Key Concepts of the INCAS Multicomputer Project

JÜRGEN NEHMER, DIETER HABAN, FRIEDEMANN MATTERN, DIETER WYBRANIETZ, AND H. DIETER ROMBACH

Abstract—**This paper gives an overview of the INCAS (INCremental Architecture for distributed Systems) multicomputer project, which aims at the development of a comprehensive methodology for the design and implementation of locally distributed systems. A structuring concept for distributed operating systems has been developed and integrated into the system implementation language LADY. The concurrent high-level programming language CSSA, based on the actor model, has been designed for the implementation of distributed applications. A substantial effort in the INCAS project is directed towards the development of a distributed test methodology. An experimental system has been implemented on a network of ten MC68000 microcomputers. Preliminary experience with the methodology has been gained from a small number of prototype applications.**

Index Terms—**Distributed operating systems, distributed programming languages, distributed systems, distributed testing, message passing, multicast communication, multicomputer.**

I. Introduction

THE INCAS project belongs to the wide class of research projects which investigate the potential benefits (and drawbacks) of multicomputer architectures as an alternative to traditional single processor systems. A *multicomputer* is a locally concentrated set of loosely coupled autonomous processing nodes of identical structure each with its own private memory. Each single node itself may consist of a tightly coupled multiprocessor system as depicted in Fig. 1.

The nodes do not share global memory and communicate solely by exchanging messages over a high bandwidth interconnection network. Multicomputers with hundreds of nodes such as BBN's Butterfly machine and systems based on the hypercube interconnection scheme (e.g., the T-series from Floating Point Systems [13] and Intel's iPSC [31]), are now commercially available and can serve as advanced testbeds for research projects.

In contrast to projects on workstation networks such as Accent [25], V-system [7], and Eden [1], the INCAS project puts its emphasis on the development of a comprehensive methodology for the design of locally distributed systems, supporting applications with large grain parallelism at the process level. Related multicomputer projects are Cm* [32], Zmob [39], CONIC [17], Micros [37], Chorus [41], and Crystal [10]. The INCAS project addresses the entire spectrum of multicomputer-related software aspects such as distributed operating systems, distributed programming languages and environments, distributed applications, and a distributed test methodology. Most other projects on parallel and distributed computing specialize in particular topics such as programming methodology [1], [2], [21], [35].

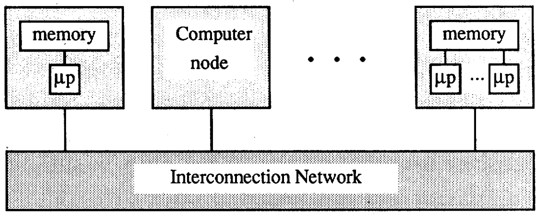


Fig. 1. General structure of a multicomputer system.

The project name INCAS reflects one of the most important features of our approach: the ability to *incrementally* configure systems of any size and desired degree of fault tolerance, according to particular application requirements.

Two distributed languages have been developed in the course of this project: the language LADY for the design and implementation of distributed operating systems and the high-level application language CSSA. The underlying philosophies of both languages date back to earlier research projects and take different approaches to structuring distributed software. Since any distributed operating system should be able to support different distributed application languages efficiently, we considered the implementation of a CSSA run-time environment in LADY as a challenge for the applicability of our design methodology.

This paper presents an overview of the INCAS project with special emphasis on both languages involved. In Section II the overall system architecture of INCAS is outlined. Section III is devoted to an in-depth discussion of our distributed operating system methodology including the language LADY. In Section IV we will focus on the distributed application language CSSA and its communication concept. We also show how run-time support for CSSA was implemented in LADY. The distributed test facilities are described in Section V. The lessons we have learned so far are summarized in Section VI. Section VII

Manuscript received September 23, 1986; revised March 6, 1987. This work was supported by the Deutsche Forschungsgemeinschaft as part of the Research Institute SFB 124 "VLSI-Design and Parallel Architectures," Kaiserslautern—Saarbrucken, West Germany.

J. Nehmer, D. Haban, F. Mattern, and D. Wybranietz are with the Department of Computer Science, University of Kaiserslautern, D-6750 Kaiserslautern, West Germany.

H. D. Rombach was with the Department of Computer Science, University of Kaiserslautern, D-6750 Kaiserslautern, West Germany. He is now with the Department of Computer Science, University of Maryland, College Park, MD 20742.

IEEE Log Number 8715357.

gives an overview of the current project status and outlines areas of future research.

II. Overall System Architecture

From a global point of view, our distributed system is structured into four logical layers as shown in Fig. 2.

The *physical network layer* consists of ten MC68000-based four-processor nodes interconnected by a logical communication ring. A separate ring is used for test and measurement purposes. The present approach to the interconnection network is based on the COM-chip used in the ARCNET [33]. The moderate transmission speed of the token-bus protocol of the COM-chip is sufficient for the development and test phase of our distributed system. In a medium term range, we intend to extend the testbed by a larger number of nodes and to replace the interconnection network by a faster system.

The *LADY Support System* provides mechanisms for process creation and deletion, local process synchronization, interprocess communication, storage management, local I/O, exception handling, etc. The *distributed operating system layer* is represented by a physically dispersed set of communicating distribution units, called teams. Each team usually consists of a cluster of tightly coupled processes which share common working space. Special teams, called agent servers, implement the run-time environment for the CSSA language.

The *distributed application layer* is formed by communicating agents, the independent execution units of CSSA.

Program development is done under UNIX® on different machines, which have access to the communication ring of the distribution testbed.

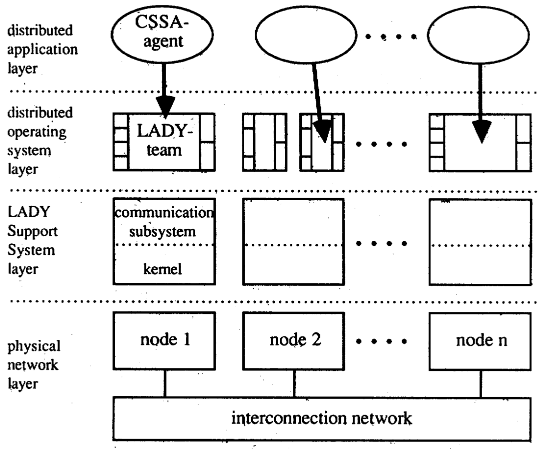


Fig. 2. The overall system architecture.

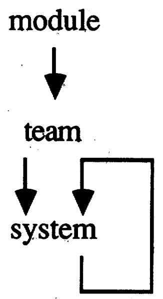


Fig. 3. Language levels of LADY.

III. The Distributed Operating System Design Methodology

A. The LADY Language and Its Underlying Structural Concepts

The LADY language reflects our view of an adequate structuring model for distributed operating systems. Experiences with early versions of LADY [23], [29] induced several iterations of the early concepts until the present design has been reached [40], [38]. The development of LADY has been motivated by the lack of expressive power with respect to higher level structures in present system implentation languages.

The structuring concepts of LADY are expressed in terms of three language levels as illustrated in Fig. 3.

Process and monitor modules constitute a *team*. Several teams and systems form a *system*. This system definition is recursive and allows nested system structures of any depth.

LADY is a strongly typed language. Each description of a module, team and system type, is divided into a *specification part* and an *implementation part*, called the body of the respective type. The specification part describes the interface a type exports to its environment.

®UNIX is a registered trademark of AT&T Bell Laboratories.

These features promote the decomposition of programs into well-structured building blocks. Due to narrow interface specifications, these building blocks can be easily exchanged between different environments. This greatly supports the customizing of distributed operating systems using predefined building blocks.

The fundamental structuring unit of LADY is the *team*. A team consists of a collection of tightly coupled *processes* which cooperatively perform a specific function. Processes within a team solely communicate via shared memory. Synchronization is achieved by using monitor modules or lower level primitives such as semaphores. Teams are *distribution units*, i.e., they have to be placed as a whole at one node. Different teams may be placed at the same node or at distinct nodes.

Teams interact with other teams solely via *message passing*. The internal structure of teams is hidden by the encapsulation of teams by a *port interface*. The port concept in INCAS is symmetric: input ports define the message interface exported by a team, while output ports define the message interface imported by a team from its environment. A process can send a message through an output port to a destination input port, only if a connection between the output and the input port has been established beforehand. A similar approach has been taken in the language NIL [35].

Two types of connections between input and output ports can be defined:

1) *logical channels*, which provide for a one-to-one link between an output and an input port;

2) *logical buses*, which provide for a many-to-many link between output and input ports, thereby offering a multicast communication capability.

Teams and systems as well as their interconnections can be dynamically generated and deleted. Teams are mobile; communication between teams does not depend on their location. The internal structure of a team is fixed at compile time and cannot be changed at run time.

B. Communication Concepts

The semantics of the message passing mechanisms in LADY can be characterized by the following properties:

- one way communication
- reliable one-to-one communication via logical channels
- unreliable communication via logical buses allowing the user to define the success of a communication.

Input and output ports as well as communication paths are *typed*, i.e., the communication paths can only transport messages of one specific type, and the connected ports must have been associated with the same type at declaration time. However, strong typing can be bypassed for special applications. Ports can be assigned a timeout value. Send and receive operations will be canceled, if they cannot complete within the time limit specified by the timeout value of the respective port. Input ports can be associated with a *buffer* of fixed length at declaration time. A parameter determines the maximum number of messages the buffer can store. A zero value indicates that no buffer is supplied.

The semantics of *one-to-one communications* via logical channels can be described as follows. A process which attempts to send a message to a receiver suspends execution until there is buffer space available at the receiving input port, or in case no buffer was associated with the input port, the receiving process is ready to accept the message; symmetrically the receiving process is suspended until the message it is waiting for is available from the buffer or directly from the sending process.

This scheme works also in the case of many output ports connected to one input port.

If no buffer has been associated with an input port, the one-to-one communication is semantically equivalent to the *synchronization send* [20]. There seems to be no general consensus on how to define synchronous communication, especially when buffers are involved (cf. [3], [20], [22]). We regard our particular approach to one-to-one communication more closely related to synchronous communication as defined in [3] or [20], than to asynchronous communication. The reason for this judgement is that in our approach only a fixed number of messages can be buffered at the receiver's side; after this buffer limit has been reached the sender gets suspended.

The *multicast* communiction based on logical buses has required some modifications to the semantics as defined for the one-to-one communications above. An arbitrary number of input and output ports can be connected to a logical bus. Logical buses offer three distinct transmission modes, which differ in their addressing selectivity:

1) a multicast message sent over a logical bus is potentially received by all input ports connected to this bus.

2) a subset of input ports connected to the same bus can be defined as a *multicast group*; a message sent to a multicast group is potentially received by all members of the multicast group; logical buses and multicast groups within buses form a two-stage multicast communication capability.

3) a message can be sent to one single input port connected to the bus.

The transmission modes are selected by different send statements. Furthermore, LADY encompasses built-in types and operations, enabling creation and deletion of buses and groups, manipulation of bus and group names, dynamic reformation of multicast groups, and connection and disconnection of ports at run time. All operations are performed locally thus avoiding a significant implementation overhead. These operations are discussed in more detail in [40].

A major problem in the design of a multicast mechanism consists in the definition of the semantics of a multicast send with special regard to reliability aspects. Our solution to a flexible multicast protocol allows a process to dynamically define the *success* of a multicast operation. The introduction of an additional port operation

<output port>.demandack(<no of acks>)

enables a process to specify the number of acknowledgments it expects as a result of a multicast. This operation changes the value of an internal port variable DEMANDACK to < no of acks >. The default value of DEMANDACK is zero. Each message carries a 1-bit information whether an acknowledgment is requested or not. In the latter case the receiving port generates no acknowledgment message. The send operation terminates if:

1) a timeout occurred at the sending output port (in this case an undefined number of receivers might have received the message), or

2) the expected number of acknowledgments was received at the output port, indicating that the message was either copied into the input port buffers or was received by the expected number of processes.

Additionally, the transparent sending of acknowledgments can be controlled explicitly by the operations

<input port>.enableack

<input port>.disableack

By default, acknowledgments are enabled. The operations demandack, enableack, and disableack, are only relevant to logical bus communications; they have no effect in one-to-one communications via logical channels. Their purpose is to provide a basis for a wide class of multicast protocols at the system programming level.

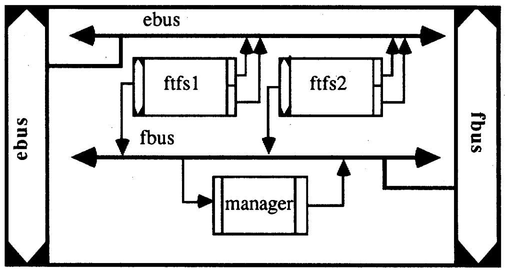


Fig. 4. The system type "file system."

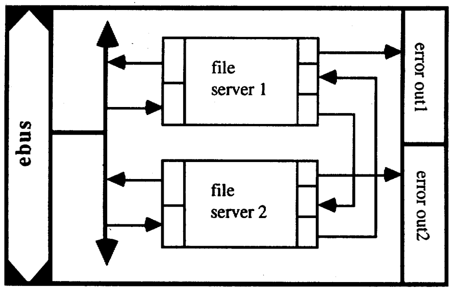


Fig. 5. The system type "fault tolerant file server."

C. An Example

We use the system illustrated in Fig. 4 as an example to discuss the structuring means of LADY in more detail.

The system "file system" consists of two system objects of type "fault tolerant file server," and a team object "manager" of type "file system manager." The team "file system manager" is responsible for assigning certain requests to a specific file server. The file system exports the logical buses "fbus" and "ebus" to its environment by the two respective bus interfaces. The system type "file system" is described at the LADY language level as follows:

```
{specification part}
SYSTEM file system =
BUS fbus type = file system message type;
BUS ebus type = file system error message type;
   fbus: fbus type;
   ebus: ebus type;
END SYSTEM file system;

{implementation part}
SYSTEM BODY file system =
   {library part}
   SYSTEM fault tolerant file server;
   TEAM file system manager;

   {declaration part}
   ftfs1: fault tolerant file server;
   ftfs2: fault tolerant file server;
   manager: file system manager;

   {connection part}
   CONNECT ftfs1.filebus TO fbus,
      ftfs1.error out1 TO ebus,
      ftfs1.error out2 TO ebus,
      ftfs2.filebus TO fbus,
      ftfs2.error out1 TO ebus,
      ftfs2.error out2 TO ebus,
      manager.in TO fbus;
      manager.out TO fbus;
END SYSTEM BODY file system;
```

Each fault tolerant file server (ftfs1 and ftfs2 in Fig. 4) is refined by the system type "fault tolerant file server" of Fig. 5.

The fault tolerant file server consists of two team objects of type "file server." Only one team is active, while the other operates in a hot-standby mode. The fault tolerant file server receives client requests and sends replies via the bus interface "filebus." Error messages and abnormal conditions are reported via the ports "error out1" and "error out2."

Finally, the internal structure of the team "manager" is shown in Fig. 6. It is defined by two processes, which communicate via a monitor. The language representations of Figs. 5 and 6 are omitted here. The interested reader is referred to [40], [38].

D. Design Rationale

After having reviewed the essentials of our structuring model for distributed operating systems, we summarize the technical arguments which guided our specific design:

1) Connection-Oriented Communication Model: The decision for a connection-oriented communication model (logical channels and buses) is motivated by our experience that distributed systems are easier to understand if connection information is provided explicitly. This interconnection information can be used for more efficient routing mechanisms.

2) One-Way Communication: In our opinion, it is mandatory for an implementation language to support various distributed applications encompassing a broad variety of communication philosophies. For this reason, we chose a lower level one-way communication mechanism over an RPC-like mechanism as used in Eden [1], Argus [21], and the V-system [7].

3) Logical Buses: Multicast communications have been recognized as a very useful means in distributed systems. We consider logical buses as an adequate concept to describe multicast structures at the language level. Their implementation is greatly simplified in case the physical transportation medium has a broadcast or multicast capability. The logical bus concept in combination with a dynamically definable success of a multicast operation serves as a basis for application-oriented multicast protocols.

4) Ports: The symmetric port approach in LADY (input and output ports) provides for a high degree of structural transparency. The adjustable buffer capacity of input ports at compile time requires no extensive buffer management by the LADY Support System. In the case of one-to-one communication via logical channels, the buffer size determines the degree of synchronization between sender and receiver. In addition, buffers are used to allow

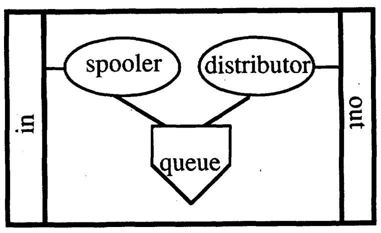


Fig. 6. The team type "file system manager."

the successful completion of multiple parallel multicast operations initiated by multiple senders on the same bus.

5) Recursive System Definitions: The introduction of recursively definable system types greatly enhances the structuring facilities for distributed systems and allows the description of hierarchical, layered, and recursive structures.

6) Teams: The choice of process clusters over single processes has the following advantages:

- typical operating system functions are more easily described by multithread control structures
- tightly coupled processes within a team can communicate more efficiently via shared memory
- performance can be increased by exploiting the potential power of multiprocessor nodes

Process clusters can also be found in languages such as SR [2], EPL [1], Argus [21], and in the V-system [7].

E. Experiences with LADY

The first version of LADY was available in the beginning of 1983. In 1983 and 1984 some applications including small batch and time sharing operating systems, a simple multiuser distributed operating system, and a process control system for distributing parcels by ZIP code, were implemented in LADY and run on a testbed of eight TI 990 microcomputers. Our primary concern with the first series of experiments was to measure the effects of a high-level, well-structured, distributed programming language on quality attributes of the resulting software such as reliability, performance, and maintainability.

In this paper the results with respect to maintainability will be briefly discussed. A series of maintenance experiments was conducted involving twelve software systems (operating systems of about 12K lines of source code each and process control systems of about 2K lines of source code each). Six of these systems were implemented in LADY, the rest was implemented in a Pascal-like language. The maintenance experiments included *corrective* (detecting and correcting seeded faults) as well as *adaptive* and *perfective* maintenance tasks. The reader is referred to [30] for a more detailed description of the experimental design, including the maintained software systems and the individual maintenance tasks.

The analysis results indicate that LADY's structural concepts have a positive impact on maintainability, especially on comprehending the structural complexity of a software system and the locality behavior of changes [30]. As an example, we want to mention the results concerning maintainability defined as the average effort in staff-hours per maintenance task. An individual maintenance task as 1) identifying and designing necessary changes (*isolation* phase), and 2) actually implementing the changes in each individual module (*correction* phase). No significant differences between LADY and Pascal systems could be detected as far as correction effort is concerned. However, the isolation effort is significantly lower for LADY systems across all three types of maintenance experiments (33 percent for corrective, 17 percent for adaptive, and 29 percent for perfective maintenance tasks). The ability to understand the complexity of LADY systems in three separate steps according to the three structural language levels is asssumed to be mainly responsible for the detected difference.

We could not only show the benefits of the structural LADY concepts compared to traditional ones exemplified by a Pascal-like language; in addition, we were able to capture the impact of structural LADY features on maintainability in a formal and quantitative way, as metrics [27], [30]. The practical importance of these metrics is particularly high due to the fact that they can already be applied during the design phase of software projects [28].

The experience gained from these experiments led to an improved version of LADY. In 1983 we started to build up a new and more powerful experimental environment based on MC68000 microcomputers. Since the summer of 1985 a first distributed operating system prototype has been operational supporting a run-time environment for the distributed application language CSSA. In this first prototype, we used a simple strategy for statically assigning team objects to physical nodes.

Presently, we are working on more sophisticated distribution strategies supporting the dynamic distribution of team objects at load time. These strategies are implemented via cooperating node managers [34].

IV. The Distributed Application Language CSSA

CSSA (Computing System for Societies of Agents) [4] is an experimental high-level programming language for expressing distributed application algorithms which involve many loosely coupled cooperating tasks. In contrast to LADY, CSSA is a more abstract language where most details of synchronization and communication tasks are hidden from the programmer. Its underlying model of distributed computation is based on the notion of *actors* originally developed by Hewitt [16]. The object-oriented message-passing philosophy of the actor-model provides clean mechanisms for exploiting parallelism and is especially well suited to distributed computing at a higher level of abstraction.

In CSSA, computations are performed by *agents*, which are active objects that communicate with other agents solely by message passing. There is no sharing of data among agents. An agent is an autonomous entity consisting of a cluster of operations that can be activated by sending messages to the agent. Each agent processes only one message at a time without interruption. Messages arriving at an agent while it is executing an operation are collected in a *mailbox*. Execution of an operation may result in any number of messages being concurrently transmitted to other agents with which the agent is acquainted.

Agents can be created dynamically, and acquaintances with other agents may be transmitted via messages. Therefore, the *agent-net*, which illustrates the potential flow of information, may change dynamically during a computation. Many agents may be sending or receiving messages at the same time.

The behavior of an agent is determined by its programmed *script*, which is an agent-schema containing operations. An agent may provide several clusters of operations. Such a cluster together with local variable declarations is called a *facet*. At any instance of time the behavior of an agent is uniquely determined by exactly one facet, the current facet. *Facetting* is accomplished by replacing the current facet by another facet. The new facet may provide some other operations or the same operations with different semantics. Therefore, facets allow the dynamic behavior of an agent to be structured.

CSSA provides a powerful set of language features for expressing communication and parallelism. Its sequential structures and data types are similar to those of Pascal. Concepts of modularization and data-abstraction have been combined in a homogeneous way to allow a structured implementation of distributed applications.

The programmer is only aware of the logical structure of the distributed system, and has no influence on the assignment of agents to processors. The physical network is made completely transparent by the underlying distributed operating system written in LADY.

During a computation the user is part of the agent-net: he resides with a multiwindow terminal or other I/O devices on a designated processor which is conceptually a specific agent, the so-called *interface agent*. It consists of a CSSA interpreter and is "programmed" dynamically by the user during the computation. In common with all other agents, the interface agent can send and receive messages from acquainted agents and create new agents. Beyond that, it comprises various features for debugging and testing distributed application programs.

A. The Communication Concept

In CSSA, communication and interactions between agents are solely performed by *asynchronous message passing*. Each agent can send a message to another acquainted agent. It is assumed that message transmission times are undefined but finite and that messages do not necessarily arrive in the same order as they were sent. Hence, all messages eventually arrive, i.e., there is no possibility of their being lost.

Data values of each type (except pointer values), including those of recursively defined complex types (arrays, records, sets) and structures built up by dynamic records and pointers, can be transmitted in messages. To provide for type checking across agents and scripts, which may be separately compiled, types as well as constants, operations, functions and procedures, may be declared in a global system library.

The basic communication construct is the asynchronous *send-statement*

send <op-name> <message> **to** <target-agent>

which does not cause the sender to wait. *Multicast* is possible by specifying a set of agents as the target.

A sender can request a *reply* by specifying

send . . . **reply to** <op-name>

The receiving agent responds to such a reply-obligation by "**reply** <message>," where the target-agent and the operation name are taken implicitly from the reply-obligation. An agent which receives a message it cannot answer on the basis of its own local knowledge, can consult another agent or pass on its reply-obligation to another agent.

Messages can be explicitly received in operations, or their arrival automatically triggers the execution of an operation. This *implicit message receipt* is the normal case for agents acting as servers and is explained first.

An agent basically consists of a set of variable declarations and several clusters of named operations:

<var-decl>
operation <name> <pattern> <assertion>
  is . . . **endoperation**
operation . . .

The global variables constitute the global state of an agent.

The agent, when not executing an operation, scans the mailbox for an executable message. A necessary condition for a message to trigger the execution of an operation is the matching of the particular operation's name and the operation name contained in the message. By the pattern and assertion, an operation describes the message it wants to receive. Using *pattern-matching* it specifies what the message should look like; this is used to test the message for structural equality with a certain pattern or data type and to break up composite data structures to extract pieces of the message and bind them to local variables. The *assertion* allows the use of an arbitrary predicate on the values of the message and the variables of the agent. The selection of a specific message from several eligible messages is assumed to be fair.

If the pattern-match succeeds and the assertion evaluates to true, the operation is executed with the actual variable-bindings, similar to the execution of a procedure. Otherwise the message remains in the mailbox without any side effects and its match is retried at a later time or tried against other operations.

Explicit message receipt, which is usually done when expecting a reply to a message sent earlier in the same operation, can be programmed using the receive-statement:

receive
  when <op-name> <pattern> <assertion> **do** . . .

```
  when . . .
  otherwise <assertion> do . . .
endreceive
```

When the otherwise-option is not present, the statement blocks the agent until a suitable message is received. A simple nonblocking receive can be programmed by specifying "**otherwise do null.**" A message can be selected by the same mechanisms (pattern-matching, assertion) as the implicit message receipt which triggers an operation.

B. An Example

The following small program demonstrates how a distributed algorithm, the so-called *echo algorithm* [6], can be programmed in CSSA. The purpose of the program is to construct a graph and then to visit and mark each node.

```
1   script NODE is
2     defines KNOW, MARK, ECHO;
3     uses MARK, ECHO;
4     type AGENT__LIST is set of agent;
5     var AGENT__LIST: NEIGHBORS;
6     var agent: ACTIVATOR;
7     facet NOT__MARKED is
8       operation KNOW (int:I; array [1 . . I]
9             of agent:N) assert I> =1 is
10        loop for J in 1 . . I do
11          put N[J] into NEIGHBORS;
12        endloop;
13      endoperation;
14      operation MARK(--> ACTIVATOR)
15          assert not empty(NEIGHBORS) is
16        send MARK(self) to NEIGHBORS;
17        replace by MARKED;
18      endoperation;
19      operation MARK(agent: SENDER)
20          assert empty(NEIGHBORS) is
21        send ECHO(self) to SENDER;
22        replace by MARKED;
23      endoperation;
24    endfacet;
25    facet MARKED is
26      operation MARK(agent: SENDER) is
27        send ECHO(self) to SENDER;
28      endoperation;
29      operation ECHO(agent: A) is
30        remove A from NEIGHBORS;
31        if empty(NEIGHBORS)
32         then send ECHO(self) to ACTIVATOR;
33        endif;
34      endoperation;
35    endfacet;
36    initial NOT__MARKED;
37  endscript
```

When starting a CSSA computation, a single agent, the interface agent, already exists. All other agents must be created dynamically. At the terminal connected to the interface agent, the user interactively writes the following CSSA statements:

```
var agent: ROOT := new NODE;
var agent: N2, N3, N4, N5, N6, N7;
N2 := new NODE; . . . ; N7 := new NODE;
```

Now seven nodes have been declared and created. To build a graph, the user sends every node a list of its neighbors:

```
uses KNOW, MARK;
send KNOW (2, N2, N3) to ROOT;
send KNOW (1, N4) to N2;
send KNOW (2, N4, N5) to N3;
send KNOW (1, N6) to N5;
send KNOW (3, N3, N5, N7) to N6;
send KNOW (2, N2, ROOT) to N7;
```

Because the node-agents are initially in the facet NOT__MARKED (line 36 and lines 7 to 24) they will eventually receive the KNOW-messages and put their acquaintances with their neighbors in the set NEIGHBORS (line 11).

The user starts computation of the echo algorithm by writing "**send** MARK (**self**) **to** ROOT;" where "self" holds as a value an acquaintance with the issuing agent. The facet NOT__MARKED contains two definitions of the operation MARK, i.e., it is overloaded. The selection is made by the assertion. If the node has no neighbors, it immediately sends an ECHO (line 21) to the sender of the MARK-message, otherwise the MARK-message is propagated to all neighbors (line16) and an acquaintance with the invocator is stored in the global variable ACTIVATOR (lines 6, 14). In either case the agent replaces the current facet by the facet MARKED (lines 17, 22). In this facet the operation acts differently: further MARK-messages are acknowledged immediately by an echo (line 27), whereas the echo to the activator is only sent after all neighboring agents have sent their echoes (line 32).

C. Implementation of CSSA

The first ideas of a sequential version of CSSA originated in 1977 [5]; a concurrent version [12] was first implemented for a multicomputer simulation system which served as a testbed for some small programs [36]. Early experiments and careful examination of other high-level distributed programming languages such as SR [2], Starmod [8], PLITS [11], AMPL [9], Argus [21], NIL [35], and Act1 [19] resulted in some revisions of the CSSA language. As part of the INCAS project, a compiler running under UNIX and generating code for a virtual stack machine was realized. The run-time environment which consists of the virtual machine and an elaborated set of specific operating system functions is realized in LADY. These functions are provided by encapsulating each single agent in a separate team object of type *agent server.*

Fig. 7 sketches the principal structure of such a team. One process is responsible for providing the virtual machine (realizing the run-time environment) and executing the compiled code of the agent. Buffering of incoming and outgoing messages is performed by dedicated processes

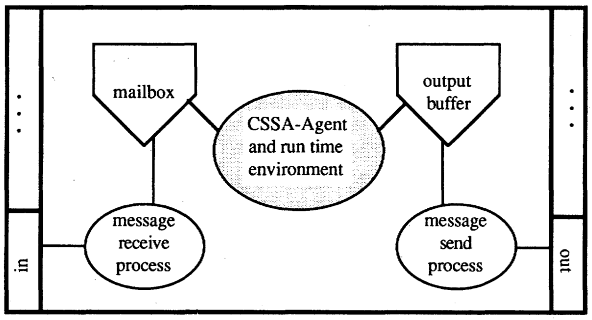


Fig. 7. A team of type "agent-server."

and two monitors. A CSSA message is packed into fixed-length LADY messages and the agent server processes take care of synchronization and flow control to guarantee reliable communication of CSSA messages. Other processes and monitors not shown in the figure are responsible for local I/O, memory management, debugging capabilities, pattern-matching, and administration of the mailbox.

The port interfaces are used to connect the agent server team to other agent servers, the interface agent, and various LADY teams realizing other functions of the distributed operating system (e.g., window management, configuration management, and file administration). Corresponding to the creation and deletion of CSSA agents, agent servers can be created or deleted. Depending on the operating system's management strategy, unused agent servers can also be preserved for subsequent agent creations. A comparison to a recently completed implementation of CSSA on a network of UNIX systems shows that our scheme of agent creation is several times more efficient than process creation under UNIX.

D. Applications and Experience with CSSA

At present, few distributed programming languages are implemented, and little experience of their use has been acquired. In order to evaluate the language, its underlying model, and the induced programming methodology, several *trial problems* have been implemented:

- A *concurrent scene labeling* system with a distributed constraint propagation algorithm [26].
- A distributed system for the solution of *word puzzles* (e.g., DONALD + GERALD = ROBERT) according to an idea by Kornfeld [18]. This problem gave rise to the development and analysis of several algorithms for solutions to the distributed termination problem [24].
- A *master-mind game* playing system. A game playing agent has several advisor-agents with different strategic behavior. The problem is to guarantee a coherent behavior in a loosely coupled system of concurrently active agents.
- A distributed *calendar and appointment* system. The calendar agents negotiate the appointments and try to fix the schedules.

According to the actor methodology, the applications have been organized as a society of communicating problem-solving experts, which cooperate to achieve their common goal. The experiments have been conducted to investigate the following questions: What classes of problems are suitable for execution in a distributed environment? What patterns of communication are most appropriate? Which organizational structures of processes are appropriate (e.g., hierarchies, heterarchies, client-server relations)? How can autonomous processes be coordinated in a decentralized way to ensure coherent behavior? Although it is still too early for an exhaustive discussion and more experience is needed, we try to give some preliminary answers in Section VI.

V. Monitoring, Testing, and Debugging

Existing concepts and methodologies for distributed computer systems generally demonstrate a substantial lack of methods and tools for monitoring, testing, and debugging these systems. Since distributed systems include asynchronous parallel processes, they show a nondeterministic and nonreproducible behavior. In many cases, subtle and sporadic errors are caused by improper synchronization among processes or race conditions. Since communication among processes introduces significant delays, and since processes run on different processors, there is a lack of adequate central control, precise global time and accurate global scale. Therefore, it is difficult to detect abnormal program behavior and to localize the erroneous processes.

We have designed a *distributed test methodology DTM* [14] which is integrated into the INCAS system. During implementation, the test tools allow users to monitor and control the tested system at different problem-oriented levels. During operation, the test system *permanently* monitors system behavior and measures system performance. The immense amount of information is condensed in easy-to-read charts and graphs on a high resolution graphic screen. The actual team network which can dynamically change during the system's lifetime is depicted graphically. To gain permanent insights into the running system, the test system provides statistical information about the monitored system (e.g., number of messages received or sent by each port, process elapsed and blocked times, communication system and operating system elapsed times and procedure running times).

The realization of these concepts is accomplished by a *test and measurement processor TMP* [15] which is part of each node of our multicomputer system. The TMP is able to run the local parts of the test and measurement software both independently and concurrently with the execution of the measured system. Each TMP consists of a local processor, a local memory, a local I/O unit, and a component which collects *test and measurement events*. The TMP's are connected to a central test station via a separate local network. The main idea of efficient execution time monitoring is that software in the measured system marks significant events but these events are categorized, processed, and displayed by dedicated hardware. (Fig. 8). With the aid of compiler information, the test

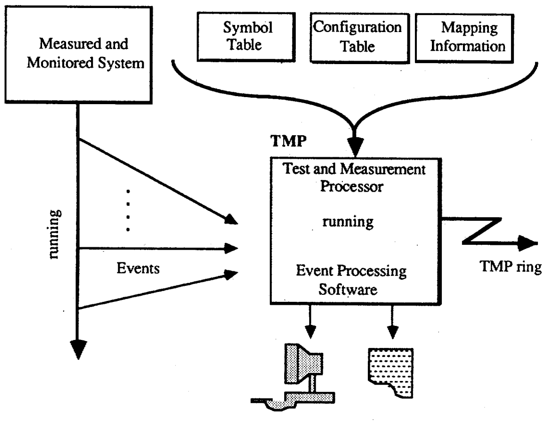

Measured and Monitored System
Symbol Table
Configuration Table
Mapping Information
TMP
Test and Measurement Processor
running
Event Processing Software
TMP ring
running
Events

Fig. 8. The test and measurement principles.

software is able to interpret events and present them in an application-oriented manner.

Preliminary analysis results show that the overhead to be expected will be *lower than 0.1 percent* for typical monitoring and performance measuring tasks. Since the overhead is negligible and the test and measurement events are permanent in the system, our test hardware and software do not change system behavior and do not significantly slow down system performance. The system always behaves the same whether the TMP is actively monitoring or not. That justifies our underlying philosophy to view testing and performance measurements in the operational state not as an occasional activity, but as an ongoing process that is inseparable from the system. The test hardware represents only about one tenth of the cost of our distributed testbed.

VI. Discussion and Lessons Learned

After six years of conceptual work, design, and implementation, which resulted in an operational prototype system, it seems worthwhile to summarize our experiences.

The INCAS project puts its emphasis on structuring aspects and design methodologies. Although the implementation and development of languages is expensive as far as the development of compilers and run-time environments is concerned, we consider it a mandatory prerequisite for the experimentation based development of good *structuring concepts*. The ad hoc extension of existing languages with additional constructs would have prohibited the adequate integration of structuring mechanisms such as facets and operations in CSSA or typed ports, buses, teams, and systems in LADY. The extensive use of these languages in various prototype applications has greatly improved our understanding about how to structure distributed systems.

Since the application layer and the operating system layer of distributed systems have rather different characteristics, we decided to implement *two different languages*. Both languages had already been used in previous projects and substantial experience had been gained. However, several modifications to the languages were necessary. Experience has indicated that our decision to support two languages which are well suited for their specific purposes but differ substantially in their underlying computational and communication models was appropriate and feasible.

Two general goals guided our design decisions concerning the *language features:*

- to provide a flexible basis for the implementation of arbitrary operating systems using high level structuring means, but to avoid any language constructs which require extensive support by lower layers of the system.
- to free users at the application level from having to consider system details by providing an application language which supports a very abstract view of a distributed system, but to accept the implementation cost associated with this abstraction.

Our experience has shown that no general preference to synchronous or asynchronous communication or implicit or explicit message receipt can be given. The appropriate *communication scheme* is highly dependent on the application. Message driven activation of operations and reliable asynchronous communication is very attractive for a high-level abstract programming language, but it requires extensive buffer management and flow control mechanisms. For system implementation languages such as LADY, a synchronous communication scheme is therefore more appropriate.

The *multicast mechanisms* in LADY and CSSA differ in their support of reliability and their capabilities of managing multicast groups. In CSSA, the sending agent has to be acquainted with all agents it wants to address by a multicast message. For this reason, CSSA can only take advantage of one feature of multicasts: the multiple sending of the same message to several distinct receivers is avoided. In LADY, the sending team needs no information about the teams which are members of the multicast group. Any team may autonomously leave or join a multicast group without informing the other group members: the multicast message is potentially received by all teams which belong to the specified group. This distributed management of multcast groups in LADY is more adequate for distributed systems than the approach in CSSA. As a result, we intend to enhance the multicast capabilities of CSSA by supporting the receipt of a message by an unknown number of receivers.

In our operating system, multicast communication patterns range from completely unreliable to highly reliable multicasts, although for many cases an unreliable multicast is sufficient. The reliable multicast communication has been implemented in LADY by protocols based on the *logical bus concept*. This communication mechanism has proven to be extremely useful for the programming of distributed systems; in our projects it is currently used at least as frequently as the one-to-one communication mechanism.

The availability of *recursively definable system types* and *nested process clusters* distinguishes our design methodology from most other approaches. First versions of LADY did not offer recursive system types. Their introduction was promoted by students who were using LADY and complained about missing language features to impose higher level structures on a flat team net. LADY systems and teams contribute very much to the decrease in the complexity of distributed operating systems. The positive impact of the structuring means of LADY on dealing with the structural software complexity during design and maintenance is documented in [30].

In order to have a first prototype available early in the project, *efficiency* was not our primary concern when building the run-time environments and the compilers. Our current LADY Support System is an adapted version having been used in the previous project on a distinct hardware. For this reason, up to now we cannot present any significant results about the efficiency of our system. During the last year we have designed a completely new LADY Support System and improved the code generation of the LADY compiler. The new system is currently under test and we expect substantial performance improvements. We plan extensive experiments using our measurement tools to gain detailed insights into the system behavior and to derive quantitative and qualitative assessments about the performance of the new system.

The availability of adequate *debugging and measurement tools* has proven to be crucial. This field is currently receiving much more attention than was originally anticipated. The concept of the CSSA interface agent seems to be particularly useful since it provides an integrated set of high-level debugging tools at the application level and facilitates an incremental development of distributed systems. Messages sent to agents not yet programmed can be handled interactively by the user.

Only a complete programming environment with test facilities and languages tailored to the specific application needs will convince software developers to use distributed systems. We conclude that without suitable methodologies for the design and programming of distributed systems, the potential benefits of large parallel and distributed systems cannot be fully exploited. The implementation of a prototype multicomputer system as an experimental testbed for research in distributed computing was a major step towards gaining more experience.

VII. Project Status and Future Work

A first prototype of a complete system including compilers for LADY and CSSA, the LADY Support System, a distributed operating system for CSSA, and a set of small application programs, has been available since the summer of 1985. Parts of a programming environment such as a library system have been realized. Currently our hardware-based monitoring tool is being tested; it will enable us to measure performance and other characteristics of our prototype.

The LADY programming methodology is presently applied in a joint research project between our group and the data base group in our department aiming at the design of a distributed database system for nonstandard applications.

Future work in the operating systems area will be devoted to:

- Automated mapping and load balancing mechanisms
- Dynamic system reconfiguration strategies
- Exploration of fault tolerant structures

In the application area, a comfortable interactive programming interface for CSSA using the concept of interface agents is being developed. Furthermore, our interest will shift from problems concerning programming languages for distributed systems to general methods and frameworks for distributed application programming and problem solving. In order to gain further experiences with distributed programming methodologies, more challenging applications will be considered.

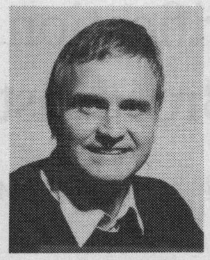
Acknowledgment

We want to acknowledge the many contributions which all the members of the INCAS project have made to the work presented in this paper, especially C. Beilken, P. Buhler, T. Gauweiler, R. Massar, R. Reske, F.-J. Stamen, R. Wagner, and B. Weitz. In addition, we want to thank all the students who helped to implement the system and keep it running. Finally, we want to thank C. Loggia Ramsey and R. Jeffery for their helpful comments on the final version of this paper, and the anonymous reviewers for helping us to express our ideas better.

References

[1] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden system: A technical review," *IEEE Trans. Software Eng.*, vol. SE-11, no. 1, pp. 43-59, Jan. 1985.
[2] G. Andrews, "The distributed programming language SR—Mechanisms, design and implementation," *Software—Practice and Experience*, vol. 12, pp. 719-753, 1982.
[3] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *Comput. Surveys*, vol. 15, no. 1, Mar. 1983.
[4] C. Beilken and F. Mattern, "The distributed programming language CSSA—A very short introduction," Dep. Comput. Sci., Univ. Kaiserslautern, Internal Rep. 123/85, 1985.
[5] H.-P. Boehm, H. L. Fischer, and P. Raulefs, "CSSA—Language concepts and programming methodology," *SIGPLAN Notices*, vol. 12, no. 8, pp. 100-108, 1977.
[6] E. J. H. Chang, "Echo algorithms: Depth parallel operations on general graphs," *IEEE Trans. Software Eng.*, vol. SE-8, no. 4, pp. 391-401, 1982.
[7] D. Cheriton, "The V Kernel—A software base for distributed systems," *IEEE Software*, pp. 19-42, Apr. 1984.
[8] R. P. Cook, "Star-MOD—A language for distributed programming," *IEEE Trans. Software Eng.*, vol. SE-6, no. 6, pp. 563-571, 1980.
[9] R. B. Dannenberg, "AMPL: Design, implementation, and evaluation of a multiprocessor language," Dep. Comput. Sci., Carnegie-Mellon Univ., Tech. Rep., 1981.
[10] D. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer—Design and implementation experience," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 553, 1984.
[11] J. A. Feldman, "High level programming for distributed computing," *Commun. ACM*, vol. 22, no. 6, 1979.
[12] H. L. Fischer and P. Raulefs, "Design rationale for the interactive programming language CSSA for asynchronous multiprocessor systems," Institut für Informatik III, Univ. Bonn, Memo. Seki-BN-79-09, 1979.
[13] K. Frenkel, "Evaluating two massively parallel machines," *Commun. ACM*, vol. 29, no. 8, pp. 752-758, 1986.
[14] D. Haban, "DTM—A distributed test methodology," in *Proc. 6th Symp. Reliability in Distributed Software and Database Systems*, 1987.

[15] D. Haban and D. Wybranietz, ''Hardware supported monitoring in distributed computer systems,'' Univ. Kaiserslautern, Tech. Rep. 23/86, SFB 124, 1986.
[16] C. Hewitt, ''Viewing control structures as patterns of passing messages,'' *Artificial Intell.*, vol. 8, pp. 323–364, 1977.
[17] J. Kramer, J. Magee, M. Sloman, and A. Lister, ''CONIC—An integrated approach to distributed computer control systems,'' *IEE Proc.*, vol. 130, part E, no. 1, pp. 1–10, 1983.
[18] W. A. Kornfeld, ''The use of parallelism to implement a heuristic search,'' in *Proc. 7th IJCAI*, 1981, pp. 565–580.
[19] H. Lieberman, ''A Preview of Act1,'' Massachusetts Inst. Technol., AI Memo. 625, 1981.
[20] B. Liskov, ''Primitives for distributed computing,'' in *Proc. 7th Symp. Operating System Principles*, 1979, pp. 33–42.
[21] B. Liskov and R. Scheifler, ''Guardians and actions: Linguistic support for robust, distributed programs,'' *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 381–404, 1983.
[22] B. Liskov, M. Herlihy, and L. Gilbert, ''Limitations of synchronous communication with static process structure in languages for distributed computing,'' Dep. Comput. Sci., Carnegie-Mellon Univ., Tech. Rep. CMU-CS-85-168, 1985.
[23] R. Massar, ''LADY (Language for Distributed Systems)—Design and implementation of a language for distributed systems'' (in German), Ph.D. dissertation, Dep. Comput. Sci., Univ. Kaiserslautern, 1984.
[24] F. Mattern, ''New algorithms for distributed termination detection in asynchronous message passing systems,'' Univ. Kaiserslautern, Tech. Rep. 42/85, SFB 124, 1985.
[25] R. F. Rashid and G. G. Robertson, ''Accent: A communication oriented network operating system kernel,'' in *Proc. 8th Symp. Operating System Principles*, 1981, pp. 64–75.
[26] M. Reinfrank, ''SCENELAB—Scene labeling by a society of agents—A distributed constraint propagation system,'' Dep. Comput. Sci., Univ. Kaiserslautern, Memo. SEKI-85-06, 1985.
[27] H. D. Rombach, ''Quantitative evaluation of software quality characteristics based on software structure,'' (in German), Ph.D. dissertation, Dep. Comput. Sci., Univ. Kaiserslautern, 1984.
[28] ——, ''Software design metrics for maintenance,'' in *Proc. Ninth Annu. Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD, 1984, pp. 100–135.
[29] ——, ''The multicomputer project INCAS,'' in *Proc. Pacific Computer Communications Symp.*, Seoul, Korea, 1985, pp. 151–163.
[30] ——, ''A controlled experiment on the impact of software structure on maintainability,'' *IEEE Trans. Software Eng.*, vol. SE-13, no. 3, pp. 344–354, 1987.
[31] C. L. Seitz, ''The cosmic cube,'' *Commun. ACM*, vol. 28, no. 1, pp. 22–33, 1985.
[32] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, ''Cm*—A modular multi-microprocessor,'' in *Proc. AFIPS Conf.*, vol. 46, 1977, pp. 637–644.
[33] Standard Microsystems Corporation, ''COM 9026—Local area network controller LANC,'' DATAPOINT, 1984.
[34] F.-J. Stamen, ''Configuration management in distributed operating systems,'' (in German), in *Architektur und Betrieb von Rechensystemen*, NTG-Fachberichte 92, VDE-Verlag GmbH, 1986.
[35] R. Strom and S. Yemini, ''The NIL distributed systems programming language: A status report,'' *SIGPLAN Notices*, vol. 20, no. 5, pp. 36–43, 1985.
[36] H. Voss, ''Programming in a distributed environment: A collection of CSSA examples,'' Dep. Comput. Sci., Univ. Kaiserslautern, Memo. SEKI-82-01, 1982.
[37] L. Wittie, R. Curtis, and A. Frank, ''MICRONET/MICROS—A network computer system for distributed applications,'' in *Multicomputers and Image Processing*, K. Preston and L. Uhr, Eds. New York: Academic, 1982.
[38] D. Wybranietz, D. Haban, and P. Buhler, ''Some extensions of the LADY language,'' Univ. Kaiserslautern, Tech. Rep. 28/86, SFB 124, 1986.
[39] M. Weiser, S. Kogge, M. McElvany, R. Pierson, R. Post, and A. Thareja, ''Status and performance of the ZMOB parallel processing system,'' *IEEE CompCon Conf.*, San Francisco, CA, 1985.
[40] D. Wybranietz and R. Massar, ''An overview of LADY—A language for the implementation of distributed operating systems,'' Univ. Kaiserslautern, Tech. Rep. 12/85, SFB 124, 1985.
[41] H. Zimmermann, M. Guillemont, G. Morisset, and J.-S. Banino, ''CHORUS: A communication and processing architecture for distributed systems,'' INRIA, Rocquencourt, France, Tech. Rep. 328, 1984.



Jürgen Nehmer studied electrical engineering and received the Ph.D. degree in computer science from the Technical University of Karlsruhe, West Germany, in 1973.

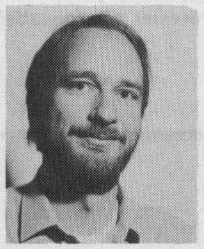
He is a Professor of Computer Science at the University of Kaiserslautern, West Germany, where he is engaged in teaching and research in the areas of distributed systems, operating systems, and software engineering. During 1971 he worked as a post-doc at the IBM Thomas J. Watson Research Center. Before joining the University of Kaiserslautern in 1979, he was a staff scientist at the Nuclear Research Center Karlsruhe doing research on real time systems.

Dr. Nehmer is a member of the IEEE Computer Society, the Association for Computing Machinery, and the German Computer Society (GI).

Dieter Haban received the Diploma in computer science from the University of Kaiserslautern, West Germany, in 1984 and is currently working toward the Ph.D. degree.

He is a Faculty Research Assistant in the Department of Computer Science at the University of Kaiserslautern. His current research interests include distributed systems, distributed testing, and computer graphics. His recent work has been involved in the design and development of test and measurement tools.

Mr. Haban is a member of the German Computer Society (GI).



Friedemann Mattern received the Diploma in computer science from the University of Bonn, West Germany, in 1983 and is currently working toward the Ph.D. degree.

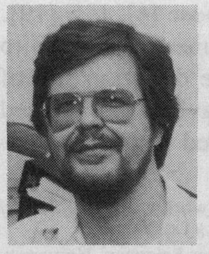
He is a Faculty Research Assistant in Computer Science at the University of Kaiserslautern. His primary research interests include distributed computing, programming language design, and compiler construction.

Mr. Mattern is a member of the IEEE Computer Society, the Association for Computing Machinery, and the German Computer Society (GI).

Dieter Wybranietz received the Diploma degree in computer science from the University of Bonn, West Germany, in 1983. At present, he is working toward the Ph.D. degree.

He is a Faculty Research Assistant in Computer Science at the University of Kaiserslautern, West Germany. His current research interests are simulation of parallel processes, programming language design, and distributed systems with special focus on operating systems and performance measurements.

Mr. Wybranietz is a member of the German Computer Society (GI).



H. Dieter Rombach received the B.S. degree in mathematics from the University of Karlsruhe, West Germany, in 1975, the M.S. degree in mathematics and computer science from the University of Karlsruhe in 1978, and the Ph.D. degree in computer science from the University of Kaiserslautern, West Germany, in 1984.

He is an Assistant Professor in the Department of Computer Science at the University of Maryland, College Park. His research interests include software methodologies, measurement of the software process and resulting products, and distributed systems. From 1978 to 1979 he was a Research Staff Member at the Nuclear Research Center, Karlsruhe, doing research on distributed systems for real time applications.

Dr. Rombach is a member of the IEEE Computer Society, the Association for Computing Machinery, and the German Computer Society (GI).