- A PORTABLE, HARDWARE AND LANGUAGE INDEPENDENT ARCHITECTURE FOR COMMUNICATION (PCA) -

Contribution to:

Activity Task 2.5

-- System Concept for Advanced CIM/AI Controller --

Title:

A PORTABLE, HARDWARE AND LANGUAGE INDEPENDENT ARCHITECTURE FOR COMMUNICATION (PCA)

Author/Company/Copyright:

Jörg - Ingo Jakob

Achievements:

A portable, hardware and language independent architecture (called PCA) for continuous exchange of messages in a heterogenous computer network was conveived. In the context of EP2434, PCA is used for implementation of a distributed heterogenous CIM/AI network. A knowledge-based application is reported.

Summary:

A portable, hardware and language independent architecture for continuous exchange of messages in a heterogenous computer network is described. The most difficult part is to overcome resource contention of message sender and receiver: under some operating systems, the receiver wants to read the message before the sender has finished writing it. Synchronization of these asynchronous tasks can be achieved by meeting two basic preconditions: (1) both computers are able to read, write and share files on a virtual disk or directory, (2) a non-interruptible rename operation is available for both computers to operate on the virtual disk or directory. It is shown how a primitive portable message passing mechanism (called PCA, the Portable Communication Architecture) can be fully implemented in high level languages, virtually without recoursing to any low level operating system or net management routines. Having implemented PCA on two or more computers connected by a network, it is then possible to implement message passing, mailboxes and remote computation mechanisms easily as variations of the PCA. Applications and transmission speed measurements are reported.

INDEX

A Portable, Hardware and Language Independent Architecture for Communication

JÖRG-INGO JAKOB

Philips GmbH Forschungslaboratorium Hamburg, Vogt-Kölln-Straße 30, D - W2000 Hamburg 54, Germany

SUMMARY

A portable, hardware and language independent architecture for continuous exchange of messages in a heterogenous computer network is described. The most difficult part is to overcome resource contention of message sender and receiver: under some operating systems, the receiver wants to read the message before the sender has finished writing it. Synchronization of these asynchronous tasks can be achieved by meeting two basic preconditions: (1) both computers are able to read, write and share files on a virtual disk or directory, (2) a non-interruptible rename operation is available for both computers to operate on the virtual disk or directory. It is shown how a primitive portable message passing mechanism (called PCA, the Portable Communication Architecture) can be fully implemented in high level languages, virtually without recoursing to any low level operating system or net management routines. Having implemented PCA on two or more computers connected by a network, it is then possible to implement message passing, mailboxes and remote computation mechanisms easily as variations of the PCA. Applications and transmission speed measurements are reported.

KEY WORDS Communication Architecture Heterogenous computer network Portability

INTRODUCTION

During implementation of CIM and AI related software in ESPRIT 2434[1], a need for a hardware and language independent communication architecture became obvious. This architecture would enable us to implement heterogenous distributed AI systems with a minimum of effort and a maximum of standardization and portability amongst software and hardware components involved. The resulting architecture, called Portable Communication Architecture (PCA), is introduced here. In a sense, each instantiation of PCA on one computer implements a communicating sequential process (C. A. R. Hoare 1978[2] and *Figure 1* below).

PCA: A HARDWARE AND LANGUAGE INDEPENDENT COMMUNICATION ARCHITECTURE

Different Types of Communication Architectures

Hardware independence of a communication architecture we define as the architecture's ability to not rely on any specific underlying computer or communication hardware for the achievement of communication between two or more programs.

Software independence of a communication architecture we define as the architecture's ability not to rely on any specific underlying language or operating system software for the achievement of communication between two or more programs

There are a number of "common" approaches to achieving hardware and/or software independence in distributed, heterogenous computer systems. These approaches can be classified into vendor-dependent, vendor-independent and vendor-spanning approaches.

An example for a *vendor-dependent approach* (taken from ESPRIT 2434) is a collection of VAXes which monitor a line for manufacturing of electronics consumer goods in one of Philips' plants in Germany. These VAXes communicate on the hardware level by

means of DEC's DECNet (DEC's own communication architecture based on Ethernet[3]). On the software level, the communication is fully handled by the operating system (in our case, VAX/VMS). VAX/VMS provides several ways of communication between VAX computers: exchanging files, or exchanging messages by a mailbox mechanism. There is nothing wrong with vendor-dependent approaches, except (1) when a computer of another vendor shall be connected to this net, or (2) when a program shall be ported to another computer from another manufacturer. In both cases, difficulties arise due to lack of compatibility of equipment, which can be resolved (at a minimum expense) by rewriting parts of the program(s) which use(s) the vendor-dependent net. However, costs can be considerably higher when extra hardware must be purchased and/or when special communication drivers need to be purchased or written.

Examples for *vendor-independent approaches* are all efforts to standardize communication protocols (most prominently, the ISO Open Systems Interconnection (OSI) reference model[4]; and in the area of manufacturing most notably MAP[5] as an instantiation of the ISO-OSI reference model). The ISO reference model distinguishes seven layers (physical layer, data link layer, network layer, transport layer, session layer, presentation layer, application layer). The number of layers (likewise: the amount of documentation) associated with the ISO reference model and its derivatives indicate the difficulties of fully implementing such all-embracing protocols.

On the other end of vendor-independent approaches there are very simple protocols (e. g. Kermit[6]) which basically need only a small communication program on each computer, and a serial link between them. These kinds of simple protocols, however, are typically very limited by their transmission speeds, safety and lack of usefulness (in the sense that they usually do not allow for "automatic" transmission of data, or that they are not properly integrated into an operating system).

A *vendor-spanning approach* is any software available on the market which is explicitely intended to enable communication between two operating systems. Examples: PCSA from Digital Equipment which serves as a bridge between VAX/VMS and MS-DOS; Novell Netware (version 3.1 and higher), which serves as a bridge between MS-DOS and the

UNIX world (TCP/IP); Symbolics' GENERA 7.2 operating system, which has a built-in communications capability obeying the DECNet standard, emulating a VAX to the rest of the connected DECNet; etc.

By contrast, the hardware and language independent communication architecture we are describing here is a *pragmatic approach*. Only the most necessary components of the above approaches are borrowed to maintain high level communication between two or more computers, and to achieve portability of code.

The central idea of our approach is to identify those components of a communication architecture which are necessary in order to establish high level communication between two or more computers, *and* which at the same time are primitives available in (nearly) every operating system and high level programming language. If we succeed in finding a non-empty intersection of components and primitives which is powerful enough to establish high level communication, we only need to find a standardized algorithm which can then be implemented in arbitrary programming languages and/or operating systems. Our main concern here is standardization and portability. Which layers our pragmatic approach actually uses of e. g. the ISO reference model is of not much concern to us, because our pragmatic approach ensures proper operation no matter what the underlying software, operating system or hardware is. In other words, we are looking for a primitive "virtual communication machine", which we then implement in various computing environments, but whose user (i. e., programmer) interface is always (nearly) the same.

The Portable Communication Architecture Approach (Virtual Communication Machine)

The following components can be identified as belonging to the class of high level communication objects *and* to the class of primitive objects available in (nearly) every operating system and high level programming language (*Table 1*):

Table 1: Necessary Components for the Virtual Communication Machine

- ASCII text files (i. e., messages)*
- a hierarchical file system (using directories and disks)
- primitive operations on files (e. g. OPEN, READ, WRITE, CLOSE, RENAME, COPY, DELETE)
- real or virtual disks or directories
- sharing of disks or directories between computers**
- a non-interruptible RENAME or COPY operation on files .

Table 1 can be used as a checklist to verify if a particular combination of hardware and software possibly will support PCA, the Portable (or *Pragmatic*) Communication Architecture.

The basic idea of the pragmatic communication architecture approach can then be formulated in the following algorithm. We start in a moment where a message (which is originating from Computer B) shall be received by Computer A, which then processes the message received and sends a response (i. e., another message) to Computer B *(Figure 1* and *Algorithm 1)*.

Figure 1 summarizes the Pragmatic Communication Architecture as two interacting state machines. Synchronization of both asynchronous processes is achieved by the interplay of the SUBMIT and WAIT operations.

---

* Binary files are suitable for messages as well. However, their portability is more restricted.

** In practice, this mostly corresponds to the availability of network software (and underlying network hardware) which simulate a virtual disk or directory. This is where all underlying layers (confer the OSI protocol) are hidden. Most network software or operating systems provide a virtual disk or directory, and this is why PCA is a portable architecture.

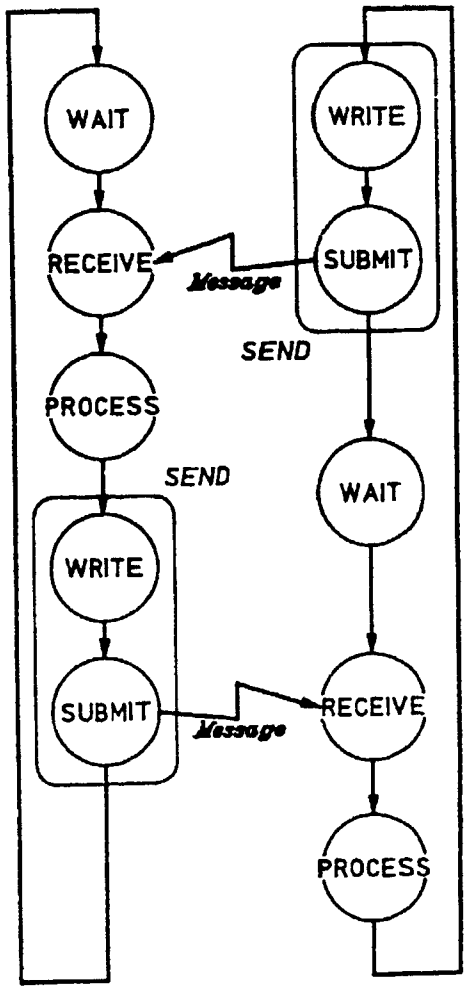
WAIT
RECEIVE
PROCESS
SEND
WRITE
SUBMIT
Message
WRITE
SUBMIT
SEND
WAIT
RECEIVE
PROCESS
Message
Computer A
Computer B

Fig.:1

(1) WAIT FOR INCOMING MESSAGE: An incoming message is always submitted to the receiver (*here:* Computer A) as a file of a particular name *Name1*, which has to appear on a particular disk and directory. Receiver (*here:* Computer A) waits for an indefinite period of time until the incoming message (the file) *Name1* appears.

(2) RECEIVE MESSAGE: Receiver (*here:* Computer A) reads the incoming message (the file) *Name1*.

(3) PROCESS MESSAGE: Process the incoming message (*here:* Computer A computes a response to the message, and computes associated side effects, if any).

(4) SEND OUTGOING MESSAGE:

(4a) WRITE OUTGOING MESSAGE: An outgoing message to the receiver (*here:* Computer B) is always written to a temporary file of a particular name *Name2* [different from the file names *Name1* and *Name3* of steps (1) and (4b)] at first. The temporary file has to appear on a particular disk or directory [often identical to the disk or directory of step (1)].

(4b) SUBMIT OUTGOING MESSAGE: To ensure that the message is not read before its writing (sending) has been finished, the message was written to a temporary file *Name2*. After completion of the outgoing message, the temporary file is renamed or copied to a file of a particular name *Name3* [different from the file names *Name1* and *Name2* of steps (1) and (4a)], which can be found by the receiver (*here:* Computer B).***

Algorithm 1: Sending a message from computer A to computer B using the proposed architecture

Algorithm 1 can as well be used to send a message from Computer B to Computer A (by simply substituting Computer B for Computer A, and Computer A for Computer B; and by substituting file name *Name1* by *Name3*, and *Name3* by *Name1*). As a net effect, computers A and B may be sending each other messages until one of them refuses to send or receive a message.

As can be seen, the components used in *Algorithm 1* are those of *Table 1*. Thus the algorithm can be easily implemented using a high level programming language, running on an arbitrary operating system.

*** This step is necessary to disable any efforts of the receiver (here: Computer B) to read the message (file) before it was properly and completely written. Appearance of this difficulty depends on the underlying operating system.

Verification of feasibility of *Algorithm 1* under the constraints of *Table 1* is particularly straightforward for high level languages which possess a well-defined file system interface. In the context of AI, the most notable example is Common Lisp's file system interface[7]* . The standards of other high level language prefer not to define a file system interface having the same consistency and expressiveness as Common Lisp's (example: ISO Pascal). However, as common sense (and the Common Lisp standard !) safeguard, all the primitive operations of Table 1 are available on commercial ISO Pascal implementations (since they are available from the underlying operating system).

Figure 2 summarizes the hardware aspects of the Portable Communication Architecture: *Figure 2a* depicts the arrangement of two computers [arrangement marked by "2" in Table 2 below]. The disk belongs either to Computer A or Computer B; *Figure 2b* depicts the arrangement of three computers, using a third Computer C as a virtual disk [marked by "3" in Table 2 below]; *Figure 2c* depicts the use of PCA one computer (either multitasking or multiuser) [arrangement marked by "1" in table 2 below].

Feasibility Study of the Portable Communication Architecture

We implemented *Algorithm 1* in various languages and under various operating systems, using various network software packages. Printed representations of integer numbers were exchanged as messages here. (However, there is no limitation on the type and size of messages as long as they are exchanged as ASCII text files). As a side effect, each communication program is incrementing the incoming integer by one before sending it out again.

Implementation details are to be found in chapter 'Common Lisp Implementation of PCA' below.

---

* With one possible exception: as experience indicates, under some operating systems there may be difficulties using Common Lisp's RENAME-FILE operation when operating via net on remote disks. An auxiliary COPY-FILE operation is not available under the Common Lisp standard, but can be easily implemented from the other primitive operations available under the Common Lisp file system interface.

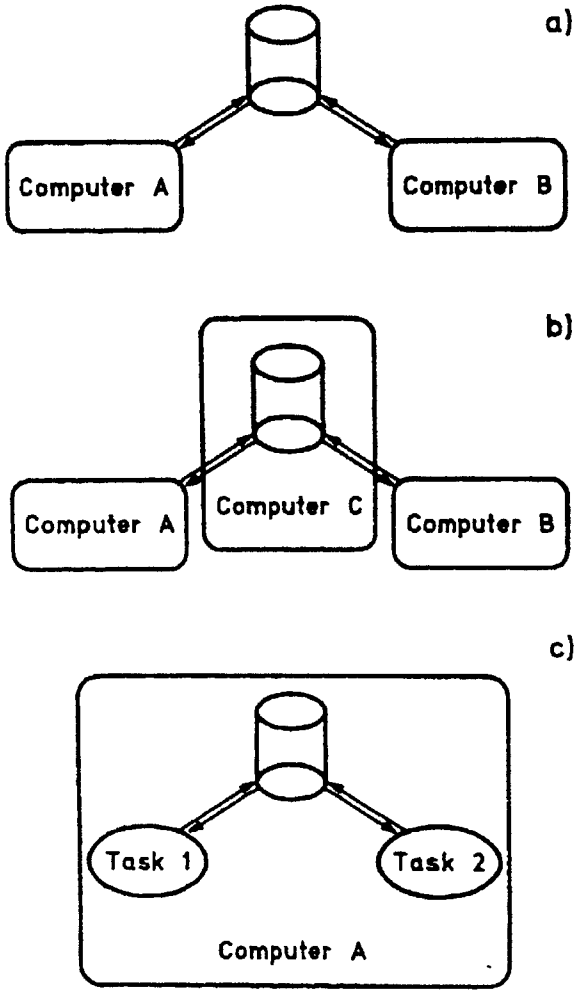
a)
Computer A
Computer B
b)
Computer A
Computer C
Computer B
c)
Task 1
Task 2
Computer A

Fig.: 2

Table 2 summaries the implementations and communications we successfully established between various high level languages, operating systems and computers:

Table 2: Successful communications implemented by the PCA. Numbers 1, 2, 3 indicate number of physical computers involved in test communication (cf. Figure 2). A double dash indicates an infeasible communication.

	Computer B			
	GENERA	MS-DOS	VAX/VMS	WINDOWS
Computer A				
GENERA 7.2	1, 2	3	2, 3	3
MS-DOS 3.3		--	2, 3	--
VAX/VMS 5.2			1, 2, 3	2, 3
WINDOWS 3.0*				1

The communications reported in *Table 2* do furthermore not depend on the high level language used for implementation. Thus, a C-based PCA may communicate with a Lisp-based PCA without knowing or recognizing it.

From the experiences reported in *Table 2*, we are confident that *Algorithm 1* may be implemented in nearly any high level language under nearly any operating system.

Measurements

Several measurements were conducted using PCA on a standalone computer (*Figure 2c*) and on an Ethernet (*Figures 2a* and *2b*). Results are reported in the following tables. All these machines are connected by the same physical Ethernet. All measurements were performed at times were the Ethernet transmission load was very low.

Measurement 1 measures transmission speed of the hardware arrangement of Figure 2c. PCA is used to measure the speed of message exchange between two tasks running on the *same* computer, directly implementing *Figure 1*. Message exchange is always done by actually writing onto the default system harddisk (no RAM disks etc. are used). For

* Inter-task communication using PCA was thus established without recoursing to Microsoft Windows 3.0's special Dynamic Data Exchange (DDE) mechanism[8].

measurement purposes, communication is terminated after 50 messages were sent and received (25 from Task 1 to Task 2, and 25 from Task 2 to Task 1). The total time needed for sending and receiving these 50 messages is taken. Each message consisting of a single integer (cf. *Listing 1* below), which is incremented by the receiver before being sent out again. Care was taken that no other tasks (except the underlying operating system) are executing. Standard operating system parameters have been taken, no tuning for speed took place. During the measurement, the delay time of function (procedure) MakeDelay (cf. *Listing 1*) was always set to zero.

Measurement 1 typically measures the sum of task switch times and the time it takes to write/read onto the local hard disk. Other times can be disregarded. Therefore, it should not matter whether these measurements were performed using a C, Pascal or Lisp based version of PCA. Results:

COMPAQ DESKPRO 486/25, Windows 3.0, Turbo Pascal 6.0:	0'10"
VAXStation 3500, VAX/VMS 4.7, VAX Pascal 3.6:	0'37"
Symbolics 3645, Genera 7.2, Symbolics Zetalisp:	1'36"
VAXStation 3500, VAX/VMS 4.7, VAXLisp 2.2:	5'00".

Discussion: "Simple" multitasking operating systems are the fastest at inter-task communication. The results of VAXLisp are disappointing; obviously, the bottleneck is not the operating system or hard disk, but the intermediate virtual Lisp machine that interprets the "binary" Lisp code. By contrast, the Symbolics Lisp Machine executes Lisp code in hardware.

Measurement 2 measures transmission speed of the hardware arrangement of *Figure 2a*. Only the measurement for Novell netware uses the arrangement of *Figure 2b* (i. e., a third computer as a virtual disk). The computers A, B, C are always of the same brand and use the same operating system. Otherwise this measurement is similar to Measurement 1. 50 messages are sent.

Measurement 2 typically measures the sum of synchronization times of the Ethernet, of transmission times via Ethernet, and the time it takes to write/read onto the hard disk. Other times can be disregarded. Therefore, it should not matter whether these measurements were performed using a C, Pascal or Lisp based version of PCA. Results:

No Name 386/33 PC-AT, PC-DOS 3.3, Turbo C++, Novell Netware 3.0:	0'03"
No Name 386/33 PC-AT, Windows 3.0/PC-DOS 3.3, Turbo C++, Novell Netware 3.0:	0'16"
VAXStation 3500 to VAXStation 3500, VAX/VMS 4.7/5.2, VAX Pascal 3.6:	2'21"
Symbolics 3640 to 3645, Genera 7.2, Symbolics Zetalisp:	3'25"
VAXStation 3500 to VAXStation 3500, VAX/VMS 4.7/5.2, VAXLisp 2.2:	7'58" .

Discussion: The transmission speed of Novell Netware 3.0 is surprising. Even when we take into account that DOS and Windows do no or little maintenance of operating system background tasks, this is somewhat astonishing, since all computers are connected to the same physical Ethernet* .

Measurements of transmission speeds of heterogenous systems (e. g., COMPAQ to VAX) were conducted but will not be reported. Thus we avoid blaming the wrong computer for slow speed.

COMMON LISP IMPLEMENTATION OF PCA

PCA was implemented in ISO Pascal, ANSI C and Common Lisp so far. Here only one implementation of the PCA is described. The Common Lisp instantiation of the PCA will be used for that purpose because it turns out to be the most concise and most readable PCA (due to the well-defined syntax and file system interface of Common Lisp).

The states of Figure 1 were implemented as follows (*Table 3*):

Table 3: Correspondence of PCA states and Lisp function names chosen

STATE	FUNCTION NAME
Wait	WaitForMessage(FileName)
Receive	ReceiveMessage(InFile)
Process	ProcessMessage(InFile)
Write	WriteMessage(TempFile)
Submit	SubmitMessage(TheFile OldFileName NewFileName)

* In case much longer files had been sent as messages, one should expect that the sum of times would equate each other asymptotically, since "physical" transmission time via net should dominate all other sum terms.

All function parameters ending with -Name are of Common Lisp data type string. All function parameters ending with -File are of Common Lisp data type stream. The Common Lisp source is to be found in *Listing 1*.

```
;;; ************** GLOBAL VARIABLES ******************

(defvar *Counter* nil)
(defvar *IOSuccessP* nil
 "Keeps track of success of last IO operation.")


;;; ************** AUXILIARY FUNCTIONS ****************

(defun MakeDelay ()
;; calls to MakeDelay take load from the network,
;; depending on the length of the delay chosen
 #+VAX (sleep 0.5)
 #+Symbolics (sleep 0.5)
 t)


;;; ************* FILE SYSTEM INTERFACE *************

(defun OpenAFile (FileName Mode)
;; returns the open stream when successful
 (setq *IOSuccessP*
  (case Mode
   (WriteMode
     (open FileName :direction :output :if-exists :supersede))
   (ReadMode
     (open FileName :if-does-not-exist nil))
   (ProbeMode
     (open FileName :direction :probe)))))

(defun CloseAFile (TheFile)
 (close TheFile))

(defun FileExistsP (FileName TheFile)
```

Listing 1 (continued) :PCA Implementation in Common Lisp

```
 (let ((Success nil))
  (setq TheFile (OpenAFile FileName 'ProbeMode))
  (setq Success *IOSuccessP*)
  (when Success (CloseAFile TheFile))
  Success))
(defun DeleteFile (FileName)
 (let ((TheFile nil))
  (when (FileExistsP FileName TheFile)
   (delete-file FileName)
  )))


;;; ********** MESSAGE SYSTEM INTERFACE ***********

(defun SubmitFile (TheFile OldFileName NewFileName)
;; Waits indefinitely until successful message submission.
;; Achieves desired synchronization.
 (loop
  (when (FileExistsP TheFile OldFilename)
   #+VAX (rename-file-by-copying OldFileName NewFilename)
   #-VAX (rename-file OldFileName NewFilename)
   (return t))))

(defun WaitForMessage (FileName)
;; Waits indefinitely for message. Returns stream when successful.
 (let ((Success nil))
  (loop
   (setq Success (OpenAFile FileName 'ReadMode))
   (unless Success (MakeDelay))
   (when Success (return t)))
  Success))

(defun ReceiveMessage (InFile)
;; assuming message is an integer (for demonstration purposes !)
  (setq *Counter* (read-from-string (read-line InFile))))

(defun SendMessage (TempFile)
;; assuming message is an integer (for demonstration purposes !)
 (format TempFile "~a~%" *Counter*))

(defun ProcessMessage ()
 ;; assuming message is an integer (for demonstration purposes !)
(incf *Counter*))
```

Listing 1: PCA Implementation in Common Lisp

The functions of the file system interface and the message system interface are designed in a way that they can easily be implemented in other programming languages in a very similar way. The Pascal and C implementations of the PCA both use functionally identical functions (procedures). Only the way return values are passed from a returning function (procedure) to its environment typically differs from language to language. Structurizing the PCA in a language-independent (and thereby portable) way was partly influenced by the concept of software "clichés" as presented in the book of Rich and Waters[9].

Complementary, *Listing 2* reports in detail about the code used for performing the above measurements.

Note that the following correspondences hold (cf. *Algorithm 1*): *Name1* = 'IN1.DAT', *Name2* = 'TEMP.DAT', *Name3* = 'IN2.DAT'. These file names (global variables `*InFileName1*`, `*TempFileName*`, `*InFileName2*`) are assembled in the main program.

When *Listing 1* and *Listing 2* are loaded into a Common Lisp system, the user must answer the question "START AS ONE=1, TWO=2 OR TEST=3 ?:". In case the user types '1', the program assumes that it shall act as Computer A (also: Task 1) of *Figure 2*; in case the user types '2', the program assumes that it shall act as Computer B (also: Task 2) of *Figure 2*. It is always necessary to load the (otherwise identical !) program two times on two computers (or as two independent tasks) to perform the above Measurements 1 and 2.

If the user types '3', the program is running a self-test. In this case, it locally runs on *one* computer only.

```
;;; ************** FURTHER GLOBAL VARIABLES ***************

#+VAX (defparameter *FileHeader*
            "999\"USER PASSWORD\"::[USER.NET]")
#+VAX (defparameter *FileName* "IN")
#+VAX (defparameter *FileExtension* ".DAT")
#+VAX (defparameter *TFileName* "TEMP.DAT")
#+VAX (defparameter *FileTrailer* ";")

#+Symbolics (defparameter *FileHeader* "s999:[USER.NET]")
#+Symbolics (defparameter *FileName* "IN")
#+Symbolics (defparameter *FileExtension* ".DAT")
#+Symbolics (defparameter *TFileName* "TEMP.DAT")
#+Symbolics (defparameter *FileTrailer* "")

(defvar *InFile* nil)
(defvar *OutFile* nil)
(defvar *TempFile* nil)
(defvar *FileName1* nil)
(defvar *FileName2* nil)
(defvar *TempFileName* nil)
(defvar *Limit* 50 "Number of messages to be sent.")


;;; ************** PCA STATE MACHINE ******************

(defun PerformDialogue (OneTwo)
 (let ((InFileName nil)
       (OutFileName nil))
  (case OneTwo
   (1
    (setq InFileName *FileName1*)
    (setq OutFileName *FileName2*))
   (2
    (setq InFileName *FileName2*)
    (setq OutFileName *FileName1*)))

  (setq *InFile* (WaitForMessage InFileName))
  (ReceiveMessage *InFile*)
  (format t " ~a~10d~%" InFileName *Counter*)
  (CloseAFile *InFile*)
  (DeleteFile InFileName)
  (ProcessMessage)
  (setq *TempFile*
   (OpenAFile *TempFileName* 'WriteMode))
  (SendMessage *TempFile*)
  (CloseAFile *TempFile*)
  (SubmitFile *TempFile* *TempFileName* OutFileName)))
```

Listing 2 (continued) : Additional code for measurements reported

```
;;; ************** MAIN PROGRAM ******************

  (format t "~%")
  (loop
   (format t " START AS ONE=1, TWO=2 OR TEST=3 ?: ")
   (setq *Choice* (read-line))
(when (member *Choice* '("1" "2" "3") :test #'string=)
 (return t)))
(format t "~%~%")

(setq *Counter* 0)

(setq *FileName1*
 (concatenate 'string *FileHeader* *FileName* "1" *FileExtension*
                      *FileTrailer*))
(setq *FileName2*
 (concatenate 'string *FileHeader* *FileName* "2" *FileExtension*
                      *FileTrailer*))
(setq *TempFileName*
 (concatenate 'string *FileHeader* *TFileName* *FileTrailer*))

(when (member *Choice* '("1" "3") :test #'string=)

   ;; start from scratch
   (DeleteFile *FileName1*)
   (DeleteFile *FileName2*)
   (DeleteFile *TempFileName*)

   ;; create file "IN1.DAT"
   (setq *TempFile*
    (OpenAFile *TempFileName* 'WriteMode))
   (format *TempFile* "~a~%" 0)
   (CloseAFile *TempFile*)
   (SubmitFile *TempFile* *TempFileName* *FileName1*))

  (time
   (loop

    (if (string= *Choice* "1") (PerformOneTwo 1)
     (if (string= *Choice* "2") (PerformOneTwo 2)
      (progn
       (PerformDialogue 1)
       (PerformDialogue 2))))

    (when (> *Counter* *Limit*) (return t))))
```

Listing 2 : Additional code for measurements reported

APPLICATIONS

Example 1: Distributed Artificial Intelligence

The author is professionally concerned with developing knowledge-based systems in the area of production planning and scheduling. We wrote a knowledge-based job shop scheduling expert system called AIPLANNER[10] which is currently available on VAX and Symbolics computer hardware. However, real factory applications require that one is fluent in many software and hardware languages, which motivated the design and realization of the PCA. Currently, we use PCA to connect PCs to the VAX host which is running the AIPLANNER job shop scheduling system. By using the PC as an intelligent graphics terminal, we can inquire AIPLANNER about manufacturing progress information, suggest changes to the current plan and use it as a data base. We are able to display manufacturing information using Microsoft Windows 3.0 based graphics, and to locally process manufacturing information on PCs when needed. We intend to use the cheap computing power of 486-based PCs to perform remote, knowledge-based optimization of the manufacturing plans. To that end, we currently are implementing a PC hardware accelerator board for performing planning and scheduling tasks using temporal logic as a description language[11]. Further plans include connecting AIPLANNER to a knowledge-based quality information system with the help of PCA; etc. There are many useful applications for PCA in diverse manufacturing environments.

Example 1 reports on a typical request sent to AIPLANNER from a remote PC, and the response sent by AIPLANNER. The request inquires on the current manufacturing plan of a workcell, which is returned within seconds. In reality, however, these "low level" details are hidden by a graphical user interface.

ASCII message send to AIPLANNER (running on a VAX under VAX/VMS) by a remote PC:

```
(get.all.orders.on.wc)
```

This message asks for all orders that are currently to be produced on the default workcell. Within a few seconds (depending on the current load of the net), AIPLANNER responds:

```
(((("A1.P681/90.E8474" P681/90 283 1 393.0 1)
("A2.P681/00R.E8484" P681/00R 500 1 398.0 2)
("A3.P681/60E.E8442" P681/60E 464 1 377.0 3)
("A4.P681/60E.E8474" P681/60E 200 1 393.0 4)
("A5.P681/60X.E8483" P681/60X 400 1 397.0 5)
("A6.P681/60.E8461" P681/60 294 1 386.0 6)
("A7.P681/60.E8473" P681/60 200 1 392.0 7)
("A8.P681/75.E8445" P681/75 90 1 380.0 8)
("A9.P681/79.E8462" P681/79 600 1 387.0 9)
("A10.P681/79.E8472" P681/79 600 1 391.0 10)
("A11.P682/00R.E8475" P682/00R 200 1 394.0 11)
("A12.P682/00.E8484" P682/00 200 1 398.0 12)
("A13.P685/02.E8464" P685/02 55 1 388.0 13)
("A14.P774/02R.E8462" P774/02R 239 1 387.0 14)
("A15.P774/02R.E8474" P774/02R 150 1 393.0 15)
("A16.P774/02.E8402" P774/02 4 1 357.0 16)
("A17.P794/02R.E8483" P794/02R 450 1 397.0 17)
("A18.P584/71.E8472" P584/71 600 1 391.0 18)
("A19.P584/71.E8474" P584/71 600 1 393.0 19)
("A20.P585/71.E8461" P585/71 600 1 386.0 20))
:OK)
```

Each sublist of the responds describes one order to be produced. The description is by name, type, amount, earliest start date and latest end date (relative to the current planning horizon).

Example 2: Message Passing Mechanism

Listing 1, using the primitive functions of *Algorithm 1*, is desribing a portable message passing mechanism. The mechanism is the underlying mechanism of Example 1. In this mechanism, both participants are sending a request, waiting indefinitely for an answer to this request, then process a new request from the answer received. This mechanism may be modified according to practical needs. More than two remote participants may be sending each other messages; or a request is sent, and the sender continues with other computations while it is waiting for an answer.

Example 3: Mailbox Mechanism

A mailbox mechanism can be implemented by modifying the message passing mechanism of Example 2. In a mailbox mechanism, a sender sends information (called mail) to a receiver. However, it is only loosely depending on receiving an answer (return mail) from the receiver. The sender might also decide to send several pieces of mail to the receiver without receiving (or processing) return mail itself. This means practically, the WAIT state of the PCA is relaxed, and messages must become distinguishable by some attribute (name and/or time stamp) since several messages may be waiting for being read in a mailbox.

Example 4: Remote Computation

Sender and receiver of messages are supposed to reside on different hardware platforms. The receiver of a message interprets the message as a request to run a certain program (possibly using certain data enclosed to the message) on its local hardware machine. After the program has finished, a result of the remote computation may be reported to the original sender by sending a return message.

ACKNOWLEDGEMENTS

Regards to Stephan Becker, Ralf Parr (both Technical University of Hamburg-Harburg) and to Dr. Randolf Isenberg (Philips GmbH Forschungslaboratorium Hamburg) for proof-reading and commenting on this paper. Ralf Parr also implemented a Smalltalk/V version of PCA. The work reported here was partially funded by the Commission of the European Communities, Directorate General XIII, under ESPRIT Project 2434 (Knowledge-Based Real-Time CIM Controllers for Distributed Factory Supervision).

REFERENCES

1. W. Meyer, Expert Systems in Factory Management: Knowledge-based CIM, Ellis Horwood, Chichester 1990.
2. C. A. R. Hoare, 'Communicating Sequential Processes', Commun. ACM, **21**, (8) 666 - 677 (1978).
3. R. M. Metcalfe and D. R. Boggs, 'Ethernet: Distributed Packet Switching for Local Computer Networks', Commun. ACM, **19**, (7) 395 - 404 (1976).
4. H. Zimmermann, 'OSI Reference Model -- The ISO Model of Architecture for Open Systems Interconnection', IEEE Trans. Communications, **28**, (4) 425 - 430 (1980).
5. General Motors Inc., MAP Specification 2.1. Contacts: General Motors Technical Center, Manufacturing Building, A/MD-39, 30300 Mound Road, Warren, Michigan 48090-9040, USA.
6. Kermit Distribution, Columbia University, Center for Computing Activities; Contacts: 612 West 115th Street, New York, NY 10025.
7. G. Steele, Common Lisp: The Language, Digital Press, Billerca 1984.
8. Microsoft Corp., Microsoft Windows Software Development Kit Guide to Programming Version 3.0, USA 1990.
9. C. Rich and R. Waters, The Programmer's Apprentice, Addison-Wesley, Reading 1990.
10. J.-I. Jakob and R. Isenberg, 'An Advanced Graphical Knowledge Acquisition Module for a Production Planning Expert System in CIM', Proc. IMACS-IFAC Symp. Modelling and Control of Technological Systems, Lille (France) May 7 - 10 1991, (1) 769 - 774.
11. J.-I. Jakob, 'Temporal Inference Accelerator Board Progress Report', ESPRIT 2434 30 Months Report, Hamburg 1991.

FIGURE CAPTIONS

Figure 1: PCA depicted as two identical, interacting state machines.

Figure 2: Possible hardware arrangements for the PCA