ב״ה

IMPLICIT COMMUNICATION IN PROGRAMMING LANGUAGES BY DECLARATION, REFERENCE, AND ASSIGNMENT

H.G. Mendelbaum and R.B. Yehezkael (Formerly Haskell)

Jerusalem College of Technology
Havaad Haleumi 21, Jerusalem 91160, Israel
E-mail address: rafi@eruvin.jct.ac.il Fax: 02-422075 Tel: 02-751111

August 1994 - אלול תשנ״ד
(Reformatted May 1999 - סיון תשנ״ט)

Abstract

With the increase of communication between computers, we need techniques for (i) communication to be handled automatically or implicitly, and (ii) flexible programs capable of handling data from different kinds of correspondents with no reprogramming. The traditional approach of handling input/output, files, inter program communication and man machine interfaces by specific statements in programming languages restricts these two aims. We propose handling these activities implicitly and in a unified manner by separating a program into a pure algorithm with no communication statements (PurAL), and separate declarations which describe externally mapped variables (DUAL description). So, without using I/O or communication statements, just assigning to an externally mapped variable will cause a value to be sent outside the program. Similarly referencing such a variable will cause a value to be fetched from outside the program (if it has not been fetched already). As the mapping of variables is described separately from the program, this enables the program to be used in various contexts (ports, files, networks, windows etc.). By specifying that a variable is mapped to data on another computer, in effect we are dealing with remote data access, i.e. the communication is implicit. By specifying that an output-variable in one program is to be mapped to input-variables in other programs, we have in effect a generalization of the pipe and tee operating system constructs. We also propose heuristics for open/close and lock/unlock, enabling these operations to be carried out automatically.

Introduction

In many cases the same algorithm can take various forms depending on the location of the data and the distribution of the code. The programmer is obliged to take into account the configuration of the distributed system and modify the algorithm in consequence in order to introduce explicit communication requests.

There are two problems: distribution of data, and distribution of code. Either a piece of code can be copied in various nodes and work with distributed data, or the whole code is split in various nodes and each part works with distributed data. Similiarly, data can be located locally and copied in various sites, or split and distributed over various places. Let us concentrate in the more common case of components of code located in different nodes and transfering data to each other.

As networking becomes commonplace, it becomes much more important to develop software which can easily access information in a distributed networked environment. Our aim is to facilitate this easy access to data at the programming language level.

Aim:
We want to use any "pure" algorithm written as if its data were in a local virtual memory, and run it either with local or remote data without changing the source code.

1) This will simplify the programming stage, abstracting the concept of data location and access mode.
2) This will eliminate the communication statements since the algorithm is written as if the data were in local virtual memory.
3) This will unify the various kinds of communication by using a single implicit method for transfer of data in various execution contexts: concurrently executing programs in the same computer, or programs on several network nodes, or between a program and remote/local file manager, etc.

For example, the programmer would write something like: x := y + 1;
where y can be "read" from a local true memory (simple variable), or from a file, or from an edit-field of a Man-Machine-Interface (for example, WINDOWS/DIALOG BOX), or through communication with another parallel task, or through a network communication with another memory in another computer.

In the same way, x can be "assigned" a value (simple local variable) or "recorded" in a file, or "written" in a window, or "sent" to another computer, or "pipe-lined" to another process in the same computer.

Proposal

We propose to separate a program into a pure algorithm and a description of the links with the external data.

1) the "pure algorithm" (PurAl) would be written in any procedural language without using I/O statements.
2) A "Distribution, Use, Access, and Linking" description (DUAL description) is used to describe the way the external data are linked to the variables of the pure algorithm. (i.e. some of the variables of the pure algorithm are externally mapped.)

To summarize in the spirit of Wirth [20] :

Program = Pure Algorithm + External Data accessed by DUAL description

or even briefer:

Program = PurAl + DUAL

Example: producer consumer, file display, file processing (ADA style)

Pure Algorithm:

```
with external_types;  USE external_types
procedure p1 is
seq1: vec;
begin
for i in seq1 'FIRST.. seq1 'LAST
loop
      data_processing (seq1 (i));     -- A --
end loop
end p1;
```

Distribution, Use, Access, and Linking Description (DUAL Description)

```
p1.seq1.site => computer3.mailbox2 => p2;
p1.seq1.access => IN sequential.increasing subscript;
p1.seq1.lock => gradual;
p1.site => comp1;
```

Pure Algorithm:

```
with external_types;  USE external_types
procedure  p2 is
seq2: vec;
j:integer:  = seq2 'FIRST
begin
loop
        exit when <exit condition>;
        j = j + 1;
        seq2 (j): = <expression>;        -- B --
end loop;
end p2;
```

Distribution, Use, Access, and Linking Description (DUAL Description)

```
p2.seq2.site => computer3.mailbox2 => p1;
p2.seq2.access => OUT sequential.increasing subscript;
p2.seq2.lock => gradual;
p2.site => comp2;
```

The previous two main procedures, work sequentially with data in a vector. The variables seq1 and seq2 are linked to the outside by the DUAL descriptions and all other variables such as i and j are local. The variables seq1 and seq2 of the two procedures are of type vec, which is defined in the package external_types, e.g. the type vec is for a vector of characters of undefined length:

```
        TYPE vec is ARRAY  ( positive range <> )  of character;
```

In the line marked "-- A --" of the procedure p1, seq1(i) is referenced and this causes a value to be fetched from outside, according to the DUAL description where p1.seq1.site indicates the location of the vector, a mailbox, and also the program(s) connected to the other side of the mailbox. Also

p1.seq1.access indicates the way of accessing the data, "IN sequential increasing subscript" means that seq1 is read from outside in sequential manner using a sequential increasing subscript. Similarly in the line marked "-- B --" of the procedure p2, the vector seq2(j) is assigned and this causes a value to be sent outside the program according to the DUAL description where p2.seq2.site indicates the location of the vector, a mailbox, and also the program(s) connected to the other side of the mailbox. Also p2.seq2.access where "OUT sequential increasing subscript" means that seq2 is written outside in sequential manner using a sequential increasing subscript.

The user can run these algorithms in various ways. With the previous DUAL descriptions, for seq1 and seq2, the programs will behave as a producer/consumer when run concurrently. However by using the modified DUAL descriptions below, they can also be run as completely independent procedures, with for example seq1 coming from a file and seq2 being put onto a screen. The procedures p1, p2 are not changed at all.

Modified DUAL Description for p1

```
p1.seq1.site => local computer.file3;
p1.seq1.access => IN sequential.increasing subscript;
p1.seq1.lock => gradual;
p1.site => comp1;
```

Modified DUAL Description for p2

```
p2.seq2.site  =>  local computer.screen;
p2.seq2.access => OUT sequential.increasing subscript;
p2.seq2.lock => gradual;
p2.site => comp2;
```

Literature

Work has been done in the field of transparent communication, i.e. masking the location of data by using an interface layer of the system, but retaining explicit communication requests at the programming language level.

1) For instance, Kramer et al. proposed to introduce interface languages (CONIC [14], REX [15], DARWIN [16]) which allow to describe in a

declarative form the distribution of the processes and data links on a network. But the algorithm of each process contain explicite communication primitives. For example, Magee and Dulay [17] use this methodology giving two versions of the Warshall's algorithm (for transitive closure): One for a sequential execution, and a second, more elaborate version for distributed execution; which includes explicit invocation of transfer of data and synchronization. Furthermore, they have a separate declaration describing the configuration and the binds between the distributed code. It would be far more convenient for the programmer, if the communication would be implicit, and were derived from the algorithm and the separate declaration of the configuration (at compile/link-time, or at run time).

2) a. Hayes et al [18] working on MLP (Mixed Language Programming) proposes using remote procedure calls by export/import of procedure names. He proposes a UTS (Universal Type System) which allows the checking of parameters passing at link time using a set of inclusion rule.

b. Purtilo [19] proposed a software bus system (Polylith) also allowing independence between configuration (which he calls "Application structure") and algorithms (which he calls "individual components"). The specification of how components or modules communicate is claimed to be independent of the component writing, but the program uses explicit calls to functions that can be remote (RPC) or local.

A separate interface language (MIL = Module Interconnection Language) explicits the binds between the local and remote procedure names and parameters. The procedures can be written in various languages; and a type checking is performed at compile/link time.

Both of these approaches are very interesting, since the use of remote procedures looks implicit, but you cannot transfer data without explicite procedure calls, it would be more interesting if the data transfer would also be implicit.

4) Miscellaneous related ideas

a. Implicit File Handling:

In the early 1980's the PDP/11 BASIC implementation included a virtual array feature, which identifies an array with a disk file. With this approach, files could be handled transparently (i.e. no read/write statements but instead assignment and reference to array variables). This corresponds to our

approach although it was limited to disk files only, the man machine interface, sequential access to data, and communication not being treated at that time. In the same way, some operating systems can access files via virtual memory mapping e.g. VAX/VMS and some versions of UNIX.

Similarly in persistent storage systems (e.g. the E programming language [13]), we find that assignment and reference are used as a "file" access method for persistent data. The possibility of using thes statements to handle communication is not discussed in [13].

Functional and Logic programming languages[3,4,12] use streams for mapping sequential data files onto lists. Indeed, in the LUCID[5] language, every variable is a stream or sequence of values. This too is a partial approach, treating only external lists and sequences via memory mapping. Our approach is more general in that we provide a unified notation for sequential and random access modes to arrays.

b. Handling the Man Machine Interface Transparently

Separating the man machine interface from the programming language has been extensively discussed over the years[2]. Some researchers considered the application part as the controlling component and the user interface functions as the slave. Others do the opposite: the user interface is viewed as the master and calls the application when needed by the I/O process. Some works (Parnas 1969)[8] described the user interface by means of state diagrams. Edmonds (1992)[9] reports that some researchers describe the user interface by means of a grammar. Some others presented an extension of existing languages, Lafauente and Gries (1989)[10].

These works handled the problems of how to define the user interface of a software system, but not the concept of treating the user interface and the application as two independant components. On the contrary, in our approach, the user interface and the application are defined separately and the link between them is explicitly described.

Some works (Hurley and Sibert 1989)[11] do build separate user interface from the applications also, but our approach is more general in that the user interface is seen as one part of a unified mechanism in which external data are accessed, the other parts of this unified mechanism being file handling, I/O, and communication.

c. Handling Communication Implicitly

Turbo Pascal has a PORT variable for low level communication with I/O Ports, and the values of these PORT variables may change asynchronously (from an external input) in parallel with program execution. This is similar in the sense of direct referencing or assigning, but the PORT fixes explicitly in the algorithm the site of the external data. Our approach also provides flexible access to distributed data and automatic locking conventions. Another difference is that Ports are for low level byte oriented communication, and our "external data" approach is also for high level "user" communication.

The idea of handling communication and I/O in an unified manner is well developed in the Hermes[1] programming language, in order to simplify the programming of heterogeneous distributed systes. It is made as an extension of a high language, adding a set of few communication primitives: send, receive, select, connect, call, reply, etc. This differs from our approach of not using special primitives but instead using assignment and reference to externally mapped variables as the means of handling communication implicitly, which has the advantage of simplifying the algorithmic language.

Universal Virtual Memory (UVM)

The realization of our proposal makes the distributed data available to various local/remote programs. The data looks like they were in the Local Virtual Memory which is mapped to Universal Virtual Memory (UVM). The UVM is located at distributed physical addresses. So, if the data are really located in the local physical memory, the access is immediate. If they are located on disk, it can be viewed as persistent storage. If they are sent or received to or from another device or remote memory, through a port, it can be viewed as communication etc.

Addressing:

In figure 1, one can see several distributed programs exchanging data on a network, each one using a software layer of UVM support based on mapping tables giving a general UVM addressing space.

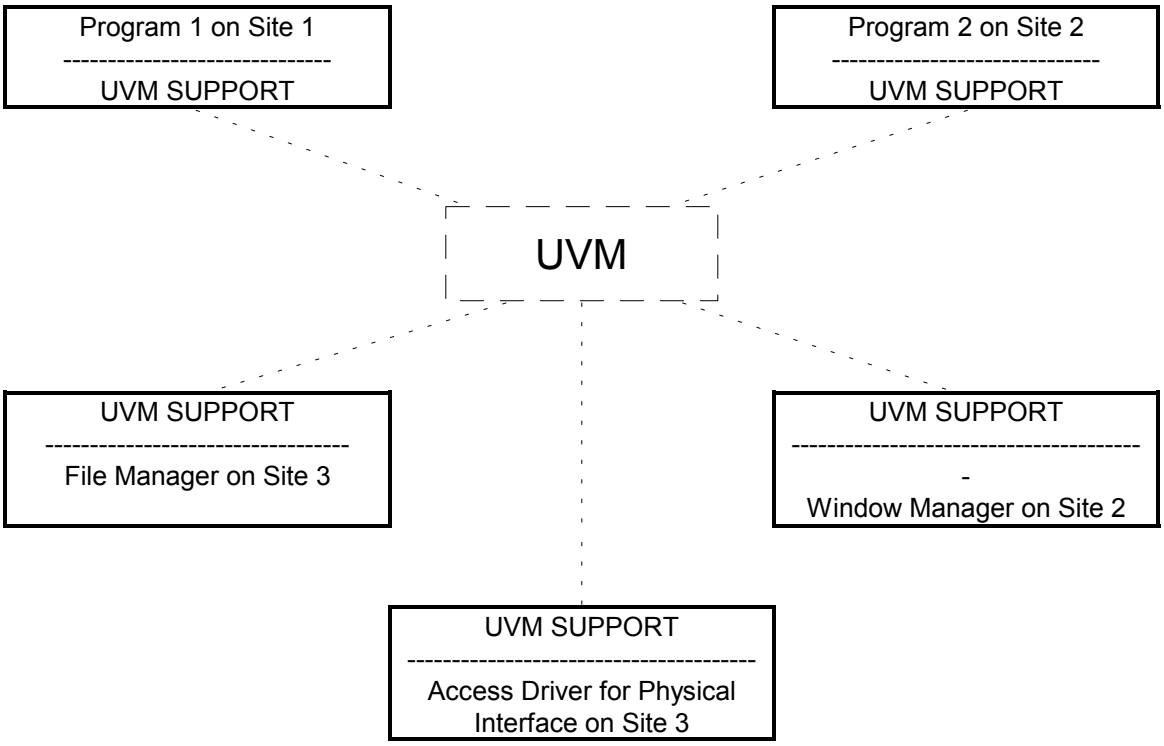

FIGURE 1: LOGICAL VIEW OF PROGRAMS AND SYSTEM COMPONENTS INTERACTING WITH EACH OTHER VIA UVM SUPPORT

In each program the external data are accessed through externally mapped variables of the pure algorithm. These externally mapped variables are associated with general UVM public names through the DUAL description. The externally mapped variables have local virtual addresses associated with them (like any other variable). Based on the DUAL description, the local virtual address is mapped to a UVM public name which consists of a UVM site name and a local name. In short, an externally mapped variable is associated with a UVM public name, where:

UVM public name = UVM site + local name.

Types of data:

The UVM support layer in each node could check the format and the type of the communicated data at initialization time when the DUAL descriptions are interchanged by the programs. Other kinds of implementation is to check the type/format of communicated data at link time (after compilation) or at run time.

Access to distributed data,UVM Support, and Mapping tables

To each pure algorithm is associated a dual description making a program, which will be executed using a UVM support layer (figure 2). The UVM support layer is responsible for finding at run time the correct address of external data (local or remote) using a mapping table (figure 3) with the appropriate access drivers and locking algorithms. In figure 3, the variable x of the pure algorithm corresponds to local virtual address 1FF8 which is associated with the UVM public name comp3.yellow_file in another computer. Referencing or assigning to this local virtual address would give immediate access to the local buffer located at physical address 200F because the "mapped flag" is on. But with the variable y, this flag is 0 so referencing or assigning to the local virtual address 1F00 will suspend program execution until the communication driver has mapped or remapped the physical memory and updated the the table. Then, if a reference needs to be made, external data is fetched based on the associated UVM public name. If an assignment needs to be made the memory receives new data and external storage can either be updated immediately or preferably when the memory needs to be freed or remapped.

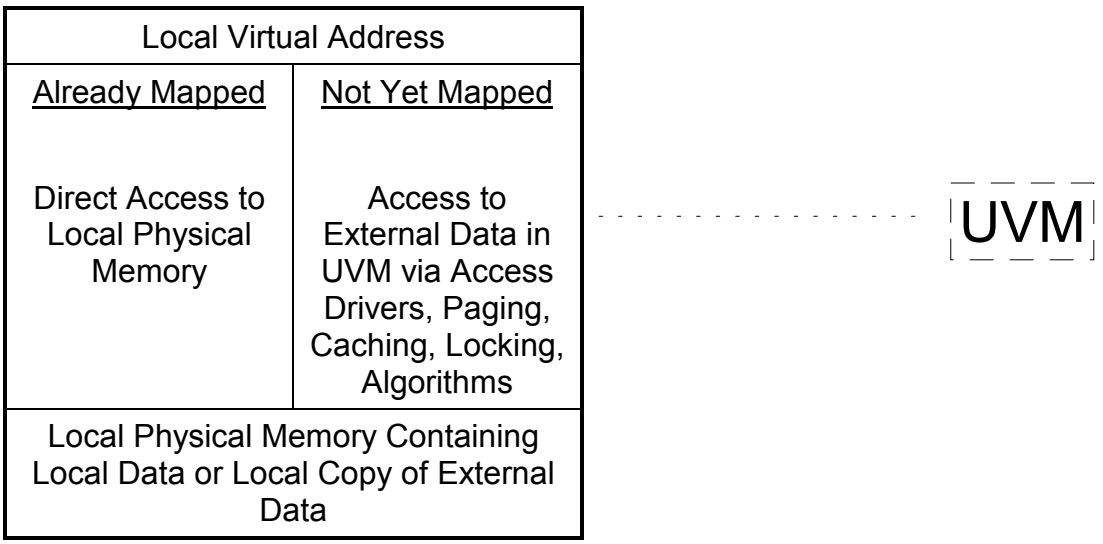

FIGURE 2 - UVM SUPPORT
(MAPPING LOCAL VIRTUAL MEMORY TO THE UVM)

Pure Algorithm (PurAl)		Distribution, Use Access and Linking Description (DUAL)					Local Physical Memory	
name	local virtual address	UVM public name site	UVM public name local name	memory type	access driver	IN/OUT and access mode	mapped flag	physical address
x	1FF8	comp3	yellow	file	com.	IN seq.	1	200F
y	1F00	comp2	shrd_mem3	shared memory	pager	INOUT	0	-
s	1000	local	pagefile1	memory	pager	INOUT	1	1F90
f	155E	local	phonebook	file	filer	IN rand.	1	220F
w	2222	local	myBOSS	window field	window manager	IN	0	-

FIGURE 3: MAPPING TABLE

Figure 4 illustrates the mapping of vectors c(..), d(..), e(..) in sequential and random access mode. Each element in a vector is allocated a unique virtual address. In this example each element in a vector is mapped to the same physical address which is the address of a buffer.

Pure Algorithm (PurAl)		Distribution, Use Access and Linking Description (DUAL)					Local Physical Memory	
name and index	local virtual address of first element	UVM public name site	UVM public name local name	memory type	access driver	IN/OUT and access mode	mapped flag	physical address of all elements
c(index)	B731	local	port3	port	serial driver	IN seq.	0	21AD
d(index)	F122	local	socket3	socket	pipe manager	IN seq.	0	AA02
e(index)	A010	comp3	file3	file	remote filer	INOUT rand.	0	2100

FIGURE 4: MAPPING VECTORS IN RANDOM AND SEQUENTIAL ACCESS MODES

Sequential access mode: Here c(i) represents the i-th item received from the port. Let us suppose the case of a sequentially increasing subscript. If the index of c in the table is 2 and c(2) is referenced, then the access is immediate. If c(4) or c(1) are accessed we have an access violation as the subscript has not increased by one. If c(3) is referenced, data is brought from the port to the buffer, the index in the table becomes 3 and then the reference takes place. Similarly the entry for d(..) illustrates how the sequence of values recieved from a pipe are mapped to a socket (or mailbox).

Random access mode: If e(1) is referenced or assigned, and the index of e in the table is 1 then the access is immediate. Then if e(3) is referenced or assigned, then as the index in the table is different, it changes to 3 and the buffer may be output to the file element corresponding to the old index. Subsequently, e(2) may be similarly accessed.

Access drivers:

There are two cases:

a) When the mapping tables indicate that the UVM public name is on the local computer, we shall use "local drivers". There are two generic kinds of local drivers, random and sequential access drivers:

   When the mapping table indicates that a random local driver is used, the actual access may be from the local memory immediately, from a disk file via paging, or via the file management system.

   When the mapping table indicates that a sequential local driver is to be used, the data are fetched sequentially as the local virtual address increases sequentially. The fetch may be immediately from local memory, from a port, a sequential device etc.

b) If the mapping tables show that the UVM public name is on another computer, we shall use "communication drivers", and the final access will be made by the "local drivers" of the remote computer.

Changes Required

A. Syntax changes:

No changes are made in the algorithmic language except for (1) no use of input output statements, and (2) the addition of unconstrained arrays.

B. Semantic changes:

We need to add rules for opening/closing the communication and locking/unlocking data which is consistent with the view of the data as if they were in local virtual memory. Also communication needs to be performed when the externally mapped variables are referenced (read) or assigned (write).

Therefore we do not to change the internal semantics of the PurAl but add an external semantics through the DUAL description. We propose that local variables (except pointers) may be externally mapped and that locking takes place in the block where they are declared (see below). Basically, this will ensure that the internal PurAl semantics are retained. More precisely we propose that only local variables of procedures be externally mapped, and for recursive procedures only the local variables of the non recursive invocation be externally mapped. A check must also be made from the DUAL description or the lock status that in a given program two active externally mapped variables are not mapped to the same external location. (By active externally mapped variable we mean an externally mapped variable whose declaration block is being executed.) These steps will ensure preservation of the internal semantics of the PurAl.

The semantic changes can be handled at *compile time* or *link time* or *run time* as follows:

1) To handle these changes at *compile time* we can use a modified compiler or preprocessor which will add specific functions in the PurAl according to the DUAL description as follows:

   (a) Request for activation and connection needs to be incorporated in the code if the DUAL description includes something like p1.var.site => ...mailbox => p2; since p1 needs p2 to be active and connected to the other side of the

mailbox. Typically when a program is started, it will send a request to activate associated (secondary local or remote) programs and and bind their externally mapped variables. These associated (secondary) programs would likewise activate associated (tertiary) programs and bind their externally mapped variables, and so on. At this start time, a copy of (relevant parts of) the DUAL description are sent to associated programs for compatibility checking (types, format, access mode, locking strategies) and mapping table building. Algorithmic processing does not start until all interconnections are established and compatibility checks completed.

(b) Open/Close operations for externally mapped variables can be added at the start and end of the PurAl or Heuristics can be used to incorporate them before first access and after last access.

(c) Lock/Unlock operations can be added at the beginning and at the end of the block where externally mapped variables are declared if the locking mode is strict. Rules can be devised for more gradual locking strategies based on first and last access of externally mapped variables.

(d) Input statements may need to be added at the beginning of the block where the externally mapped variable is declared or after each reference to an externally mapped variable.

(e) Output statements may need to be added at the end of the block where the externally mapped variable is declared or after each assignment to an externally mapped variable.

2) To handle these changes at *link time*, the compiled code is not changed. A modified linker or postprocessor uses the DUAL description to add code as above to handle the communication. This is practical only if the compiled program clearly indicates the beginning and end of each block, and if the code for each block is a separate relocatable section.

3) To handle these changes at *run time*, reference and assignment to the externally mapped variables would trigger read or write operations. The access would be handled via mapping tables using a page fault mechanism and communication drivers where necessary. Thus some changes may be needed in the operating system. The DUAL description would need to be handled by the shell or command language processor (to allocate buffers, drivers etc.). Open could be handled at the start of the program or at the first access of the externally mapped variable (first page fault). Close would occur

at the end of the program. Some pre or post processing would be needed to handle the locking and may be desirable for early closing.

Conclusion

The approaches of Darwin and Polylith-MIL are oriented towards a centralized description of an application distributed on a dedicated network. So all the binds and instances of programs are defined initially at configuration time.

Our approach is aimed towards a general network in which each program knows only of the direct binds with its direct correspondents. In one sense this approach is more dynamic (similar to phone calls), on the other hand establishing the interconnection seems to be less sure. However in requiring that algorithmic processing does not start if the interconnection fails, we provide some degree of reliability but perhaps not as great of the policy of centrally specified and configured and initialized application network as used for example in Darwin. We think our approach is justified in the context of interconnected programs in a general network. That's why we have the need of a pure algorithm not containing explicit statements to communicate with the outside in various applications and configurations.

References

[1] "Hermes: A Language for Distributed Computing", Robert E. Strom et.al. Research Report, IBM T.J. Watson Research Center, Oct. 18th, 1990.
[2] "The Seperable User Interface", Editor Ernest Edmonds, Academic Press, 1992.
[3] "Report on the Programming Language Haskell, A Non-strict Purely Functional Language", Paul Hudak et.al., Yale University Research Report No. YALEU/DCS/RR-777, 1st March 1992.
[4] "Concurrent Prolog - Collected Papers Vols 1,2" edited by Ehud Shapiro, MIT Press, 1987.
[5] "LUCID, the Dataflow Programming Language", W.W. Wadge and E.A. Ashcroft, Academic Press, 1988.
[6] "Res Edit Complete", P. Alley and C. Strange, Addison Wesley, 1991.
[7] "Interface Builder", Expertelligence Corp., 1987.

[8] "On the Use of Transition Diagrams in the design of a User Interface for an Interactive Computer System", D.L. Parnas, Proceedings of the National ACM Conference 1969, pp. 379-385, also appears in [2].
[9] "Emergence of the Seperable User Interface", E. Edmonds, appears as the introduction of the book he edited, see [2].
[10] "Language Facilities for Programming User-Computer Dialogues", J.M. Lafuente and D. Gries, IBM Journal of Research and Development 1978, Vol. 22 No. 2, pp. 122-125, also appears in [2].
[11] "Modelling User Interface-Application Interactions", W.D. Hurley and J.L. Sibert, IEEE Software, January 1989, pp. 71-77, also appears in [2].
[12] "Functional Programming: Application and Implementation", P. Henderson, Prentice Hall, 1980"
[13] "The Design of the E Programming Language", J.E. Richardson, M.J. Carey and D.T. Schuh, Research Report, Computer Sciences Department, University of Winconsin.
[14] "Constructing Distributed Systems in Conic", J. Magee, J. Kramer, and M.S. Sloman, Vol 15, No. 6, pp 663 - 675, 1989.
[15] "An Introduction to Distributed Programming in REX", J. Kramer, J. Magee, M. Sloman, N. Dulay, S.C. Cheung, S. Crane, and K. Twiddle, in "Proceedings of Esprit, Brussels, 1991.
[16] "Structuring Parallel and Distributed Programs", J. Magee, N. Dulay, and J. Kramer, in "Proceedings of the International Workshop on Configurable Distributed Systems, London, 1992.
[17] "MP: A Programming Environment for Multicomputers", J. Magee and N. Dulay, appears in "Programming Environments for Parallel Computers", edited by N. Topham, R. Ibbett, and T. Bemmerl, Elsevier Science Publishers B.V. (North Holland), 1992
[18] "A Simple System for Constructing Distributed Mixed Language Programs", R. Hayes, S.W. Manweiler, and R.D. Schlichting, Software Practice and Experience, Vol 18, No. 7, pp 641 - 660, 1988.
[19] "The Polylith Software Bus", J.M. Purtilo, ACM Transactions on Programming Languages and Systems, Vol 16 No. 1, pp 151 - 174, 1994
[20] "Algorithms + Data Structures = Programs", N. Wirth, Prentice Hall, 1976