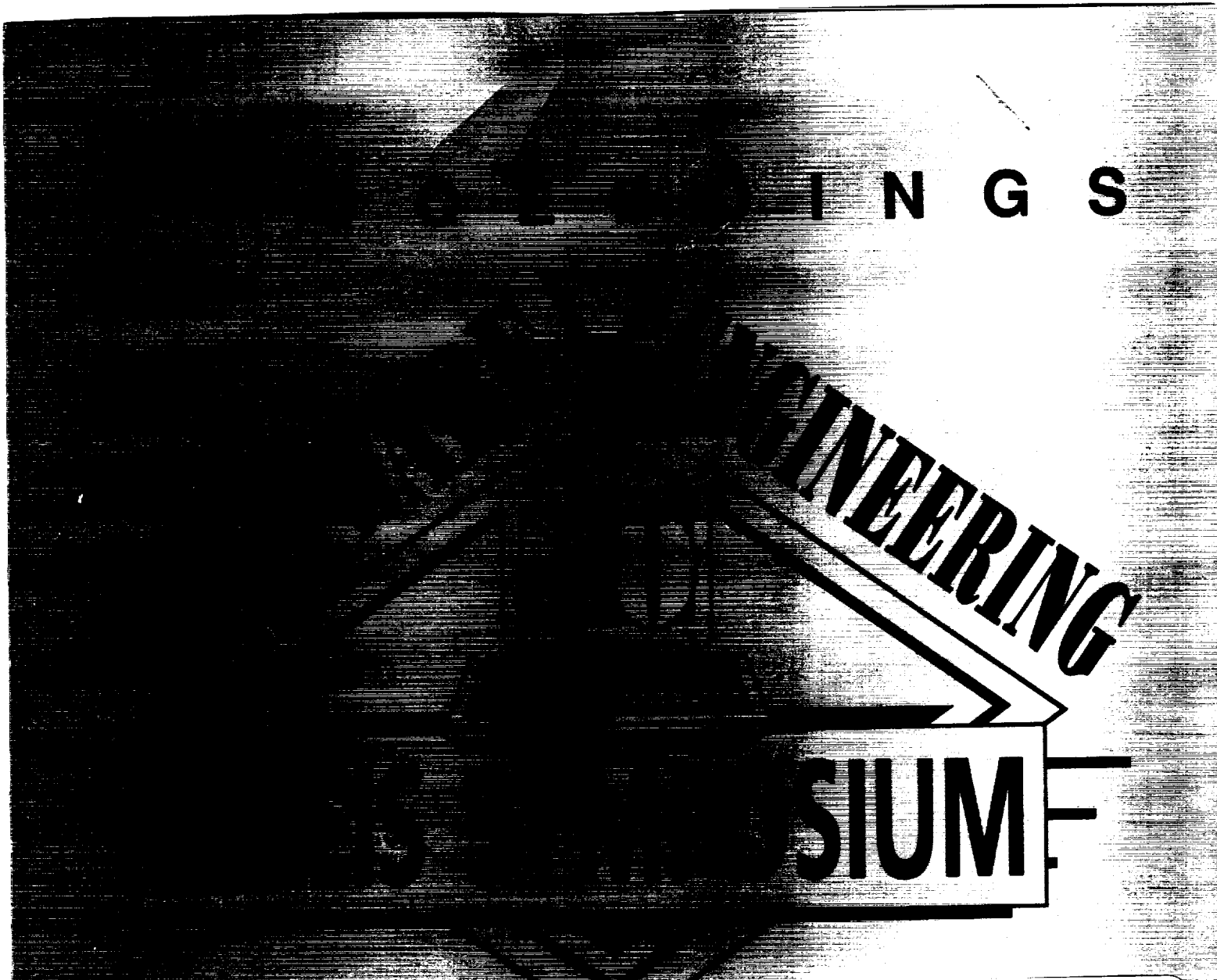
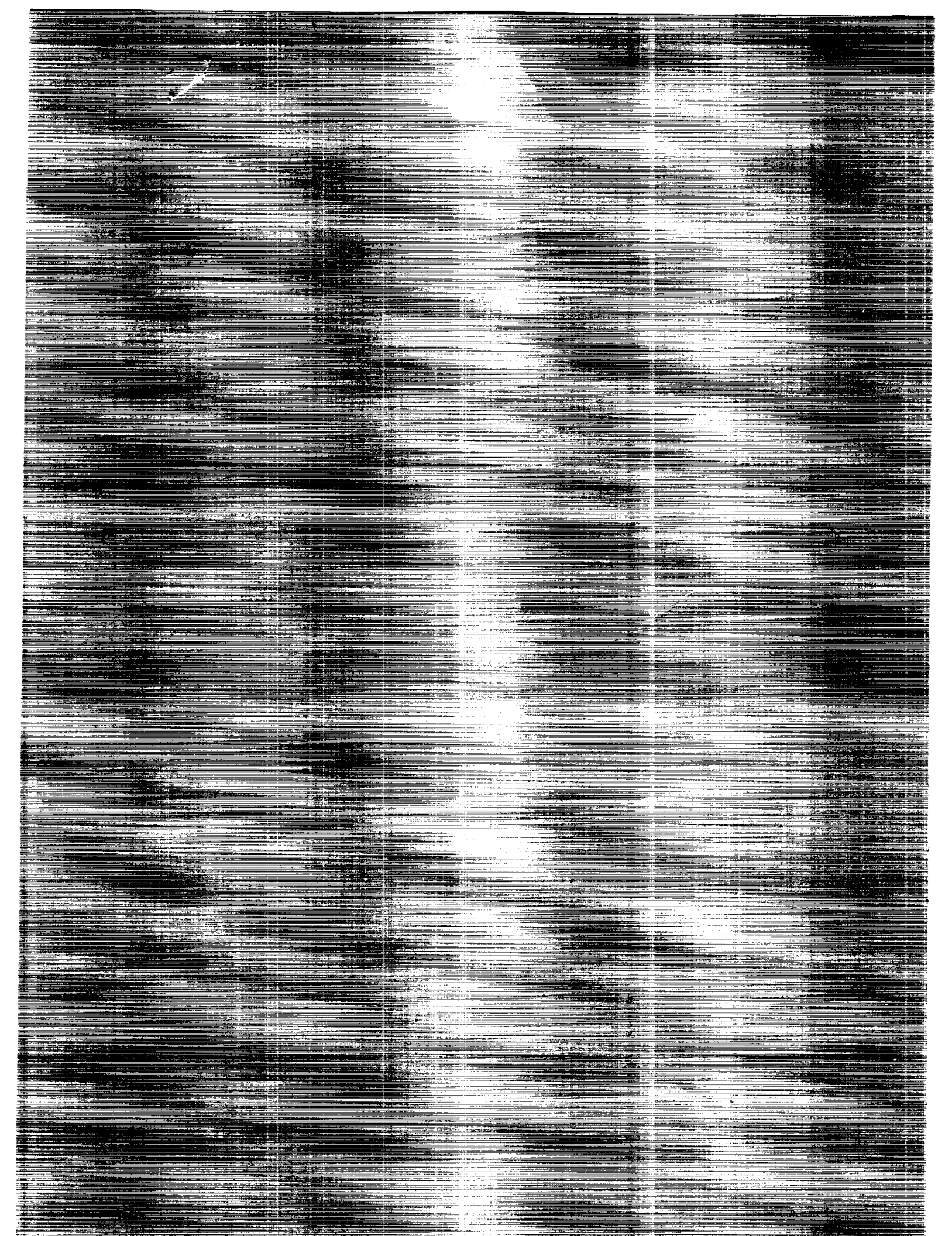


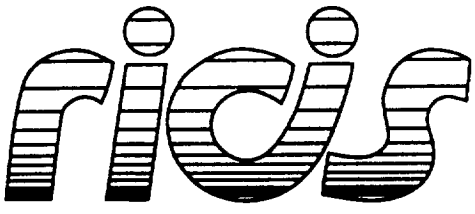


RICIS Symposium '90

Software Engineering:
Aerospace Applications & Research Directions

November 7 & 8, 1990

Houston, Texas



University of Houston-Clear Lake

NASA/Johnson Space Center

ricis

Research Institute for

University of Houston-Clear Lake

Computing & Information Systems

SOFTWARE ENGINEERING

Aerospace Applications & Research Directions

RICIS Symposium 1990

There is international concern about the impact of computer systems upon life and property issues within all elements of a world wide community of interacting social, financial, medical, and technical organizations. Software Engineering has achieved recognition as the discipline that captures the paradigms that must be used to produce high integrity software within limited time and budget constraints. The aerospace community has become a leader in applying these disciplines to software that provides control of life and property. During the decade of the 1990's these disciplines need to be applied to non-aerospace life and property critical computer systems.

The RICIS '90 Symposium has been organized to provide a review of current and future applications of software engineering paradigms. Distinguished professionals from industry, government, and universities have been invited to participate and present their views and experiences regarding research, education, and future directions of software engineering.

We trust that you will find this symposium to be informative and enjoyable.

Rodney L. Bown
Technical Co-Chair

Sadegh Davari
Technical Co-Chair

2700 Bay Area Blvd. · Box 444 · Houston, TX 77058-1098
(713) 283-3800 · FAX (713) 283-3810

University of Houston-Clear Lake & NASA/Johnson Space Center

This conference is one in a series of conferences presented under the auspices of the University of Houston-Clear Lake's Software Engineering Professional Education Center (SEPEC), which is the education and training branch of the Research Institute for Computing and Information Systems (RICIS). The University of Houston-Clear Lake founded RICIS in cooperation with NASA/ Johnson Space Center and the aerospace community.

The Mission of RICIS

The institute's mission is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from the University of Houston-Clear Lake, NASA/Johnson Space Center, the aerospace and computing industries, and other research organizations.

The Mission of SEPEC

The mission of the Software Engineering Professional Education Center is to provide education and training for software professionals with an emphasis on large, complex distributed systems. SEPEC also serves as a test bed for research and innovation in software engineering education and training.

ADA Users' Symposium

RICIS '90 shares the week with another key software engineering activity, the Third Annual NASA Ada Users' Symposium. Topics to be covered on Tuesday, November 6, will complement RICIS '90. The Ada Users' Symposium is hosted by NASA/JSC, MITRE Corporation, and UH-Clear Lake; and will be held at the NASA/Gilruth Center. It is free; however, pre-registration is recommended. Contact SEPEC at (713) 282-2223 for additional information and registration.

Conference Steering Committee

General Co-Chairs:

A. Glen Houston, *Director, RICIS, UH-Clear Lake*
Robert B. MacDonald, *Assistant for Research and Education-Mission Support Directorate, NASA/JSC*

Technical Co-Chairs:

Rod L. Bown, *Associate Professor of Computer Science, UH-Clear Lake*
Sadegh Davari, *Assistant Professor of Computer Science,* UH-Clear Lake

Administrative Co-Chairs:

Glenn B. Freedman, *Director, SEPEC, UH-Clear Lake*
Don Myers, *Coordinator, SEPEC, UH-Clear Lake*
Glen Van Zandt, *Human Resource Development Specialist, NASA/JSC*

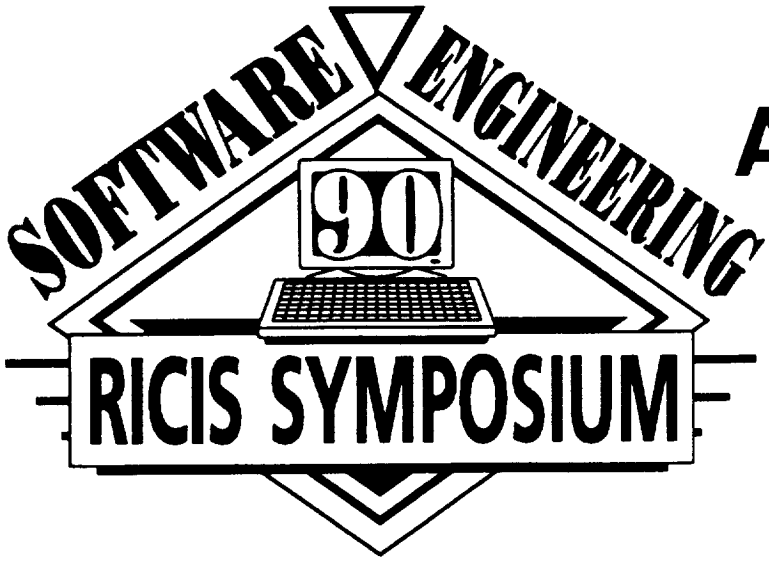
Conference Overview

Wednesday, November 7

8:00 - 5:00	Registration
8:30 - 12:00	**Tutorials**
12:00 - 1:00	Lunch for tutorials
1:15 - 1:45	**Welcome & Introductions**
1:45 - 2:45	**Keynote Address**
2:45 - 3:00	Break
3:00 - 5:00	**Session 1**
5:00 - 6:00	Wine & Cheese Reception

Thursday, November 8

8:00 - 3:00	Registration
8:30 -10:00	**Session 2**
10:00 -10:15	Break
10:15 -11:45	**Session 3**
11:45 - 1:30	**Lunch Speaker**
1:30 - 3:00	**Session 4**
3:00 - 3:15	Break
3:15 - 5:00	**Session 5**



Aerospace Applications & Research Directions

November 7

1:15 - 1:45 p.m.

Welcome & Introductions

A. Glen Houston
Director, RICIS, University of Houston-Clear Lake

Thomas M. Stauffer
President, University of Houston-Clear Lake

Daniel A. Nebrig
Associate Director, Johnson Space Center

Robert B. MacDonald
Manager, Research and Education, Information Systems and University Programs Directorate, NASA/JSC

Keynote Address

1:45 - 2:45 p.m.

Introduced by **Sadegh Davari**, *Technical Co-Chair, University of Houston-Clear Lake*

David Weiss, *Software Productivity Consortium*

SYNTHESIS: Integrating Product and Process

Many software developers share the goal of making software products easier to produce by the manufacturing process. The problem is how to organize the production process and the products to eliminate rework. The solution lies in viewing system production as creating different members of a family, rather than creating a new system each time requirements change. Synthesis is a proposed systematic process for rapidly creating different members of a program family. This talk will be a discussion of the goals of Synthesis and the Synthesis process. The technology needed and the feasibility of the approach will be briefly discussed. Finally, the status of current efforts to implement Synthesis methodologies will be given.

Break

2:45 - 3:00 p.m.

Session 1
3:00 - 5:00 p.m.

Lessons Learned in Software Engineering

Chair: **Gary Raines**, *Manager, Avionics Systems Development Office, NASA/JSC*

Report From NASA Ada Users' Group

John R. Cobarruvias

Flight Data Systems Division, NASA/JSC

Software: Where We Are & What Is Required In The Future

Jerry Cohen

Boeing Aerospace and Electronics

Managing Real-Time Ada

Carol A. Mattax

Hughes Aircraft Corp., Radar Systems Group

5:00 - 6:00 p.m. **Wine and Cheese Reception**

November 8

Session 2
8:30 -10:00 a.m.

Software Engineering Activities at SEI

Chair: **Clyde Chittister**, *Program Director of Software Systems, Software Engineering Institute, Carnegie Mellon University*

SERPENT User Interface Management Systems

Reed Little

Senior Member, Technical Staff, SEI

A Task Description Language for Distributed Applications

Dennis Doubleday

Senior Member, Technical Staff, SEI

Break
10:00 - 10:15 a.m.

Session 3
10:15 - 11:45 a.m

Software Reuse

Chair: **Robert Angier**, *IBM Corporation*

Recent Reuse Research Activities

Will Tracz

IBM System Integration Division

Ada Net

John McBride

Planned Solutions

Lunch
11:45 - 12:30 p.m.

Lunch Speaker
12:30 - 1:30 p.m. **Ed Berard,** *Berard Software Engineering Inc.*

Ada in the Software Engineering Marketplace

Session 4
1:30 - 3:00 p.m.

Software Engineering: Issues for Ada's Future

Chair: **Rod L. Bown**, *University of Houston-Clear Lake*

Assessment of Formal Methods for Trustworthy Computer Systems

Susan Gerhart
Microelectronics and Computer Technology Corp. (MCC)

Issues Related to Ada 9X

John McHugh
Computational Logic Inc.

Posix and Ada Integration in SSFP

Robert A. Brown
Charles Draper Laboratory, Inc.

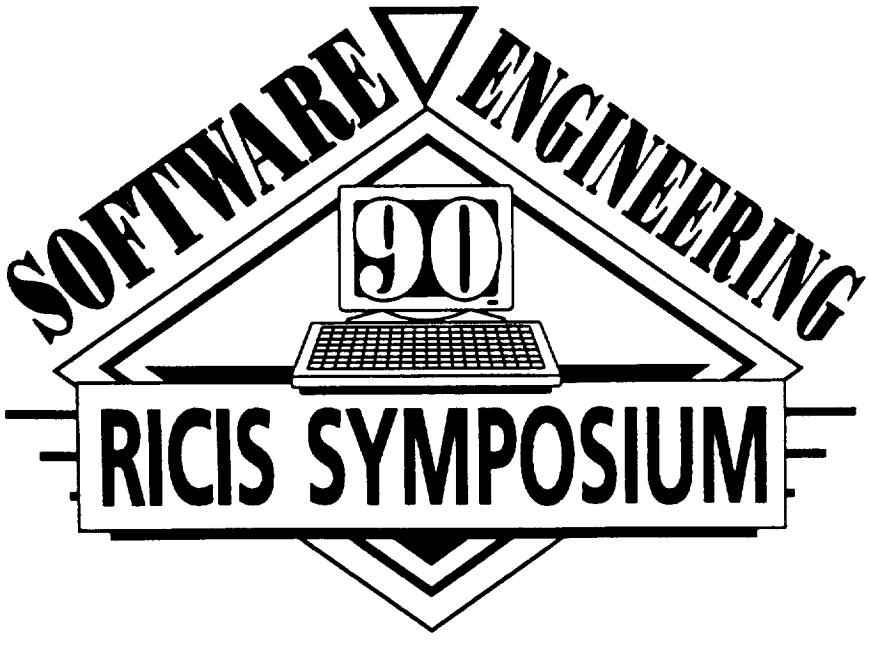
Break
3:00 - 3:15 p.m.

Session 5
3:15 - 5:00 p.m

Ada Run-Time Issues

Chair: **Alan Burns**, *University of York (U.K.)*

Key members of the Ada Run-Time Environment Working Group (ARTEWG) will discuss projections for the next release of the Catalog of Interface Features and Options (CIFO).



N91-22725

Synthesis: Intertwining Product and Process

David M.Weiss

Keynote Address 1:45 -2:45 Nov. 7, 1990

SYNTHESIS: INTEGRATING PRODUCT AND PROCESS

David M. Weiss

Software Productivity Consortium

Abstract

A current trend in manufacturing is to design the manufacturing process and the product concurrently. The goal is to make the product easy to produce by the manufacturing process. Although software is not manufactured, the techniques needed to achieve the goal of easily producible software exist. The problem is how to organize the software production process and the products to eliminate rework. The solution lies in viewing system production as creating different members of a family, rather than creating a new system each time requirements change. Engineers should be able to take advantage of work done in previous developments, rather than restating requirements, reinventing design and code, and redoing testing.

Synthesis is a proposed systematic process for rapidly creating different members of a program family. Family members are described by variations in their requirements. Requirements variations are mapped to variations on a standard design to generate production quality code and documentation. The approach is made feasible by using principles underlying design for change. Synthesis incorporates ideas from rapid prototyping, application generators, and domain analysis. This talk wll be a discussion of the goals of Synthesis and the Synthesis process. The technology needed and the feasibility of the approach will be briefly discussed. Finally, the status of current efforts to implement Synthesis methodologies will be given .

Topics

- Synthesis vision
- Components of a Synthesis process
  - The application engineering and domain engineering processes
- Application Engineering: Building the application
- Domain Engineering: Building the application engineering environment
- Summary

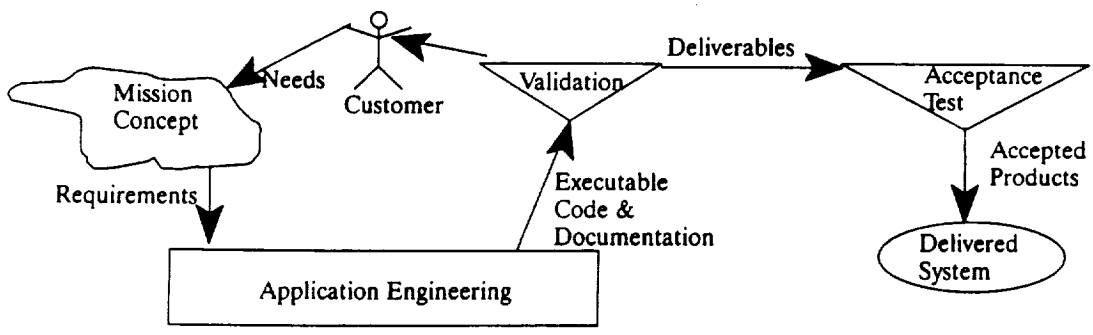
Typical Problems

- Ill–defined and changeable requirements
- Confusion of requirements, design, code
- Transformational barriers
  - Requirements –>Design –> Code
  - Requirements –> Test
- Rediscovery and reinvention

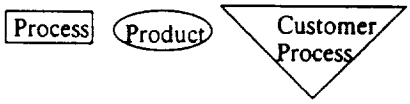
Goals

- Bring the customer into the production loop for validation
- Separate the concerns of requirements determination and validation from design, coding, and testing
- Respond rapidly to changes in requirements
- Rapidly generate deliverable products
  - generate code and documentation
  - achieve high productivity
  - achieve high quality
- Achieve systematic reuse
  - capture and leverage expertise
  - reuse systems

The Synthesis Application Development Process: Idealized



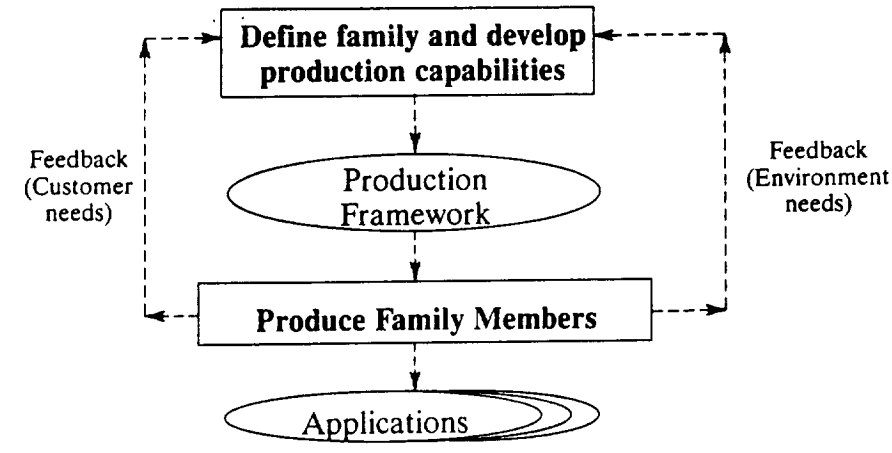
Key:



Approach

- Integrate the development process and the product
  - Design for producibility
  - Concurrent engineering for software
- Reorganize the software development process
  - Evolve a family rather than build single systems
- Develop systematic approach to building flexible application generators
- Use existing technology

A Synthesis Process



Examples of Similar Approaches

- YACC, LEX—parser/compiler applications
- Tedium—flexible application generator (MIS orientation)
- Systematica—generation of CASE tools
- Toshiba software factory
  - Generation of power plant software
  - Standardized design
  - Automated generation of 70%–80% of delivered code
- Spectrum
  - Prototype application engineering environment
  - Standardized design
  - Automated generation of code

Synthesis

Any methodology for constructing software systems as instances of a family of systems

- Process + Methods + Workproducts

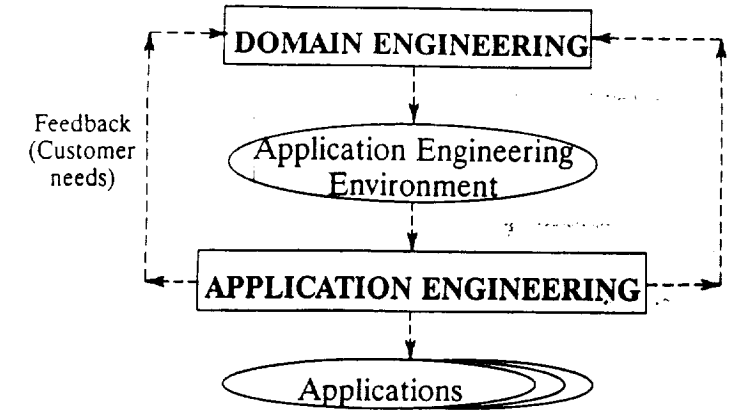
Components of Synthesis

- *Application Engineering*: An iterative process for constructing application systems.
  - Requirements are described by an *application model*
  - Rapidly determine requirements and generate deliverable software
- *Application Engineering Environment*: A framework (automated or manual) that supports a prescribed application engineering process.
  - Rapid prototyping and generation of systems
  - Automated generation of members of program families
  - Automated reuse of systems

Components of Synthesis (continued)

- *Domain Engineering*: A repeatable, iterative process for the design and development of both a family of systems and an application engineering process for the family.
  - Systematic development of product families for member company domains
  - Missing step in current processes
- *Domain Model*: A specification for an application engineering environment.
  - Conceptual framework (Language for specifying application models)
  - Reuse architecture (Standard, adaptable design)
  - System composition mapping (Map from language to reuse architecture)
- *Domain*: (1) A business area
  (2) A family of applications to be created within a business area

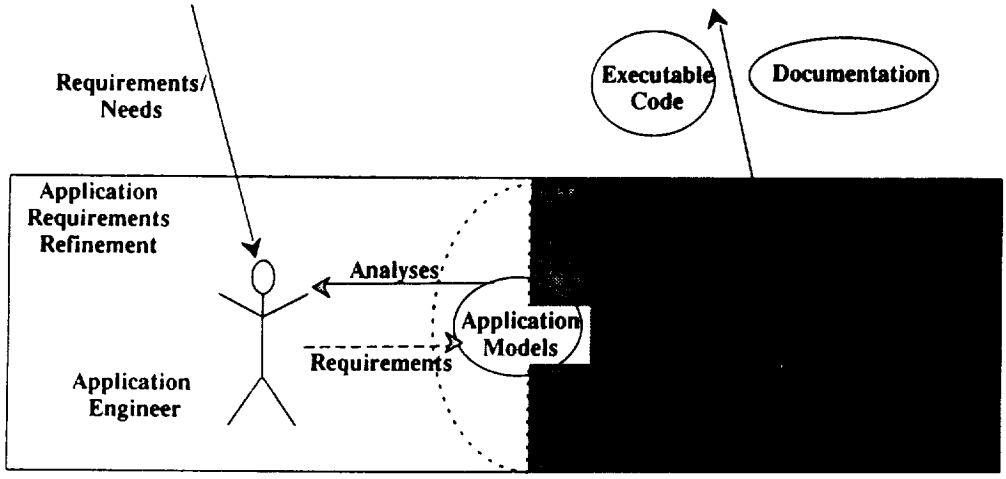
A Synthesis Process



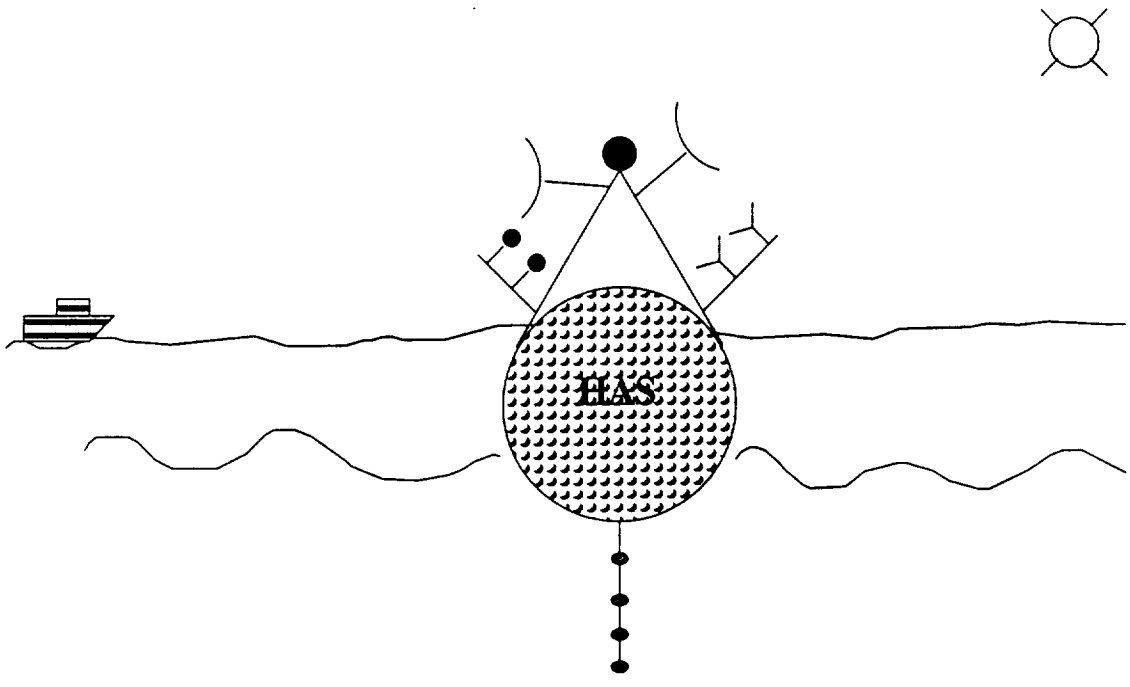
Key:

Product | Process

The Application Engineering Process



Application Engineering for Host-at-Sea (HAS) Buoy Systems



The Host-at-Sea (HAS) Buoy System

HAS Buoys drift at sea and monitor and report on environmental conditions. A typical HAS Buoy:

- Is equipped with a set of *sensors* that monitor *environmental conditions*, such as air and water temperature and wind speed. The value of a particular condition at a given time is determined by averaging sensor readings.
- Can determine its *location,* using the *Omega* or some other navigation system.
- Maintains a *history* of the environmental data it has collected, a history of its location, and a correlation between the two.
- Is equipped to transmit and receive *messages* via radio.
- Periodically transmits *messages* containing current weather information.
- Responds to requests that it receives, via radio, to transmit more detailed reports and to transmit weather *history* information.

The Host-at-Sea (HAS) Buoy System (continued)

HAS Buoys drift at sea and monitor and report on environmental conditions. A typical HAS Buoy:

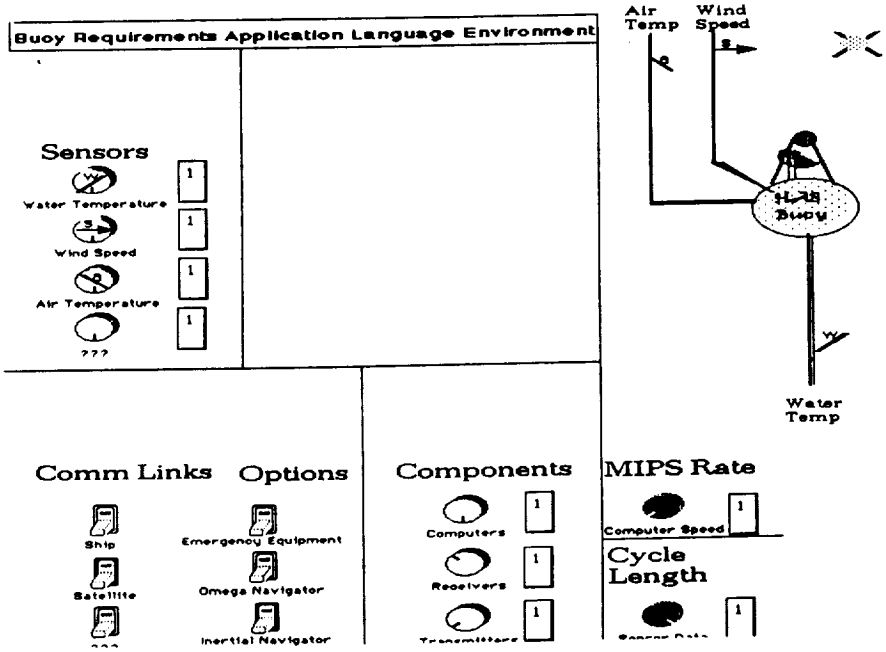
- Is equipped with an *emergency switch*, which, when flipped, causes the buoy to transmit an SOS signal in place of its periodic wind and temperature reports.
- Has a red light that it can turn on to be used in emergency rescue operations.
- Can accept location data from passing ships, via radio messages.
- Performs *built-in tests* (BIT) to determine if its computer and sensors are operating properly.

HAS as a Real Time System

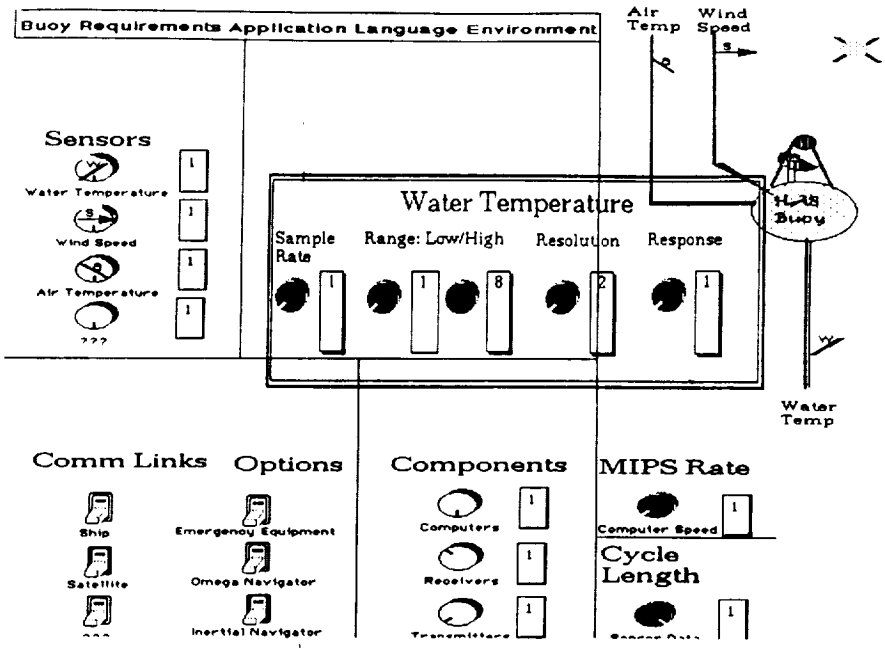
- Meet real-time deadlines for sampling sensors, sending messages
- Perform several functions concurrently, i.e., message broadcasting, data collection
- Reorder processing priorities based on occurrence of external events
- Receive and respond to requests from other systems
- Maintain history data base
- Handle exceptions, such as resource failures, without human intervention
- Missing: human-computer interface

SOFTWARE PRODUCTIVITY CONSORTIUM

The Buoy Application Engineering Environment



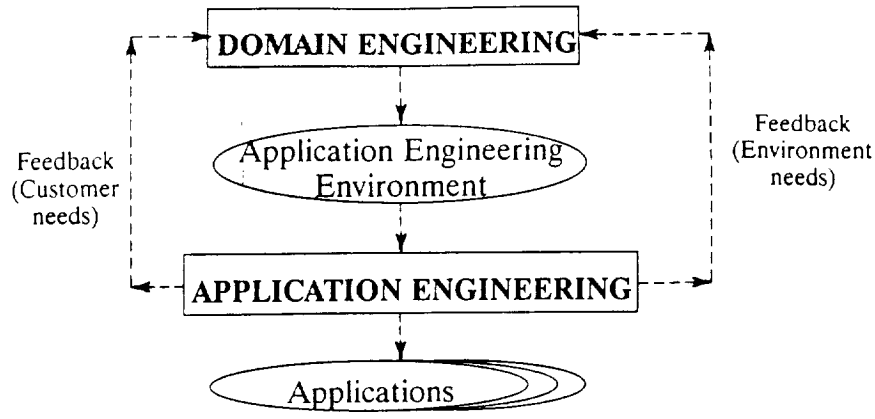
The Buoy Application Engineering Environment



Characteristics of the Buoy Application Engineering Environment

- Focuses on requirements decisions, independent of design and implementation
- Focuses on variabilities used to describe members of the Buoy family
- Generates code and documentation for the application engineer (transparently)
- Simulates Buoy operations in a form meaningful to the customer
- Analyzes consistency, completeness, cost, and performance of the application model
  - -- Do I have enough MIPS to do the job?
  - -- Have I specified unnecessary redundancy?
  - -- How much will it cost?

A Synthesis Process

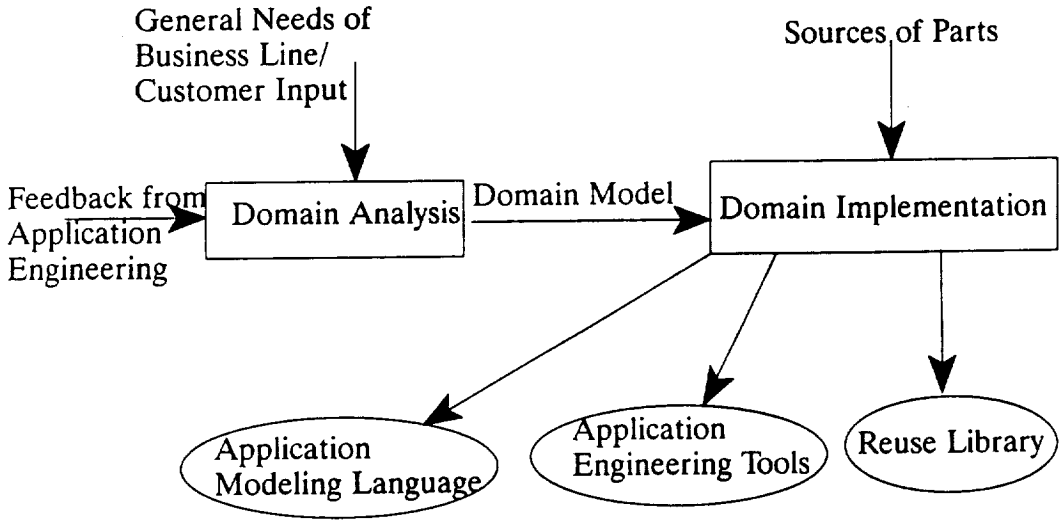


Key:

Product | Process

SOFTWARE PRODUCTIVITY CONSORTIUM

Domain Engineering: Building the Environment



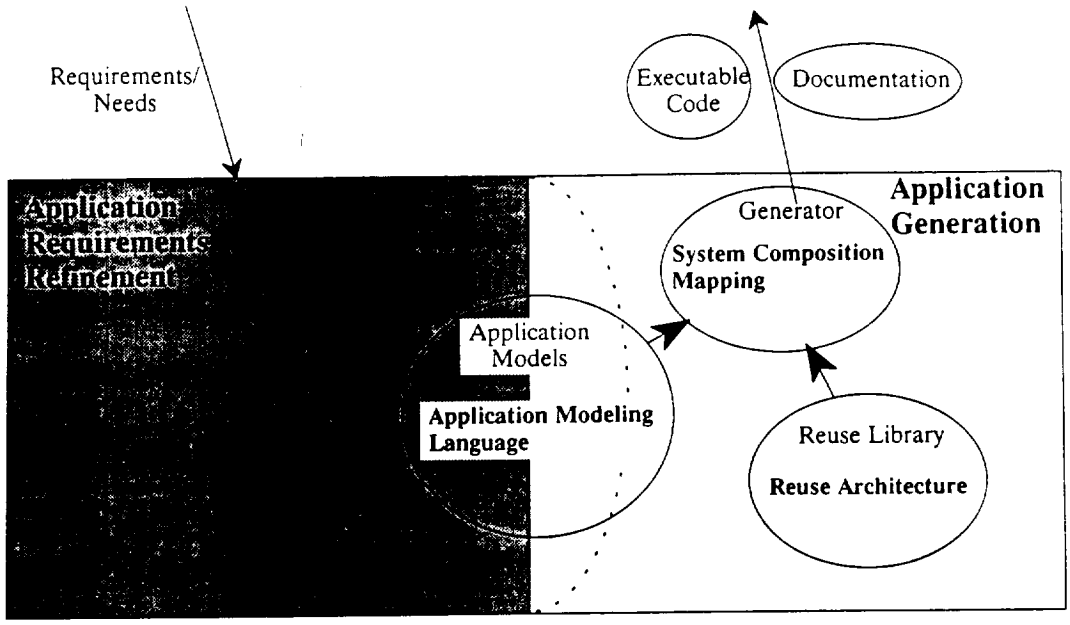
Domain Analysis

A Process For Refining/Creating Specifications For An Application Engineering Environment

The Domain Model

- Conceptual Framework
  - Application Modeling Language: Language for stating requirements
- Reuse Architecture
  - System software architecture organized for ease of reuse through adaptation
  - Specifications for parts for reuse library
- System Composition Mapping
  - Process for selecting and adapting parts for generation of code and documentation
  - Mapping from the modeling language to corresponding entities in the reuse architecture

Application Engineering Using The Domain Model



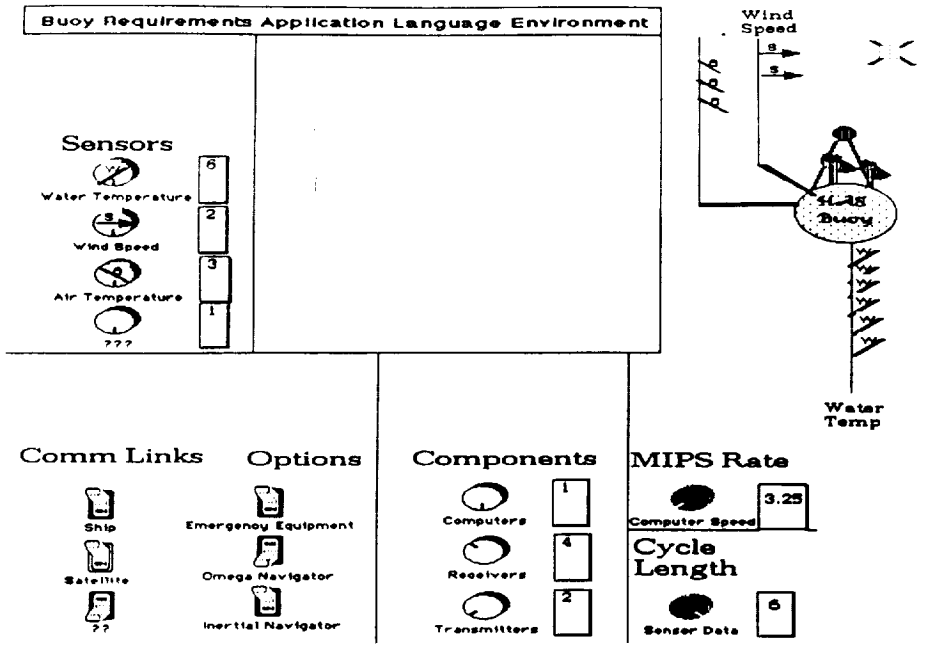
SOFTWARE PRODUCTIVITY CONSORTIUM

Application Modeling Language

Goal: Representation of requirements in application terms

- Support engineering decision making
- Specialize for the domain
- Separate representation, semantics, presentation
- Permit representation of expected variations

Application Modeling Language for The HAS Buoy Domain



Reuse Architecture

Goal: Define mechanically adaptable products for the domain

- Create information hiding class hierarchy to manage changeability and guide design decomposition
  - Divide software components into classes according to:
    1) Relative likelihood of change
    2) Origin of change (functional requirements, environment, software design decisions)
    -- Each class is an abstraction
  - Write specifications for components to be stored in reuse library
- Create process structure (for real-time systems) to support reconfigurability
- Identify dependencies among components to support systematic reuse

The System Composition Mapping

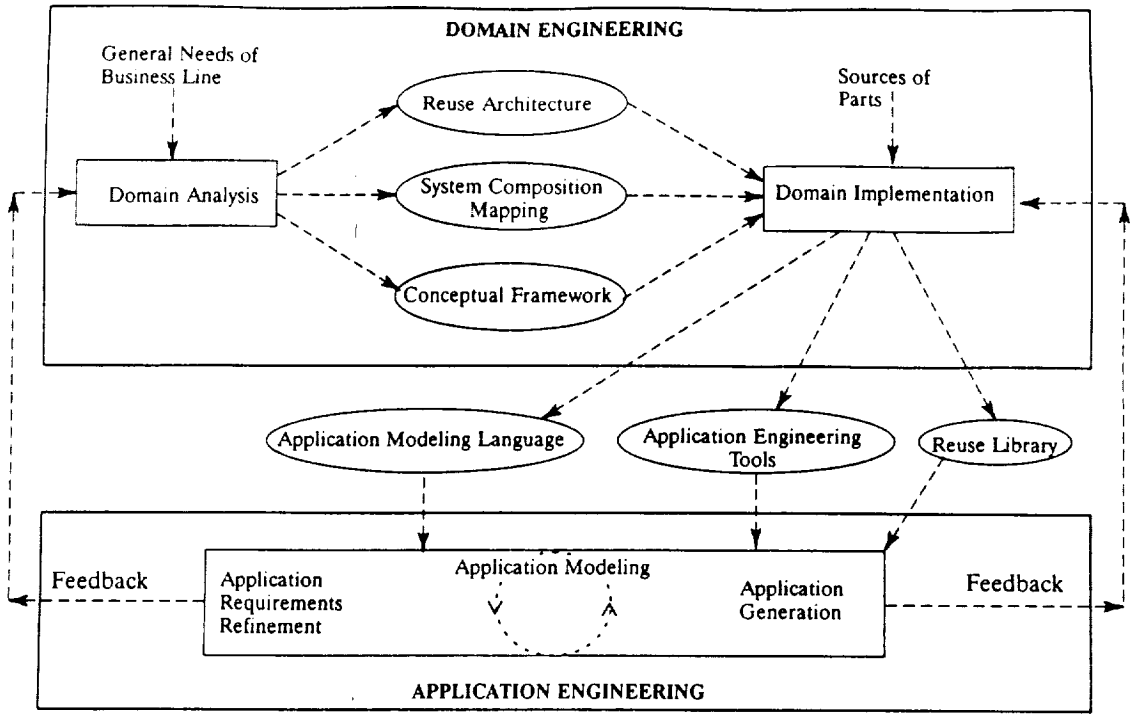
Goal: Select and adapt classes to compose applications that satisfy application models

- For each set of variations described in an application model, select appropriate parts from the reuse library
- Instantiate (Adapt) parts as determined by the application model

Domain Implementation

A Process for Refining/Creating An Application Engineering Environment

Synthesis: Major Products and Processes



SOFTWARE PRODUCTIVITY CONSORTIUM

Synthesis

- **Synthesis**: Any methodology for the construction of software systems as instances of a family of systems having similar descriptions.
- **Synthesis process**: Any systematic process for producing a reuse architecture, application modeling language, and system composition mapping within an application domain.

Key Synthesis Concepts

- ***Families of Systems***

  Domains are formalized as families of systems that share many common features. Software systems are derived as instances of a family,
  not as single unique systems.

- ***Model-Based Specification and Analysis***

  Specify requirements and system-building decisions precisely in an application model suitable for analysis,
  not constantly rework solution–specific representations.

- ***Reuse Architecture Designed for Adaptation***

  Creation and pre–planned reuse of mechanically adaptable subsystems based on engineering decisions,
  not opportunistic search and match with "reusable" parts.

- ***System Composition Mapping***

  Mapping from variations in an application model to adaptations in all deliverables for the implementing subsystems,
  not just tracing to possibly affected components.

Summary

- The technology to improve the software production process exists
- Reorganizing software production to take advantage of the family viewpoint is the key to improvement
  - One organization that concentrates on continually improving production of family members (process oriented)
  - One organization that concentrates on determining requirements for family members (project oriented)
- Similar reorganizations are happening in engineering fields
  - customer involvement
  - shorter time to market
  - more variation across product line

Session 1

Lessons Learned in Software Engineering

Chair: **Gary Raines**, *Manager, Avionics Systems Development Office, NASA/JSC*

Report from NASA Ada User's Group

John R. Cobarruvias
Flight Data Systems Division, NASA/JSC

Paper not available at time of printing.

Software: Where We Are & What is Required in the Future

Jerry Cohen
Boeing Aerospace and Electronics

Paper not available at time of printing.

Managing Real-Time Ada

Carol A. Mattax

Hughes Aircraft Corp., Radar Systems Group

Paper not available at time of printing.

Session 2

Software Engineering Activities at SEI

Chair: **Clyde Chittister**, *Program Director of Software Systems, Software Engineering Institute, Carnegie Mellon University*

Serpent:
A User Interface Management System

Reed Little, Software Engineering Institute, Carnegie Mellon University
Len Bass, Software Engineering Institute, Carnegie Mellon University
Brian Clapper, Naval Air Development Center
Erik Hardy, Software Engineering Institute, Carnegie Mellon University
Rick Kazman, Software Engineering Institute, Carnegie Mellon University
Robert Seacord, Software Engineering Institute, Carnegie Mellon University

Abstract

Prototyping has been shown to ease system specification and implementation, especially in the area of user interfaces. Other prototyping approaches do not allow for the evolution of the prototype into a production system or support maintenance after a system is fielded. This paper presents a set of goals for a modern user interface environment and Serpent, a prototype implementation that achieves these goals.

Introduction

The advent of the modern graphics-oriented workstation is placing increasing emphasis on quality of the user interface. End users are increasingly more demanding that software should be both functional and easy to use. In response, both software and hardware vendors must pay more attention to the user interfaces that accompany their products. However, it is very time consuming and expensive to construct a user interface: in some systems, the user interface development and maintenance cost exceeds 50% of the total software cost [1]. And if history is any indication, this cost is going to get more expensive in the future. The trend is to make these systems more "user friendly", which implies that the user interface needs to be more complex and robust, and thus more costly.

The current state-of-the-practice in the specification, design, implementation, and maintenance of interactive computer systems usually does not give the user interface of the system sufficient consideration. In general, software engineering techniques currently used for the development of systems are usually an ad hoc combination of "tricks" and "tools", with little regard for formalism and standardization. Further, the process of user interface development is labor-intensive. Current user interface development tools and methods inadequately address this problem. In particular, while more and more vendors are providing user interface toolkits and graphics packages, these packages typically require extensive and specific knowledge of a particular toolkit or user interface library. These packages also require the use of conventional, procedural languages such as C and Ada. These languages are not particularly well-suited to user interface specification and implementation, so the user is forced into worrying about low-level syntactic issues.

The Case for Evolutionary Development

One major problem with the software engineering of a user interface is that it is difficult to design a user interface and know a priori (before implementation) if it is "good". In fact, there are generally multiple, and often conflicting, definitions of "good". Some of the criteria used in the definition of "good" are:

1. does the operator "like" it?
2. does it support the mission goal? and

This work was sponsored by the Department of Defense.

3. is it fast enough?

The current methods used to build interactive systems can result in user interfaces that are non-intuitive for the operator to use and sometimes do not perform the necessary functions. Additionally, the user interface is often intertwined with the non-user interface parts of the system, making the task of modification and extension of the user interface during the sustaining engineering phase of the system extremely difficult.

In many respects, the user interface component is no different from the other components of a system. The user interface benefits from the accepted software engineering techniques, such as the determination of the specification of what is to be done before the design of how to do it, etc. However, user interfaces are especially difficult to build, and using a standard sequential method of construction (commonly known as the water-fall method) is not appropriate.

Practice has shown that it is better to use an iterative method, where there is specification, design, implementation, test, evaluation, and a return to specification again [2]. Frequently, there are several iterations of the specification to evaluation path. It is a fact of human nature that it is easier for people to determine what it is that they do not like about a user interface than it is for them to unambiguously specify what they want in a user interface.

Previous User Interface Approaches

Early prototyping efforts were marked by intense coding in traditional programming languages of both the user interface and the underlying application. This approach is cumbersome and error-prone, due to the low-level semantics of these languages. Using this process, changes to the user interface specification may force major changes in the application program. Even though the prototype may have only addressed some limited portion of the overall requirements, there is a natural tendency to use it as a basis for the deliverable product.

Later, specialized prototyping languages were developed, employing specific shorthand notations to generate corresponding function invocations [6]. These languages are usually fairly arcane, not unlike RPG and its successors, in that the user interface designer must be intimately familiar not only with the language, but also with the built-in functions. One of the big drawbacks to this approach is that after the prototype has been built, the user interface must be recoded (using the prototype as requirements) due to the performance and maintenance issues; there is no smooth transition from prototype to product.

With the advent of fourth generation languages and the increased use of computers for management information systems came the concept of rapid prototyping [4]. This approach is marked by the application of database concepts to software development: changing a value in the database causes a resultant change in the presentation. One major advantage over other approaches is that, for each function that can be invoked by the user, there is a corresponding program-callable routine. Once the user interface is specified, the appropriate calls can be made by the application program. However, if the user interface changes, the application program must be changed.

The explosion in workstation capabilities in the last few years has sparked many new ideas about how to use these capabilities for user interface development [9, 10, 8, 3, 5, 7], leading to a multitude of tools and environments, such as Prototyper, XVT, UIL, Granite, Autocode, and MIKE. However, each tool is marked by the use of a specific language and/or interactive tools tailored to the capabilities of a particular

platform and/or to the specific user interface toolkit supported. Application support in these packages usually takes the form of a fixed set of functions that can be invoked as necessary by the application, or a set of functions that are dynamically generated by the prototyping tool to implement the user interface. Again, if the user interface changes, the application must be changed to invoke the new functions.

Finally, user interface technology is evolving rapidly. Today's leading edge data presentation theory becomes tomorrow's commonplace toolkit, giving way to some previously unimagined technology. None of the above approaches adequately provides for the effective integration and use of new toolkits.

Goals of a Modern User Interface Environment

In 1987 the Software Engineering Institute started the User Interface Project to address perceived problems in user interface development and to assist the transition of user interface design and development technology into practice. Out of this effort arose a set of goals for the next generation of user interface environments:

1. In any computer system, there should be a true separation of concerns between the application and the user interface. This is simply the concept of modularity: the application should not try to perform the functions of a user interface, and vice versa. One should be able to develop the application independently of the user interface, in a language appropriate to the semantics of the application; similarly, user interface development should be independent of the application.

2. The user interface specification, design, and implementation should be simple and straightforward; prototyping should be fairly easy using the mechanisms provided by the environment. Non-programmers should be able to perform these activities with a minimum of training. The mechanisms used to perform these activities should not have to change, even though the user interface style or underlying user interface toolkit may change.

3. It must be possible to prototype the interface and functionality of a system without an application. The user interface support mechanisms should be sufficiently rich to support reasonably sophisticated prototypes. As the prototype matures, facilities should be provided to add an application, in pieces or all at once, thus providing evolutionary development.

4. Existing systems should be able to take advantage of new toolkits as they become available, without affecting the application portion of the system. The mechanisms for incorporating these new toolkits should be relatively simple.

5. Performance, when the environment is used strictly as a prototyping vehicle, should be reasonable, although special performance considerations may have to be made when used in production.

User Interface Management System (UIMS)

One tool which meets the above goals is the UIMS. A UIMS is generally composed of four parts:

1. a dialogue, which specifies how information is to be presented to the operator and how to respond to operator commands,

2. a dialogue manager, which is responsible for interpreting the dialogue during the execution of the system,

3. a realization component, which is responsible for the actual physical interface between the operator and the system, and

4. the application, which is responsible for all the non-user interface functionality of the system.

A UIMS can be thought of as software oriented "erector set" that is tailored for the development of user

interfaces. The UIMS provides an environment where it is very easy and fast to change the form and function of a user interface. This provides the ability to quickly prototype and change the user interface during the system specification, design, and implementation phases. A UIMS also enforces the separation of what is to be presented to the operator from the how it is presented. This provides a very convenient mechanism for the decoupling of the user interface from the rest of the system, which makes maintenance and the changes to the user interface easier.

Serpent

Starting with the above goals, the User Interface Project developed a user interface environment known as Serpent. Serpent is a UIMS, using the standard Seeheim model [11], that supports the development and execution of the user interface of a software system. Serpent supports incremental development of the user interface from the prototyping phase through production to maintenance. Serpent encourages the separation of concerns between the user interface and the functional portions of an application. Serpent is easily extended to support multiple toolkits.

Architecture

Figure 1 shows the overall architecture for Serpent. The architecture is intended to encourage the proper separation of functionality between the application and the user interface portions of a software system. The three different layers of the architecture provide differing levels of control over user input and system output. The presentation layer is responsible for layout and device issues. The dialogue layer specifies the presentation of application information and user interactions. The application layer provides the actual system functionality.

The *presentation layer* controls the end-user interactions and generates low-level feedback. This layer consists of various toolkits that have been incorporated into Serpent. A standard interface has been defined which simplifies adding new toolkits. Each toolkit defines a collection of interaction *objects* visible to the end user.

The *dialogue layer* specifies the user interface and provides the mapping between the presentation and application layers. The dialogue layer determines which information is currently available to the end user and specifies the form that the presentation will take, as previously defined by the dialogue specifier (the individual responsible for creating the user interface specification, or *dialogue*). The dialogue layer acts like a traffic manager for communication between application and toolkits. The presentation level manages the presentation; the dialogue layer tells the presentation what to do. For example, the presentation layer manages a button that the end user can select; the dialogue layer informs the presentation layer of the position and contents of the button and will act when the button is selected.

The *application layer* performs those functions that are specific to the application. Since the other two layers are designed to take care of all the user interface details, the application can be written to be presentation-independent; there should be no dependency in the application on a specific toolkit.

The data that is passed between different layers is known as *shared data*. Data passed between an application and the dialogue layer is referred to as *application shared data*, while data passed between a toolkit and the dialogue layer is called *toolkit shared data*. A *shared data definition* provides the format of the data.

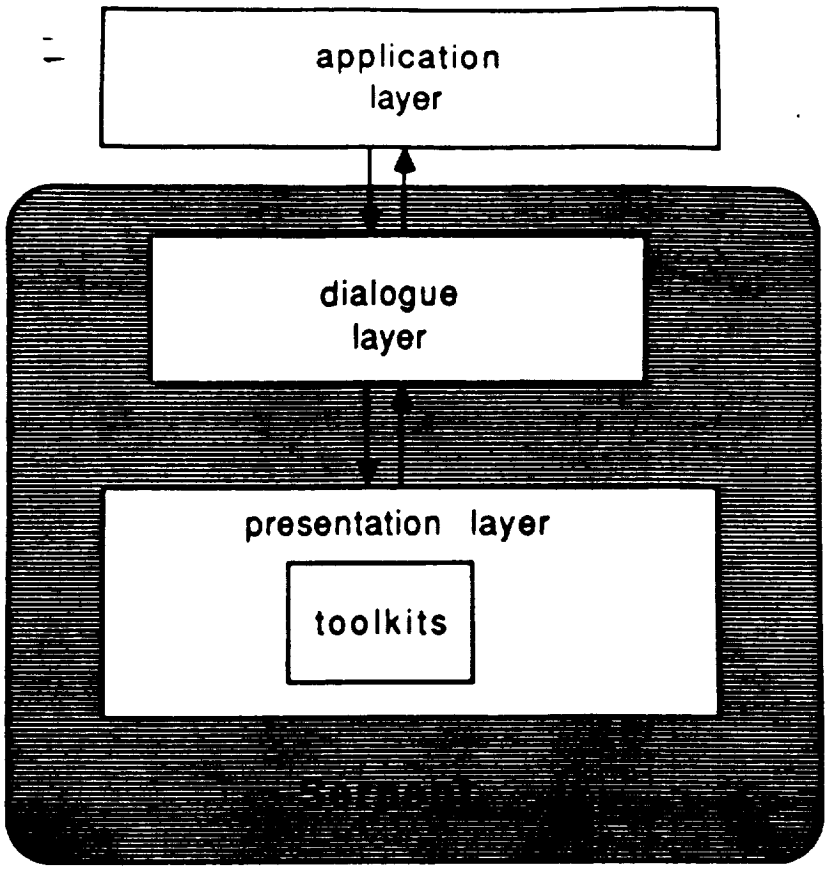


Figure 1: Serpent Architecture

Slang

In Serpent, user interface dialogues are specified in a special-purpose language called Slang. Slang provides a mechanism for defining the presentation of information to, as well as interactions with, the end user. A Slang program defines and enumerates a collection of interaction objects and allowable actions to be available to the end user. Slang provides variables for intermediate storage and manipulation, along with a full complement of primitive arithmetic operations.

The interaction objects available to the dialogue writer are defined by the toolkit. Each toolkit defines a set of primitive objects that may be used in a dialogue. Each object has a collection of *attributes* that define its presentation and a collection of *methods* that determine how the end user can interact with that object.

In Slang, dependencies between items are automatically enforced. That is, suppose variable V depends on the value of some object attribute A. If A changes (perhaps due to some end user action), the value of V is re-evaluated automatically. This important and powerful feature allows the dialogue writer to build complex, interdependent interaction objects simply by referencing data items; the dependencies are automatically determined and enforced by the the Serpent system.

Slang also allows a dialogue writer to group arbitrary objects into logical collections called *view controllers* that may be created or destroyed as a unit. Specifying a view controller in Slang defines a view controller *template*; each template has a *creation condition* that defines when an *instance* of the template should come into existence. The existence of a view controller instance and its child objects can be controlled by the values of Slang variables or by the creation, modification, or destruction of application data. When a view controller instance's creation condition is no longer valid, it and its associated objects are destroyed. Multiple instances of a view controller template may exist at any time. A view controller serves two main purposes:

1. It maps specific application data onto display objects with which the end user can interact.
2. It controls the existence of a series of related objects.

Application Program Interface (API)

From the application developer's perspective, Serpent behaves like a database management system. Shared data is a "common" database manipulated by the application, the presentation layer (usually in response to end-user actions), or the dialogue layer (in response to actions within the dialogue).

The application can add, modify, or delete shared data. Information provided to Serpent by the application is available for presentation to the end user. The application has no direct interface to the presentation layer and therefore cannot affect how data is presented to the user. When end user actions cause the dialogue to change the application shared database, the application is automatically informed. In this sense, the application views Serpent as an *active database manager.*

Saddle

The type and structure of data that is maintained in the shared database is specified in a *shared data definition* file, defined in a language called Saddle. This data definition corresponds to the database concept of *schema.* A shared data definition file is created once for each application and once for each toolkit that is integrated into Serpent.

The shared data definition file is processed to produce a language-specific description of shared data. Processors currently exist for Ada and C. If the application is written in C, the processor will generate structure definitions that can be included into the application program. If the application is written in Ada, the processor will generate package specifications.

Input/Output Toolkit Integration

Given that Serpent manipulates objects, the toolkits that are integrated most easily are those that are object-oriented. The successful integration of object-oriented graphics systems and their associated toolkits has been a major proof of Serpent's ability to separate presentation concerns from application concerns.

The process of integrating a toolkit into Serpent is conceptually simple. It can be logically divided into three parts:

1. the objects with which the end user will interact must be determined, along with their behavior;
2. these objects must be defined to Serpent through the use of Saddle; and
3. "glue" code must be written to allow the toolkit to communicate with the dialogue manager, through Serpent's shared database facility.

If a toolkit already has an object orientation, then the first and third integration steps are usually

straightforward. If it does not, then a set of objects and their attributes which conform to the Serpent model must be built on top of the toolkit.

Toolkit integration presents other practical difficulties. The integrator has to decide how much of the underlying toolkit to expose to a dialogue writer, whether to change any of the default behavior of the system, and whether to make the system more robust by, for instance, performing error checking that the toolkit does not handle.

The User Interface Development Process Using Serpent

Slang was designed explicitly for user interface specification. A Slang dialogue writer is not burdened with the technical and procedural details necessary to manipulate specific interaction objects; those details are hidden in the presentation layer. The dialogue writer merely specifies the objects that make up the user interface and indicates how they relate to one another and to the end user; the Serpent runtime system manages the interaction objects. The dialogue specifier needs to be familiar with the characteristics of various objects, such as knowing that an Athena widget set label widget appears as a rectangle on the screen; however, the specifier does not need to know how to tell the Athena toolkit library how to display such a widget.

Slang dialogues can be executed without of an application, allowing the building, testing, and refinement a prototype before designing and implementing the rest of a system. Often, however, a prototype requires the existence of some application functionality, if only to initialize display values. Slang's rich set of primitive operations allow the user interface designer to "mock up" application operations in the prototype dialogue. Once the prototype has been refined, the simulated application behavior is removed from the dialogue and the real application is added.

A Simple Example

Perhaps the best way to illustrate the simplicity of prototyping with Slang is by example. Figure 2 shows the screen display for a counter demonstration, using the X Toolkit Athena widget set. The box labeled "PRESS" is a command widget that can be selected by the user via a mouse. The box above the command widget is a label widget containing the current value of the counter. When the user selects the button labeled "PRESS", the value in the label widget is incremented by 1.

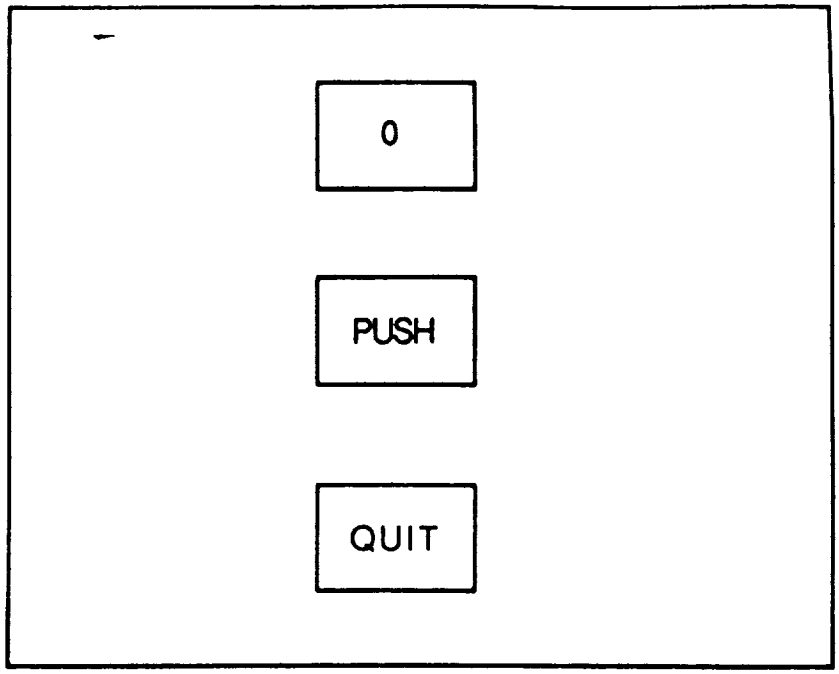


Figure 2: A Simple Example

In Slang this example is implemented as follows:

```
VARIABLES:
   counter: 0;

OBJECTS:
   /*
      width, height, vert_distance, and horiz_distance
      are all specified in pixels
   */
   background: form_widget
   {ATTRIBUTES:
         width:  640;
         height: 645;
   }

   display:  label_widget
   {ATTRIBUTES:
         parent:          background;
         width:           60;
         height:          40;
         vert_distance:   150;  /* from upper left of parent */
         horiz_distance:  310;  /* from upper left of parent */
         label_text:      counter;
   }

   push_button:  command_widget
   {ATTRIBUTES:
         parent:          background;
         width:           60;
         height:          40;
         vert_distance:   250;  /* from upper left of parent */
         horiz_distance:  310;  /* from upper left of parent */
         label_text:      "PRESS";

      METHODS:
         notify:
            {counter := counter + 1;
            }
    }
```

The **background** object provides a form on which to locate the other objects. The **display** object defines the label widget containing the current value of the counter; note that the **label_text** field, which controls what is actually displayed in the form widget, is dependent on the value of the global variable **counter**. When the value of the variable changes, all items that depend on it are re-evaluated. Put more simply, if **counter** changes, the text displayed in the **display** object will change automatically.

The **push_button** object defines the command widget that the end user will select in order to increment the value displayed on the screen. When the user selects the button, the presentation layer captures the event and communicates it to the dialogue via a *notify* method, causing the associated code snippet to be executed. In this case the **counter** variable is incremented, which in turn causes the label in the **display** object to be changed.

Dependencies and type conversions are managed automatically by the Serpent runtime system, allowing the dialogue writer to focus on user interface issues, rather than syntactic details. For example,

the **counter** variable is an integer; the **label_text** attribute of the **display** object is a string. Slang converts the **counter** value to a string before assigning it to the the **label_text** attribute; the dialogue writer merely needs to specify the dependence between the variable and the attribute. Further, the attributes for every interaction object take reasonable defaults, so the dialogue writer does not need to specify a value for every possible characteristic of an object.

In short, Slang is designed to minimize the amount of information the dialogue writer needs to specify in order to manipulate interaction objects.

Status

The initial implementation of Serpent was done under ULTRIX 2.2 on DEC microVAX II and III workstations. Serpent was also easily ported to run under SUNOS 3.5 or higher on SUN2 and SUN3 workstations and DECStation 3100 & 5000 platforms . We expect porting to similar UNIX platforms to be relatively straightforward.

Applications can be written in either C or Ada, and simple mechanisms exist to extend Serpent to support other high level languages. Serpent was implemented predominantly in C, with additional support software written as shell scripts.

Currently, two different interfaces to X Window System toolkits have been written for Serpent: one implements a subset of the Athena widget set and the other implements the Motif widget set. In addition, Lockheed's Softcopy Map Display System has been integrated.

An interactive What-You-See-Is-What-You-Get (WYSIWYG) graphical editor that hides most of the details of the user interface specification is available. The editor provides for fast feedback, so that the entire application system need not be executed, or even exist, to begin to "get a feel" for the interface.

Serpent is available from the Software Engineering Institute and MIT through anonymous ftp. It is also contained in the X11R4 contrib release from MIT.

Conclusions

As a result of our experiences in developing user interfaces with Serpent, we have concluded that Serpent offers the following advantages over other user interface development approaches:

1. The active database model for applications allows the true separation of application issues from user interface issues, thus ensuring modularity. Application writers are also free from the syntactic drudgery inherent in programming large, complex toolkits.
2. The constraint mechanisms implemented via automatic dependency updates ensure that all participants (application, dialogue manager, and toolkit) are synchronized in terms of the state of the system.
3. Serpent's language-independent interface definition and inter-process communication mechanisms help in achieving modularity. Application developers are not constrained to work in a single language.
4. Serpent's toolkit integration support reduces the integration process to a series of concise, well understood steps. Once a particular toolkit is integrated, its objects are available for use in any dialogue.
5. Due to Serpent's inherent separation of concerns, system developers can experiment with different user interface styles, and even different toolkits, without changing either the application code or the API. This also provides for the injection of new toolkits and user interface paradigms into an existing system, while minimizing the system portions which are

affected.

Serpent has achieved the goals of a modern user interface environment set forth earlier. The user interface specification mechanisms are simple and direct; changes in the user interface are made easily, without changing the application. The application program interface is simple and easy to use and enforces a true separation between the application and the user interface portions of the system. Prototyping is accomplished rapidly, with reasonable provision for application functionality simulation. Serpent's toolkit integration mechanisms allow a new toolkit to be incorporated into Serpent easily without affecting the application. Finally, Serpent is itself a prototype, implementing the goals listed above. Even so, performance is quite reasonable, and we are continually making improvements, although we would not yet recommend it for time-critical production environments.

References

[1] Boehm, Barry W.
A Spiral Model of Software Development and Enhancement.
Computer 21(5), May, 1988.

[2] Boehm, Barry W.
Improving Software Productivity.
Computer 20(9), September, 1987.

[3] Colborn, Kate.
OSF Determines User Interface; Choices Could Affect the Development of Applications Software.
EDN, December, 1988.

[4] Fisher, Gary E.
Application Software Prototyping and Fourth Generation Languages.
Technical Report, National Bureau of Standards, May, 1987.

[5] Foley, James, et al.
Defining Interfaces at a High Level of Abstraction.
IEEE Software, January, 1989.

[6] Hanner, Mark Allen.
Gambling on Window Systems.
UNIX Review, December, 1988.

[7] Kasik, David J., et al.
Reflections on Using a UIMS for Complex Applications.
IEEE Software, January, 1989.

[8] Kolodziej, Stan.
User Interface Management Systems.
Computerworld, July 8, 1987.

[9] Myers, Brad A.
Tools for Creating User Interfaces: An Introduction and Survey.
Technical Report CMU-CS-88-107, Carnegie Mellon University, 1988.

[10] Myers, Brad A.
The Garnet User Interface Development Environment: a Proposal.
Technical Report CMU-CS-88-153, Carnegie Mellon University, 1988.

[11] Pfaff, G. (Ed.).
User Interface Management Systems.
Springer-Verlag, Berlin, 1985.

N91-22727

Prototyping Distributed Simulation Networks

Dennis L. Doubleday
Software Engineering Institute

Prototyping Distributed Simulation Networks

Dennis L. Doubleday
Software Engineering Institute

Abstract

Durra is a declarative language designed to support application-level programming. In this paper we illustrate the use of Durra to describe a simple distributed application: a simulation of a collection of networked vehicle simulators. We show how the language is used to describe the application, its components and structure, and how the runtime executive provides for the execution of the application.

1. Programming at the Application-Level

Many distributed applications consist of large-grained tasks or programs, instantiated as processes, running on possibly separate processors and communicating with each other by sending messages of different types.

Since the patterns of communication between the processes can vary over time and the speeds of the individual processors can differ widely, the developers may need explicit control over the allocation of processors to processes in order to meet performance or reliability requirements. Processors are not the only critical resource. The resources that must be allocated also include communication links and message queues. We call this network of various processor types, links, and queues a *heterogeneous machine.*

Currently, users of a heterogeneous machine network follow the same pattern of program development as users of conventional processors: Programmers write individual tasks as separate programs, in the different programming languages (e.g., C, Lisp, Ada) supported by the processors, and then hard code the allocation of resources to their application by explicitly assigning specific programs to run on specific processors at specific times. This coupling between the component programs and the built-in knowledge about the structure of the application and the allocation of resources often prevents the reuse of the programs in other applications or environments. Modification of the application during development is often expensive, time-consuming, and error-prone. The problem is compounded if the application must be modified while running in order to deal with faults or mode changes. We claim that developing distributed applications for a heterogeneous machine is qualitatively different from developing programs for conventional processors. It requires different kinds of languages, tools, runtime support, and methodologies. In this paper we address some of these issues by presenting a language, Durra. We briefly describe the language and its distributed runtime support environment and then present, as an example distributed application, a simple simulation of a network of vehicle simulators.

The rest of this paper is organized as follows. Section 2 briefly describes the Durra language and runtime environment. Section 3 discusses the problem we are attempting to address in the realm of

This work is sponsored by the U.S. Department of Defense. The views and conclusions contained in this document are solely those of the author(s) and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

networked simulation devices. Section 4 describes the work we have done to date toward that end.

2. Introduction to Durra

Durra [2] is a language designed to support the development of distributed, large-grained concurrent applications running on heterogeneous machine networks. A Durra application description consists of a set of *task descriptions* and *type declarations* that prescribe a way to manage the resources of the network. The application description describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

2.1. The Durra Language

Task descriptions are the building blocks for applications. A task description includes the following information (Figure 1): (1) its interface to other tasks (**ports**); (2) its **attributes**; (3) its functional and timing **behavior**; and (4) its internal **structure**, thereby allowing for hierarchical task descriptions.

```
task task-name
  ports                          -- Used for communication between a process and a queue
    port-declarations

  attributes                     -- Used to specify miscellaneous properties of the task
    attribute-value-pairs

  behavior                        -- Used to specify task functional and timing behavior
    functional specification
    timing specification

  structure                    -- A graph describing the internal structure of the task
    process-declarations             --Declaration of instances of internal subtasks
    bind-declarations              -- Mapping of internal ports to this task's ports
    queue-declarations                   -- Means of communication between processes
    reconfiguration-statements             -- Dynamic modifications to the structure
end task-name
```

Figure 1: A Template for Task Descriptions

The interface information declares the ports of the processes instantiated from the task. A port declaration specifies the direction and type of data moving through the port. An **in** port takes input data from a queue; an **out** port deposits data into a queue:

```
ports
    in1: in heads;
    out1, out2: out tails;
```

The attribute information specifies miscellaneous properties of a task. Attributes are a means of indicating pragmas or hints to the compiler and/or runtime executive. In a task description, the developer of the task lists the actual value of a property; in a task selection, the user of a task lists the desired value of the property. Example attributes include author, version number, programming language, file name, and processor type:

```
attributes
   author = "jmw";
   implementation = "program_name";
   Queue_Size = 25;
```

The behavioral information specifies functional and timing properties of the task. The functional information part of a task description consists of a pre-condition on what is required to be true of the data coming through the input ports, and a post-condition on what is guaranteed to be true of the data going out through the output ports. The timing expression describes the behavior of the task in terms of the operations it performs on its input and output ports. For additional information about the syntax and semantics of the functional and timing behavior description, see the Durra reference manual [1].

The structural information defines a process-queue graph and possible dynamic reconfiguration of the graph.

A process declaration of the form

process_name : **task** *task_selection*

creates a process as an instance of the specified task. Since a given task (e.g., convolution) might have a number of different implementations that differ along different dimensions such as algorithm used, code version, performance, or processor type, the task selection in a process declaration specifies the desirable features of a suitable implementation. The presence of task selections within task descriptions provides direct linguistic support for hierarchically structured tasks.

A queue declaration of the form

queue_name [*queue_size*]: *port_name_1* > *data_transformation* > *port_name_2*

creates a queue through which data flow from an output port of a process (port_name_1) into the input port of another process (port_name_2). Data transformations are operations applied to data coming from a source port before they are delivered to a destination port.

A port binding of the form

task_port = *process_port*

maps a port on an internal process to a port defining the external interface of a compound task.

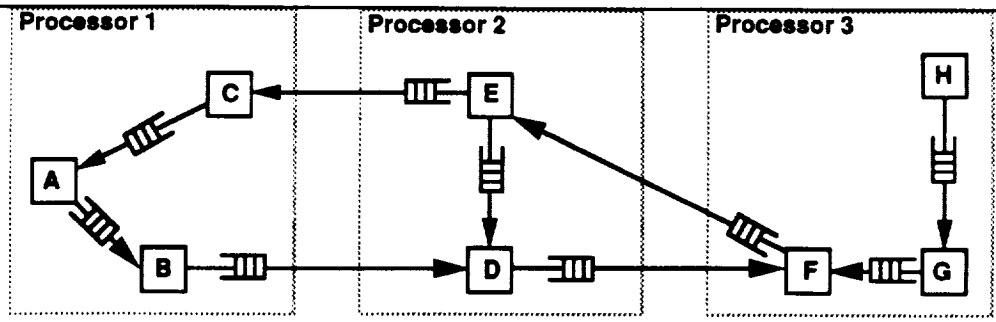
A reconfiguration statement of the form

```
if condition then
    remove   process-names
    process  process-declarations
    queues   queue-declarations
end if;
```

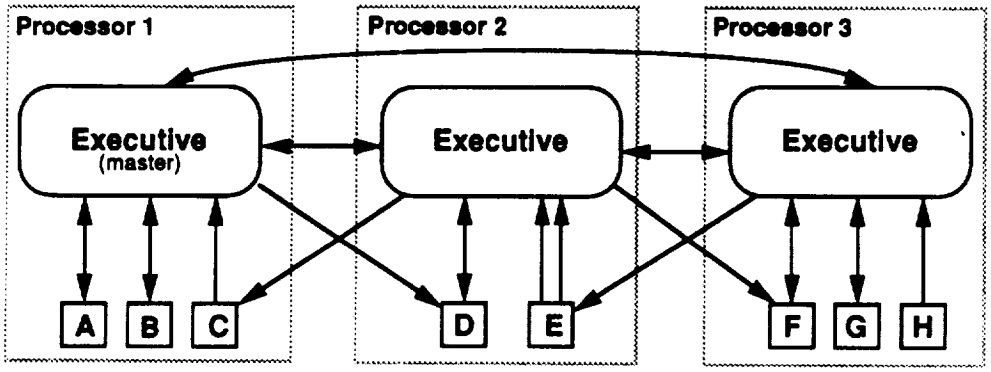
is a directive to the executive. It is used to specify changes in the current structure of the application (i.e., process-queue graph) and the conditions under which these changes take effect. Typically, a number of existing processes and queues are replaced by new processes and queues, which are then connected to the remainder of the original graph. The reconfiguration predicate is a Boolean expression involving time values, queue sizes, and other information available to the executive at runtime.

2.2. The Durra Runtime Environment

There are two classes of active components in the Durra runtime environment: the application processes and the Durra executives. As shown in Figure 2, an instance of the executive runs on each processor while the processes are distributed across the processors in the system.



a -- Process Graph with Processor Allocation



b -- Actual Communication Patterns

Figure 2: The Durra Runtime Environment

The executives interpret the resource allocation commands produced by the Durra compiler, monitor reconfiguration conditions, and implement the necessary changes in the application structure.

The component processes making up a Durra application are instances of independent tasks (programs) that can be written in any language for which a Durra interface has been provided (currently, there are Durra interfaces for both C and Ada). The Durra interface is a collection of procedures that provide communication and control primitives. The component processes use the interface to communicate with the Durra executives and, indirectly, with other application processes. For a more detailed discussion of the Durra runtime environment, see [3].

3. Distributed Simulation Networks

The development of large networks of heterogeneous simulation and training devices often presents problems related to the performance and interconnectivity of the network components. There is a need to evaluate various design alternatives before committing to a specific implementation. Problems arise in several areas:

- Multiple protocols. Cooperating devices are often written using different communication protocols because they rely on predetermined standards or technologies. When communicating devices use different protocols, it is necessary to translate messages in a way that is transparent to the communicating agents. This message translation consumes time and reduces performance.

- Multiple levels of fidelity. When developing hierarchical networks of simulation and training devices, it is often the case that the time scales (i.e., granularity), amount of data, and level of detail in the data are not compatible between levels or devices. Thus, there is a need to filter (i.e., reduce) data moving up in the hierarchy and to pad (i.e., augment) data moving down the hierarchy. This is a different type of 'translation' from the protocol translation described above. The translating programs in this case need to have a thorough understanding of the application to compensate for the mismatch in the levels of detail.
- Multiple technologies. When connecting devices that use different hardware technology, the developers of the distributed application need to compensate for differences in speed, performance, and fault-tolerance requirements.

This collection of problems is just an illustration of the issues that must be addressed by the developers before implementing the network. A useful technique is to develop prototypes using emulators of the component software and hardware devices. The emulators are easier to implement than the real devices and can more easily be reconfigured into alternative structures. Experiments can be conducted under various load conditions and measurements of performance can be derived from these experiments.

4. Using Durra to Prototype Simulation Networks

We are using Durra to develop a tool for testing and evaluating various network configurations. We are implementing the tool as a distributed application consisting of clusters of emulators. These emulators are responsible for interpreting specifications of hypothetical application tasks. We use the Durra language to describe the various components of the system, their ports and message queues, and the types of messages exchanged between components. We use the Durra runtime environment to execute the application and perform dynamic reconfigurations of the application, to emulate mode changes, and to evaluate their impact on performance.

The final version of our tool will include at least four types of emulators:

1. **Generic simulation device emulators**: These programs will mimic the I/O behavior of a generic networked simulation device. Scripts specifying the behavior of the emulated device(s) will be developed. Differences in I/0 behavior between different types of simulation devices can be emulated through variations in these scripts. The initial scripts consist simply of position updates and timing instructions. Eventually they should be more representative of actual networked simulation sessions; this could be accomplished by adaptation of I/O logs of an actual simulation session.
2. **LAN emulators**: These emulators will model communications delay in the network (e.g., token ring delay). This kind of emulation can likely be accomplished via buffer tasks in the Durra runtime, which would mean that no executable version of these emulators need be developed.
3. **Intelligent gateway emulators**: These programs will model the effect of various message-filtering and protocol translation techniques on the networked simulation's use of processor and communications resources.
4. **Console emulator**: This program will provide an interactive user interface to the simulation environment, allowing the experimenter to change emulation parameters, inject faults, and collect data.

4.1. Example: A Simple Network Specification

In this section we present a Durra specification of a simple network of simulators. In this example, we instantiate a user console and two LAN emulators, each consisting of a group of three simulators and one gateway process. The reader should note that there is nothing special about this configuration--another version consisting of some other grouping could just as easily have been constructed from the same primitive building blocks.

The following is the Durra description of the message type used for communications between the application components. The message type description is purposely a very general one. A generic description of the message type allows us in the actual implementation of the type to use a variant record to represent both simulator position updates and command messages and easily combine both types of messages in a single data stream.

```
type message is array of byte;
```

At the lowest level of the structure we have the descriptions of the primitive tasks, the *simulator*, the *gateway*, and the *console*. The *simulator* task has one output port, through which it emits its position updates, and one input port, through which it receives position updates and user commands. The *gateway* task has one input port and two output ports; port *to_wan* sends messages outside the LAN and port *to_lan* distributes remote messages to the simulators in its LAN. The *console* task is the application user's interface to the tool; it accepts a set of user commands and forwards them to the *gateway* task for each LAN in the configuration. The *gateways* may in turn forward those messages to the simulators in their respective LANs if the nature of the command requires it.

```
task simulator
    ports
        in1  : in message;
        out1 : out message;
    attributes
        version = "2";
        implementation = "simulator";
end simulator;

task gateway
    ports
        in1    : in message;
        to_lan : out message;
        to_wan : out message;
    attributes
        version = "2";
        implementation = "gateway";
end gateway;

task console
    ports
        to_lan : out message;
    attributes
        xwindow = "-geom 80x24+0+0 -title CONSOLE";
        implementation = "console";
end console;
```

The Durra task *lan* encapsulates the internal structure of the LAN itself. This instantiation of a LAN includes one *gateway* task and three *simulator* tasks, as well as three built-in Durra buffer tasks. The

buffer tasks implement the routing of message traffic between the component tasks of the LAN. Task *gate_merge* merges local and remote messages intended for the local *gateway*. Task *gate_mb* merges messages from the local simulators and then distributes them to both the *gate_merge* task and the *lan_mb* task. The *lan_mb* task merges those local messages with the remote messages forwarded from the *gateway* and distributes them all to each of the local simulators. Note that, given this structure, each *simulator* will receive its own updates; these can either be ignored by the *simulator* or used as a check to ensure that its own updates are being distributed properly.

```
task lan
  ports
    in1  : in message;
    out1 : out message;
  structure
    process
      gate : task gateway   attributes version = "2"; end gateway;
      sim1, sim2, sim3 :
            task simulator attributes version = "2"; end simulator;
      gate_merge : task merge
                     ports
                       from_lan, from_wan : in message;
                       to_gate            : out message;
                     attributes mode = fifo;
                   end merge;
      gate_mb    : task merge_broadcast
                     ports
                       from1, from2, from3 : in message;
                       to_gate, to_lan     : out message;
                     attributes mode = fifo;
                   end merge_broadcast;
      lan_mb     : task merge_broadcast
                     ports
                       from_gate, from_lan : in message;
                       to1, to2, to3       : out message;
                     attributes mode = fifo;
                   end merge_broadcast;
    queues
      qgate_in[10]    : gate_merge.to_gate >> gate.in1;
      qgate_out[10]   : gate.to_lan        >> lan_mb.from_gate;
      qsim1_in[10]    : lan_mb.to1         >> sim1.in1;
      qsim2_in[10]    : lan_mb.to2         >> sim2.in1;
      qsim3_in[10]    : lan_mb.to3         >> sim3.in1;
      qsim1_out[10]   : sim1.out1          >> gate_mb.from1;
      qsim2_out[10]   : sim2.out1          >> gate_mb.from2;
      qsim3_out[10]   : sim3.out1          >> gate_mb.from3;
      qmb_to_gate[10] : gate_mb.to_gate    >> gate_merge.from_lan;
      qmb_to_lan[10]  : gate_mb.to_lan     >> lan_mb.from_lan;
    bind
      in1       =       gate_merge.from_wan;
      out1      =       gate.to_wan;
end lan;
```

At the highest level of abstraction, the Durra task *internet* provides the view of the application as a console process controlling two connected, but independent, local area networks. These LAN simulators may be distributed to separate physical processors. Figure 3 shows a graphical view of

the structure of the application.

```
task internet
  structure
    process
      lan1: task lan attributes processor = net1; end lan;
      lan2: task lan attributes processor = net2; end lan;
      uc  : task console attributes version = "xterm"; end console;

      uc_b   : task broadcast
                   ports
                       from_uc          : in message;
                       to_lan1, to_lan2 : out message;
                 end broadcast;

     lan1_m, lan2_m :
             task merge
                ports
                    from_uc, from_lan   : in message;
                    to_lan              : out message;
                attributes mode = fifo;
             end merge;

    queues
      quctob     : uc.to_lan      >> uc_b.from_uc;
      qucbto1    : uc_b.to_lan1   >> lan1_m.from_uc;
      qucbto2    : uc_b.to_lan2   >> lan2_m.from_uc;
      q1tom[10]  : lan1.out1      >> lan2_m.from_lan;
      q2tom[10]  : lan2.out1      >> lan1_m.from_lan;
      qmto1[10]  : lan2_m.to_lan >> lan2.in1;
      qmto2[10]  : lan1_m.to_lan >> lan1.in1;
end internet;
```

Only three of the aforementioned Durra tasks, the *simulator*, the *gateway*, and the *console* have actual implementations associated with them. The *lan* task's behavior is defined constructively from the behavior of the *simulator* and the *gateway*, the three buffer tasks (whose behavior is implemented in the Durra executive), and the connections between them all. Similarly, the behavior of the *internet* task derives from the connections between its components, the two instantiations of the *lan* task and the *console*.

5. Conclusions

Application-level programming, as implemented by Durra, separates the structure of an application from its behavior. This separation provides developers with control over the evolution of an application during application development as well as during application execution. During development, an application evolves as the requirements of the application are better understood or as they change.

This evolution takes the form of changes in the application description, modifying task selection templates to retrieve alternative task implementations from the library, and connecting these implementations in different ways to reflect alternative designs. During execution, an application evolves through mode changes or in response to faults. This evolution takes the form of conditional,

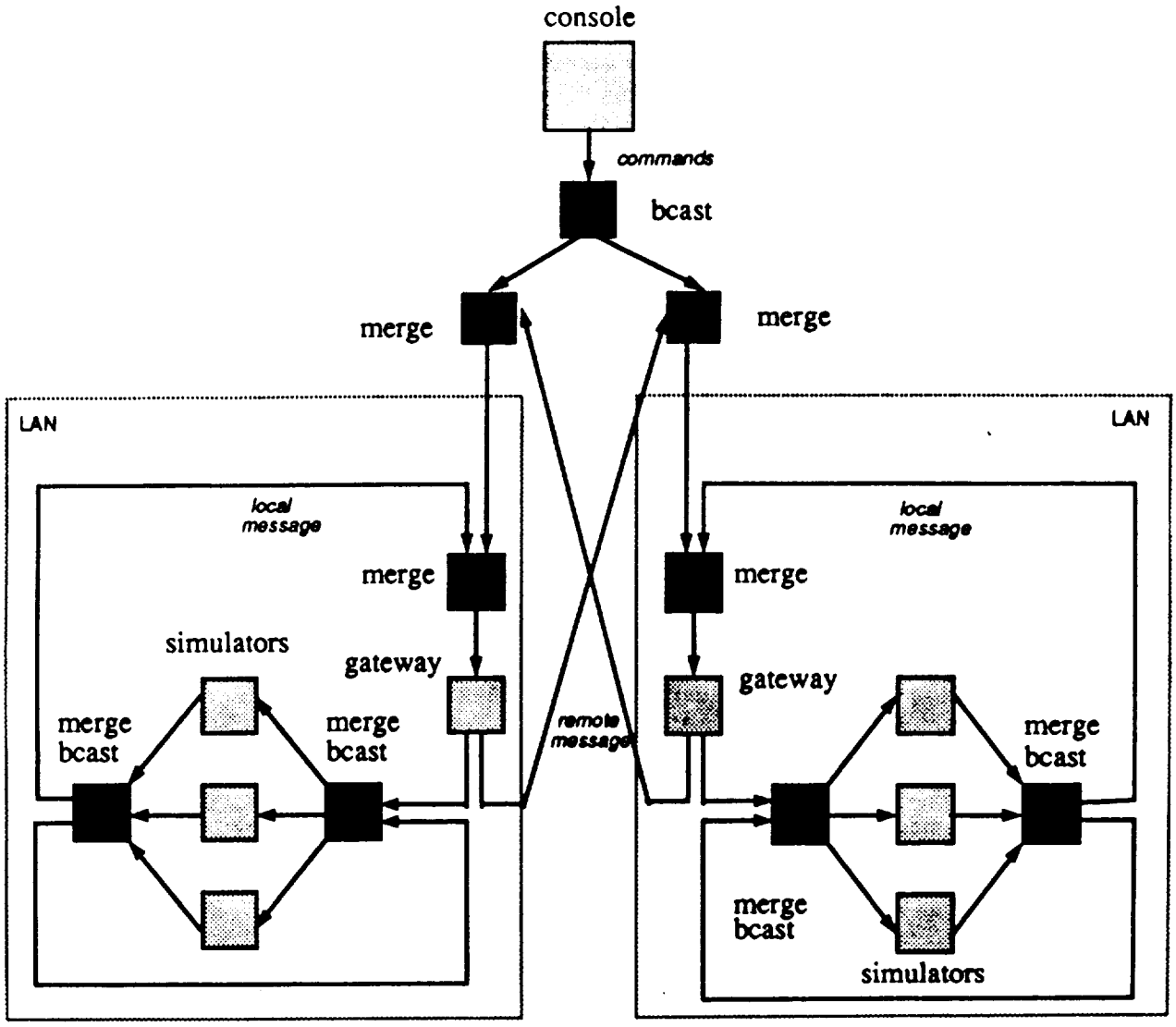


Figure 3: Structure of the Application

dynamic reconfigurations, removal of processes and queues, and instantiation of new processes and queues without affecting the remaining components. This approach to application-level programming is similar in spirit to the *constructive* approach of CONIC [4]. We illustrated this method for developing distributed applications by describing the implementation of a simple prototyping tool for modelling various configurations of networked simulators. We wrote Durra task and application descriptions and used them to control the evolution of the application, both during the development and during the execution.

References

[1] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language.
Technical Report CMU/SEI-86-TR-3 (DTIC AD-A178 975), Software Engineering Institute, Carnegie Mellon University, December, 1986.

[2] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language Reference Manual (Version 2).
Technical Report CMU/SEI-89-TR-34, Software Engineering Institute, Carnegie Mellon University, September, 1989.

[3] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, M.J. Gardner.
Developing Fault-Tolerant Distributed Systems.
Technical Report, Software Engineering Institute Technical Review 1989, 1990.

[4] J. Kramer and J. Magee.
A Model for Change Management.
In *Proceedings of the IEEE Workshop on Trends for Distributed Computing Systems in the 1990's*, pages 286-295. IEEE Computer Society, September, 1988.

Session 3

Software Reuse

Chair: **Robert Angier**, *IBM Corp.*

Research Directions in Software Reuse

November 8, 1990

Will Tracz

MD 0210
IBM Federal Sector Division
Owego, NY 13827
(607) 751-2169
net: OWEGO@IBM.COM

Unclassified

Session 3

Software Reuse

Chair: **Robert Angier**, *IBM Corp.*

Research Directions in Software Reuse

November 8, 1990

Will Tracz

MD 0210
IBM Federal Sector Division
Owego, NY 13827
(607) 751-2169
net: OWEGO@IBM.COM

Unclassified

Overview

"*Currently, **software is put together one statement at a time. What we need is to put software together one component at a time.***" – **Barry Boehm, at the Domain Specific Software Architecture (DSSA) Workshop, July 11-12, 1990.**

Topics

- Darpa/ISTO **Megaprogramming**
  - Domain Analysis and Modelling
  - Rapid Prototyping
  - Software Understanding
  - Formal Methods
- Recent Workshops
  - Realities of Reuse - January 1990
  - Methods and Tools for Reuse - June 1990
- 3-C Model for Software Components

Megaprogramming Motivation

"***Megaprogramming is the type of thing you can go into a 3-star general's office and use to explain what DARPA is going to do for them to make their software less expensive and have better quality.***" – Barry Boehm, at the ISTO Software Technology Community Meeting, June 27-29, 1990.

"***Software productivity improvements in the past have been accidental because they allow us to "work faster". DARPA wants people to "work smarter" or to avoid work altogether.***" – Barry Boehm, at the Domain Specific Software Architecture (DSSA) Workshop, July 11-12, 1990.

Megaprogramming Vision

- Megaprogramming is a "giant step" toward increasing
  - "development productivity,
  - maintenance productivity,
  - reliability,
  - availability,
  - security,
  - portability,
  - interoperability and
  - operational capability ."
- Megaprogramming will incorporate proven, well-defined components whose quality will evolve.
- Megaprogramming requires the modification of the traditional software development process.
- Domain-specific software architectures need to be defined and implemented with open interfaces according to software composition principles, and open interface specifications.
- Additional environmental capabilities are needed to provide software understanding.

Megaprogramming Software Team

"*Configuration = Components + Interfaces + Documentation*

Software Team = Configuration + Process + Automation + Control." – Bill Scherlis, at the ISTO Software Technology Community Meeting, June 27-29, 1990.

Megaprogramming Software Team Goal

To create an environment to:

1. "manage systems as *configurations* of components, interfaces, specifications, etc.,
2. increase the *scale of units* of software construction (to modules), and
3. increase the *range of scales* of units of software interchange (algorithms to subsystems)."

Key Elements of Megaprogramming Software Team

- **Component sources** — currently, components under consideration are from reuse libraries (e.g., SIMTEL20 or RAPID) or COTS (Commercial Off-The-Shelf) software (e.g., GRACE or Booch components). Application generator technology is desirable to provide for adaptable modules. Re-engineered components (e.g., CAMP) could provide additional resources.
- **Interface definitions** — currently, there exists an ad hoc standard consisting of Ada package specifications and informal documentation. It is desirable to develop a Module Interconnect Formalism (MIF) with hidden implementations supported by formal analysis and validation tools.
- **System documentation** — currently, simple hypertext systems are supporting the textual documentation associated with software components. It is desirable to create a repository-based, hypermedia environment that provides traceability between artifacts and supports the capture, query, and navigation of domain knowledge.

Key Elements of Megaprogramming Software Team

- **Process structure** — currently, there exists no predictable software development process. It is desirable to develop an evolutionary development life cycle with support to domain engineering, integrated requirements acquisition, and reverse/re-engineering.
- **Process Automation** — currently, CASE tools are either stand-alone or federated (e.g., Unix'). It is desirable to integrate the tools and create a meta-programming environment to support process description and refinement.
- **Control/Assessment** — currently, only a priori software metrics and process instrumentation exists. It is desirable to integrate the measurement process with tool support and to create an cost-estimation capability.

' Unix is a trademark of AT&T Bell Laboratories

Megaprogramming Resources

- STARS (Software Technology for Adaptable Reliable Systems) SEE (Software Engineering Environment)
- Arcadia
- CPS/CPL (Common Prototyping System/Common Prototyping Language)
- DSSA (Domain Specific Software Architectures)
- POB (Persistent Object Bases)
- SWU (Software Understanding)
- REE (Re-Engineering)

Interface and architecture codification will be supported by a Module Interconnect Formalism (MIF), which is an outgrowth of the CPS/CPL program.

Goal of MIF

To adequately describe a software component such that its selection and use can be accomplished without looking at its implementation.

Component Interface

- entry points,
- type definitions
- data formats (e.g. Ada package specification),
- a description of its functionality,
- side effects,
- performance expectations,
- degree and kind of assurance of consistency between specification and implementation (reliability), and
- appropriate test cases.

SWU Design Record

The design record will provide a "common data structure for system documentation and libraries".

The suggested data elements in a design record include:

- code,
- test cases,
- library and DSSA links,
- design structure,
- access rights,
- configuration and version data,
- hypertext paths,
- metric data,
- requirement specification fragments,
- PDL texts,
- interface and architecture specifications,
- design rationale,
- catalog information, and
- search points.

Megaprogramming Software Interchange

"Software Interchange = Software Team + Convention + Repository + Exchange." – Bill Scherlis, at the ISTO Software Technology Community Meeting, June 27-29, 1990.

Megaprogramming Software Interchange Goal

To "enable wide-area commerce in software components."

Elements of Megaprogramming Software Interchange

- **Conventionalization** – *currently, conventions are emerging. It is desirable to create a cooperative decision and consensus mechanism that supports adaptable, multi-configuration libraries, which present a standard search capability.*
- **Repository/Inventory**– *currently, repositories support code storage only. It is desirable to retain, assess, and validate other software assets such as architectures, test cases, specifications, designs, and design rationales.*
- **Exchange/Brokerage** – *current intellectual property rights and government acquisition regulations are stifling a software component industry. It is desirable to populate certain application domains (via DSSA) and to support the creation of an electronic software component commerce by*
  - *defining mechanisms for access control,*
  - *authentication/certification, and*
  - *establishing composition conventions.*

Realities of Reuse Workshop

January 4-5 1990
Syracuse, NY

The goal of the workshop was to

"... serve as a forum for sharing practical experiences and methodologies

- *for specifying and designing software for reuse,*
- *for defining the level and kinds of components that can be reused, and*
- *for incorporating reuse philosophies into organizations".*

Highlights

Software Reuse: Representing a Reusable Software Collection
William Frakes, Software Productivity Consortium

- *IR approach is the best way to go about organizing a library.*
- *other approaches (keyword, faceted, semantic net, hypertext) require significant amounts of effort to set up and to catalog.*

Realities of Language Support for Reuse: What we desire - What we have.
Larry Latour, University of Maine

- *Code and type inheritance*
- *parameterization*
- *granularity of change*
- *algorithm parameterization.*

Highlights

Library-Base Software Design Methodology
David Musser, RPI

- *The following are myths:*
  1. *generic software is not efficient,*
  2. *generic software is hard to find, and*
  3. *software libraries only address the implementation level.*
- *Rationale:*
  1. *algorithms can be more complex and efficient than any simple ones that a programmer would tend to write from scratch.*
  2. *Library can be organized into a semantic net that a user could easily navigate to find what was needed.*
  3. *80% of the effort to build a library is writing the specifications that could be reused at high level design time.*

Highlights

Reusable Specifications for Requirements Prototyping and System Construction
Donald Hartman, International Software Systems, Inc.

- *Proto system that ISSI built for RADC.*
- *Graphical input language for drawing data flow diagrams, then simulating them (if the contents of the nodes is real code).*
- *One can also watch the data flow nodes fire.*

Designing for Reuse: Is Ada Class Conscious?
Sholom Cohen, Software Engineering Institute

- *Feature Analysis*
- *Commonality Analysis to develop a generic architectures.*

Highlights THIRD ANNUAL WORKSHOP: METHODS & TOOLS FOR REUSE

June 13-15 1990
Syracuse, NY

Highlights

- *If you are not teaching software reuse, you are not teaching software engineering (Bob Cook - University of Virginia)*
- *The (throw everything into a) "Bag" approach was the style of software reuse in the 80's, the "Generic Architecture" approach is the style for the 90's.*
- *"Cloning" (a new-to-me term) is a form of unplanned reuse (salvaging) popular at HP and other companies.*
- *What is needed to stimulate software reuse are handbooks that describe the architectures of applications along with their design rationale.*
- *GOTO's were found bad in the 70's for the same reason that Top Down Decomposition will be found bad in the 90's -- failure to modularize complexity*

Highlights THIRD ANNUAL WORKSHOP: METHODS & TOOLS FOR REUSE

- *A good interface specification has enough information so the (re-) user doesn't have to look at the code to figure out what it does and how to use it.*
- *One (large) problem that people have failed to realize is that software reuse doesn't stop at retrieval.*
- *Data flow diagrams provide too much information to be included in the functional specification of a reusable software component.*
- *Domain Analysis research projects are actively being addressed at TRW, Bell Labs, UNISYS, ESPRIT, Magnovox, CONTEL, MCC and SPS.*

Highlights THIRD ANNUAL WORKSHOP: METHODS & TOOLS FOR REUSE

- *SPS (Software Productivity Solutions) speculated that in 6 years they have increased their programmer productivity an order of magnitude through*
  1. *simple black box reuse (function libraries)*
  2. *parameterized black box reuse (Ada generics)*
  3. *large component reuse (modules/Ada packages)*
  4. *inheritance (required object-oriented programming language)*
  5. *parameterized application generators*

  NOTE: they indicated the switch to OOPL was the greatest facilitator of reuse.
- *Best malaprop: "Generics are something you use when you can't afford the name brand."*

Paper Summaries

KAPTUR: KNOWLEDGE ACQUISITION FOR PRESERVATION OF TRADEOFFS AND UNDERLYING RATIONALES
Sidney C. Bailin, CTA INCORPORATED

- *Roll-your-own hypertext system for capturing design decisions.*
- *An impressive domain analysis case study in tools to support reuse.*

REUSE OF SOFTWARE KNOWLEDGE: A PROGRESS REPORT
Prem Devanbu, AT&T BELL LABORATORIES

- *Knowledge Base to assist in software reuse.*

HYPERBOLE: A RETRIEVAL-BY-REFORMULATION INTERFACE THAT PROMOTES SOFTWARE VISIBILITY
Patricia Carando, Schlumberger Laboratory for Computer Science

- *Generic user interface and data analysis architecture to analyze well data.*
- *Graphical workstation tool (500-600 classes).*

Paper Summaries

AN EMPIRICAL FRAMEWORK FOR SOFTWARE REUSE RESEARCH
Bill Frakes, Software Productivity Consortium

- *Determine the relationships between the dependent variables in model*
  1. *quality,*
  2. *productivity, and*
  3. *reuse*

THE 3C MODEL OF REUSABLE SOFTWARE COMPONENTS
Stephen Edwards, Institute for Defense Analyses

- *Emphasis on the maintenance payback from using the 3C model.*

THE THREE CONS OF SOFTWARE REUSE
Will Tracz, IBM Corporation

- *The gospel according to Will.*

Paper Summaries

DESIGNING FOR SOFTWARE REUSE IN ADA
Sholom Cohen, SEI/Carnegie-Mellon University

- *Implementation implications of using the 3C model in regards to hierarchies of parameterized models.*
- *Coupling inversion – where context is fixed for implementation efficiencies within the generic architecture.*

THE PRACTITIONER REUSE SUPPORT SYSTEM (PRESS): A TOOL SUPPORTING SOFTWARE REUSE
Cornelia Boldyreff, Brunel University

- *ESPRIT 1094 Practitioner Project (one of many reuse projects funded by ESPRIT).*
- *Capture and reuse software concepts from designs through code.*
- *Questionnaire was passed out to the team company to assist in domain analysis*
- *"canonical" form for describing software components developed.*

Paper Summaries

REUSE AT HEWLETT-PACKARD LABORATORIES
Martin L. Griss, Hewlett-Packard Laboratories

- *Hypertext tools.*
- *Object-Oriented Design.*

BEYOND RETRIEVAL: UNDERSTANDING AND ADAPTATION IN SOFTWARE REUSE
Karen Huff & Ronnie Thomson, GTE Laboratories Inc.

- *SATURN (Software Adaptation Through Understandable Reuse Notation)*

THE STARLITE INTELLECTUAL REUSE PROJECT
Robert P. Cook, University of Virginia

- *Reusable operating system and system modelling components*

Conceptual Model
Reusable Software Components

- Context
- Concepts
- Content
  - Context
  - Concepts
  - Content

Conceptual Model
Context

- **"Language shapes thought"**
  - Inheritance
  - Genericity/Parameterization
  - Importation
- **Binding time**
  - Compile time
  - Load/Bind time
  - Run Time

Conceptual Model
Concepts

- **Concept:** – *What*
- **Content:** – *How*
- **Context:**
  1. **Conceptual** – *relationship*
  2. **Operational** – *with/to what*
  3. **Implementation** – *trade – offs*

Context: what is needed to complete the definition of a concept or content within an environment. (*Latour*)

Software Components
Formal Foundations

- **Horizontal Structure**
  1. type inheritance
  2. code inheritance
- **Vertical Structure**
  - implementation dependencies
  - virtual interfaces
- **Generic Structure**
  - variations/adaptations

Conceptual Model
Example

- Concept: Stack
  - *Operational Context:* Element/Type
  - *Conceptual Context:* Deque
  - *Implementation Context:* Sequence

Conceptual Model
Example

- Stack **Implementation**
  1. Inherit Deque
  2. Use an array
  3. Use a linked – list
     - memory management
     - no memory management
     - concurrent access

Megaprogramming Example

Stack − > Deque

```
make Deque [ Triv ] is
    Stack [ Triv ]
        * ( rename ( Push = > Push_Right )
                   ( Pop = > Pop_Right )
                   ( Stack = > Deque )
        * ( add Push_Left, Push_Right )
end;
```


Hyperprogramming Example

Make with View

```
make Integer_Set is
     LIL_Set [ Integer_View ]
end;

view Integer_View :: Triv = > Standard is
     types (Element = > Integer);
end;
```


Megaprogramming Example

Make with Vertical Composition

```
make Short_Stack is
     LIL_Stack
         − − horizontal composition
     needs (List_Theory = > List_Array)
         − − vertical composition
end;
```

LILEANNA Example

Package Expressions

```
make New_Ada_Logic_Interface is
    Identifier_Package +
    Clause_Package*(hide Copy) +
    Substitution_Package +
    DataBase_Package +
    Query_Package*(add function Query_Fail (C: Clause;
                                            L: List_Of_Clauses)
                                              return Boolean)
                  *(rename ( Query_Answer => Query_Results ))
end;
```

Ada Net

John McBride
Planned Solutions

Paper not available at time of printing.

Session 4

Software Engineering: Issues for Ada's Future

Chair: **Rod L. Bown** , *University of Houston-Clear Lake*

Assessment of Formal Methods for Trustworthy Computer Systems

Susan Gerhart

Microelectronics and Computer Technology Corp. (MCC)

Paper not available at time of printing.

Issues Related to Ada 9X

John McHugh
Computational Logic, Inc.

Paper not available at time of printing.

POSIX and Ada Integration In The Space Station Freedom Program

Dr. Robert A. Brown
The Charles Stark Draper Laboratory, Inc.

This paper discusses the integration of real-time POSIX and real-time, multiprogramming Ada in the Space Station Freedom Data Management System. Use of POSIX as well as use of Ada has been mandated for Space Station Freedom flight software. However, POSIX and Ada assume execution models that are not always compatible. This becomes particularly true once Ada has been extended to support multiprogramming. This paper points out the conflicts between POSIX and Ada multiprogramming execution models and describes the approach taken in the Data Management System to resolve those conflicts.

Session 5

Ada Run-Time Issues

Chair: **Alan Burns**, *University of York (U. K.)*