ב״ה

Towards Implicit Communication and Program Distribution

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

April 1995 - ניסן תשנ״ה
(Minor changes made October 2001 - תשרי תשס״ב)

Abstract

With the increase of communication between computers, we need (i) techniques for 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 and distribution of programs (DUAL declarations). 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). Another approach, is to perform the input of an externally mapped variable when its storage is allocated (e.g. block entry) and the output when its storage is freed (e.g. block exit). 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 rules for open/close and lock/unlock, enabling these operations to be carried out automatically.

NOTE: A shortened version of this paper appears in CDROM Conference Proceedings, 4th World Multi-Conference on: Circuits, Systems, Communications and Computers (CSCC 2000), Vouliagmeni, Athens, Greece, July 2000. It also appears in "Advances in Physics, Electronics and Signal Processing Applications", pp. 402-409, edited by N. E. Mastorakis, World Scientific and Engineering Society Press, 2000.

CONTENTS

1 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.

With the proliferation and increased use of networks, it has become very 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.

2 PROPOSAL

We propose to separate a program into a pure algorithm and a declarative 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" declarative description (DUAL declaration) 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.) The DUAL declaration is also used to describe the distribution of the programs.

To summarize in the spirit of Wirth [20] :

Program = Pure Algorithm + External Data accessed by DUAL declaration

or even briefer:

Program = PurAl + DUAL

2.1 The Producer Consumer Example and Variations (in ADA style)

Pure Algorithm 1:

```
WITH external_types;  USE external_types
PROCEDURE c is
seq1: vec;
i:positive:=1;
BEGIN
  LOOP
    data_processing (seq1 (i));       -- A --
    i:=i+1;
  END LOOP;
EXCEPTION
  WHEN constraint_error => EXIT;     -- out of range on seq1, i.e. end of data
END c;
```

DUAL declarations for c (Distribution, Use, Access, and Linking Declarations)

```
c'site => comp1;
c.seq1'site => mailbox2;
```

```
c.seq1'access => IN sequential_increasing_subscript;
c.seq1'lock => gradual;
```

Pure Algorithm 2:

```
WITH external_types;  USE external_types
PROCEDURE  p is
seq2: vec;
j:positive:  = 1;
BEGIN
  LOOP
    EXIT when <exit condition>;
    seq2 (j): = <expression>;          -- B --
    j = j + 1;
  END LOOP;
END p;
```

DUAL declarations for p (Distribution, Use, Access, and Linking Declarations)

```
p'site => comp2;
p.seq2'site => mailbox2;
p.seq2'access => OUT sequential_increasing_subscript;
p.seq2'lock => gradual;
```

Pure Algorithm 3:

```
WITH external_types;  USE external_types
PROCEDURE mailbox2 IS    -- one to one mailbox
SUBTYPE data_type IS integer;
place:positive:=1;
BEGIN
  LOOP
    block1: DECLARE inline, outline: vec (place..place);
    BEGIN
      outline(place) := inline(place);  place := place + 1;
    EXCEPTION
      WHEN constraint_error => EXIT;    -- out of range on inline, i.e. end of data
    END block1;
  END LOOP;
END mailbox2;
```

DUAL declarations for mailbox2 (Distribution, Use, Access, and Linking Declarations)

```
mailbox2'site => computer3;
mailbox2.inline'site => comp2:p.seq2;
mailbox2.inline'access => IN sequential;
mailbox2.outline'site => comp1:c.seq1;
mailbox2.outline'access => OUT sequential;
```

Some Comments on the above Programs

The first two algorithms are written at the user programmer level and we have hidden the details of the communication in the package "external_types". Actually non standard use is being made of ADA unconstrained array feature but this is hidden from the user programmer. The third algorithm is written at the system level and illustrates how a simple mailbox could be handled. Local declarations are used inside a loop to define which elements of a vector, are being processed. This means that externally the vector is unconstrained, but internally the bounds, which are given by the current value of "place", define which elements are being processed. The algorithm is programmed in completely standard ADA and the exception mechanism is used to detect the "end of data" condition ("constraint_error").

The procedures in algorithms 1 and 2, work sequentially with data in a vector. The variables seq1 and seq2 are externally mapped by the DUAL declarations 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 data_type;
```

In the line marked "-- A --" of the procedure c, seq1(i) is referenced and this causes a value to be fetched from outside, according to the DUAL declaration where c.seq1'site indicates the location of the vector, a mailbox. Also c.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 sequentially increasing subscript. Similarly in the line marked "-- B --" of the procedure p, the vector seq2(j) is assigned and this causes a value to be sent outside the program according to the DUAL declaration where p.seq2'site indicates the location of the vector, a mailbox. Also regarding p.seq2'access, "OUT sequential_increasing_subscript" means that seq2 is written outside in sequential manner using a sequentially increasing subscript.

The user can run these algorithms in various ways. With the previous DUAL declarations, for seq1 and seq2, the programs will behave as a producer/consumer when run concurrently.

Other distributed use of the same algorithms are possible by only modifying the DUAL declarations. With the modifications below, they are run as completely independent procedures, with for example seq1 coming from a file and seq2 being put onto a screen. The procedures c, p are not changed at all.

Modified DUAL declaration for c

```
c.seq1'site => local_computer.file3;
c.seq1'access => IN sequential_increasing_subscript;
c.seq1'lock => gradual;
c'site => comp1;
```

Modified DUAL declaration for p

```
p.seq2'site  =>  local_computer.screen;
p.seq2'access => OUT sequential_increasing_subscript;
p.seq2'lock => gradual;
p'site => comp2;
```

2.2 Generalizations of the Producer Consumer Example

Let us reconsider the producer consumer classical case, where in each example we shall assume that all programs are concurrently executing and communicating via a mailbox which are all on the local computer. All consumers use procedure c (pure algorithm 1) and all producers use procedure p (pure algorithm 2) as described previously. In our approach there are no changes in the Pure Algorithms only new DUAL declarations. We shall use vector notation in the DUAL declarations where c(1..n) will denote the n consumer programs c(1), c(2), ... c(n), and p(1..n) will denote the n producer programs p(1), p(2), ... p(n).

Many Consumers but one Producer: Suppose that there are multiple copies of the consumer program c on various sites, which are communicating with a single producer program p.

DUAL for p

```
p'site => local_computer;
p.seq2'site => mailbox5;
p.seq2'access => OUT sequential;
```

DUAL for multiple copies of c

```
computer = (sparc, sun, eruvin.jct.ac.il);
multiple c'site => computer(i) for i in 1..3;
c.seq1'site => mailbox5;
c.seq1'access => IN sequential;
```

DUAL for mailbox5

```
mailbox5'site => local_computer;
computer = (sparc, sun, eruvin.jct.ac.il);
mailbox5.inline'site => local_computer:p.seq2;
mailbox5.inline'access => IN sequential;
mailbox5.outline(i)'site => computer(i):c.seq1 for i in 1..3;
mailbox5.outline(i)'access => OUT sequential  for i in 1..3;
```

Many Producers but one Consumer: Suppose that there are multiple copies of the producer program p on various sites, which are communicating with a single consumer program c.

DUAL for c

```
c'site => local_computer;
c.seq1'access => IN sequential;
c.seq1'site => mailbox3;
```

DUAL for multiple copies of p

```
computer = (sparc, sun, eruvin.jct.ac.il);
multiple p'site => computer(i) for i in 1..3;
p.seq2'site => mailbox3;
p.seq2'access => OUT sequential;
```

DUAL for mailbox3

```
mailbox3'site => computer2;
computer = (sparc, sun, eruvin.jct.ac.il);
mailbox3.outline'site => c.seq1;
mailbox3.outline'access => OUT sequential;
mailbox3.inline(i)'site => computer(i):p.seq2 for i in 1..3;
```

mailbox3.inline(i)'access => IN sequential for i in 1..3;

Comments on the use of a DUAL declaration for a mailbox

With this approach we propose allowing several DUAL declarations using the same physical mailbox, providing the programs associated each DUAL declaration are all different. In this way when a mailbox is allocated to a program, associated programs can be uniquely and unambigously identified and activated. The disadvantage of this approach is that it is more static way of using mailbox requiring predefinition of all possible connections using the mailbox. On the other hand it makes the connections more secure.

NOTE: There can be several types of mailboxes e.g. permanent or temporary, mutual exclusion or broadcast, one to many or many to one, etc.

3 LITERATURE SURVEY

3.1 Transparent Communication

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 allows the user to describe centrally and in a declarative form the distribution of the processes and data links on a network. But the algorithm of each process contain explicit 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 central and separate declarations 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) Hayes et al [18] working on MLP (Mixed Language Programming) proposes using remote procedure calls (RPC's) 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.

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

3) Comparison of DUAL with Darwin, MLP-UTS, and Polylith-MIL

The appof Darwin, MLP-UTS, 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 in DUAL is aimed towards a non dedicated 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 (see later), we provide some degree of reliability but perhaps not as great of the policy of centrally specified and configured and initialized application network. Our claim is that our approach is better suited to interconnected programs in a non dedicated network.

3.2 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 these 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.

3.3 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, Lafuente 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.

3.4 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.

4 HANDLING DATA DISTRIBUTION AND COMMUNICATION

We now discuss various aspects about handling data distribution and communication implicitly.

4.1 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.

a. 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.

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 declaration. The externally mapped variables have local virtual addresses associated with them (like any other variable). Based on the DUAL declaration, 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.

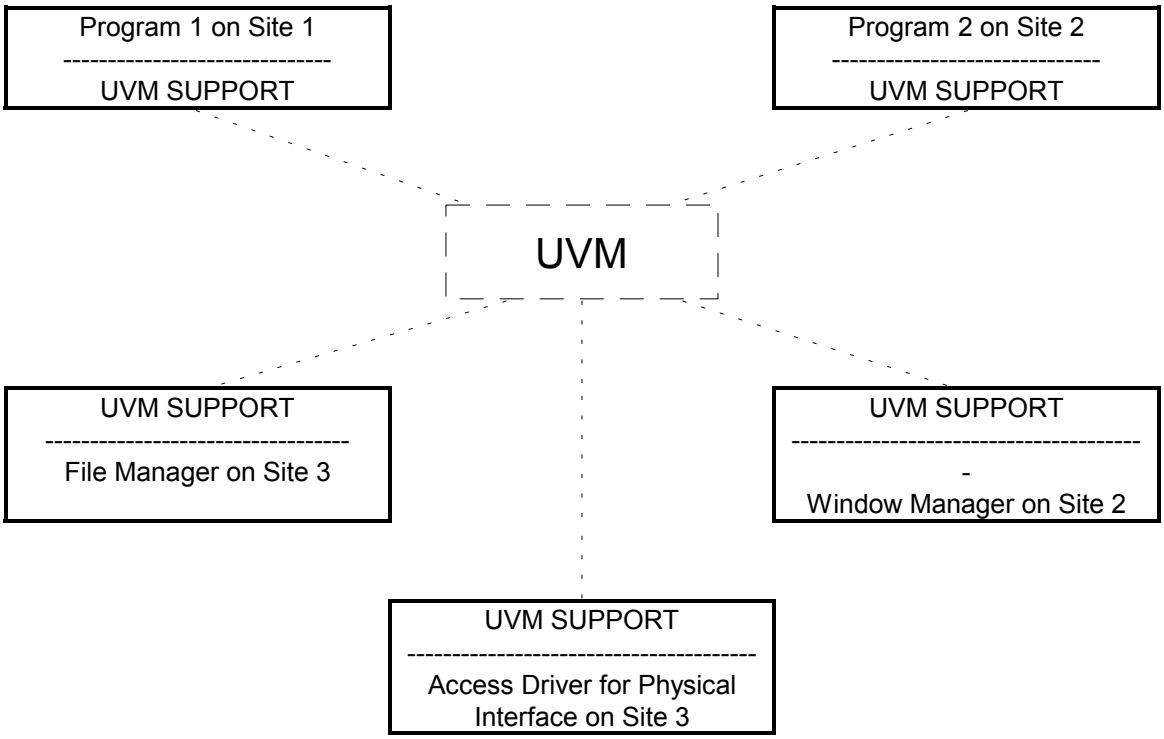

FIGURE 1: LOGICAL VIEW OF PROGRAMS AND SYSTEM COMPONENTS INTERACTING WITH EACH OTHER VIA UVM SUPPORT

b. 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 declarations 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.

c. Access to distributed data,UVM Support, and Mapping tables

To each pure algorithm is associated a DUAL declaration 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).

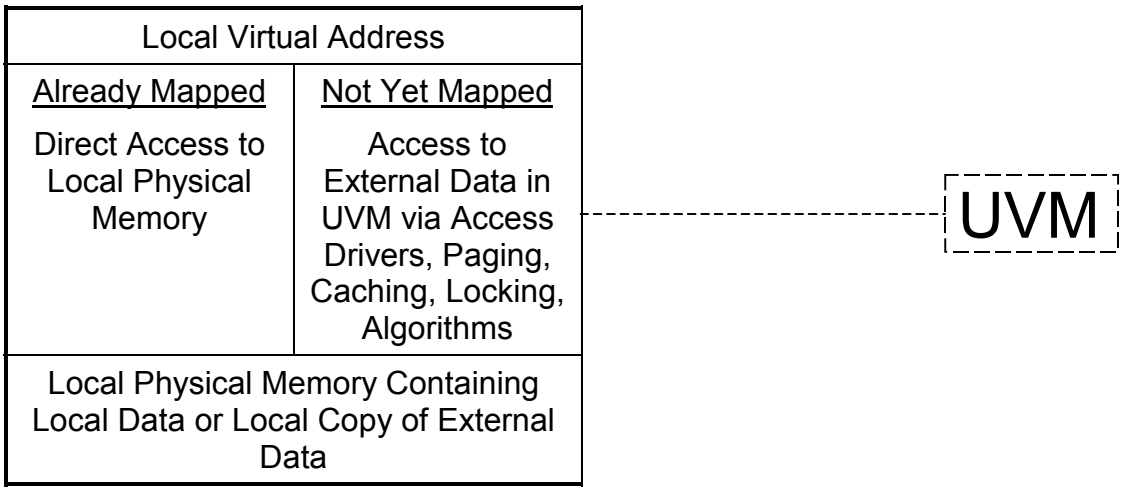

FIGURE 2 - UVM SUPPORT
(MAPPING LOCAL VIRTUAL MEMORY TO THE UVM AND TO LOCAL PHYSICAL MEMORY)

. Access drivers:

There are two cases:

i) • When the mapping (figure 2) indicates 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 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 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.

ii) If the mapping shows 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.

4.2 Syntax Changes Required

No changes are made in the algorithmic language except for (i) no use of input output statements, and (ii) the use of unconstrained arrays (for mapping vectors to files).

4.3 Semantic Changes Required

We need to add rules for opening/closing the communication and locking/unlocking data which are 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 have to change the internal semantics of the PurAl but add an external semantics using the DUAL declaration. We propose that local variables (except for 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 declaration 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 declaration as follows:

(a) Request for activation and connection needs to be incorporated in the code if the DUAL declaration includes something like c.var'site => mailbox since c needs whatever is connected to the other side of the mailbox to be active at run time (more details later).

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

(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 (see later).

(d) Input statements may need to be added at the beginning of the block where the externally mapped variable is declared and/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 and/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 declaration 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 declaration 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.

4.4 Open and Close Rules

In the following explanation we assume that EMV denotes an externally mapped variable.

"Open" may be handled at *compile time* or *run time* according to the first occurence or access of EMV. However, "close" can be handled at *compile time* based on the last occurence of EMV, but as the last access of EMV can not be identified, the way to handle "close" at *run time* is on exit from the main procedure.

Compile time: Let F_EMV denote the earlier of:

- the first occurence of EMV itself in the body of the main procedure, or
- the first procedure call in the main procedure which has the potential of causing an access to EMV.

Similarly, L_EMV denotes the later of:

- last occurence of EMV itself in the body of the main procedure, or
- the last procedure call in the main procedure which has the potential of causing an access to EMV.

(In the above, Warshall's algorithm for transitive closure can be used to identify which procedures have the potential of causing an access of EMV.)

Compile time rules:

Insert "open" before F_EMV in the main procedure. However, if the open is inside a "loop" or a conditional construct, move it before the "loop" and the conditional construct.

Insert "close" after L_EMV in the main procedure. However, if the "close" is in a "loop" or a conditional construct, move it after the "loop" and after the conditional construct.

Run time: open EMV on its first access and close it when exiting the main procedure.

4.5 Lock and Unlock Rules

Locking strategies: So as to avoid deadlock in implicit communication we propose the following approach for entering/leaving blocks and automatic locking and unlocking of variables. We present three possibilities.

a) Locking on open and unlock on close: All data associated with an externally mapped variable are locked from open to close in accordance with the open/close rules described earlier.

b) Strict Locking and Unlocking: Entry to a block would take place only if its externally mapped variables are unlocked. More precisely, we also check if externally mapped variables of inner blocks or blocks which potentially may be invoked by inner procedure/function calls are also unlocked. (As before, Warshall's algorithm for transitive closure can be used to identify which blocks potentially may be invoked through procedure and function calls.) On entry to the outer bnlock only its externally mapped variables are locked. When leaving the block, all its externally mapped variables are unlocked.

c) Gradual Locking and Early Unlocking: On entry to a block no checks are made. On the first use of any of its externally mapped variables, a check is made to see if all variables of the block and variables of inner blocks (and variables which may be involved by inner procedure/function calls) are unlocked and then the said variable is locked in its entirety. More specifically, if its first access is a reference, wait for the externally mapped variable to receive a value from outside and lock the externally mapped variable but if its firsaccess is an assignment, lock it prior to the assignment.

Access Mode	Early Unlocking Rule
IN sequential OUT sequential IN OUT sequential IN1 OUT sequential IN OUT1 sequential	Unlock current element when next element is used.
IN1 OUT1 IN1 OR OUT1	Unlock each element separately after use.
IN1 OUT1	Unlock each element after second use.

FIGURE 3 - EARLY UNLOCKING RULES

As a general principle regarding unlocking, we unlock remaining locked data when leaving the block where an externally variable is declared. Early unlocking is possible, and situations where this can take place are given in the following table where we have also included the possibility of single reference and assignment access modes. In particular, "IN1" means that once the external mapped variable is fetched it will never be never be fetched again by the program. Similarly "OUT1" means that the externally mapped variable will only be assigned once. Also "IN1 OR OUT1" means that the externally mapped variable may be accessed only once, but this access may be a reference or assignment whereas "IN1 OUT1" means that there are at most two accesses where the first may be a reference and the second an assignment or vice versa.

5 HANDLING CODE DISTRIBUTION

Typically when a program is initiated, it will use the DUAL declaration of the mailboxes it uses to 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 initialization time, a copy of (relevant parts of) the DUAL declaration 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.

5.1 Using Already Existing Distribution

Let us consider a network with already existing general purpose programs on various sites: for instance a file manager, an algorithm for FFT computation, etc. We can then use the DUAL declarations to connect them so that the exchange of data can be properly coordinated and synchronized (like in our producer/consumer examples, see 2.1, 2.2). We propose two ways of handling the parallelism in the execution, in this case.

a) At initialization time, the first activated program will send (according to its DUAL declaration) a request to its mailboxes which in turn would send activation and connection requests to correspondent programs connected to the other side of the maiboxes (see above). Then the rules for open/close and

lock/unlock, in 4.4 and 4.5 can be applied to synchronize the exchange of data on reference and assignment or block entry and exit.

b) At runtime, each time a data transfer is necessary, according to the DUAL and according to assignment and reference in the algorithm, a connection is requested and the transfer fulfilled. Again this must be in accordance with the rules of 4.4 and 4.5, so that the internal semantics of the pure algorithm are preserved.

5.2 Static Distribution at Initialization Time

Suppose we have a specific application in which there are several programs which need to be distributed on a network. The distribution can be handled from one site which will use the DUAL declarations to send the various programs to various sites and build the connections between them. For example in the parallel version of Warshall's algorithm below, the DUAL declaration indicates that there should be ten copies of the procedure "or_rows" on ten computers while the matrix "mat" is located on "local_computer:/data/file2". The communication is implicit, through parameter passing and remote procedure calls.

Pure Algorithm

```
PROCEDURE warshall IS
-- Warshall's algorithm for the transitive closure of a square boolean matrix

SUBTYPE indexrange IS integer RANGE 1..10;
TYPE boolean_vector IS ARRAY (indexrange) OF boolean;
TYPE square_matrix IS ARRAY (indexrange) OF boolean_vector;

mat:square_matrix;

 FUNCTION or_rows(v1,v2:IN boolean_vector) RETURN boolean_vector IS
 v:boolean_vector;

 BEGIN
  FOR finish IN indexrange LOOP
   IF v1(finish) OR v2(finish)
   THEN v(finish) := true;
   ELSE v(finish) := false;
   END IF;
```

```
   END LOOP;
   return v;
 END or_rows;

 PROCEDURE transitive_closure(a:IN OUT square_matrix) IS
 b:square_matrix;

 BEGIN
   FOR middle IN indexrange LOOP
     FOR start IN indexrange LOOP
       IF a(start)(middle)
       THEN b(start) := or_rows(a(start),a(middle));
       ELSE b(start) := a(start);
       END IF;
     END LOOP;
     a := b;
   END LOOP;
 END transitive_closure;

BEGIN -- warshall
  transitive_closure(mat);
END warshall;
```

DUAL declaration

```
computer = (sparc, sun, shekel.jct.ac.il, c1, c2, c3, c7, c9, c11 , eruvin.jct.ac.il);
warshall'site => local_computer;
multiple warshall.or_rows'site => computer(i) for i in 1 .. 10;
warshall.mat'site => local_computer:/data/file2;
warshall.mat'access => IN OUT random;
```

Notes:

1) The only difference between the serial and parallel version of Warshall's algorithm is the use of an additional matrix "b" and the use of the statement "a:=b" to transfer the contents of "b" back to "a". Thus as "a" is only referenced in the inner loop of procedure transitive_closure, all iterations of the inner loop can be executed in parallel.

2) In this example, the distribution of code takes place at initialization time (the default). It is also possible to specify that the distribution take place at run time when a procedure or function is called (see next section - "on_call").

3) In section 6, we shall return to this example and discuss how parallel execution is handled.

5.3 Dynamic Distribution at Run TIme

In some applications we do not know in advance the exact location of the procedures which need to be copied and distributed. In this case, the DUAL can indicate the location of the main procedure and data at initialization time, but at run time the procedure calls will cause the automatic distribution of procedures on available processors. For instance in the "merge sort" example below, the DUAL indicates that the procedure "sort" may be distributed on eight computers but it would be only at run time that the distribution would take place, based on availability. As the procedure "merge" runs on the same computer as "sort", a copy of the procedure "merge" is also distributed together with the procedure "sort".

Pure Algorithm

```
PROCEDURE merge_sort IS
first: CONSTANT INTEGER :=1;
last: CONSTANT INTEGER :=17;
TYPE vec IS ARRAY( INTEGER RANGE <>) OF INTEGER;
v: vec(first..last);

 PROCEDURE merge (vec_1,vec_2 : IN  vec; vec_3 : OUT vec) IS
 i_vec1:INTEGER:=vec_1'FIRST;
 i_vec2:INTEGER:=vec_2'FIRST;
 i_vec3:INTEGER:=vec_3'FIRST;

 BEGIN
  WHILE i_vec2 <= vec_2'LAST AND i_vec1 <= vec_1'LAST
  LOOP  -- merge
   IF vec_1(i_vec1) <= vec_2(i_vec2)
   THEN
    vec_3(i_vec3):=vec_1(i_vec1);
    i_vec1:=i_vec1 + 1;
   ELSE --  vec_1(i_vec1) > vec_2(i_vec2)
    vec_3(i_vec3):=vec_2(i_vec2);
    i_vec2:=i_vec2 + 1;
   END IF;
   i_vec3:=i_vec3+1;
```

```
    END LOOP;  -- merge

    FOR j IN i_vec1..vec_1'LAST LOOP --supplement in vec_1
      vec_3(i_vec3):=vec_1(j);
      i_vec3:=i_vec3+1;
    END LOOP;

    FOR k IN i_vec2..vec_2'LAST LOOP --supplement in vec_2
      vec_3(i_vec3):=vec_2(k);
      i_vec3:=i_vec3+1;
    END LOOP;
  END merge;

  PROCEDURE sort (x: IN OUT vec) IS
  mid: INTEGER:=(x'FIRST+x'LAST)/2;
  x_left:vec(x'FIRST..mid):=x(x'FIRST..mid);
  x_rigth:vec(mid+1..x'LAST):=x(mid+1..x'LAST);

  BEGIN
    IF x'LENGTH > 1
    THEN
      sort(x_left);
      sort(x_rigth);
      merge(x_left,x_rigth,x);
    END IF;
  END sort;

BEGIN -- merge_sort
  sort(v);
END merge_sort;
```

DUAL declaration

```
computer = (sparc, sun, c12, c14, c21, c23, c29 , eruvin.jct.ac.il);
file = (lo_computer:/mydir/datafile1);
mergesort'site => local_computer;
multiple mergesort.sort'site => computer(i) for i in 1..8 on_call;
mergesort.merge'site => same_as_caller;
mergesort.v'site => file(1);
mergesort.v'access => IN OUT random;
```

Notes:
1) Actually, the above algorithm was not written with parallelism in mind. However, the DUAL declaration enabled us to specify this parallelism with no change to the pure algorithm.
2) In this example, the distribution of code takes place at run time when a procedure or function is called. This is indicated in the DUAL declaration by the "on_call" keyword in the multiple site declaration.
3) In section 6, we shall return to this example and discuss how parallel execution is handled.

6 HANDLING PARALLELISM IN THE CODE

6.1 Conventions for Parallel Execution of Procedure and Function Calls

In the previous examples of parallel Warshall's Algorithm and merge sort, we discussed policies for distribution. We wish to exploit parallelism inherent in the pure algorithm without incorporating this into the pure algorithm. Instead, we propose conventions which are designed to ensure equivalance of sequential and parallel execution of procedure calls and exploit the parallelism from the DUAL declaration. (Functions are treated like a procedure with an extra OUT parameter for returning the result.)

1) These rules apply if pointers are not part of the parameters.
2) Wait if necessary for the data to be passed as parameters to be unlocked, and then lock the data against the parent procedure. (So if the parent wishes to use these data or to pass it to another child, it must wait for these data to be unlocked.) Make copies if possible of the data passed as an IN parameter and if a copy has been made, unlock the data for the parent procedure. Even if a copy is made of data passed as an OUT parameter, keep it locked against the parent. When asynchronous execution of the child procedure completes, unlock the remaining locked parameters for the parent procedure.
3) In general, wait for variables to become unlocked during execution.

Returning to the parallel algorithms of Warshall and merge sort presented in the previous section, we observe the following.

1) In Warshall's algorithm, the above conventions ensure that there may be up to ten copies of or_rows in execution concurrently (as the matrix has dimensions 10 x 10). Waiting typically occurs at the statement "a:=b;" in the procedure "transitive_closure" and then another ten copies of or_rows can be

executed concurrently. So nothing is gained by allowing more than 10 concurrently executing copies of "or_rows".

2) In the merge sort example, the above conventions ensure that the two recursive calls to sort may be executed concurrently and that the merge will not take place until the two recursive calls to sort are completed

6.2 Facilitating Automatic Locking and Synchronization

In section 4.5 and the previous subsections, we saw that the following programming language features facilitate automatic locking and synchronization during the parallel execution of procedure and function calls.

1) "IN", "OUT" and "IN OUT" qualifiers of parameters.
2) Passing slices of one dimensional arrays as parameters.

The above features are part of the ADA programming language. The following proposed additions would further facilitate the use of parallelism and automatic locking and synchronization conventions.

1) "IN1" and "OUT1" qualifiers on parameters and variable declarations and the obvious combinations.
2) "IN1 OR OUT1" qualifier on parameters and variable declarations.
3) Passing slices of any array as parameters.
4) The use of an additional declaration when an inner procedure or function (or block) needs to access variables of an outer procedure or function (or block). This would be particularly effective when used together with slices.

7 CONCLUSION

Our proposal has the following key features:

1) The separation of a program into a pure algorithm (PurAl) and a DUAL declaration. This yields flexible programs capable of handling different kinds of data distribution with no change to the pure algorithm.

2) Implicit or automatic handling of communication via externally mapped variables and generalizations of assignment and reference to these variables. This provides unified device independent view and processing of internal data and external distributed data at the user programming language level.

3) Default locking conventions which preserve internal semantics of the programming language and ensure equivalence between sequential and parallel execution of procedure and function calls.

4) Programs need only know of the direct binds with distributed correspondents (mailbox driver, file manager, remote task, window manager etc.). This avoids the need for a central description of all the interconnections.

The last three features follow naturally from the first feature. That's why we have the need for separation between a pure algorithm not containing explicit communication statements and a description of its links with the outside. This is what gives the flexibility of use in various applications and configurations.

8 ACKNOWLEDGMENTS

The authors would like to thank Y. Gordin and J. Levian, of the Jerusalem College of Technology, as well as A. Frank, O. Kremien, P. Ravid and Y. Wiseman of Bar Ilan University for their helpful discussions and assistance.

9 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, IEEE Transactions on Software Engineering, 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