A Lisp-based OCCAM Interpreter

MIKE STARKEY

UUCS-91-002

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

March 6, 1991

Abstract

The OCCAM *programming language is an implementation of Communicating Sequential Processes and is used in a number of different areas. These areas usually require explicitly describing small-grain paralleslism.* OCCAM *programs formed by such descriptions can be tested for correctness by executing them on commercially availabe transputers. Unfortunately, this environment requires that all components be written in* OCCUM *instead of being able to describe parts of the program with behavioural models written in a more powerful language. The interpreter described here solves this problem. It allows programs to be written in* OCCAM *with behavioural descriptions in Lisp. A number of large programs which take advantage of this powerful environment have been implemented and tested.*

University of Utah, Department of Computer Science
Technical Report UUCS-91-002

A Lisp-based OCCAM Interpreter

MIKE STARKEY (*starkey@cs.utah.edu*)
University of Utah
Dept. of Computer Science
3190 Merrill Engineering Building
Salt Lake City, Utah 84112

Abstract. The OCCAM programming language is an implementation of Communicating Sequential Processes and is used in a number of different areas. These areas usually require explicitly describing small-grain parallelism. OCCAM programs formed by such descriptions can be tested for correctness by executing them on commercially available transputers. Unfortunately, this environment requires that all components be written in OCCAM instead of being able to describe parts of the program with behavioural models written in a more powerful language. The interpreter described here solves this problem. It allows programs to be written in OCCAM with behavioural descriptions in Lisp. A number of large programs which take advantage of this powerful environment have been implemented and tested.

1 Introduction

> Entia non sunt multiplicanda praeter necessitatem.
> *William of Occam (Occam's razor)*

The OCCAM programming language conforms well to Occam's Razor since it is not excessively complicated. This language, based on Hoare's Communicating Sequential Processes[3] is very simple. The original language definition contains only 22 reserved words and allows very low-level descriptions of explicit parallelism.

Apart from being a powerful programming language, OCCAM is also a design formalism[6]. As a programming language, OCCAM has been used for applications in areas such as computer graphics[2]. As a design formalism, OCCAM has recently been used as an asynchronous circuit description language[1]. Each of the OCCAM primitives can be translated into a small self-timed asynchronous circuit. Combining these small circuits by way of an OCCAM program creates an entire circuit through translation. As long as each of the components has been designed to perform in an electrically correct manner, and the combination of these components maintains correctness, the resulting circuit should at least be electrically correct.

```
a) PROC buffer (CHAN in<1> out<1>)
     VAR temp<1>:
     SEQ
       temp := 0
       WHILE TRUE
         in ? temp
         out ! temp
```

```
b) (proc buffer ((chan in<1> out<1>))
       (seq ((var temp<1>))
           (set temp 0)
           (while true
               (? in temp)
               (! out temp))))
```

Figure 1: An example of OCCAM written in the traditional style (a) and the Lisp-like format (b). Note that using parentheses does not preclude also using indentation.

Although programs written in OCCAM can be proven by construction to be mathematically or electrically correct, this construction does not assure that the program executes the desired function. The programmer must verify that it does. Verifying functionality provides motivation for being able to run these programs interpretively to watch or test their behaviour. The commercially available transputers[4] which were designed around the OCCAM language can be used to test any such programs which are written entirely in OCCAM. Here the power of the interpreter becomes apparent. Programs can be written which use the OCCAM constructs for describing parallelism. Instead of each process consisting of only one primitive, a process can be a much larger entity. It is assumed that the behaviour of this larger process is known as far as its interface to the OCCAM program, and that this behaviour can be described. The language used for describing such processes in the interpreter is Lisp.

2 The OCCAM Programming Language

OCCAM programs consist of a number of single-primitive processes which are connected by constructs that describe their execution method. These execution methods include being executed in parallel, sequentially or dependent on certain conditions or guards. In the specification of OCCAM, the set of processes which may be executed within these constructs is delimited by the number of indenting characters before the primitive. This method of process delineation is good for small sets of processes, since the hierarchy can easily be seen. However as sets of processes become large characters are either lost off the right hand edge of the screen or paper, or the processes become so long that the structure is difficult to view.

Due to the problems caused by many indentations, a Lisp-like syntax borrowed from [1] is used to create programs. Lisp statements are built by sur-

rounding a function name followed by its arguments in parentheses. Programs are constructed by hierarchically imbedding parenthesized expressions within others. The program structure is easily viewed by matching sets of parentheses (Figure 1).

An additional benefit to using the Lisp-like format is for parsing the program. Lisp includes some powerful features for processing and parsing lists. A list in Lisp is a set of lists or atoms (data values) encapsulated within parentheses. Using this approach, an OCCAM program becomes a complex list which can easily be manipulated in Lisp.

```
Program      =  Process
Process      =  Primitive | Construct
Primitive    =  Assignment | Input | Output | (SKIP)
Assignment   =  (SET Variable Expression )
                (SETFUNC Funcvar Func-call )
Input        =  (? Channel [ Variable ] )
Output       =  (! Channel [ Expression ] )
Construct    =  Seq-Construct
                | (PAR Decl {Process} )
                | (ALT {Guarded-Proc} )
                | (IF {Cond-Proc} )
                | (WHILE Cond Process )
Seq-Construct =  {Process} | (SEQ Decl {Process} )
Guarded-Proc =  ( Guard Process)
Guard        =  ( Cond Input ) | ( Cond (SKIP))
Cond-Proc    =  ( Cond Process)
Decl         =  ( {Decl-Part} )
Decl-Part    =  (CHAN {chan-name [ < int > ]} )
                | (VAR {var-name [ < int > ] } )
                | (FUNCVAR { ( funcvar-name function-name ) } )
Variable     =  var-name | (SUBSEQ var-name int int )
Channel      =  chan-name | (SUBSEQ chan-name int int )
Funcvar      =  funcvar-name
Expression   =  Variable | Funcvar
                | ( gate-name { Expression })
Func-call    =  ( function-name { Expression } )
Cond         =  boolean-expression
Macro        =  (PROC macro-name Decl {Process} )
```

Figure 2: Lisp-like OCCAM Syntax[1].

The general OCCAM commands which are supported are not specific to any application. However, since this system was designed primarily for describing asynchronous circuits, a number of application specific commands have been

```
a) (defun addition-function (inputs)
      (+ (car inputs) (cadr inputs)))

b) (describe-function ''+'' 'adder-function)

c) (add-function ''+'' '(16 16 16))

d) (funcvar sum<16> adder)

e) (setfunc sum (+ 45 32))
```

Figure 3: Steps required to define (a), register (b,c) and use (d,e) a behavioural model of a process in the interpreter.

added. The flexibility of these additional commands allows them to be easily used for other applications. The commands described in Figure 2 provide support for behavioural descriptions of processes, but do not demonstrate how to associate these descriptions with the program.

2.1 Adding Behavioural Descriptions of Processes

To associate behavioural descriptions with the OCCAM program, a number of steps are required. First, a Lisp function must be written to perform the appropriate behaviour (Figure 3a). The interface can have any number of inputs and one output. The value of the output will be placed in the desired variable on return from the function. If the function must maintain some state between invocations, a closure can be used. The Lisp function is then registered with the system (Figure 3b). Next, the widths of the inputs and output are specified to permit type checking (Figure 3c). The function can now be used in the system wherever the programmer desires. To use the function, a variable is defined which is restricted to accept only the value from that function (Figure 3d). The assignment can then be performed by evaluating the function at that time and placing the value in the variable (Figure 3e).

Simple functions can also be defined (Figure 4). In asynchronous circuits, these functions represent gates. The differences between these definitions are required to direct the translation of these programs to circuits. In the translation, functions and gates have different implementations. For other applications, gates are sufficient. A library of useful gates which may have different definitions required for certain applications can be maintained. Many of the useful gates are included in the interpreter base, but these definitions can be overridden.

```
a) (defun and-function (inputs)
      (logand (car inputs) (cadr inputs)))

b) (describe-gate ''and'' 'and-function)

c) (add-gate ''and'' '(1 1 1))

d) (if (and i 1) (set i 0))
```

Figure 4: Steps required to define (a), register (b,c) and use (d) a simple behavioural model in the interpreter.

2.2 Extracting Parts of Variables

The variable type used in this version of OCCAM is a string of bits of any length. To assist in decoding individual bits or groups of bits from a variable, an instruction is included which defines a variable as containing a sub-sequence of bits from another variable. The syntax of this function is (DEFINE *sub-var-name* (SUBSEQ *var-name int int*)) This definition of the *sub-var-name* variable is global. Wherever it is used, the *var-name* variable must be defined.

2.3 Providing User Interaction

While debugging or running a program, it is sometimes useful to display certain variables or to direct the program with different user inputs. Printing messages or prompting the user for data can be done in three ways. One method is to define a behavioural model of a process in Lisp which has the side effect of printing or obtaining data. This method is very flexible, since the data can be formatted by the associated Lisp code as desired. The data can also be printed or read from a file.

Another method of transferring data to and from the user is to use predefined channels and gates. Two channels have been predefined to allow data transfer. Sending data to the OUTPUT channel causes data to be displayed on the screen. The channel definition is modified slightly by allowing not just bit strings of data but also character strings for debugging purposes. Receiving on the INPUT channel will prompt the user for data. The command syntax is the same as with any other channel command. These two channels are global and can be redefined by the user as desired.

The third method is to use a pair of special gates. These gates help to monitor and affect expression evaluation. The GET gate gets data from the user and effectively replaces that instance of the GET in the expression with

the entered value. The PUT gate displays the value of an expression. It is transparent to the expression evaluation and therefore does not affect the program execution.

3 Interpreting An OCCAM Program

A simple example of an OCCAM program is included in Figure 5. This program implements a two stage FIFO buffer. This example demonstrates a number of OCCAM constructs and primitives and will be used in subsequent sections to demonstrate how the interpreter is used and the types of information it can display.

```
(proc buffer ((chan in<1> out<1>))
    (seq ((var temp<1>))
        (set temp 0)
        (while true
            (? in temp)
            (! out temp))))

(proc environment ((chan env-in<1> env-out<1>))
    (par ((var temp<1>))
        (! env-out 1)
        (! env-out 2)
        (! env-out 3)
        (? env-in temp)
        (? env-in temp)
        (? env-in temp)))

(par ((chan buffer-in<1> mid<1> buffer-out<1>))
    (buffer buffer-in mid)
    (buffer mid buffer-out)
    (environment buffer-out buffer-in))
```

Figure 5: Simple example of an OCCAM program.

3.1 User Interface

The user interface of the simulator is menu driven. The menu includes many commands which allow files to be loaded, processes or channels to be traced, program execution to be graphically displayed, and programs to be terminated. A help function and a command to automatically generate documentation have also been included.

```
Commands
--------

Only enough characters to identify the command are required.

?              - Prints help summary
base-change    - Change the base of displayed numbers
channel-trace  - Set channel trace on
execute-limit  - Set execution limit for simulation
documentation  - Produce the documentation in LaTex form
function       - Describe a function definition
gate           - Describe a gate definition
help           - Print help summary
introduction   - Print the introduction to the simulator
kill-program   - Kill the current program
load-program   - Load a new program
no-trace       - Turn off all traces
occam          - Display OCCAM syntax
process-trace  - Set process trace on
quit           - Quit the simulator
restart        - Restart program
simulate       - Perform simulation
view-program   - View the program structure
xdisplay       - Start X display

During trace and view operations, the values of variables will be
printed between brace brackets '{}' in the selected numerical base
whenever they are known. Widths will be printed as specified, between
'<>'. For example, a set process may be displayed as:
   SET x<4> (and a{1} b{0}){0}
```

Figure 6: Menu provided by the interpreter.

3.2 Tracing Program Execution

Tracing program execution is a very important feature in the interpreter. The user can set the granularity of the traces to print channel information only (Figure 7) or process and channel information (Figure 8). Alternatively, the trace can be disabled.

```
Sending 1 on channel BUFFER-IN
Received 1 on channel BUFFER-IN
Sending 2 on channel BUFFER-IN
Sending 1 on channel MID
Received 1 on channel MID
Sending 1 on channel BUFFER-OUT
Received 1 on channel BUFFER-OUT
Received 2 on channel BUFFER-IN
Sending 3 on channel BUFFER-IN
Sending 2 on channel MID
Received 2 on channel MID
Sending 2 on channel BUFFER-OUT
Received 2 on channel BUFFER-OUT
Received 3 on channel BUFFER-IN
Sending 3 on channel MID
Received 3 on channel MID
Sending 3 on channel BUFFER-OUT
Received 3 on channel BUFFER-OUT
Deadlock reached
```

Figure 7: Channel trace of the example in Figure 5. Note that macro expansions have replaced the local names with the global names.

3.3 Graphical Display of Program Execution

The interpreter can be used to analyze the execution of a program. In some applications, knowing the critical processes can help to optimize certain parts of the program. A critical process is one which is very active. This activity is easily viewed by means of a graphical display. As each process becomes active, it is highlighted. Very active processes can be easily detected by monitoring the frequency at which they are highlighted. The example in Figure 9 shows highlighted processes at one instance of program execution; it also provides a display of the program structure.

```
Executing Process: SEQ
Executing Process: SET TEMP<1> 0
Executing Process: SEQ
Executing Process: SET TEMP<1> 0
Executing Process: PAR
Executing Process: ! BUFFER-IN<1> 1
Sending 1 on channel BUFFER-IN
Executing Process: ! BUFFER-IN<1> 2
Executing Process: ! BUFFER-IN<1> 3
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executing Process: SEQ
Executing Process: WHILE TRUE{1}
Executing Process: ? BUFFER-IN<1> TEMP<1>
Received 1 on channel BUFFER-IN
Executing Process: SEQ
Executing Process: WHILE TRUE{1}
Executing Process: ? MID<1> TEMP<1>
Executing Process: PAR
Executing Process: ! BUFFER-IN<1> 1
Executing Process: ! BUFFER-IN<1> 2
Sending 2 on channel BUFFER-IN
Executing Process: ! BUFFER-IN<1> 3
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executing Process: SEQ
Executing Process: WHILE TRUE{1}
Executing Process: ! MID<1> TEMP{1}
Sending 1 on channel MID
Executing Process: SEQ
Executing Process: WHILE TRUE{1}
Executing Process: ? MID<1> TEMP<1>
Received 1 on channel MID
Executing Process: PAR
Executing Process: ! BUFFER-IN<1> 2
Executing Process: ! BUFFER-IN<1> 3
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executing Process: ? BUFFER-OUT<1> TEMP<1>
Executed 5 primitives
```

Figure 8: Process trace of the example in Figure 5.

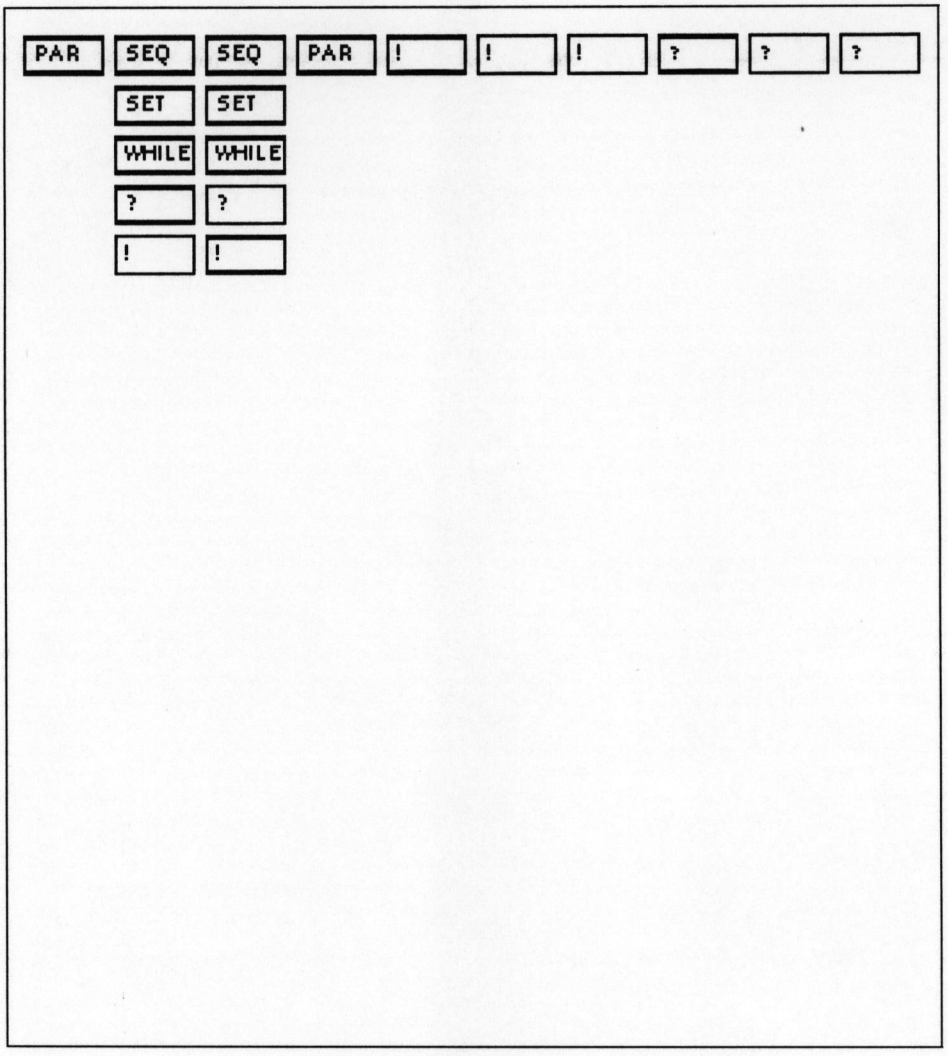

Figure 9: Graphical display of the example in Figure 5. Active processes are highlighted as they execute.

4 Modeling Parallelism

The interpreter is written in Common Lisp[5] and executes on a uniprocessor machine by simulating the parallelism defined in OCCAM. To simulate this parallelism, the program is first structured into a tree where the depth of the tree is controlled by sequential processes and the breadth of the tree depends on the number of processes executed in parallel. Figure 9 demonstrates the structure of the example program. Once the program is in a tree structure, it can be traversed. This traversal is performed in a breadth first manner to ensure that each parallel process is evaluated fairly. To speed up this execution, queues of active child processes are maintained and processes are removed from their parent's queue as they complete.

Sends and receives are also handled in this way. In OCCAM, channel communication blocks until both partners in the transfer are ready. If a send is reached, it will mark the channel as being ready to send. The program execution will continue in a round-robin fashion until the corresponding receive is reached. At this point the data will be transferred and the channel marked as data received. When the sending process is reached again, it will complete. The mechanism is similar if a channel is blocked waiting on a receive.

A side effect of this model is that deadlock can easily be detected. If in one entire loop through all active processes, no process is executed then deadlock has occurred. Deadlock in this case means that there must be a pending send or receive which is blocked and no corresponding receive or send will be executed to unblock it. Detecting livelock in a program can still be difficult.

5 Results

This interpreter has been used to debug and test some large programs. The largest program developed in this environment is an asynchronous Reduced Instruction Set Computer (RISC)[7]. This program is approximately 500 lines of OCCAM. The interpreter simulated the RISC by allowing test programs to be "executed" on the RISC. Another large program implemented, run and tested in the interpreter environment is an asynchronous cache controller[8].

6 Conclusions

This interpreter is important for implementing, testing and debugging OCCAM programs. These three components of program development are greatly simplified by using an interpreter instead of a compiler and transputers. The interpreter's user interface provides a useful set of commands to aid in the development process.

The programs which are developed can be combinations of OCCAM and Lisp. OCCAM can be used to describe the control flow of a program while Lisp provides an easy way to describe the behaviour of processes. Through the interpreter, these methods can be combined to build very powerful programs.

References

1. BRUNVAND, E., AND SPROULL, R. F. Translating concurrent communicating programs into delay-insensitive circuits. Tech. Rep. CMU-CS-89-126, Carnegie Mellon University, 1989.

2. FAY, D. Working with occam: a program for generating display images. *Microprocessors and Microsystems 8* (1984), 3–15.

3. HOARE, C. A. R. *Communicating Sequential Processes.* Prentice Hall, 1985.

4. INMOS. *Communicating Process Architecture.* Prentice Hall, 1988.

5. KESSLER, R. R. *LISP, Objects, and Symbolic Programming.* Scott, Foresman and Company, 1988.

6. MAY, D., AND TAYLOR, R. Occam - an overview. *Microprocessors and Microsystems 8*, 2 (1984), 73–79.

7. STARKEY, M., AND FARHANG, A. R. C-risc ii: An asynchronous reduced instruction set computer. CS 572 Project Report, 1990.

8. YIH, B. An asynchronous cache controller. CS 572 Project Report, 1990.