CC-MODULA: A MODULA-2 TOOL TO TEACH CONCURRENT PROGRAMMING

Rafael Morales-Fernández
Juan José Moreno-Navarro

Departamento de Lenguajes y Sistemas
Informáticos e Ingenieria de Software
Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo - Boadilla del Monte
28660 Madrid - SPAIN

ABSTRACT

The present work shows an educational experience at University level in the field of concurrent programming. CC-Modula, a tool to teach concurrent programming using a single language, is presented. It could also be considered as a contribution to the development of methods to implement concurrent mechanisms, in the frame of centralized and distributed operating systems.

CC-Modula is a Modula-2 package that allows the use of the best known abstract mechanisms of concurrency. CC-Modula handles parallelism between processes and contains mechanisms to synchronize them. Mechanisms based on shared storage as well as mechanisms based on message passing that implement the CSP schema are included.

INTRODUCTION

Concurrent programming has been usually taught in operating systems courses. But this idea does not correspond with the actual development of applications in the industry. Most of the current applications include real time features.

For this reason we mantain that concurrent programming is a discipline that should be included in all the undergraduate computer science curricula. This could be achieved as an independent course and its aim must be the introduction of the best known abstract mechanisms of concurrency, avoiding details about how concurrency is implemented. Only after this course, a student should learn about this implementation. Recently a panel in the 19th SIGCSE Technical Symposium on Computer Science Education [15] was a focus for a further discussion on the subject.

Historically, abstractions for concurrency have been developed as follows:

Concurrent programming, defined as cooperating processes, impose the need of comunicating and synchronizing them. A first approach to this problem was to use mechanisms based on the availability of shared memory. The best known of these are Semaphores, Critical Regions, Conditional Critical Regions and Monitors. A complete description of these mechanisms can be found in [1] and [2]. These mechanisms are used in some languages, and Concurrent Pascal [3] is, perhaps, the most interesting one.

The solution of using shared memory is not applicable when dealing with a distributed and decentralized system. This problem brings about the idea of treating communication between processes through message exchanges. The most celebrated proposal in this field, usually used as a reference, is the Hoare's CSP schema ([7] and [8]), in which the sending and the reception of messages are the basic communication and synchronization operations.

Another interesting feature of CSP is the possibility of programming selective receptions in a nondeterministic way by using the guarded command construction introduced by Dijkstra. Modern languages, like Occam [9] and, in a certain sense, Ada [6], take advantage of this idea.

Modula-2 is a high level programming language developed by Wirth [13]. It is a descendant of Pascal and has modularity as its main feature. Even though it does not provide real concurrency, it allows programming with coroutines, thus obtaining a quasi-concurrency.

The fact that Modula-2 does not include concurrent programming operations has been considered by some authors as an advantage. They claim that Modula-2 provides the facilities to work with concurrency by giving the programmer the possibility to design his own mechanisms as he needs them.

There are some experiences in the use of Modula-2 to develop concurrent mechanisms. Wirth in [13] describes a module for signal handling that allows concurrent execution of a number of processes at the same time. However, parallel command nesting is not possible in this implementation. Sewry [11] improves Wirth's module by adding the possibility of terminating processes, and including a COBEGIN..COEND structure for executing processes in parallel. As before, this structure cannot be nested. Sewry [12] again develops some ideas to implement monitors. Wirth [14] implements some aspects of CSP, like a parallel command (that can be nested, but restricted to two processes) and the reception and the sending of messages.

Brumfield [4] talks about the previous idea that teaching concurrent programming is not only a task of operating systems instructors. He also proposes Modula-2 as a language to practice with concurrent programming. He has developed a Modula- 2 Process Manager that provides the user with types and procedures to write concurrent programs. The mechanisms allowed are process handling, semaphores, events and message links. It seems that a process can create another process but making explicit the suspension of the current process. The work of Brumfield supposes a good advance in the development of mechanisms to work in concurrency in Modula-2, but we believe that the syntax he proposes is something different of the most known concurrent languages. Moreover, it already does not implement some mechanisms and some of them are incomplete.

Although these proposals are very interesting, they are not complete and contain several restrictions.

Our tool, CC-Modula, implements all the classic mechanisms of concurrency based on shared storage (semaphores, critical regions, conditional critical regions and monitors). It also includes a complete implementation of the CSP schema (message exchange via channels and selective reception by using guarded commands). These mechanisms can be used in a program as extensions of the Modula-2 language.

The advantages of CC-Modula with respect to other languages are its own generality (it includes a large set of mechanisms) without the need of using different languages, and the possibility of working within the frame of Modula-2. During our previous experience in teaching concurrent programming, we had to use several different compilers (Concurrent Pascal, Ada, Occam) instead of a single tool. Furthermore these languages are not available for many of the computer systems. CC-Modula proves to be the adequate solution to these problems.

CC-MODULA

CC-Modula includes several Modula-2 extensions. All of them can be used in Modula-2 programs. A Modula-2 user only needs some additional syntax. Every new construction has been designed by trying to adapt it to the most useful proposals for each structure and in the Modula-2 syntax style. The next sections describe these extensions.

Process Declaration.

Every process in a concurrent program must be declared like a parameterless procedure, using the keyword TASK instead of PROCEDURE.

```
TASK name;
  ..procedure body..
END name;
```

Parallel Execution.

A COBEGIN..COEND structure denotes parallel execution of a set of processes.

```
COBEGIN
  Process1;
  ...
  ProcessN
COEND
```

Each process is invoked by its name. A process body can contain other COBEGIN..COEND statements.

Semaphores.

A semaphore must be declared as:

```
VAR sem : ( BSEMAPHORE | SEMAPHORE )
                        [INIT initial value]
```

A semaphore can be accessed by the procedures:

```
Wait (sem)
Signal (sem)
```

that implement the operations P and V respectively.

Critical Regions.

Critical regions are introduced by declaring:

```
VAR name : SHARED type
```

Shared variables can only be used inside a REGION statement.

```
REGION name DO
  ..statements..
END
```

Conditional Critical Regions.

A critical region can be accessed conditionally by means of the AWAIT statement in the form

```
        REGION name DO
          ..statements..
          AWAIT (boolean condition);
          ..statements..
        END
```

The condition to wait for can be any boolean expression, usually referring to the shared variables.

Monitors.

A monitor must be declared like a module in the form:

```
        MONITOR list of instance names;
        PUBLIC  procedure names list;
        VAR  ..local variables declaration..;

        PROCEDURE  Proc1 ( parameters );
          ......
        PROCEDURE  ProcN ( parameters );

        BEGIN
          ..initialization statements..
        END MONITOR
```

This declaration generates a set of distinct monitor instances, each one with its own name. Public procedures are called by qualifying its name in the form: MonitorName.ProcName (arguments).

Operations inside a monitor can be explicity delayed, if necessary, using "CONDITIONS" that work in the same way of "queues" in Concurrent Pascal.

```
        VAR  cond : CONDITION;
```

The operations to work with them are

```
        Delay ( cond )

        Continue ( cond )
```

Message Exchange.

If there is no shared storage (for instance, in a distributed system) we can not have shared variables. The synchronization and cooperation between processes can be done only by means of message exchanges.

This communication is made through channels. There is a predefined CHANNEL type, from which variables of this type can be declared.

```
        VAR  chann : CHANNEL;
```

Any process can send messages over a given channel, but only one can receive from it. Some auxiliary operations for handling channels are provided.

Communication primitives are introduced as procedures:

```
        Send (message, channel)

        Receive (message, channel)
```

Messages sent over a channel may be of any type. The agreement upon the format of messages in the reception is a responsibility of the programmer using the Send and Receive procedures in a consistent way.

In the original CSP proposal, process names are used to specify communication source and destination. By using channel names instead of process names, CC-Modula makes it easier to implement remote procedure calls via message exchange.

Nondeterministic Reception.

Guarded commands, originally proposed by Dijkstra, are included in CSP to allow selective message reception. Using an Ada-like notation this construction can be written as:

```
        SELECT
          WHEN condition1,
               Receive (message1, Channel1)
            DO  .. action1 ..
          ...
          WHEN conditionN,
               Receive (messageN, ChannelN)
            DO  .. actionN ..
          [ELSE  .. default action ..]
        END
```

where each action is a sequence of statements.

EXAMPLES

In order to show how CC-Modula can be used to write concurrent programs, we present here a detailed example developing different alternative solutions to it. The example is the well known bounded buffer problem. The programs can be found as an appendix of the text.

The following solutions are presented: the classic solutions using semaphores, conditional critical regions and monitors; a solution using CSP; and a version of the problem in Ada is also included. It is possible to make a comparison between all the abstract mechanisms for concurrency by studying the different version.

The Ada version has been included in order to compare it with the CSP one. There are no big differences between both versions, so concurrent features of the Ada language can be transcribed into CC-Modula in a rather straightforward manner.

FUTURE VERSIONS

As a future work we are going to develop some improvements of CC- Modula:

- A timeout system for the communication between processes. This timeout will also be available for the receptions in a SELECT statement.

- A PAUSE statement to delay a process for a certain amount of time, specified as an argument of the PAUSE procedure.

- Fault-Tolerance mechanisms using FT-Actions (like atomic actions, see [10]) and operations to use them.

- More detailed and useful error messages during compilation and execution.

- A Trace Facility to allow the user to have a complete vision of the process life (i. e. writting a message every time a process executes a CC-Modula operation if the Trace Facility is activated).

- Debug facilities in order to know which is the cause of a DEADLOCK during the execution of a program. The system can show a table with all the processes and the reason, and the number of the line, they are waiting for.

All of them will configure a future version of CC-Modula that is being developed now. Specially the last three features (which deals with the developping of concurrent programs) are very interesting. One of the major problems in writting parallel programs are the detection of deadlocks, livelocks, etc. The tool gives the hints to solve the problem and the user learns the appropiate methodology to develop such this programs.

Furthermore, the accomplish of CC-Modula with a window system supposes a very nice and useful way to test concurrent programs.

CONCLUSIONS

CC-Modula is a concurrent programming laboratory. In a single package we provide the user the main types and basic primitives for process comunication, synchronization and parallelism based on shared storage or message passing.

The basis of all these mechanisms are provided in the Kernel Module. A partial description of this module can be found in [5], that shows how to implement CSP primitives. It also contains a longer example that illustrates how Ada concurrent features can be transcribed into CC-Modula in a rather straighforward manner.

The Kernel Module is written in standard Modula-2. Our implementation works under the VAX-11 Modula-2 compiler developed at the Fachbereich Informatik, Hamburg Universität or under a PC computer using the Logitech Modula-2 compiler. Both versions are available in our address.

From the technical point of view, the main contribution is the implementation of:

- a COBEGIN..COEND structure that can be nested

- the critical regions and the conditional access to them

- the CSP primitives including the selective reception by means of guarded commands.

All the mechanisms are allowed as Modula-2 extensions with the help of a pre-processor. Programs written in CC-Modula are compiled by a standard compiler after a pre-processing step and then linked with the Kernel Module. Moreover, CC-Modula syntax is quite similar to the more celebrated concurrent languages (like Concurrent Pascal or Ada).

It is possible to use this laboratory for several tasks. Currently, we are using it to:

- Teach concurrent programming at the University. We have collected several examples and developed alternative solutions by using different mechanisms (if possible) in order to compare them. One of these examples has been shown here. The collection of applications includes from introductory exercises to more elaborated examples.

- Clarify the interest, limitations and practical use of the different concurrent mechanisms. Their use in distributed systems is specially interesting to us.

- Test parallel computing algorithms. The debug facilities will help to find and solve the errors.

ACKNOWLEDGEMENTS

The authors would like to thank professor M. Collado for their helpful comments and contributions to this paper.

REFERENCES

[1] Andrews, R., F. B. Schneider: Concepts and Notations for Concurrent Programming. ACM Computing Surveys V. 15 N. 1, March 1983, pp 3-43.

[2] Brinch Hansen, P.: Operating System Principles. Prentice Hall, 1973.

[3] Brinch Hansen, P.: The Programming Language Concurrent Pascal. IEEE Trans. Softw. Eng. V. 1, N. 2, February 1975, pp 199-207.

[4] Brumfield, J. A.: Concurrent Programming in Modula-2. Proceedings 18th SIGCSE Technical Symposium on Computer Science Education, SIGCSE Bulletin V. 19, N. 1, February, 1987, pp 191-200.

[5] Collado, M., R. Morales, J. J. Moreno: A Modula-2 Implementation of CSP. ACM Sigplan Notices. V. 22, N. 6, June 1987, pp 25-38.

[6] DoD: Reference Manual for the Ada Programming Language. ANSI/MIL-STD 1815A. 1983.

[7] Hoare, C. A. R.: Communicating Sequential Processes. Comm. ACM V. 21 N. 8, August 1978, pp 666-677.

[8] Hoare, C. A. R.: Communicating Sequential Processes. Prentice Hall. 1985.

[9] INMOS Limited: Occam-2 Programming Manual. Prentice Hall. 1987.

[10] Jalote, P., R. H. Cambell: Atomic Actions for Fault- Tolerance Using CSP. IEEE Transactions on Software Engineering. V. 12, N. 1 January 1986, pp 59-68.

[11] Sewry, D. A.: Modula-2 Process Facilities. ACM Sigplan Notices V. 19, N. 11 November 1984, pp 23-32.

[12] Sewry, D. A.: Modula-2 and the Monitor Concept. ACM Sigplan Notices V. 19, N. 11 November 1984, pp 33-41.

[13] Wirth, N.: Programming in Modula-2 and Report on the Programming Language Modula-2. Springer-Verlag. 1982.

[14] Wirth, N.: Schemes for Multiprogramming and their Implementation in Modula-2. Technical Report 59 ETH Zurich Institut für Informatik. 1984.

[15] Panel: Concurrency in the Undergraduate Curriculum. 19th SIGCSE Technical Symposium on Computer Science Education, SIGCSE Bulletin V. 20, N. 1, February 1988, pp 42.

APPENDIX: EXAMPLES

SOLUTION USING SEMAPHORES

```
MODULE BoundedBuffer;

CONST Size = << ... >>;

TYPE MessageType = ARRAY [1..2] OF CHAR;

VAR   sem : BSEMAPHORE INIT TRUE;
      full : SEMAPHORE INIT 0;
      free : SEMAPHORE INIT Size;
      b : ARRAY [1..Size] OF MessageType;
      in, out : INTEGER;
      count : INTEGER;

TASK Producer;
VAR   message : MessageType;
      ch : CHAR;
BEGIN
  FOR ch := 'a' TO 'z' DO
    message [1] := ch;
    message [2] := CHR (ORD (ch) + ORD ('A') - ORD ('a'));
    Wait (free);
    Wait (sem);
    b[in] := message;
    in := in MOD Size + 1;
    count := count + 1;
    Signal (sem);
    Signal (full)
  END
END Producer;

TASK Consumer;
VAR   message : MessageType;
BEGIN
  REPEAT
    Wait (full);
    Wait (sem);
    message := b[out];
    out := out MOD Size + 1;
    count := count - 1;
    Signal (sem);
    Signal (free);
  UNTIL  message [1] = 'z'
END Consumer;

BEGIN   (* BoundedBuffer *)
  in := 1;
  out := 1;
  count := 0;
  COBEGIN
    Producer;
    Consumer
  COEND
END BoundedBuffer.
```

SOLUTION USING CONDITIONAL CRITICAL REGIONS

```
MODULE BoundedBuffer;

CONST  Size = << ... >>;

TYPE MessageType = ARRAY [1..2] OF CHAR;

VAR  Buffer : SHARED
              RECORD
                 b : ARRAY [1..Size] OF MessageType;
                 in, out, count : INTEGER
              END;

TASK Producer;
VAR  ch : CHAR;
     message : MessageType;
BEGIN
  FOR ch := 'a' TO 'z' DO
    message [1] := ch;
    message [2] := CHR (ORD (ch) + ORD ('A') - ORD ('a'));
    REGION  Buffer  DO
       WITH Buffer DO
          AWAIT ( count < Size );
          b[in] := message;
          in := in MOD Size + 1;
          count := count + 1;
       END
    END;
  END
END Producer;

TASK Consumer;
VAR  message : MessageType;
BEGIN
  REPEAT
    REGION  Buffer  DO
       WITH Buffer DO
          AWAIT ( count > 0 );
          message := b[out];
          out := out MOD Size + 1;
          count := count - 1;
       END
    END
  UNTIL  message [1] = 'z'
END Consumer;

BEGIN   (* BoundedBuffer *)
   in := 1;
   out := 1;
   count := 0;
   COBEGIN
      Producer;
      Consumer
   COEND
END BoundedBuffer.
```

SOLUTION USING MONITORS

```
MODULE BoundedBuffer;

CONST  Size = << ... >>;
TYPE  MessageType = ARRAY [1..2] OF CHAR;

MONITOR Buffer;
IMPORT  Size;
PUBLIC Append, Take;

VAR  b : ARRAY[1..Size] OF MessageType;
         count, in, out : INTEGER;
         nonempty, nonfull : CONDITION;

PROCEDURE Append ( message : MessageType);
BEGIN
   IF  count = Size  THEN Delay (nonfull) END;
   b [in] := message;
   count := count + 1;
   in := in MOD Size + 1;
   Continue (nonempty)
END Append;

PROCEDURE Take ( VAR message : MessageType);
BEGIN
   IF  count = 0  THEN Delay (nonempty) END;
   message := b[out]; count := count - 1;
   out := out MOD Size + 1;
   Continue (nonfull)
END Take;

BEGIN  (* Buffer *)
   in := 1; out := 1; count := 0
END MONITOR;

TASK Producer;
VAR message : MessageType;
   ch : CHAR;
BEGIN
   FOR ch := 'a' TO 'z' DO
       message [1] := ch;
       message [2] := CHR (ORD (ch) + ORD ('A') - ORD ('a'));
       Buffer.Append (message)
   END
END Producer;

TASK Consumer;
VAR message : MessageType;
BEGIN
   REPEAT
      Buffer.Take (message);
   UNTIL  message [1] = 'z'
END Consumer;

BEGIN  (* BoundedBuffer *)
   COBEGIN
      Producer;
      Consumer
   COEND
END BoundedBuffer.
```

SOLUTION USING CSP

```
MODULE  BoundedBuffer;

CONST  Size = << ... >>;
TYPE  MessageType = ARRAY [1..2] OF CHAR;
VAR  BuffIn, BuffOut, Return : CHANNEL;

TASK Buffer;
VAR  b: ARRAY [1..Size] OF MessageType;
       in, out, count : INTEGER;
       rc: CHANNEL;
       more: BOOLEAN;
BEGIN
  count := 0; in := 1; out := 1; more := TRUE;
  GetChannel(BuffIn); GetChannel(BuffOut);
  LOOP
    SELECT
       WHEN (more AND (count > 0)), Receive (BuffOut, rc) DO
          Send(rc, b[out]);
          IF b[out] [1] = 'O' THEN more := FALSE END;
          out := out MOD Size + 1; count := count - 1
       WHEN (more AND (count < Size)), Receive (BuffIn, b[in]) DO
          in := in MOD Size + 1; count := count + 1
       ELSE EXIT (* Not more *)
    END  (* SELECT *)
  END (* LOOP *)
END Buffer;

TASK Producer;
VAR  ch : CHAR;
      message : MessageType;
BEGIN
  FOR ch := 'a' TO 'z' DO
    message [1] := ch;
    message [2] := CHR (ORD (ch) + ORD ('A') - ORD ('a'));
    Send(BuffIn, message)
  END;
  message [1] := 'O'; message [2] := 'O';
  Send (BuffIn, message)
END Producer;

TASK Consumer;
VAR  message : MessageType;
BEGIN
  GetChannel(Return);
  REPEAT
    Send(BuffOut, Return);
    Receive (Return, message);
  UNTIL message [1] = 'O'
END Consumer;

BEGIN (* BoundedBuffer *)
  COBEGIN
    Producer;
    Consumer;
    Buffer
  COEND
END BoundedBuffer.
```

SOLUTION USING ADA

```
PROCEDURE Bounded_Buffer IS
  TYPE  MessageType IS  STRING (1..2);

  TASK Buffer IS
    ENTRY Append (x : IN  MessageType );
    ENTRY Take (y : OUT  MessageType );
  END Buffer;
  TASK Producer;
  TASK Consumer;

  TASK BODY Buffer IS
    Size: CONSTANT INTEGER:= << ... >>;
    b : ARRAY (1..Size) OF  MessageType ;
    inc, outc: INTEGER RANGE 1..Size := 1;
    count: INTEGER RANGE 0..Size := 0;
    more : Boolean := TRUE;
  BEGIN
    LOOP
      SELECT
           WHEN more  AND  ( count > 0 ) =>
             ACCEPT Take (y : OUT  MessageType ) DO
               y := b(outc);
             END Take;
             IF ( b ( outc, 1 ) = 'O' ) THEN more := FALSE; END IF;
             outc := outc MOD Size + 1; count := count -1;
        OR WHEN more  AND  ( count < Size ) =>
             ACCEPT Append (x : IN  MessageType ) DO
               b(inc) := x;
             END Append;
             inc := inc MOD Size + 1; count := count + 1;
        OR TERMINATE
      END SELECT;
    END LOOP;
  END Buffer;

  TASK BODY Producer IS
    change : CONSTANT INTEGER := CHARACTER'POS('A') - CHARACTER'POS('a');
    ch : CHARACTER;
    message : MessageType;
  BEGIN
    FOR  ch IN  'a'..'z'  LOOP
      message(1) := ch;
      message(2) := CHARACTER'VAL (CHARACTER'POS(ch) + change);
      Buffer.Append (message);
    END LOOP;
    Buffer.Append ("OO");
  END Producer;

  TASK BODY Consumer IS
    message : MessageType;
  BEGIN
    LOOP
      Buffer.Take (message);
      EXIT WHEN (message(1) = 'O');
    END LOOP;
  END Consumer;
BEGIN
  NULL;
END Bounded_Buffer;
```