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

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.

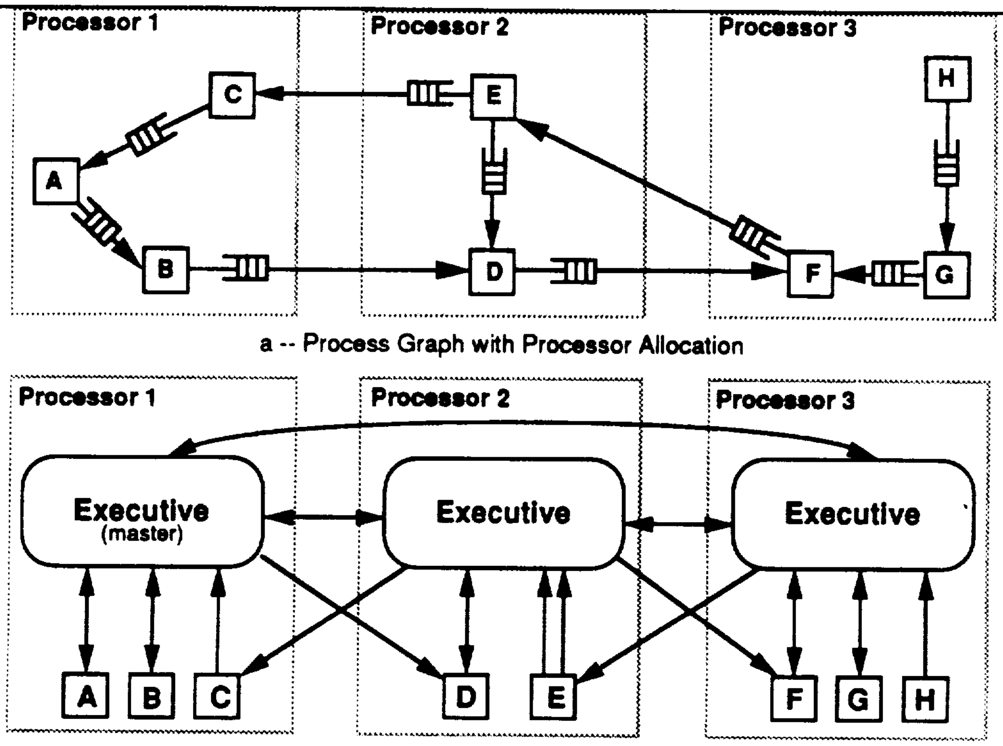

Processor 1
Processor 2
Processor 3
A
B
C
D
E
F
G
H
a -- Process Graph with Processor Allocation
Executive
(master)
Executive
Executive
b -- Actual Communication Patterns

**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,

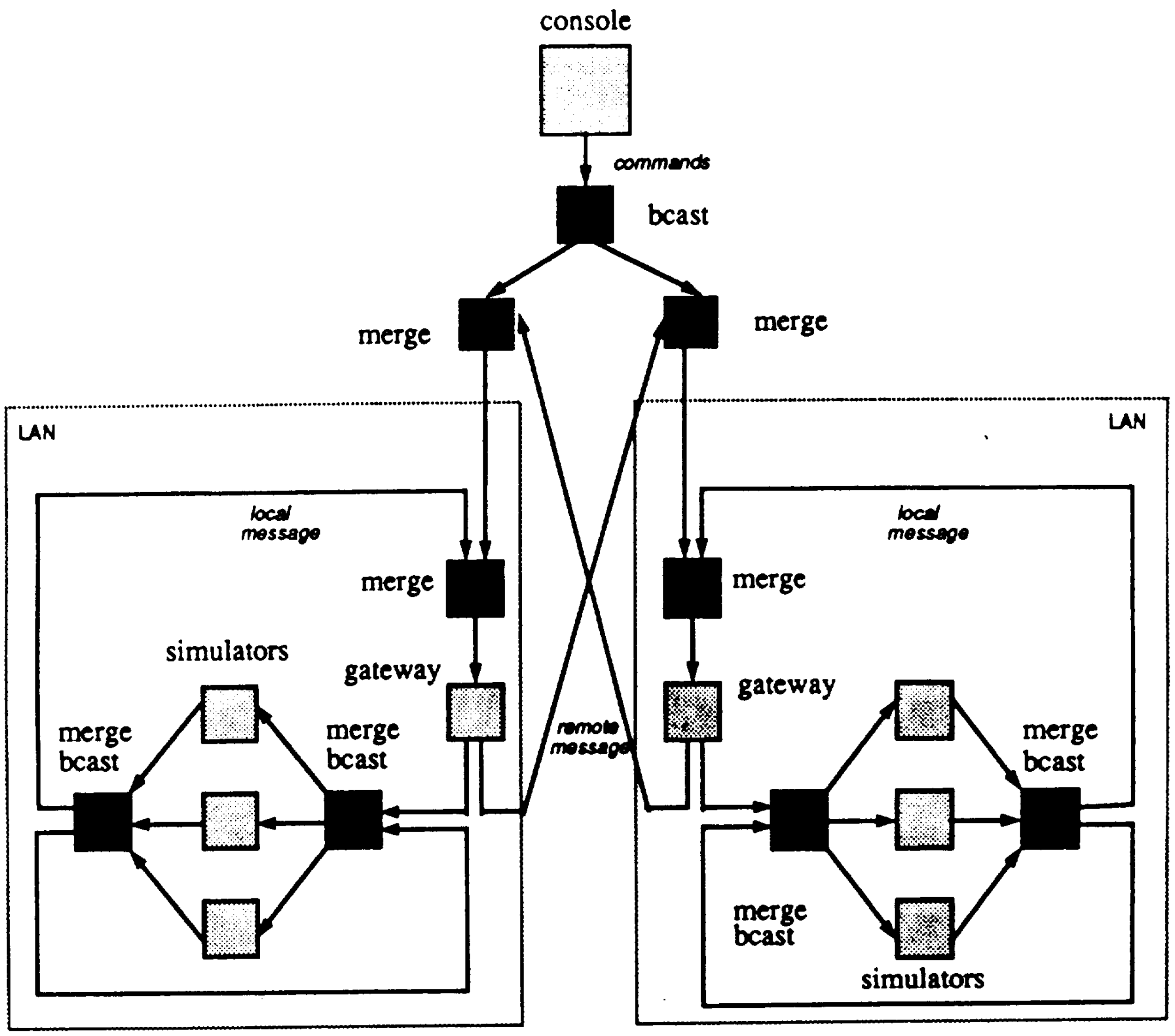


**Figure 3:** Structure of the Application

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.*

Session 3     10:15 - 11:45 a.m.          Nov. 8

# 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.*

Session 3   10:15 - 11:45 a.m.   Nov. 8

# 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

IBM

## 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[1]). 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.

[1] 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. Ballin, 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**

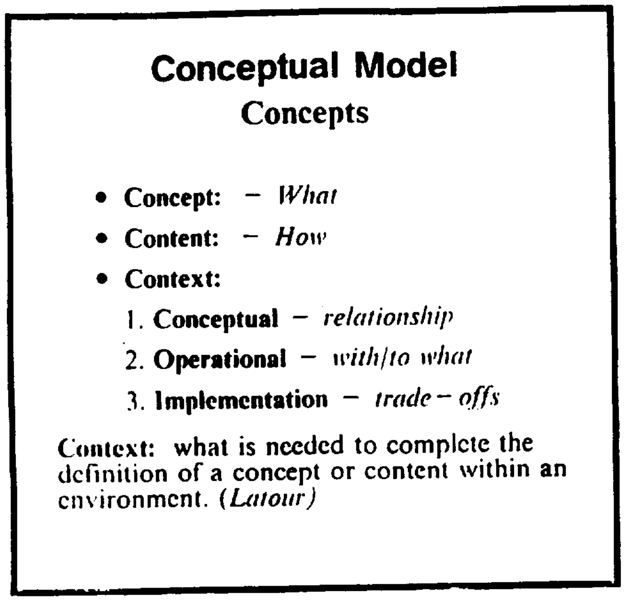
- *Reusable operating system and system modelling components*

## 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

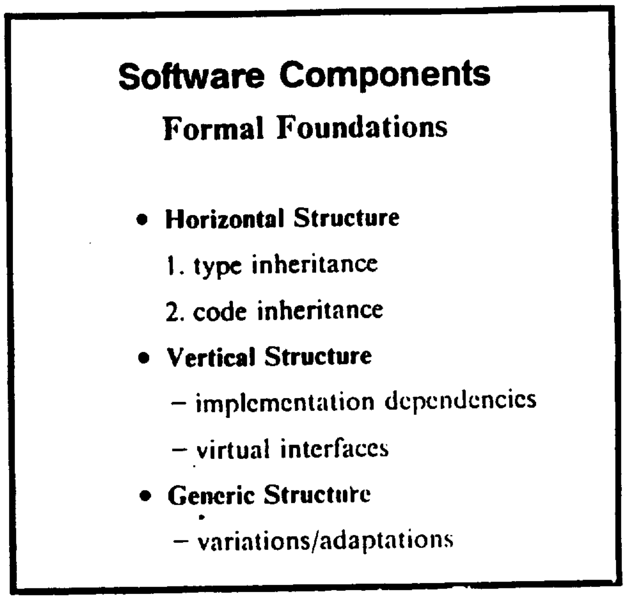


## Conceptual Model

### Concepts

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

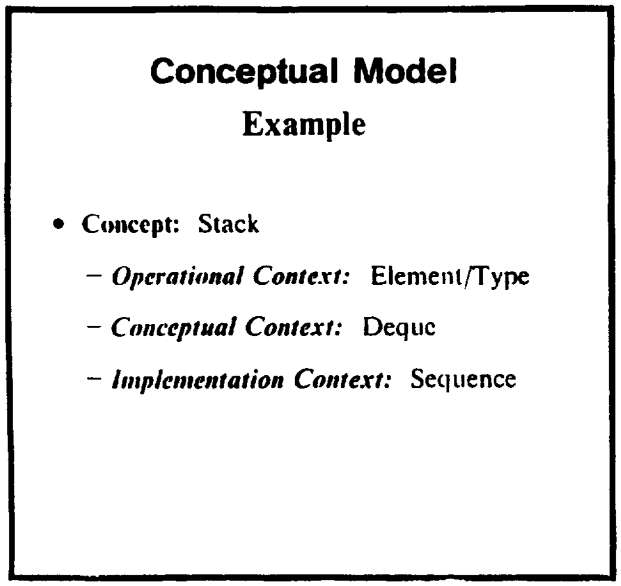
**Context:** what is needed to complete the definition of a concept or content within an environment. (*Latour)*



## Software Components

### Formal Foundations

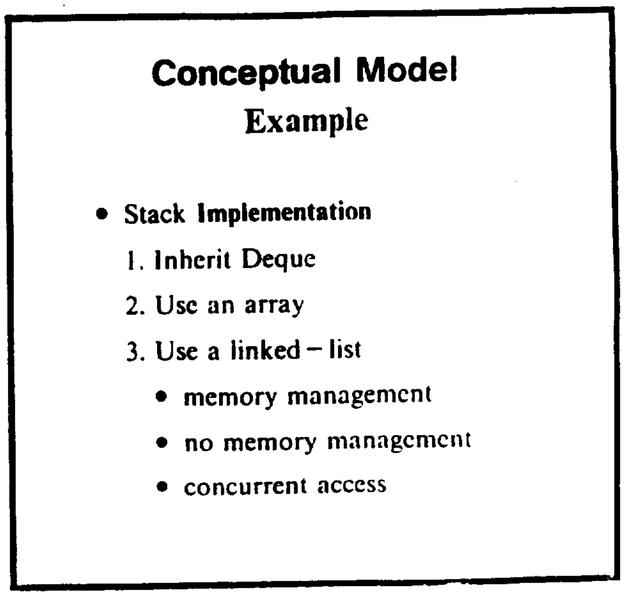
- **Horizontal Structure**
  1. type inheritance
  2. code inheritance
- **Vertical Structure**
  - implementation dependencies
  - virtual interfaces
- **Generic Structure**
  - variations/adaptations



## 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.)*