational Views of Programs," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium*, The Association for Computing Machinery, NY, 1984
- [10] *The REFINE User's Guide*. Reasoning Systems, Palo Alto, CA, 1985
- [11] Perry, D. "Software Interconnection Models," *Ninth International Conference on Software Engineering*, IEEE Computer Society Press, 1987

An Approach to the Support
of Dynamic Extensibility
in Nonstop, Distributed
Target Environments

by
Charles W. McKay
Software Engineering Research Center
University of Houston - Clear Lake

Abstract

An increasing number of applications for computer automated systems require large, complex, nonstop, distributed target environments. Many of these systems must also support a growing and changing set of demands for the deployment and operation of mission and safety critical (MASC) components throughout a long, incrementally evolving life cycle. Thus far, most of the reported research on the development and application of knowledge based paradigms to the life cycle issues of software engineering has not focused on the specific challenges of such systems. In particular, the paradigm is more often applied in the capture of requirements and the development and representation of solution specifications from which automated generations of implementations may be produced. Subsequent changes in requirements lead to new specifications and a new generation of an implementation which is typically intended to replace the old one by stopping the system, loading the new code, and beginning execution. The Portable Common Execution Environment (PCEE) is a NASA sponsored research project to reduce the life cycle risks of supporting a changing set of MASC components in nonstop, distributed systems. These issues of dynamic extensibility have led to a proposed solution approach which exploits a knowledge based paradigm. This paper reports the major features of this approach which is now under study for the host development environment portion of the PCEE project.

Key words: nonstop, distributed systems, mission and safety critical (MASC), portable common execution environment (PCEE), knowledge based (KB) paradigms

Introduction

An increasing number of applications require large, complex, nonstop, distributed target environments which must support mission and safety critical (MASC) components through a long,

incrementally evolving life cycle. Examples include the Space Station Freedom Program (SSFP), moon colonization and other programs of NASA and her international partners. The Portable Common Execution Environment (PCEE) is a research project at NASA's Software Engineering Research Center at the University of Houston - Clear Lake. The program is focused on the reduction of risks in the life cycle issues of supporting MASC components in such applications. Although knowledge based, life cycle paradigms clearly have much to offer in the initial development of such systems, much of the international research has not been focused on the problems of dynamic extensibility in incrementally evolving, nonstop systems. (Eg, Green et al, '83) Specifically, the paradigm is more often applied to the capture of requirements and the representation and development of solution specifications from which implementations can be automatically generated. Subsequent changes in requirements result in changed solution specifications and a newly generated implementation rather than direct modification of the original code. However, when lives, health, property, the environment and the success of missions depend upon continuous operation of the target environment resources even while modifications and extensions to the applications and systems software are being made, then both the paradigm and approaches to its implementation must be reconsidered. The results must avoid a requirement of shutting the old system down and bringing up a new one each time there is a need to accommodate the timely introduction of safe and reliable changes. This paper reports the early investigation of an approach to these issues in the host environment portion of the PCEE project. The paper begins with an elaboration of the conceptual foundation underlying the approach now under study. The paper concludes with the status and projections for the future.

Key Terms

The following terminology will be used to explain the underlying conceptual models in three environments addressed by the PCEE project. The first underlying model describes a host environment where computing solutions to automation problems are proposed, developed, and sustained. Another underlying model describes a target environment where the solutions from the host environment will be deployed and operated. The third underlying model describes an integration environment which bridges the first two. A tutorial introduction to these conceptual models and the prerequisite concepts and principles is provided in AIRMICS, '89 with additional discussion in the report by McKay, Burgett, and Collins (1989).

An entity is a representation which describes either a logical or a physical "thing" having both an individual existence and a distinct identification. A relationship is an ordered association between two entities where the ordered association has

both an individual existence and a distinct identification. An attribute is a property of an entity or of a relationship where the property has both an individual existence and a distinct identification. Note that attributes may include information on: values and constraints, predicates and triggers, temporal and spatial semantics, and normal and exception processing. (Triggers are actions to be automatically initiated when their predicate is satisfied. Together with entities and relationships, they may also include semantic information on: information related to other information, information related to behavior, behavior related to other behavior, behavior related to context, etc.)

Abstract types define the sets of legal: contexts, values, and operations which may be visible at the interface to any instance of the abstract type. These contexts allow abstract types to be constructed (via visible relationships) from other abstract types. Examples of abstract types include but are not limited to: data/information, processes/behaviors, interface sets, etc.

Objects are instances of abstract types. They should be used to structure and control complexity by abstracting a design decision and encapsulating its associated implementation details. Each object may consist of three parts: context information, an abstract interface specification (AIS), and an implementation part. For a given context, the AIS of an object specifies:

1. the services and resources to be provided, consumed, or affected by the object
- and for each service and resource,
2. how well it is to be supported (e.g., precision)
 3. under what circumstances (e.g., normal versus exception processing).

Note that the concept of services implies operations performed by an object on behalf of a user of the object whereas the concept of resources implies "raw" entities available to or from an object. Thus a resource may be a type definition or a value of a data variable.

Objects communicate by messages which specify: context, services, and resources to be exchanged between the communicating objects. Note that messages convey the temporal and spatial semantics of: normal versus exception processing (i.e., context), control flow (i.e., services), and data flow (i.e., resources) for each communication between a source object and a destination object. This is in sharp contrast to the effects of more traditional analysis and design methodologies which separately consider control flow and data flow and lose much of the temporal and spatial semantics (as well as context) which describe control and data flow interactions.

Objects may be classified into three categories based upon the visibility of threads of control in the object's AIS: passive,

neutral, and active. Passive objects offer services and, possibly, resources but they must borrow the thread of execution of the caller in order to perform the operations associated with the requested service. This means the thread of control is suspended in the calling object and "loaned" to the called object which will return it after using it to perform the requested service. Ada packages with procedures and functions visible in their AIS (but no tasks) are examples of passive objects. (Many languages offer only passive objects. Hence the confusion in the literature is exacerbated.) Such passive objects may serve as triggers used in knowledge based systems.

Neutral objects contain resources only (e.g., type declarations and values). They neither borrow a thread of control (i.e., no services are provided) nor possess their own. Examples include the relations of a relational data model (i.e., data defined in terms of other data types and data).

Active objects possess their own visible thread of control as in Ada tasks. Not only does this allow program structuring to exploit parallelism inherent in the environment, but the firewalled semantics of each active object allow program structuring to reflect separation of concerns so that a failure in a "nice-to-have" active object does not necessitate a failure in a "have-to-have" active object. Active objects may serve as pattern matching demons in knowledge based systems. (I.e., they have their own action rules and advance their own state in response to or in search of a match for their predicates.)

The reader should note that each object has both an individual existence and a distinct identification. Therefore an object can readily be associated with an entity in an EA/RA form of model. Thus its external relationships to other objects/entities are depicted within the model. Furthermore, the context, AIS and implementation parts of an object can all be internally decomposed into an EA/RA model. In a sense, the external relationships among objects only provide information visible in the context and AIS parts since implementation details are encapsulated and hidden in the implementation parts. Thus it is reassuring to know that the same form of modeling that represents the "whats, how well, and under what circumstances" at the public visibility level is also an appropriate form of modeling to represent the hidden "how tos". The reader should also note that the relationships between communicating objects/entities can be readily associated with messages and their temporal and spatial semantics.

Just as objects, messages, and abstract types can be used to superimpose a useful discipline upon model representations in EA/RA form, at least two higher levels of discipline can be superimposed on objects to facilitate constructing and sustaining well engineered computing systems. The next level of discipline is based upon a host environment construct called stable frameworks. The succeeding level of discipline is based upon a

target environment construct called stable interface sets. Essentially, stable interface sets allow collections of objects to be deployed and operated such that all objects, their interrelationships, and their key object and relationship properties have well understood structure and behavior. Stable frameworks are basically a configuration and change control construct in the host environment that allow objects to be successively combined and integrated until they reach the status of a stable interface set which is ready for deployment.

Knowledge Based Models

Object based, semantic models in EA/RA form may be used to represent a solution to a given problem space. Within such models, a knowledge based paradigm may be used to capture expert knowledge and associated knowledge handling as an important added value to the more traditional paradigms that may constitute the bulk of the solution. The draft standard (ANSI X3H4, 1985) for the Information Resource Dictionary System (IRDS) offers a promising opportunity to develop, sustain, and exploit standard, machine independent, on-line representations of object based, semantic models which integrate value added knowledge based paradigms with more traditional designs. More will be said about this in the next section.

Mylopoulos and Levesque (Brodie et al, 1984) classify knowledge representation for such systems into four categories: semantic nets, logical schemes, procedural schemes, and hybrids of these three (eg, frames). The two part basis of the classification rests on the concepts of states and state transformations. State is defined as the set of all entities and their relationships at any one time in any one model. State transformation within a model causes creation/destruction or modification of entities or changes in their relationships. Attributes are an important factor in states, state transformations, and the ability to reason about the model.

The first category, semantic nets, focuses principally on states (ie, E's and R's) of a model. The second category, logical schemes, focuses principally on assertions about states. Typically, such a scheme uses a collection of logical formulas to provide a partial view of a state. Modifications occur when a logical formula is added or deleted from the knowledge representation. By contrast, the third scheme, procedural, focuses on state transformations. Here a knowledge representation consists of a global database of assertions plus an associated collection of theorems (also called demons in this approach) which watch over the database and are activated whenever the database is modified or searched.

There are purists in each of the three categories. Claimed advantages by each camp include the following. Logical scheme

advocates point to the easy support for inferencing rules based on clean semantics. Critics point out the difficulty in representing procedural and heuristic knowledge and the difficulty in organizing and sustaining large, complex knowledge representations. Network scheme advocates point to the ease of organizing and graphically representing large, complex knowledge representations plus the ease of associating access paths with relationships. Critics point out the lack of formal semantics and standard terminology (although IRDS is a major step toward the resolution of abused and overloaded terms). Fans of the procedural scheme point to the ease of specifying direct interactions among facts with no wasteful searching. Critics emphasize the difficulty of understanding and modifying such representations.

In this same reference, Brodie (Brodie et al, 1984) makes the point that real world applications can be characterized by:

- . static properties of structure
- . dynamic properties of behavior (eg, operations and their properties and relationships)
- . integrity rules (eg, constraint management) over structure and behavior.

He concludes that none of the three approaches is recognized to be superior in addressing all three issues.

Since none of the three models in "pure" form have been clearly demonstrated to be adequate for modeling large, complex, distributed applications, hybrid approaches deserve consideration. This is particularly true of the issues of dynamic extensibility in nonstop environments with MASC components. The next section introduces the hybrid scheme recommended in this paper.

Recommended Approach

As pointed out earlier, objects, messages, and abstract types can be used to impose a discipline on semantic modeling in EA/RA form. Furthermore, IRDS provides a standard for representing instances of EA/RA models via extensible and tailorable schemas and dictionaries. Thus, whereas a stable framework may depict relationships among Abstract Interface Specifications (AIS) of an integrated collection of objects (as well as the attributes of these AIS's and their R's), one may "look beneath" this macroscopic view into either the AIS or the implementation part of any object and see a further decomposition of the internals into EA/RA form. Such recursive modeling can be easily represented in IRDS.

More unusual as part of the approach, however, is the proposed special treatment of attributes combined with active interfaces in the run time environment to the IRDS compatible

dictionaries and schemas. The application knowledge will be captured in the metadata and metadata processes of these dictionaries and schemas. Access to sharable and persistent objects in the object base will be controlled by interactions with these active interfaces.

In IRDS, attributes are typed just as entities and relationships are. This means that for each attribute, there is a known (ie, constrained) set of legal values and legal operations for each legal context of use. For example, a sensor input may have legal values constrained to type integer and further constrained to range between 4 and 20. Just as these constraints can be represented in IRDS, so can a legal extension called a predicate set which is to be associated (ie, related) with this attribute. As an example, each legal sample value reported from this sensor could satisfy a predicate that triggers setting a status bit in a bit vector representing sensors reporting legal values since the last scan. Note that the shared and persistent variable where the value is to be stored is accessed through its identifier reference to the appropriate dictionary. Transparent to the code which provides the value, the dictionary entry is checked by a constraint enforcement procedure to find the definition of legal values. This procedure is a passive object also modeled in the dictionary. Since the 4 to 20 predicate is satisfied, another procedure call is made to the associated trigger. This too is a procedure modeled in the dictionary. This trigger sets the appropriate status bit in the bit vector which is also modeled in the dictionary. Note that the code that provides the sensor value contains no notion of the integrity checks and status reporting that are effected via the active dictionary.

To illustrate the concept of demons, consider this sensor example further. Suppose that low values less than 4 are associated with a predicate set that further discriminates these legal values into two categories: a value of 0 (probably indicating an "opened" circuit) or other-than-0 (probably indicating a "drifted" amplifier in need of recalibration). If the predicate for less-than-4 and other-than-0 is satisfied by the next reported value, a call may be issued to a recalibration demon associated with the predicate. The demon (eg., an Ada task modeled in the dictionary) might accept this call and thus release the thread-of-control of the caller so that it may proceed. Later, the demon may report the results of the attempted recalibration. It may even signal another demon responsible for maintaining a statistical profile of all calibration and maintenance metrics. Once again the code that reported the sensor value has no notion of what's going on behind the scenes. (Nor should it have. Remember that these sensor values were identified as shared and persistent data. Therefore the reporting code should not be responsible for direct storage and manipulation of either this data or its associated meta data/knowledge.)

To support dynamic extensibility, a new meta relationship can be defined to serve as a link from a self identifying, self

descriptive instance of a schema and its associated dictionary to a newly created descendant schema and its associated dictionary. By legal extensions to IRDS to include definitions of controlled inheritance and extensibility, the descendant schema can be defined in the host environment with respect to the parent schema. The changes can then be installed via messages under the control of an interactive integration environment. Thus if, in the preceding example, it becomes desirable to modify the definition of one attribute of a set of software processes for the sensors, to add two new attributes, and to inherit all other attributes that now exist for the set, then the null value in the link field of the current schema for this set can be replaced with a reference to a new descendant schema with its own dictionary for future instances of the abstract types defined in the new schema. The integration environment may then be used to control the individual blocking and replacement of old sets with new ones (possibly after an observation period of monitoring the concurrent operation of old and new) or the old sets may continue at some locations while new ones are used in others. In this latter case, a query of a commonly defined attribute can result in information form new and old alike. A query of the two new attributes may result in information reflecting only the new descendants in the family tree with an indication that there are ancestors that lack these attributes.

Summary

In summary, the proposal is to legally extend IRDS (which may be done for any IRDS application according to specified rules) to define a new level of metaknowledge associated with attributes. Ie, constraints, predicates, triggers, and demons where each is modeled in the schemas and dictionaries as a form of meta-meta objects (eg, triggers, demons, predicates), meta-meta relationships (eg, predicate-calls-trigger), and meta-meta attributes (eg, 0 and other-than-0 for less-than-4) associated with the attributes of relationships and entities. The extensions also include the concepts of links to descendant schemas and associated dictionaries with controlled inheritance from the ancestor schemas. Semantic queries may be made against a family tree with differing attributes among members of differing generations.

The disadvantages of such an approach appear to include the additional complexity of metaknowledge concerning some key attributes and relationships (although this too is simply another level of recursion). Another potential disadvantage that must inevitably be considered is performance (although recent experiences with multi-processors indicate the potential to resolve this problem). The advantages of such an approach appear to include the following. First, Brodie's three characteristics of real world applications can clearly be supported. Also, since inferencing rules can be mapped to constraint management in a very

similar way to the preceding example, the principal benefit of the logical model has been retained, although arguably in a less straight forward manner. Furthermore, the authors believe that, unlike the logical scheme, the approach offers far less difficulty both in organizing and sustaining large, complex knowledge representations and in representing procedural and heuristic knowledge.

The proposed approach may complicate the current ease within the network scheme of organizing and representing large, complex knowledge representations. The ease of associating access paths with relations is unchanged. However IRDS helps improve the network disadvantage of non-standard terminology and a combination of precise modeling and formal modeling certainly improves semantic representations.

Fans of the procedural scheme should note that the approach has fully retained the ability to specify direct interactions among facts with no wasteful searching, although arguably some ease of specification has been sacrificed. The author believes that the highly recursive approach to modeling in EA/RA form permitting macroscopic views (eg, object AIS's integrated into a stable framework), microscopic views (eg, E's, A's, and R's of the internal contexts, services, and resources of an AIS), and "submicroscopic views" (eg, associating predicates, triggers and demons with attributes) will ultimately facilitate both understanding and modifying such representations.

Status

The PCEE project is in its first year. High level design and assembly of the first increments of needed physical resources for the host environment are nearing completion.

Acknowledgements

The author gratefully acknowledges the many contributions of the team members of the PCEE project as well as those on other projects that contributed substantially to the concepts and principles now in use. In particular, two projects in this latter category include AdaNET (NASA sponsored) and DOBIS (IBM sponsored). Opinions and conclusions are solely the responsibility of the researchers and do not necessarily reflect those of the sponsors.

Bibliography

AIRMICS '89, Ada Reusability Guidelines, section titled "Conceptual and Implementation Models" by Charles McKay, to be published in 1989 by AIRMICS.

ANSI X3H4, Parts 1..4 (Draft Proposed) American National Standard Information Resource Dictionary System, American National Standards Institute, New York, 1985.

Brodie, M., Mylopoulos, J., Schmidt, J. (editors), On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages, Springer Verlag, New York, 1984.

Green, C., Luckham, D., Balzer, R., Cheatham, T., Rich, C., Report on a Knowledge Based Software Assistant, Rome Air Development Center, RADC-TR-83-195, 1983.

McKay, C., Burgett, D., Collins, G., Distributed Object Based Information Systems (DOBIS), SERC/HTL@UHCL, (IBM Contract 2-5-51539), 1989.



MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.