Extended (by adding section D) version of the paper published on pages 75–79 in the
Proceedings of the First International Modula-2 Conference
October 11-13, 1989, Bled, Yugoslavia

Modula-P
A Language for Parallel Programming

Jürgen Vollmer
GMD Research Group at the University Karlsruhe
Haid-und-Neu-Str. 7, D-7500 Karlsruhe 1, FRG

1 Introduction

Communicating Sequential Processes (CSP) [**?**] provide a powerful framework for the description and analysis of parallel programs. The notion has been used as the underlying model of *Occam* [**?**], an elegant language designed by D. May. However, programmers sometimes consider this language as too frugal: there are only limited data structuring mechanisms, procedures can not be recursive, and the important concept of data abstraction using modules is missing. This paper introduces the language *Modula-P* as an attempt to overcome these shortcomings: it extends the sequential language *Modula-2* by the CSP model of parallel programming.

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

```
PAR p1 | ... | pn END
```

Synchronous communication between processes is done via typed channels. The statement

```
channel ! expression
```

outputs a value to *channel* which may be read by a further process using

```
channel ? variable
```

Processes may wait for several events simultaneously by means of an ALT statement.

```
ALT
    guard_1 : stmts_1
  | ...
  | guard_n : stmts_n
END
```

A guard may contain a Boolean expression and an input request from a channel.

A compiler translating Modula-P into code for Transputer [**?**] systems was developed as an extension of our existing Modula-2 compiler.

This paper describes the syntax and semantic of the language extensions.

2 The language Modula-P

2.1 The PAR statement

The PAR statement specifies the concurrent execution of its components.

```
PAR p1 | ... | pn END
```

The processes p_1, ..., p_n are executed in parallel. The process executing this PAR statement is suspended until all of its son processes p_1, ..., p_n have terminated. The component p_i may be either a statement sequence, referred to as a *local process* or a call of a *global process* (see below).

The general form of the PAR statement is:

```
ParStatement =
    PAR Process "|" Process END.
Process =
    [[replicator] LocalProcess] |
    [[replicator] GlobalProcess].
LocalProcess =
    StatementSequence.
GlobalProcess =
    ProcedureCall [";"].
```

2.2 Communication between processes

Channels are used for unidirectional synchronous communication between two concurrent executed processes. Unidirectional means, that a process may only output to or input from a channel, but it is not allowed to mix these operations. The term synchronous specifies, that the process which reaches its communication statement first will wait until the second process reaches the corresponding statement. Then the communication takes place, i.e. the message is passed, and both processes continue independently.

Channels are treated in the same manner as Modula-2 variables, i.e. they have a type and must be declared. A type and a variable declaration looks like:

```
TYPE channel = CHANNEL OF BaseType;
VAR  c1 : CHANNEL OF BaseType;
VAR  c2 : channel;
```

BaseType may be any other Modula-P type.
Before the first communication over a channel can

take place, the channel must be open with the standard procedure *OPEN*. A channel is opened as long as both processes, which used it are not terminated. For variables of type channel are input and output operations defined.

```
channel ? variable
```

is used for input from a channel.

```
channel ! expression
```

is used for output to a channel. The variable and the expression must be assign compatible to the base type of the channel.

TIMER is a special predefined channel from which the actual system time may be read. TIMER need not be opened. The base type of this TIMER channel is the new scalar type *TIME*. Objects of type TIME may be compared to equality and unequality. The new standard module *SysTime* specifies further operations for objects of type TIME.

```
TIMER ? AFTER expression
```

denotes a delay statement. The process executing this statement is suspended until the current system time is later the the time specified by the expression. The general form of the channel operations is:

```
type =
    ... | ChannelType | TIME.
ChannelType =
    CHANNEL OF ChannelBaseType.
ChannelBaseType = type.

ChannelStatement =
    InputStatement |
    TimerInputStatement |
    DelayStatement |
    OutputStatement.
InputStatement =
    channel "?" designator.
TimerInputStatement =
    TIMER "?" designator.
DelayStatement =
    TIMER "?" AFTER time_expression.
OutputStatement =
    channel "!" expression.
channel = designator.
time_expression = expression.
```

2.3 The ALT statement

The ALTernate statement may be used for simultaneous waiting to several events. Events may be communication with other processes or time events. The syntactic structure is:

```
ALT
    guard_1 : stmts_1
  | guard_2 : stmts_2
    ...
  | guard_n : stmts_n
  ELSE stmts_0
END
```

The processes is suspended until one of the guards *guard*$_1$, ... *guard*$_n$ will be ready. From the set of ready guards one arbitrary guard *guard*$_i$ is selected and the corresponding statements stmts$_i$ of the alternative are executed.

There exists three types of guards: simple guards, channel guards, and time guards. They look like:

```
bool_expression
bool_expression , channel ? variable
bool_expression , TIMER ? AFTER
                            time_expr
```

The bool_expression may be omitted for channel and time guards. The bool_expression will be evaluated first if it is present; if it is omitted, it is assumed to be TRUE. A guard is ready if the evaluation of the bool_expression yields in TRUE and

- for the simple guard, no other condition is necessary, .
- for the channel guard, another process waits for communication over *channel*,
- for the time guard, the actual system time is later than the time specified by *time_expr*.

Before executing the statements of a selected channel alternative, the communication over this channel takes place.

The ELSE part of the ALT statement may be omitted. If it is present and none of the bool_expressions is evaluated to TRUE the statements stmts$_0$ are executed. If the ELSE part is omitted and none of the bool_expressions is evaluated to TRUE a runtime error is raised.

The general form of the ALT statement is

```
AltStatement =
    ALT alternative {"|" alternative}
       [ELSE StatementSequence]
    END.
alternative =
    [[replicator] guard ":"
                StatementSequence]].
guard = expression |
    [expression ","] InputStatement |
    [expression ","] DelayStatement.
```

2.4 Replicators

The components of a PAR or ALT statement may be replicated. A replicated process component has the form:

```
[ident : lower_bound TO upper_bound] p
```

(upper_bound - lower_bound + 1) processes are started all executing p in parallel. Each process knows its value of ident, that is each process gets a unique identification value in the range [lower_bound .. upper_bound], which is accessed using ident.

A replicated alternative has the form:

```
[ident : lower_bound TO upper_bound]
         guard : stmts
```

(upper_bound - lower_bound + 1) alternatives are set up, waiting for the guards.

Constraints for the use of the replicator variable:

- The type of ident must be ordinal.
- The type of lower_bound and upper_bound must be assign compatible to that of ident.
- The control variable ident is not allowed to be a component of a structured variable, nor may be imported or parameter of a procedure.
- The control variable may be used only like a constant.

If (upper_bound - lower_bound + 1) ≤ 0 then no component (process or alternative) is created.

The general form of a replicator is

```
replicator =
    "[" ident ":" lwb "TO" upb "]".
lwb = expression.
upb = expression.
```

2.5 Modula-P on distributed systems

The main design problem of a language for programming distributed systems based on Modula-2, comes from the fact, that Modula-2 has a concept for the formulation of abstract data types with memory, which make use of global (level 0) variables. This feature is expressed in Modula-2 by the module concept. Allowing concurrent processes, the variable access problem arises, i.e. what should be done if several processes want to write a variable at a time? Another question comes from the fact that processes may run on different processors, which don't share a common memory. The question here is, on which processor are the global variables allocated, and how is the access realized?

The answer of CSP is not favouring shared storage, because it is another way for process interaction.

Two answers are possible for this problems, first forbid all global variables and second invent language constructs which make the use of global variables possible and save. The first answer is very restrictive and the resulting language is neither a superset of Modula-2 nor should be called Modula-*xy*.

Our solution is to allow shared memory, only if father and son process run on the same processor. Syntactically this is expressed by writing as component process of a PAR statement just a sequence of statements. This son process is named a *local process*. The variable access synchronization problem in this case is left to the programmer.

If the process needs abstract data types with memory and no shared memory is desired, the *process module* encapsulation mechanism is provided. A process declared in a process module is called a *global process*. A global process may define global variables, but one global process can not access the variables of another global process. As a consequence, global processes may run on different processors, without the problems mentioned above.

Both process kinds are specified by the syntax, but it must not be specified by the program on which processor the processes are executed.

Our solution allows the runtime system to distribute global processes over different processors and ensure that processes using common storage are running on the same processor. An an automatic mapping of channels to hardware connections is possible too. Hence the design goals of preserving the programmer from thinking about the allocation of hardware resources and the independence of the program text from the network architecture are fulfilled.

2.6 Process declaration

There are two ways (as a local or global process) to specify the actions a process should execute.

2.6.1 Local processes

Local processes are specified by writing the statement sequence the process should execute as a component of the PAR statement. This implies that this process has access to objects from its father process, which means that shared storage is allowed. No synchronisation is done by the system if several parallel executed local processes access the same variable.

A local process runs on the same processor as the father process, which executes the PAR statement.

A local process is not allowed to contain a RETURN statement nor an EXIT statement, which is related to a LOOP outside of that local process.

2.6.2 Process modules and global processes

The encapsulation in a process module is something like a Modula-2 program module. As a program module a process module may import other modules, and only the name of the program or process is visible outside. While a program module is invoked form the operating system level, a global process is initiated, when the PAR statement containing a call to this process is executed. Like starting a program module causes the bodies of all imported modules to be executed, similarly for the modules imported by that process module is done each time the global process is invoked. And again, like each running program module has its own memory (from the operating system view), a global process has its own memory as long it is active. As concurrently executed Modula programs have no access to variables of each other, a global process has no access to variables of another global process. Unlike a program module a process module may be called with parameters. The formal parameters of a global process may only be value parameters. The types used in the formal parameter list are either predefined types or are implicit imported, hence this type identifier must be qualified. These qualified identifiers are known only inside this parameter list. The formal parameters are declared at level 0.
A process module looks like

```
PROCESS MODULE p (ch : m.channel;
                  x  : m.type;
                  i  : CARDINAL);
(* imports *)
(* local declarations *)
BEGIN
(*the action executed by that process*)
END p.

MODULE prog;
IMPORT P;
 ....
BEGIN
  PAR ....
    | (* invoke the global process *)
      p (ch, x, i)
  END
END prog.
```

The general form of a process module is:

```
ProcessModule =
    PROCESS MODULE ident
         [FormalParameters] ";"
         {import}
         block ident ".".
```

A process module is another compilation unit.
`CompilationUnit = ... | ProcessModule.`
Note, that

- there exists no shared memory with other global processes.
- interaction between global processes is only possible using communication over channels.
- invoking a global process implies that the bodies from all (transitive) imported modules are executed.

3 The Modula-P Transputer development system

The Modula-P Transputer development system [**?**], based on the Modula-2 system MOCKA [**?**] is an implementation of the language Modula-P. The target processor of the system is the Transputer from INMOS. It includes a compiler for the language Modula-P (and hence for Modula-2 too), an automatic *Make* facility, an assembler for the Transputer machine code, a linker and a executive that allows programs to be run on a Transputer. The system is available for a host SUN 3/60 workstation (with SUN OS), to which the Transputer board is connected via a VME bus. The system is written in Modula-2 and runs either on a Transputer or as a cross system on a Sun-3 workstation. The Transputer is used without an own operating system, but full access to the UNIX environment of the host computer is supported.

4 Further work

Further work will be done, to run the system on a multi Transputer network. Another project is to use the HELIOS operating system.

Appendix

A Standard procedures

There is one new standard procedure dealing with channels.

```
PROCEDURE OPEN (VAR channel : ChannelType);
(*************************************)
(* Opening a channel, must be done once *)
(* before the first communication takes *)
(* place.                               *)
(*************************************)
```

B Standard modules

This module provides some procedures, for handling the system time of the computer (here for the Transputer T800). It may be adapted to other systems.

```
DEFINITION MODULE SysTime;
(*****************************************)
(* The properties of 'TIME' are dependent  *)
(* on the computer system running the      *)
(* program. But the procedures below are   *)
(* system independent.                     *)
(* For a Transputer T800 system:           *)
(* The system clock is a cyclic time clock.*)
(* For the low level Transputer T800 clock *)
(* the cycle is approximately 76 hours.    *)
(* A consequence is that a group of times  *)
(* are only unambiguous if they are all    *)
(* contained within a half cycle.          *)
(* Hence (x after y) AND (y after z) does  *)
(* not imply (x after z).                  *)
(* See Transputer manual [INMOS, 1988])    *)
(*****************************************)
CONST TicksPerSec  = (1000 * 1000) DIV 64;
(* 'TicksPerSec' ticks of the low priority *)
(* clock froms exactly one second. The low *)
(* priority clock ticks every 64           *)
(* microseconds.                           *)
TYPE  TimeDiff = INTEGER;
(* Values of type 'TimeDiff' are always *)
(* measured in system clock ticks.      *)
(* Positive Values specify the future,  *)
(* negative the past.                   *)

PROCEDURE Plus (time : TIME;
                diff : TimeDiff) : TIME;
(* returns the system time 'time' + 'diff' *)
PROCEDURE Diff (time1 : TIME;
                time2 : TIME) : TimeDiff;
(* returns the difference 'time1'-'time2' *)
PROCEDURE After (time1 : TIME;
                 time2 : TIME) : BOOLEAN;
(* returns TRUE iff 'time2' specifies a *)
(* time which is later as 'time1'       *)
PROCEDURE Delay (diff  : TimeDiff);
(* delays the process calling this *)
(* procedure for 'diff' ticks      *)
END SysTime.
```

C Example programs

C.1 n to 1 Multiplexer

The first example shows the usage of the replication feature of Modula-P. The program fragment denotes a multiplexer, with n producers and one consumer. An arbitrary number of channels may be multiplexed, using the open array feature of Modula-P.

```
DEFINITION MODULE m;
TYPE ElementType    = ...;
TYPE ElementChannel = CHANNEL OF ElementType;

PROCEDURE generate (process_nr : CARDINAL;
                    VAR elem : ElementType);
(*  generate some data *)
PROCEDURE output (elem : ElementType);
(* output some data *)
END m.

PROCESS MODULE producer
        (process_nr : CARDINAL;
         channel    : m.ElementChannel);
(* process generating a stream of data *)
(* and send them over channel          *)
FROM m IMPORT
     ElementChannel, ElementType, generate;
VAR elem : ElementType;
BEGIN
  LOOP
    generate (process_nr, elem);
    channel ! elem;
  END;
END producer;

PROCESS MODULE multiplex
        (chns : ARRAY OF m.ElementChannel);
(* reads data from the channels and pass *)
(* them to a single consumer.            *)
FROM m IMPORT
     ElementChannel, ElementType, output;
VAR i : CARDINAL; elem : ElementType;
BEGIN
  LOOP
    ALT [i : 0 TO HIGH (chns)]
          chns [i] ? elem : output (elem)
    END;
  END;
END multiplex;
```

```
MODULE prog;
FROM m IMPORT
     ElementChannel, ElementType;
IMPORT multiplex, producer;

PROCEDURE doit(chns:ARRAY OF ElementChannel);
(* a simple n-to-1 multiplexer *)
VAR i : CARDINAL;
BEGIN
  (* first all channels must be opened *)
  FOR i := 0 TO HIGH(chns) DO
    OPEN(chns [i])
  END;
  PAR [i : 0 TO HIGH (chns)]
         producer (i, chns [i])
   | multiplex (chns)
  END;
END doit;
END prog;
```

C.2 Pipeline

The next example shows a parallel program computing prime numbers. It seems silly to compute prime numbers in this way, but this example shows how to construct an arbitrary large pipeline of processes, using recursion.

```
MODULE p_prime;
(*************************************)
(* a concurrent program, computing prime *)
(* numbers                           *)
(*************************************)

FROM  InOut IMPORT WriteInt;
TYPE  IntChannel = CHANNEL OF INTEGER;

PROCEDURE pipe (prime   : INTEGER;
                in      : IntChannel);
(*****************************************)
(* idea: 1. 2 is the first prime number.   *)
(*       2. for each positive number N     *)
(*          greater than 2: if it is not   *)
(*          dividable by any prime less    *)
(*          than N it is prime.            *)
(* method: adapted sieve of Eratostenes.   *)
(* Each prime number is represented by a   *)
(* process. The processes are chained to   *)
(* pipeline. A number is given to a the    *)
(* next process, if it is not dividable by *)
(* the prime number represented by this    *)
(* process. If there                       *)
(* is no next process, the number is prime *)
(* and a new process  is generated.        *)
(* The process is terminated, if a         *)
(* number < 0  is read from the channel.   *)
(*****************************************)
VAR out : IntChannel; x, y : INTEGER;
BEGIN
  (* If this process is generated with *)
  (* prime < 0, this process must be   *)
  (* terminated.                       *)
  IF prime < 0 THEN RETURN END;

  WriteInt (prime,8); WriteLn;
  OPEN (out);
  PAR
     LOOP
       in ? x;
       IF   x < 0
       THEN (* terminate next process *)
            (* in the pipeline.       *)
            out ! -1;
            EXIT (* and terminate self *)
       END;
       IF   x MOD prime = 0
       THEN (* forget the number, *)
            (* is not prime       *)
       ELSE out ! x (* give it to   *)
                    (* next process *)
       END;
     END;
   | out ? y;
     pipe (y, out) (* generate new process *)
  END;
END pipe;
```

```
PROCEDURE do_it (max: INTEGER);
(*****************************************)
(* starts the "first" prime number process *)
(* (i.e. for 2) and the test number        *)
(* generator.                              *)
(* Remark: only odd numbers>2 may be prime *)
(*****************************************)
VAR out : IntChannel; i : INTEGER;
BEGIN
  OPEN (out);
  PAR
     pipe (2, out)
   | (* test number generator *)
     FOR i := 3 TO max BY 2 DO out ! i END;
     out ! -1 (* termination *)
  END;
END do_it;

BEGIN  (* main *)
 do_it (100000);
END p_prime.
```

D Syntax

The syntactic rules described here are an extension of the Modula-2 syntax given in [**?**]. The syntax is given in EBNF.

```
CompilationUnit =
    ... | ProcessModule.
ProcessModule =
   PROCESS MODULE ident
       [FormalParameters] ";"
       {import}
       block ident ".".
```

The rule statement is extented for the non terminal symbols ParStatement, AltStatement and ChannelStatement.

```
statement =
    [  ... |
     ParStatement |
     AltStatement |
     ChannelStatement ].
ParStatement =
    PAR Process {"|" Process} END.
Process =
    [[replicator] LocalProcess] |
    [[replicator] GlobalProcess].
LocalProcess = StatementSequence.
GlobalProcess = ProcedureCall [";"].
AltStatement =
    ALT alternative { "|" alternative }
    [ELSE StatementSequence]
    END.
alternative =
    [[replicator] guard ":"
                StatementSequence]].
guard = expression |
    [expression "," InputStatement |
    [expression "," DelayStatement.
replicator =
    "[" ident ":" lwb "TO" upb "]".
lwb = expression.
upb = expression.
ChannelStatement =
    InputStatement |
    TimerInputStatement |
    DelayStatement |
    OutputStatement.
InputStatement =
    channel "?" designator.
TimerInputStatement =
    TIMER "?" designator.
DelayStatement =
    TIMER "?" AFTER time_expression.
OutputStatement =
    channel "!" expression.
channel = designator.
time_expression = expression.
```

The type rule is extended for the non terminal symbol ChannelType and and the terminal symbol TIME.

```
type = ... | ChannelType | TIME.
ChannelType =
    CHANNEL OF ChannelBaseType.
ChannelBaseType = type.
```

The underscore character ("_") is now allowed as significant letter.

```
letter = "A" |..| "Z" | "a" |..| "z" | "_".
```

References

[Hoare, 1985] Hoare, C. (1985). *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, Inc.

[INMOS, 1984] INMOS, editor (1984). *occam Programming Manual*. Prentice-Hall, Inc.

[INMOS, 1988] INMOS, editor (1988). *The Transputer instruction set - a compiler writers' guide*. Prentice-Hall, Inc.

[Schröer, 1988] Schröer, F. (1988). Das GMD Modula-2 Entwicklungssystem. *GMD-Spiegel*.

[Vollmer, 1989] Vollmer, J. (1989). Kommunizierende sequentielle Prozesse in Modula-2; Entwurf und Implementierung eines Transputer – Entwicklungssystems. Master's thesis, Universität Karlsruhe, Fakultät für Informatik.

[Wirth, 1985] Wirth, N. (1985). *Programming in Modula-2*. Springer Verlag, Heidelberg, New York, 3 edition.