An Occam Compiler

Kees Bot
Edwin Scheffer

Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

This document describes the implementation of an **Occam** to **EM** compiler. The lexical analysis is done using **Lex**. For the semantic analysis the extended LL(1) parser generator **LLgen** is used. To handle the Occam-specific features as channels and parallelism some library routines are required.

1. Introduction

Occam [1] is a programming language which is based on the concepts of concurrency and communication. These concepts enable today's applications of microprocessors and computers to be implemented more effectively.

An Occam program consists of a (dynamically determined) number of processes communicating through channels. To communicate with the outside world some predefined channels are needed. A channel has only one writer and one reader; it carries machine words and bytes, at the reader/writer's discretion. The process with its communication in Occam replaces the procedure with parameters in other languages (there are no procedures in Occam).

In addition to the normal assignment statement, Occam has two more information-transfer statements, the input and the output:

```
chan1 ? x        -- reads a value from chan1 into x
chan2 ! x        -- writes the value of x onto chan2
```

Both the outputting and the inputting processes wait until the other is there. Channels are declared and given names. Arrays of channels are possible.

Processes come in 5 varieties: sequential, parallel, alternative, conditional and repetitive. A process starts with a reserved word telling its nature, followed by an indented list of other processes. (Indentation is used to indicate block structure.) It may be preceded by declarations. The processes in a sequential/parallel process are executed sequentially/in parallel. The processes in an alternative process have guards based on the availability of input; the first to be ready is executed (this is waiting for multiple input). The conditional and repetitive processes are normal **IF**s and **WHILE**s.

Producer-consumer example:

```
CHAN buffer:                    -- declares the channel buffer
PAR
  WHILE TRUE                    -- the producer
    VAR x:                      -- a local variable
    SEQ
      produce(x)                -- in some way
      buffer ! x                -- and send it
  WHILE TRUE                    -- the consumer
    VAR x:
    SEQ
      buffer ? x                -- get a value
      consume(x)                -- in some way
```

Processes can be replicated from a given template; this combines with arrays of variables and/or channels.

Example: 20 window-sorters in series:

```
CHAN s[20]:                       -- 20 channels
PAR i = [ 0 FOR 19 ]              -- 19 processes
  WHILE TRUE
    VAR v1, v2:
    SEQ
      s[i] ? v1; v2               -- wait for 2 variables from s[i]
      IF
        v1 <= v2                  -- ok
          s[i+1] ! v1; v2
        v1 > v2                   -- reorder
          s[i+1] ! v2; v1
```

A process may wait for a condition, which must include a comparison with **NOW**, the present clock value.

Processes may be distributed over several processors; all processes under a **VAR** declaration must run on the same processor. Concurrency can be improved by avoiding **VAR** declarations, and replacing them by **CHAN** declarations. Processes can be allocated explicitly on named processors and channels can be connected to physical ports.

2. The Compiler

The compiler is written in **C** using LLgen and Lex and compiles Occam programs to EM code, using the procedural interface as defined for EM. In the following sub-sections we describe the LLgen parser generator and the aspect of indentation.

2.1. The LLgen Parser Generator

LLgen accepts a Context Free syntax extended with the operators '`*`', '`?`' and '`+`' that have effects similar to those in regular expressions. The '`*`' is the closure set operator without an upperbound; '`+`' is the positive closure operator without an upperbound; '`?`' is the optional operator; '`[`' and '`]`' can be used for grouping. For example, a comma-separated list of expressions can be described as:

```
expression_list:
      expression [ ',' expression ]*
    ;
```

Alternatives must be separated by '`|`'. C code ("actions") can be inserted at all points between the colon and the semicolon. Variables global to the complete rule can be declared just in front of the colon enclosed in the brackets '`{`' and '`}`'. All other declarations are local to their actions. Nonterminals can have parameters to pass information. A more mature version of the above example would be:

```
expression_list(expr *e;)       {  expr e1, e2; } :
        expression(&e1)
        [ ',' expression(&e2)
                                {  e1=append(e1, e2);   }
        ]*
                                {  *e=e1;   }
      ;
```

As LLgen generates a recursive-descent parser with no backtrack, it must at all times be able to determine what to do, based on the current input symbol. Unfortunately, this cannot be done for all grammars. Two kinds of conflicts are possible, viz. the **alternation** and **repetition** conflict. An alternation confict arises if two sides of an alternation can start with the same symbol. E.g.

```
plus:   '+' | '+' ;
```

The parser doesn't know which '`+`' to choose (neither do we). Such a conflict can be resolved by putting an **if-condition** in front of the first conflicting production. It consists of a **"%if"** followed by a C-expression between parentheses. If a conflict occurs (and only if it does) the C-expression is evaluated and parsing

continues along this path if non-zero. Example:

```
plus:
      %if (some_plusses_are_more_equal_than_others())
      '+'
    |
      '+'
    ;
```

A repetition conflict arises when the parser cannot decide whether "`productionrule`" in e.g. "`[ productionrule ]*`" must be chosen once more, or that it should continue. This kind of conflicts can be resolved by putting a **while-condition** right after the opening parentheses. It consists of a **"%while"** followed by a C-expression between parentheses. As an example, we can look at the **comma-expression** in C. The comma may only be used for the comma-expression if the total expression is not part of another comma-separated list:

```
comma_expression:
      sub_expression
      [ %while (not_part_of_comma_separated_list())
          ',' sub_expression
      ]*
    ;
```

Again, the **"%while"** is only used in case of a conflict.

Error recovery is done almost completely automatically. All the LLgen-user has to do is write a routine called *LLmessage* to give the necessary error messages and supply information about terminals found missing.

2.2. Indentation

The way conflicts can be resolved are of great use to Occam. The use of indentation, to group statements, leads to many conflicts because the spaces used for indentation are just token separators to the lexical analyzer, i.e. "white space". The lexical analyzer can be instructed to generate 'BEGIN' and 'END' tokens at each indentation change, but that leads to great difficulties as expressions may occupy several lines, thus leading to indentation changes at the strangest moments. So we decided to resolve the conflicts by looking at the indentation ourselves. The lexical analyzer puts the current indentation level in the global variable *ind* for use by the parser. The best example is the **SEQ** construct, which exists in two flavors, one with a replicator and one process:

```
seq i = [ 1 for str[byte 0] ]
    out ! str[byte i]
```

and one without a replicator and several processes:

```
seq
    in ? c
    out ! c
```

The LLgen skeleton grammar to handle these two is:

```
SEQ         {   line=yylineno; oind=ind; }
[     %if (line==yylineno)
      replicator
      process
    |
      [ %while (ind>oind) process ]*
]
```

This shows clearly that, a replicator must be on the same line as the **SEQ**, and new processes are collected as long as the indentation level of each process is greater than the indentation level of **SEQ** (with appropriate checks on this identation).

Different indentation styles are accepted, as long as the same amount of spaces is used for each indentation shift. The ascii tab character sets the indentation level to an eight space boundary. The first indentation level found in a file is used to compare all other indentation levels to.

3. Implementation

It is now time to describe the implementation of some of the occam-specific features such as channels and **NOW**. Also the way communication with UNIX† is performed must be described. For a thorough description of the library routines to simulate parallelism, which are e.g. used by the channel routines and by the **PAR** construct in Appendix B, see [6].

3.1. Channels

There are currently two types of channels (see Figure 1.) indicated by the type field of a channel variable:

- An interprocess communication channel with two additional fields:
  - A synchronization field to hold the state of an interprocess communication channel.
  - An integer variable to hold the value to be send.
- An outside world communication channel. This is a member of an array of channels connected to UNIX files. Its additional fields are:
  - A flags field holding a readahead flag and a flag that tells if this channel variable is currently connected to a file.
  - A preread character, if readahead is done.
  - An index field to find the corresponding UNIX file.

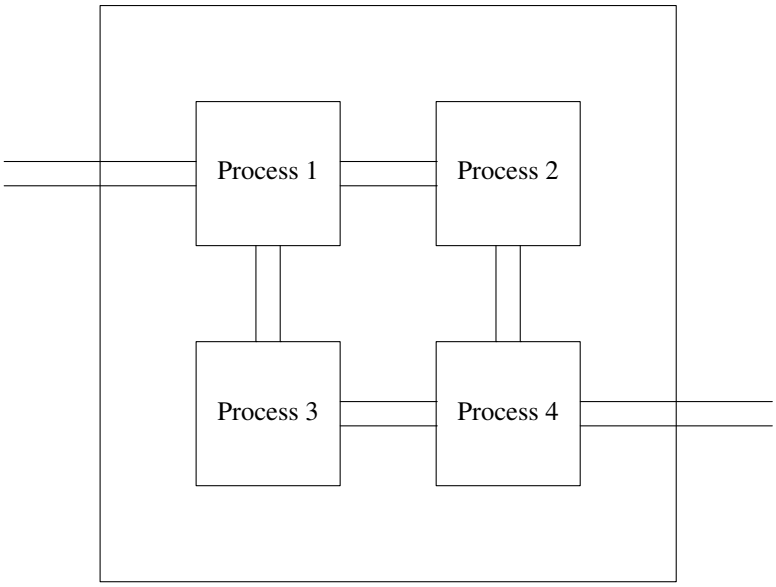

Figure 1. Interprocess and outside world communication channels

The basic channel handling is done by `chan_in` and `chan_out`. All other routines are based on them. The routine `chan_any` only checks if there's a value available on a given channel. (It does not read this value!) `C_init` initializes an array of interprocess communication channels.

The following table shows Occam statements paired with the routines used to execute them.

Occam statement	Channel handling routine	Called as
CHAN c: CHAN c[z]:	c_init(c, z) chan *c; unsigned z;	c_init(&c, 1); c_init(&c, z);
c ? v	chan_in(v, c) long *v; chan *c;	chan_in(&v, &c);

† UNIX is a trademark of Bell Laboratories

Occam statement	Channel handling routine	Called as
c ? b[byte i]	cbyte_in(b, c) char *b; chan *c;	cbyte_in(&b[i], &c);
c ? a[i for z]	c_wa_in(a, z, c) long *a; unsigned z; chan *c;	c_wa_in(&a[i], z, &c);
c ? a[byte i for z]	c_ba_in(a, z, c) long *a; unsigned z; chan *c;	c_ba_in(&a[i], z, &c);
c ! v	chan_out(v, c) long *v; chan *c;	chan_out(&v, &c);
c ! a[i for z]	c_wa_out(a, z, c) long *a; unsigned z; chan *c;	c_wa_out(&a[i], z, &c);
c ! a[byte i for z]	c_ba_out(a, z, c) long *a; unsigned z; chan *c;	c_ba_out(&a[i], z, &c);
alt c ?	int chan_any(c) chan *c;	deadlock=0; for(;;) { if (chan_any(&c)) { ....

The code of `c_init`, `chan_in`, `chan_out` and `chan_any` can be found in Appendix A.

3.1.1. Synchronization on interprocess communication channels

The synchronization field can hold three different values indicating the state the channel is in:

- **C_S_FREE**: Ground state, channel not in use.
- **C_S_ANY**: Channel holds a value, the sending process is waiting for an acknowledgement about its receipt.
- **C_S_ACK**: Channel data has been removed by a receiving process, the sending process can set the channel free now.

A sending process cannot simply wait until the channel changes state C_S_ANY to state C_S_FREE before it continues. There is a third state needed to prevent a third process from using the channel before our sending process is acknowledged. Note, however that it is not allowed to use a channel for input or output in more than one parallel process. This is too difficult to check in practice, so we tried to smooth it a little.

3.2. NOW

NOW evaluates to the current time returned by the time(2) system call. The code is simply:

```
long now()
{
    deadlock=0;
    return time((long *) 0);
}
```

The "deadlock=0" prevents deadlocks while using the clock.

3.3. UNIX interface

To handle the communication with the outside world the following channels are defined:

- **input**, that corresponds with the standard input file,
- **output**, that corresponds with the standard output file,
- **error**, that corresponds with the standard error file.
- **file**, an array of channels that can be subscripted with an index obtained by the builtin named process "`open`". Note that **input=file**[0], **output=file**[1] and **error=file**[2].

Builtin named processes to open and close files are defined as

```
proc open(var index, value name[], mode[]) = ..... :
proc close(value index) = ..... :
```

To open a file 'junk', write nonsense onto it, and close it, goes as follows:

```
var i:
seq
    open(i, "junk", "w")
    file[i] ! nonsense
    close(i)
```

Errors opening a file are reported by a negative index, which is the negative value of the error number (called *errno* in UNIX).

Bytes read from or written onto these channels are taken from occam variables. As these variables can hold more than 256 values, some negative values are used to control channels. These values are:

- **EOF** (-1): Eof from file channel is read as -1.
- **TEXT** (-2): A -2 written onto any channel connected to a terminal puts this terminal in the normal line oriented mode (i.e. characters typed are echoed and lines are buffered before they are read).
- **RAW** (-3): A -3 written onto any channel connected to a terminal puts it in raw mode (i.e. no echoing of typed characters and no line buffering).

To exit an Occam program, e.g. after an error, a builtin named process `exit` is available that takes an exit code as its argument.

3.4. Replicators and slices

Both the base and the count of replicators like in

```
par i = [ base for count ]
```

may be arbitrary expressions. The count in array slices like in

```
c ? A[ base for count ]
```

must be a constant expression however, the base is again free.

4. Particular details

4.1. Lower case/Upper case

Keywords must be either fully written in lower case or in upper case, thus **PAR** is equivalent to **par** but **Par** is not a keyword. Identifiers may be of mixed case. Different styles are used in our examples just to indicate what's accepted by the compiler.

4.2. File inclusion

The C preprocessor is applied to the input file before compilation, so that files containing useful **PROC** and **DEF** declarations can be used in the program by using the **#include**-directive of the preprocessor.

4.3. Substitution

Named processes are not textually substituted. A procedure call is used instead. The semantics of occam substitution imply this by letting a global variable (i.e. not declared inside the named process' body) be found where the named process is defined and not where it is substituted.

4.4. ANY

According to the occam syntax the **ANY** keyword may be the only argument of an input or output process. Thus,

```
c ? ANY; x
```

is not allowed. Because it was easy to add, and it was used by some programs, our compiler allows it. (If portability is an issue, usage of this feature is not advisable).

4.5. Configuration

The special configuration keywords like **PLACED**, **ALLOCATE**, **PORT** and **LOAD** are not implemented. Only **PRI** works because **PAR** and **ALT** work the same without it.

5. Conclusions

Writing the compiler was very straightforward using the LLgen parser generator. Its extended grammar and its way of conflict resolving were of great use to us, especially the indentation handling could be implemented quite easily. The automatic error recovery given by LLgen took a great weight of our shoulders.

A set of parallelism simulation routines makes implementing **PAR** constructs very simple. And we consider it a necessity to have such a layer to shield the compiler writer from these details.

The translation to EM code was fairly direct, no great tricks were needed to make things work. Only the different sizes of words and pointers that are given as parameters to the compiler must be carefully watched. Variables or pointers must sometimes be handled with double word instructions for different word or pointer sizes.

6. Acknowledgement

We want to thank Dick Grune for his description of Occam which is used in the introduction.

7. References

[1] INMOS limited, *OCCAM Programming manual*, Prentice-Hall, 1984.

[2] C. J. H. Jacobs, *Some Topics in Parser Generation*, Informatica Rapport IR-105, Vrije Universiteit, Amsterdam, October 1985.

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[4] M. E. Lesk, *Lex - A Lexical Analyser Generator*, Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murrey Hill, New Jersey, October 1975.

[5] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, J. W. Stevenson, *Description of a Machine Architecture for use with Block Structured Languages*, Informatica Rapport IR-81, Vrije Universiteit, Amsterdam, 1983.

[6] K. Bot and E. Scheffer, *A set of multi-process primitives for stack based machines*, Vrije Universiteit, Amsterdam, 1986.

8. Appendix A: Implementation of the channel routines

```
/* $Id: ocm_chan.h,v 1.4 1994/06/24 10:08:30 ceriel Exp $ */
/*
 * (c) copyright 1987 by the Vrije Universiteit, Amsterdam, The Netherlands.
 * See the copyright notice in the ACK home directory, in the file "Copyright".
 */
/*      ocm_chan.h - channel definitions */
#include <stdio.h>
#include "ocm_parco.h"

typedef union channel {
        struct {                /* Interprocess channel */
                char _type;     /* Channel type, see note */
                char synch;     /* State in channel synchronization */
                long val;       /* Transmitted value */
        } c;
        struct {                /* File channel */
                char _type;     /* Dummy field, see note */
                char index;     /* Index in the file array */
                char flgs;      /* Status flags: in use & readahead */
                char preread;   /* Possible preread character */
        } f;
} chan;
#define type            c._type /* Channel type */
/* Note: The channel type should not be part of each structure in chan. But
 * the C alignment rules would make chan about 50% bigger if we had done it
 * the right way. Note that the order of fields in a struct cannot be a problem
 * as long as struct c is the largest within the union.
 */

#define C_T_CHAN        0       /* Type of a interprocess channel */
#define C_T_FILE        1       /* Type of a file channel */

#define C_S_FREE        0       /* IP channel is free */
#define C_S_ANY         1       /* IP channel contains data */
#define C_S_ACK         2       /* IP channel data is removed */

#define C_F_EOF         (-1L)   /* File channel returns EOF */
#define C_F_TEXT        (-2L)   /* File channel becomes line oriented */
#define C_F_RAW         (-3L)   /* File channel becomes character oriented */

#define C_F_INUSE       0x01    /* File channel is connected to a UNIX file */
#define C_F_READAHEAD   0x02    /* File channel has a preread character */

extern chan file[20];  /* Array of file channels */
extern FILE *unix_file[20];    /* Pointers to buffered UNIX files */

void c_init();

void chan_in(), cbyte_in(), c_wa_in(), c_ba_in();
void chan_out(), c_wa_out(), c_ba_out();

int chan_any();
```

```
/* $Id: channel.c,v 1.12 1994/06/24 12:28:06 ceriel Exp $ */
/*      channel.c - basic channel handling routines */
#include <errno.h>
#ifndef __BSD4_2
#include <signal.h>
#endif
#include <sgtty.h>
#include "ocm_chan.h"

static void disaster();

void c_init(c, z) register chan *c; register unsigned z;
/* Initialise an array of interprocess channels declared as: CHAN c[z]. */
{
        do {
                c->type=C_T_CHAN;
                (c++)->c.synch=C_S_FREE;
        } while (--z!=0);
}

void chan_in(v, c) long *v; register chan *c;
/* Reads a value from channel c and returns it through v. */
{
        switch(c->type) {
        case C_T_FILE:
                if ((c->f.flgs&C_F_READAHEAD)!=0) {
                        *v=(c->f.preread&0377);
                        c->f.flgs&= ~C_F_READAHEAD;
                } else {
                        register FILE *fp= unix_file[c->f.index];

                        *v= feof(fp) ? C_F_EOF : getc(fp);
                }
                break;
        case C_T_CHAN:
                deadlock=0;             /* Wait for value to arrive */
                while (c->c.synch!=C_S_ANY) resumenext();

                *v=c->c.val;
                c->c.synch=C_S_ACK;     /* Acknowledge receipt */
                break;
        default:
                disaster();
        }
}
```

```
void chan_out(v, c) long v; register chan *c;
/* Send value v through channel c. */
{
        switch(c->type) {
        case C_T_FILE: {
                register FILE *fp= unix_file[c->f.index];
                struct sgttyb tty;

                if ((v& ~0xff)==0)      /* Plain character */
                        putc( (int) v, fp);
                else
                if (v==C_F_TEXT) {
                        gtty(fileno(fp), &tty);
                        tty.sg_flags&= ~CBREAK;
                        tty.sg_flags|= ECHO|CRMOD;
                        stty(fileno(fp), &tty);
                } else
                if (v==C_F_RAW) {
                        gtty(fileno(fp),&tty);
                        tty.sg_flags|= CBREAK;
                        tty.sg_flags&= ~(ECHO|CRMOD);
                        stty(fileno(fp), &tty);
                }
        }       break;
        case C_T_CHAN:
                deadlock=0;             /* Wait until channel is free */
                while (c->c.synch!=C_S_FREE) resumenext();

                c->c.val=v;
                c->c.synch=C_S_ANY;     /* Channel has data */

                deadlock=0;             /* Wait for acknowledgement */
                while (c->c.synch!=C_S_ACK) resumenext();

                c->c.synch=C_S_FREE;    /* Back to normal */
                break;
        default:
                disaster();
        }
}
```

```
#ifndef __BSD4_2
static void timeout();
#endif

int chan_any(c) register chan *c;
{
#ifdef __BSD4_2
#include <fcntl.h>
        int flags;
#endif
        switch (c->type) {
        case C_T_FILE:
                if ((c->f.flgs&C_F_READAHEAD)!=0)
                        return 1;
                else {
                        register FILE *fp= unix_file[c->f.index];

                        if (feof(fp))
                                return 1;
                        else {
                                extern int errno;
                                register ch;

                                deadlock=0;
                                        /* No deadlock while waiting for key */

                                /* Unfortunately, the mechanism that was used
                                   here does not work on all Unix systems.
                                   On BSD 4.2 and newer, the "read" is
                                   automatically restarted. Therefore, on
                                   these systems, we try it with non-blocking
                                   reads
                                */
#ifdef __BSD4_2
                                flags = fcntl(fileno(fp), F_GETFL, 0);
                                fcntl(fileno(fp), F_SETFL, flags | O_NDELAY);
                                errno = 0;
                                ch = getc(fp);
                                fcntl(fileno(fp), F_SETFL, flags);
                                if (errno == EWOULDBLOCK) {
                                        clearerr(fp);
                                        return 0;
                                }
#else
                                signal(SIGALRM, timeout);
                                alarm(1);

                                errno=0;
                                ch=getc(fp);

                                signal(SIGALRM, SIG_IGN);
                                alarm(0);

                                if (errno==EINTR) {
                                        clearerr(fp);
                                        return 0;
                                }
#endif
                                else {
                                        if (!feof(fp)) {
                                                c->f.flgs|=C_F_READAHEAD;
                                                c->f.preread=ch;
```

```
                                        }
                                        return 1;
                                }
                        }
                }
        case C_T_CHAN:
                return c->c.synch==C_S_ANY;
        default:
                disaster();
        }
}

#ifndef __BSD4_2
/* The ch=getc(fp) in the above function calls read(2) to do its task, but if
 * there's no input on the file (pipe or terminal) then the read will block.
 * To stop this read from blocking, we use the fact that if the read is
 * interrupted by a signal that is caught by the program, then the read returns
 * error EINTR after the signal is processed. Thus we use a one second alarm
 * to interrupt the read with a trap to timeout(). But since the alarm signal
 * may occur *before* the read is called, it is continuously restarted in
 * timeout() to prevent it from getting lost.
 */

static void timeout(sig)
{
        signal(SIGALRM, timeout);
        alarm(1);
}
#endif

static void disaster()
{
        write(2, "Fatal error: Channel variable corrupted\n", 40);
        abort();
}
```

9. Appendix B: Translation of a PAR construct to EM code using the library routines to simulate parallelism

Translation of the parallel construct:

```
par
    P0
    par i = [ 1 for n ]
        P(i)
```

is

```
 lal -20          ; Assume 20 bytes of local variables at this moment
 cal $parbegin    ; Set up a process group
 asp 4
 cal $parfork     ; Split stack in two from local -20
 lfr 4
 zne *23          ; One end jumps to second process, other continues here
 lor 0            ; Static link
 cal $P0
 asp 4
 bra *24          ; Jump to the outer parend
23
 cal $parfork     ; Fork off 'par i = ...'  process
 lfr 4
 zne *25          ; One end jumps to end of outer par
 lal -20          ; Place break just above i
 cal $parbegin    ; Set up another process group for the P(i)
 loc 1
 stl -24          ; i:=1
 lol n            ; Assume n can be addressed this simply
 stl -28          ; A nameless counter
 bra *26          ; Branch to counter test
27
 cal $parfork     ; Fork off one P(i)
 lfr 4
 zne *28          ; One jumps away to increment i, the other calls P(i)
 lol -24
 lor 0
 cal $P
 asp 8
 bra *29
28
 inl -24          ; i:=i+1
 del -28          ; counter:=counter-1
26
 lol -28
 zgt *27          ; while counter>0 repeat loop
29
 cal $parend      ; Wait for the P(i) to finish, then delete group
 bra *24          ; Jump to the higher up meeting place with P0
25                ; Note that the bra will be optimized away
24
 cal $parend      ; Wait for both processes to end, then delete group
```