Published in the Proceedings of the GI workshop *Parallelrechner und Programmiersprachen (PARS)*

in Schloß Dagstuhl, Germany, 26.2.- 28.2.1992, pages 59 – 67

Modula-P – a Language for Parallel Programming and the Implementation of its Channel Communication*

Jürgen Vollmer
Gesellschaft für Mathematik und Datenverarbeitung†

Abstract

Hoare's CSP (Communicating Sequential Processes) provide a powerful framework for the description and analysis of parallel programs. *Modula-P* extends Modula-2 with the concurrent features of CSP, i.e. parallelly executed processes, synchronous message passing and simultaneous waiting for input from several channels. Modula-P tackles some problems other CSP-based languages don't solve. The language and the implementation of its channel communication is presented.

1 The language

Modula-P [Vollmer 89, Vollmer $^{et\ al}$ 92] is a language for programming parallel MIMD[1] computers. It is a superset of Modula-2 [Wirth 85] enriched with language constructs based on the *Communicating Sequential Processes (CSP)* [Hoare 85].

Modula-P provides the `PAR` statement to initiate concurrent execution of its components.

$$\boxed{\texttt{PAR } p_1 | \cdots | p_n \texttt{ END}}$$

Processes is either declared as *local* or *global* processes. Local processes are simple statement sequences. They share variables with the parent process. The language doesn't provide synchronization on variable accesses. A global process is declared in a `PROCESS MODULE`, which is syntactically similar to the main module of a Modula program, except that value parameters may be passed. Each global process has its own instance of all (transitively) imported global variables. It doesn't share any variables with other global processes. Communication between global processes is done solely using *channels* (see below). Global processes may be executed on different processors. The distribution may be controlled by specifying a placement for the global process (`AT expression`) in the `PAR` statement. If no placement is given a default strategy is used.

Synchronous communication between processes is done via typed channels. A channel is declared as a usual Modula variable:

```
VAR channel :  CHANNEL OF OtherType
```

The statement

```
channel !  expression
```

writes the value computed by *expression* to *channel* which a further process may read and assign to *variable* by the statement

```
channel ?  variable
```

Channels must be initialized using the standard procedure `OPEN`. This is usually done by the parent process of the communicating processes. Notice: Messages are routed automatically through the network by a runtime system.

The language knows the notion of an abstract clock. Its values are of the type `TIME` and are read from the predefined channel `TIMER`. Processes may delay themselves for some time by the statement

```
TIMER ? AFTER expression
```

*This research was supported in part by ESPRIT under ESPRIT project 5399 (COMPARE).

†Vincenz–Prießnitz–Straße 1, D–7500 Karlsruhe 1, email: vollmer@karlsruhe.gmd.de

[1]Multiple Instructions Multiple Data

Processes may wait for several events simultaneously by means of an ALT statement. A guard may contain an optional boolean expression and an input request from a channel or a time out condition. If a guard of an alternative is ready (i.e. the optional boolean expression evaluates to TRUE and a process wants to communicate via the channel, or the time has passed), the corresponding statement is executed. If the optional boolean expression $\texttt{bool}_i$ evaluates to FALSE, the alternative is discared. If all alternatives are discarded, $\texttt{stmts}_0$ is executed.

```
ALT bool₁,chn ?  var :  stmts₁ | ··· | boolₙ,TIMER ? AFTER expr :  stmtsₙ ELSE stmts₀ END
```

Similarly to Modula-2's ARRAY OF WORD (BYTE) parameter passing mechanism there exists an untyped channel, too. A formal parameter of type ARRAY OF WORD (BYTE) is compatible to all types in Modula-2. This may be used to program very flexible, but type unsafe general routines. Analogously, any type of message may be passed over a CHANNEL OF ANY, using only the size of the message and the address of the source / destination:

```
channel !  source_address, message_size
channel ?  destination_address, message_size
```

The message size of the sender and receiver must match, and the receiver process has to provide enough memory for storing the received message, before. This kind of channel may used as a guard of an ALT statement, too.

Processes and alternatives may be replicated using a *replicator* inside of a PAR or ALT statement. lower and upper may be variables of any enumeration type, INTEGER, or CARDINAL. Each process or alternative, respectively, gets a unique value assigned, which is in the range [lower .. upper]. This value may be accessed in the process or alternative through the replicator variable i, respectivly and used like a constant.

```
PAR [i :  lower TO upper] process(i) | ··· END
ALT [i :  lower TO upper] bool[i],chn [i] ?  x [i] :  stmts(i) | ··· END
```

2 Program examples

2.1 Producer-Consumer

The following is the Modula-P solution for the *producer - consumer* problem: Several producers generate some data, which must be consumed by a single consumer process, e.g. several printer jobs use a single printer (see figure 1).

All processes are declared in process modules. The procedure doit executes the producer and consumer processes in parallel. The Modula-2/Modula-P *open array* language feature is used, to express that a variable number of processes is used. The number of producers depends on the number of declared channels. The consumer emits a timeout message, if in a given time span no data is sent by any producer.

2.2 Pipeline parallelism[2]

The example shows a parallel program computing prime numbers. It seems silly to compute prime numbers this way, but this example shows how to construct an arbitrary large pipeline of processes using recursion. One can think of two kinds of processes: a generator and several worker processes chained by channels. The generator produces odd numbers and sends them to the first worker (which has number 2). Each worker process represents a prime number. It reads numbers from its input channel, where the first number is its prime number. If a read number is divisible by the worker's own prime number, this number is not prime; hence it is discarded. Otherwise it must be sent to the next worker process for further examination. A negative number is interpreted as the termination signal. When the worker process is called two further processes are started. One of them checks the read numbers. The other starts recursively the next worker, which waits for input of its prime number or the termination signal, sent by the checking process (see figure 2).

3 Problems with channels

What kind of difficulties arise when designing a channel communication protocol? To answer this question, we examine the behavior of Modula-P channels in detail.

[2]The idea for this example was taken from an Ada program, see: "An Ada Tasking Demo", Dean W.Gonzalez, Ada letters, Volume VIII, Nr. 5 , Sept,Okt 1988, page 87 ff

```
PROCESS MODULE producer
      (nr  : CARDINAL;
       (* I'm producer "nr" *)
       chn : defs.tDataChannel);
IMPORT defs;

PROCEDURE produce
      (VAR data : defs.tData);
BEGIN ...  END produce;

VAR data : defs.tData;
BEGIN
  LOOP
    produce (data);
    chn ! data;
  END;
END producer.
```

```
PROCESS MODULE consumer (chns : ARRAY OF defs.tDataChannel);
IMPORT defs, sysTime, netIO;
PROCEDURE consume (nr  : CARDINAL; data : tData);  ...

VAR data : defs.tData; time : TIME;
BEGIN
  LOOP
    TIMER ? time;
    time := sysTime.plus(time, 10 * sysTime.TicksPerSec);
    ALT (* wait for messages from a producer or timeout *)
       [i : 0 TO HIGH (chns) ] chns [i] ? data :
          consume (i, data);
     | TIMER ? AFTER time : (* if time has passed *)
          netIO.WriteString ("no results since 10 sec");
          netIO.WriteLn;
    END;
  END;
END consumer.
```

```
...  IMPORT producer, consumer; ...
PROCEDURE doit (chns : ARRAY OF defs.tDataChannel);
VAR i : CARDINAL;
BEGIN
  FOR i := 0 TO HIGH (chns) DO OPEN (chns[i]) END; (* initialize channels *)
  PAR (* execute producers and consumer in parallel *)
      [i : 0 TO HIGH (chns)] producer (i, chns [i]);
   | consumer (chns);
  END;
END doit;
```

Figure 1: Producer - consumer with timeout message program example

```
MODULE p_prime;
FROM defs IMPORT IntChannel;
(* TYPE IntChannel = CHANNEL OF INTEGER *)
FROM netIO IMPORT ReadInt;
IMPORT worker;
VAR max, i: INTEGER; out : IntChannel;
BEGIN
  netIO.ReadInt (max);
  OPEN (out);
  PAR
     worker (out) (* start first worker *)
   | (* test number generator *)
     out ! 2; (* first prime number *)
     FOR i:=3 TO max BY 2 DO out ! i END;
     out ! -1 (* termination *)
  END
END p_prime.
```

```
PROCESS MODULE worker (in : defs.IntChannel);
FROM defs IMPORT IntChannel;
FROM netIO IMPORT WriteInt; FROM net IMPORT left;
VAR out : IntChannel; prime, x : INTEGER;
BEGIN
  in ? prime;
  IF prime < 0 THEN RETURN ELSE WriteInt(prime) END;
  OPEN (out);
  PAR
     LOOP (* read numbers, -1 terminates *)
       in ? x;
       (* pass termination signal and terminate *)
       IF x < 0 THEN out ! -1; EXIT END;
       IF x MOD prime # 0 THEN out ! x; END;
     END;
   | worker (out) AT left() (* start next worker *)
  END;
END worker.
```

Figure 2: The *pipeline* program example

Three "objects" are involved in communication: two processes (sender and receiver) and the channel. The channel is opened by the parent process and then passed as parameter to the processes, when they are started in a `PAR` statement. The sender and receiver process may change while the channel exists, since it may be passed to another process as parameter, and hence the "identification" of the communication partner may change between two communications. Unfortunately, it is not always possible[3] to inform the other communication partner about that change. The problem is now how a message "finds its way" form the sender to the receiver.

For instance, look at the program fragment in figure 3. After the process *Process*$_2$ has communicated over the channel *ch* this channel is used by the child process *Process*$_3$. After termination of this child process the father *Process*$_2$ can use this channel again. Both sender and receiver may change between to communications.

```
Parent process          Process1 (ch)          Process2 (ch)          Process3 (ch)
VAR ch:CHANNEL OF ···     ⋮                      ⋮                      ⋮
   ⋮                    LOOP                   ch ?  y;               ch ?  z;
OPEN (ch);                ch !  x;             PAR                      ⋮
PAR                         ⋮                      Process3 (ch)
    Process1 (ch)       END                      | ···
  | Process2 (ch)                              END
END;                                           ch ?  y; (*)
```

Figure 3: Changing communication partners

A common solution is to leave forward references when a channel is passed to another process. The message uses these references to "find its way". But the flexibility of Modula-2 causes some problems, e.g. the process decides after some time to use the channel field of a variant record and not the other fields. Hence before passing it may be unknown whether the channel is really passed and used in the process, so no forward reference may be stored. Furthermore this method is very expensive, since a message must follow all forward references, while there may be a shorter "way" from the sender to receiver.

4 Channel agent protocol

An implementation for the changing communication partner problem would be to use a central instance for each channel, a so-called *agent* [Hoffart 91], which is known to the sender and the receiver. Before each communication, the partners inform the agent about their current identification. The agent then propagates this information to them. For this protocol it is necessary that the identification of the agent of a channel is known by the processes using this channel. The only invariant information on channels is that they have to be opened before they are used. Hence, the agent can be implemented by the runtime system of that processor on which the channel was opened. So, agents of different channels may be distributed over the entire network depending on the dynamic behaviour of a program.

The protocol is based on a simple set of signals, which may be sent asynchronously. For each signal, the receiver of the signal (the agent, the sender or receiver Modula-P process) is known in before.

4.1 Communication without `ALT`

In detail a channel communication where the receiver doesn't execute an `ALT` statement looks like shown in figure 4.

Assume a sender arrives at its output operation first. (Notice: it doesn't matter which partner will reach its communication statement first.)

1. The sender sends a signal *SReady* to the channel agent, containing its identification. Then it waits to receive the partner's identification sent by the agent.
2. When the receiver process has reached the corresponding input operation it sends *RReady* to the agent containing its identification. Now it waits for the proper data from the sender.

[3]due to user interaction, procedure variables, variant records without tag field etc.

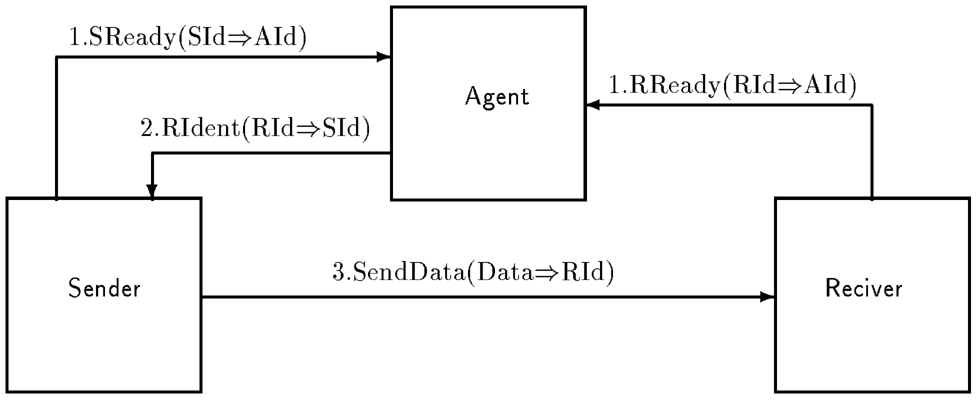

Figure 4: Channel agent protocol

3. After having both identifications the agent can pass the receiver identification to the sender process with *RIdent* and its work is done for this time.
4. Having received the partner identification the sender is able to transmit the data directly to the receiver with *SendData*.

After the data exchange both processes may run independently again. This procedure has to be done every time a Modula-P channel communication takes place.

Since each of these signals are sent asynchronously, synchroneity (i.e. blocking sender and receiver) of the entire message passing on the Modula-P level must be guaranteed in some way. This is based on an acknowledgement protocol. Where the *SendData* message is interpreted to be the acknowledgement of the *RReady* and *RIdent* message going from the receiver to the sender via the agent.

4.2 Execution of an `ALT` statement

Now we turn to the case of channel input guards contained in `ALT` statements. During the time a receiver process waits for a channel guard, the information that a sender is ready has to be passed to this receiver process. It can continue then, i.e. select a sender for input and execute the corresponding alternative of the `ALT` statement.

Let's have a more detailed look to the different patterns of interaction between the `ALT`-receiver and any of the senders:

a) The sender does not get not ready to send at all.

b) A sender gets ready, but it is not selected by the receiver.

c) A sender is ready and it is selected by the receiver.

The following protocol is performed now (see figure 5):

1. The receiver process informs all agents of the channels involved in the `ALT` statement that it executes an `ALT` statements and waits for communication by sending the signal *EnableS*. After that the receiver process suspends itself.
2. A sender process gets ready to send a message and sends *SReady* to the channel's agent (as done before). Now it waits for a response from its agent, i.e. for the *RIdent* message. Steps 1 and 2 may take place in any order.
3. If the agent knows that there is a ready sender, it signals this by sending *SReady*. The receiver ignores all but the first *SReady* signals.
4. The next step is to choose one ready sender out of the set of ready senders. The channel agent of the selected sender gets the *SelectS* signal, which initiates communication of the user data as shown before. The receiver executes then the statements of the chosen alternative of the `ALT` statement.
5. All other agents get a *DisableS* signal, to inform that the receiver is no more waiting for communication.

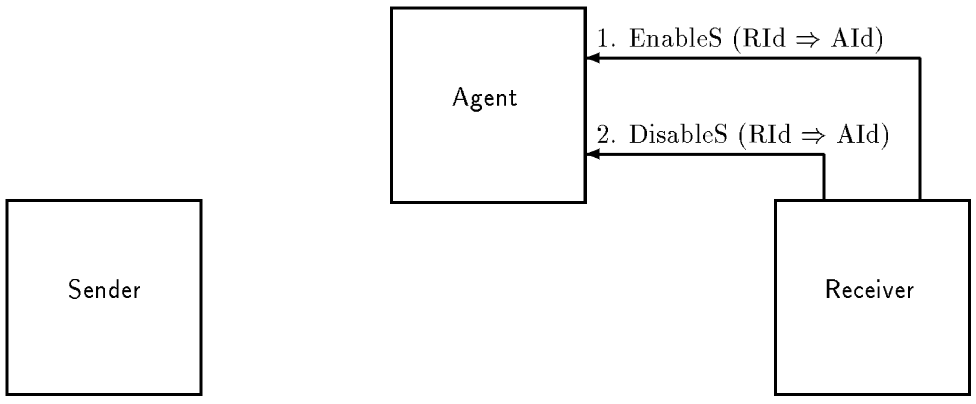

Figure 5a: Receiver is ready, but sender not.

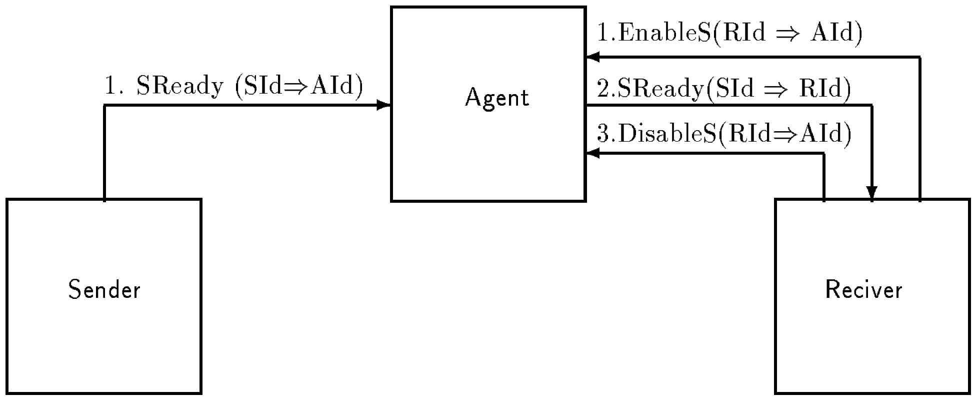

Figure 5b: Receiver and sender are ready, but sender is not selected.

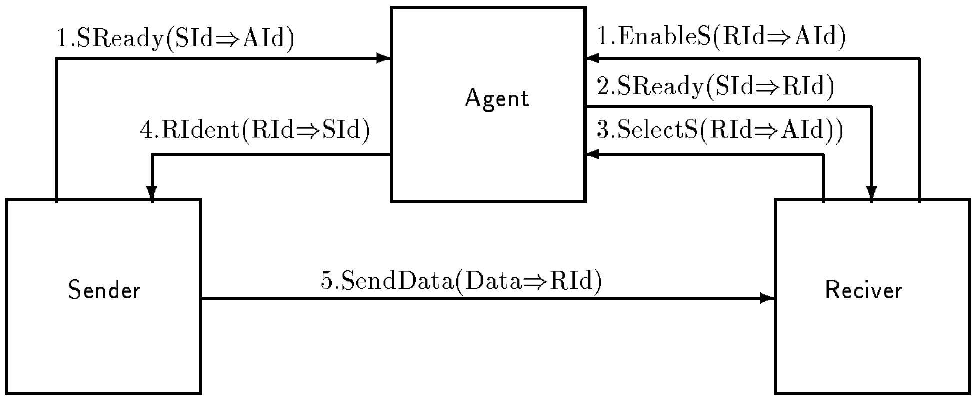

Figure 5c: Receiver and sender are ready, and sender is selected.

Figure 5: Channel agent protocol, executing `ALT`.

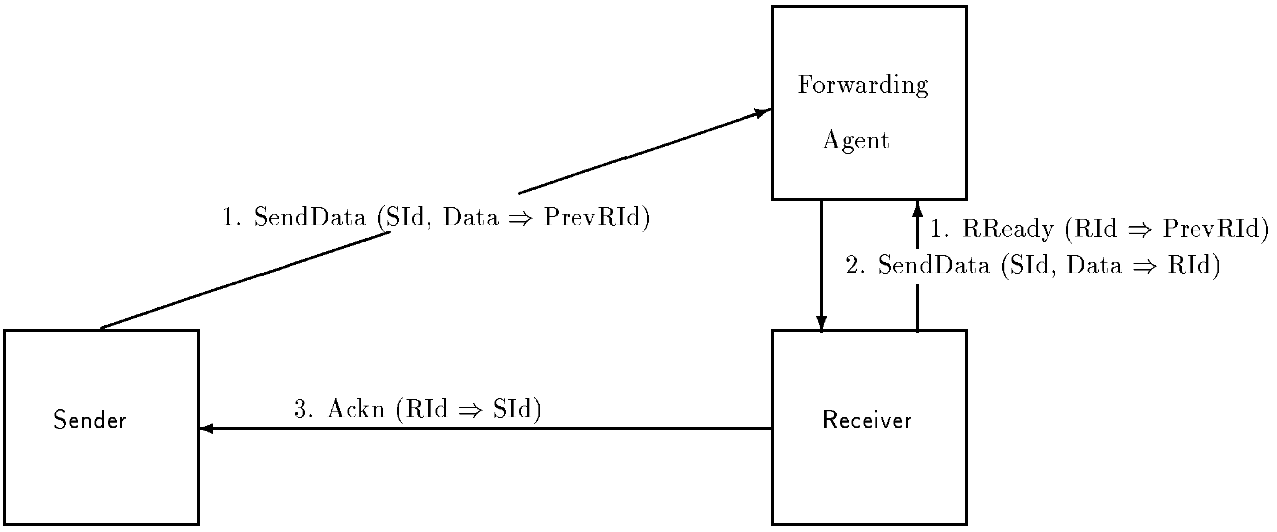
Forwarding Agent
1. SendData (SId, Data ⇒ PrevRId)
1. RReady (RId ⇒ PrevRId)
2. SendData (SId, Data ⇒ RId)
Sender
3. Ackn (RId ⇒ SId)
Receiver

Figure 6a: Receiver has changed since last communication

5 Restricting the communication

Often the "forward" passing of the channels is sufficient, i.e. in *Process*$_2$ of figure 3 the communication marked with (*) is not allowed, since the parent processes reuse a channel, used by a child process before. Guided by the assumption that only channels are only passed forward, a more efficient implementation for the changing communication partner problem is possible:

A new kind of channel agent, called *forwarding channel agent* is used to forward a message addressed to the process A to the new process A'[4]. This agent is needed only at the first communication after a change of a communication partner. If the receiver doesn't change, no agent is needed, since the sender knows the receiver's current identification. The forwarding of a messages is initiated by the receiver which knows that it has changed (just by memorizing the identification of the process which did the last communication).

The case that the sender changes needs no special treatment, since in this protocol, the Modula-P communication is always initiated by the sender. Initially before the very first communication, a forwarding channel agent is used by sender and receiver, acting on the processor the channel was opened.

The protocol is called "adaptive forwarding protocol", since the message is forwarded only when the receiver changes, and then the sender is informed about the new receiver.

The programmer may pass the `OPEN` procedure additional parameters to specify that this protocol should be used for communication.

The next two sections describe this idea with more details.

5.1 Communication without `ALT`

Assume the receiver changes, then the sender send its message to the forwarding channel agent of the receiver. The receiver itself knows that it has changed since the last communication (since it has another identification) and asks its *forwarding agent* to forward the data. An acknowledgement signal must be sent to ensure the blocking semantics of the Modula-P communication, together with this the identification of the current sender is sent (see figure 6a).

Now assume that the receiver has not changed since the last communication, then the sender sends its message directly to the current receiver (see figure 6b).

5.2 Execution of an `ALT` statement

Now we turn to the case of channel input guards contained in `ALT` statements. This is done very similar to the complete channel agent protocol. Assume that the receiver has changed, then the channel agent with the

[4]The process A passes the channel to A' and A' uses it for communication.

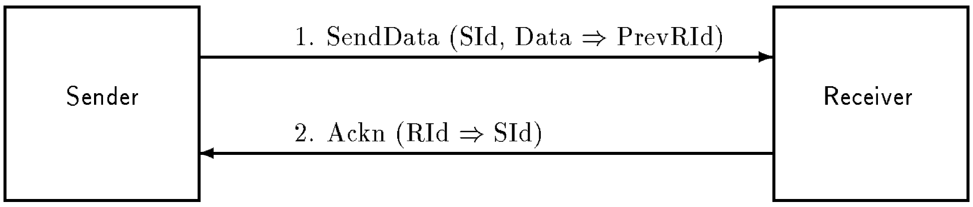

Figure 6b: Receiver has not changed since last communication

Figure 6: Adaptive protocol, no `ALT` executed

identification of the "previous receiver" is informed that the current receiver executes an `ALT` statement by *EnableS*. If a message has arrived (*SendData* at the channel agent, it informs the receiver about a sender is ready using *SReady*. After all senders are informed that the receiver has finished its waiting (*DisableS*, one ready sender is selected (*SelectS*) and the message is forwarded and the *Ackn* signal is send to the sender. Note, the data are not allowed to be forwarded to the receiver, until the sender is selected, because, if it not selected, and the receiver changes, the data are on the wrong processor. If the sender has not changed, these actions are then performed analogously without sending signals.

6 MOPS: The Modula-P OPerating System

Some special problems arise implementing Modula-P on a distributed computer system. First, several (local and global) processes may be executed on a single processor, and second global processes may be started on different processors. The execution of several processes on a single processor (in a timeshared way) may be controlled by the hardware (e.g. Transputer) or some software. Third a user should not be concerned how a message find its way from the sender to the receiver in the network. Last but not least, the distribution of global processes as well as the remote start and termination of such processes have to be done automatically. All these tasks are performed by the runtime system, called *MOPS*,

One main aim was to let MOPS work absolutely transparent. It is an entirely distributed system running on every processor of the network as processes independently from the user processes. For efficiency we haven't built up a layered communication model but used the abilities of transputer hardware: independent process management, autonomous process scheduling on one transputer with own scheduling queues. Programs with MOPS run on different transputer network configurations without any change in the source files. Furthermore, no recompilation caused by changes in the network topology is necessary.

MOPS is implemented by several separate processes. They run on high priority to satisfy the requests with a higher priority than the user program, which runs at low priority. MOPS consists of three parts (see figure 7):

1. The *procedural interface*. User processes access MOPS by calling these procedures. Calls to these procedures are inserted into the produced code by the compiler. These procedures are executed as part of the user process and send *MOPS messages* to the other parts of MOPS.
2. The *Command Handler* implements the channel agent protocols, distributes the local messages to the user processes, passes external messages to the link handler, and does the other tasks, like starting global and local processes, etc.
3. The *link handler* is a set of separate processes, which deal with the physical external Transputer channels. For each input and output link there is a separate process. They have two main tasks: First, they accept MOPS messages from user processes (which are produced by calling procedures from the MOPS interface) and second they forward messages which are not bound for this processor to the MOPS running on one of its neighbor processors.

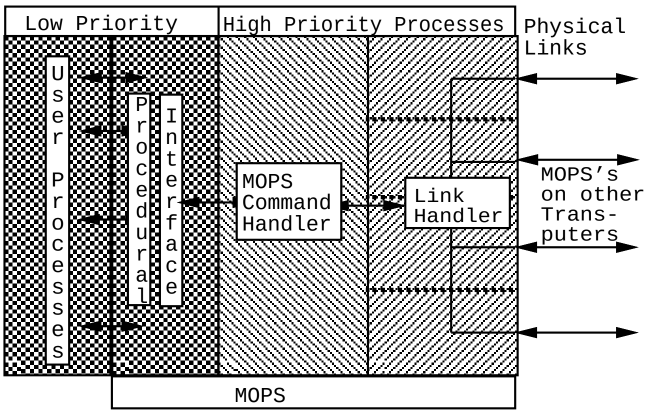

Figure 7: Structure of MOPS

7 Future work

The Transputer version will be ported to new Transputers (T9000) and to a SPARC based processor network. Analysis methods will be developed, which free the user from choosing the one or the other communication protocol. The more general question is: how to optimize explicit parallel programs.

8 Summary

The programming language Modula-P and the development system *MOCKA-P* offer a very powerful programming environment for Transputer-based parallel computers. The programmer gets a clear and intuitive model of parallelism, which is soundly based on the theory of CSP. As usual for modern programming languages, Modula-P supports programming "in the small" as well as "in the large", and frees the programmer from details allocating hardware resources. The presented communication protocols show how powerful Modula-P's channels are.

Acknowledgements

Thanks to the students Markus Armbruster, Claas Hinrichs, Ralf Hoffart, Jens Hübel, and Jürgen Richter who did parts of the implementation.

References

[Hoare 85] C.A.R Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice–Hall, Inc., 1985.

[Hoffart 91] Ralf Hoffart. Übersetzung einer parallelen Programmiersprache für ein verteiltes Rechnersystem – Modula-P auf Transputernetzwerken. Master's thesis, Universität Karlsruhe, March 1991.

[Vollmer 89] Jürgen Vollmer. Modula-P, a language for parallel programming. *Proceedings of the First International Modula-2 Conference October 11-13, 1989, Bled, Yugoslavia,* pages 75–79, 1989.

[Vollmer $^{et\ al}$ 92] Jürgen Vollmer and Ralf Hoffart. Modula-P, a language for parallel programming; definition and implementation on a transputer network. In *Proceedings of the 1992 International Conference on Computer Languages ICCL'92, Oakland, California,* pages 54–64. IEEE, IEEE Computer Society Press, Los Alamitos, California, April 1992.

[Wirth 85] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, Berlin, Heidelberg, New York, Tokyo, third, corrected edition, 1985.