Communicating Process Architectures 2000
P. H. Welch and A. W. P. Bakkers (Eds.)
IOS Press, 2000

Post-Mortem Debugging in KRoC

David C. WOOD and Frederick R. M. BARNES
Computing Laboratory, University of Kent at Canterbury, CT2 7NF
{D.C.Wood@ukc.ac.uk, frmb2@ukc.ac.uk}

Abstract. A simple post-mortem debugging facility has been added to KRoC [1], to identify and locate run-time errors, including deadlock. It has been implemented for the SPARC/Solaris and i386/Linux versions of KRoC.

1. Introduction

A limitation of KRoC has always been its lack of debugging facilities. Some users, especially students, are not convinced by the argument that a properly constructed program does not contain bugs.

Interactive debuggers are complex to write and hard to use, particularly in a system with concurrent processes. For these reasons, a simple post-mortem debugger has now been provided. When KRoC is used in debugging mode, it automatically prints debugging information when an error is detected.

Separate versions of this debugger have been written for the SPARC and the i386. The architectures and translation mechanisms used for KRoC on these two targets are sufficiently different for it to be worth describing them separately [2].

2. SPARC version

The main ideas for post-mortem debugging were developed first on the original SPARC version of KRoC.

2.1. *Debugging directives*

On the SPARC, compilers for C, Fortran, etc. insert debugging directives (`stabs` and `stabn`) into their assembly-language output [3]. These cause extra information to be added to the symbol table of the object code, which in turn can be interpreted by debugging programs like `adb`, `gdb`, and `dbx`.

An experimental version of `octran` has been use to generate similar directives, but it appears that the structure of occam* programs is too unfamiliar for the standard debuggers to handle, and this approach has been abandoned. In any case, one very important error, deadlock, could hardly be handled by standard debuggers.

2.2. *Special code*

KRoC has always, by default, generated code to check for run-time errors, like arithmetic overflow and array-bound violations, and options are provided to control what, if anything, happens when such an error occurs. Some of these checks are simply translations of the

* occam is a trademark of the SGS-THOMSON Microelectronics Group.

transputer code from the compiler; others, like integer overflow on most machines, are added by `octran`.

This mechanism has been expanded to provide more information.

2.2.1. *Run-time errors*

In order to identify errors detected during translation (which, of course, never occur!), `octran` has always kept track of the file name and line number of the occam source, as indicated in comments generated by the compiler, and also the line numbers of the transputer and target assembly-language files. The debugging option causes this information to be recorded as a data structure in the object code, and when an error occurs a post-mortem routine is called to print it.

In most cases the cost of this extra debugging code, in speed and size of the object program, is small.

A few kinds of errors require special treatment.

2.2.2. *Floating-point errors*

At appropriate points in the code, the compiler generates transputer `fpchkerr` instructions to check if a floating-point error has occurred. By default, `octran` ignores these, because the SPARC hardware is set to trap such errors when they actually occur.

However, for some checking modes, these checks were already implemented. So for debugging purposes, hardware trapping is turned off, and the check instructions are generated, with the extra information required, exactly as for other run-time errors. This involves a significant run-time overhead, because the floating-point state register has to be checked explicitly, which is slow on the SPARC [4].

2.2.3. *Division by zero*

Like many machines, the SPARC treats integer division by zero as a floating-point error, and traps it. In debugging mode, `octran` checks if the divisor is zero before all integer division operations.

2.2.4. *Deadlock*

One of the most frustrating errors in an occam program is deadlock, because it is a property of a whole network of processes, rather than being localized at a single point.

Deadlock occurs when the scheduler finds no processes on the run queue or waiting for an external event (keyboard input or a timeout, in KRoC). In standard occam, any remaining processes must be waiting on a channel. Hence, the processes involved in the deadlock must have attempted to start a communication that has never finished.

To identify such processes, `octran` generates extra code for each communication over a channel, setting a flag before the communication and clearing it afterwards.

The workspace of each occam process has a header, of between two and five words, used to contain control information. One of these words is the link, `pw.Link`, used when the process is on the run queue. Since any communicating process may be put on this queue, this location must be available, so it is used to contain the flag.

The post-mortem routine searches the workspace for such flags, and so identifies all communications currently blocked.

2.3. *Output*

For any run-time error detected, the the following information is printed:

- The type of error (overflow, range, floating-point, etc.), insofar as it can be determined; or, for deadlock, the instruction (`in`, `out`, `outbyte`, `outword`, `altwt`, or `taltwt`*) concerned.
- The name of the `PROC` or `FUNCTION` in which it occurred.
- The file name and approximate line number of the occam source file.
- The file names and line numbers of the transputer and target assembly-language files.
- The contents of the virtual transputer registers. Only the active elements of the evaluation stacks are given.
- A few words of the workspace around the workspace pointer, and of the target code around the instruction pointer.
- The process and timer queues.

Most of this information is given in hexadecimal, except for the floating-point stack registers, which are printed in an appropriate format.

In `ALT` constructs, the relationship between occam source and assembly language is complicated; for deadlock in an `ALT` the line number reported is near the end of the construct.

2.4. *Main `PROC` header*

The main `PROC` of a program in KRoC must have the form

```
PROC proc.name (CHAN OF BYTE keyboard, screen, error)
```

where the channels correspond to the standard UNIX† input and output streams. Previously this was not checked, giving rise to the possibility of mysterious run-time errors that could not be located by the post-mortem mechanism. This has now been corrected.

2.5. *Implementation*

The debugging option requires extra code to be generated when checking for run-time errors and in communications, and additional routines in the run-time system to interpret the information this provides.

2.5.1. *Extra code generated*

Normally, run-time errors are reported through conditional trap instructions; for example:

```
cmp     %l0,%i4          !- error if negative
tgeu    18               !+ RANGE
```

where the operand of the trap instruction indicates the type of error (16 greater than the error code).

---

* A timer `ALT` can deadlock, if the timer guard has a precondition that evaluates to `FALSE`.
† UNIX is a trademark of AT&T Bell Laboratories in the USA and other countries.

For debugging, this is replaced by a conditional branch over some extra code, and a debugging record is assembled in the `.data` segment:

```
        cmp     %l0,%i4         !- error if negative
        blu     9f
        sethi   %hi(0f),%o0     !*
        call    $$debug
        or      %o0,%lo(0f),%o0 !*
        .data                   !* debugging record
        .align  4
0:      .word   0x12002000,17,31,58,LP0,LF0,LS0,LD0
LP0:    .asciz  "out_repeat"
LF0:    .asciz  "small_utils.lib"
LS0:    .asciz  "nos.kt8"
LD0:    .asciz  "nos.s"
        .text
9:
```

The hexadecimal digits of the first word of the debugging record encode, from left to right:

- The type of the routine (in this case `PROC`);
- The depth of the integer evaluation stack;
- The depth of the floating-point evaluation stack;
- The types (`REAL32` or `REAL64`) of the entries in the floating-point stack;
- The type of error (in this case `RANGE`).

Then follow the line numbers in the occam, transputer assembly-language (`nos.kt8`), and SPARC assembly-language (`nos.s`) files, and pointers to the names of the routine and files concerned.

Since this is the first such record, all the strings are required.

In the normal case when the branch is taken, the `sethi` in its delay slot is also executed. This is unavoidable, but harmless (as it happens, `nop` is actually a special case of `sethi`).

Communications are implemented as kernel calls; for example, without debugging:

```
        call    $$out
        inc     8,%o7           !* adjust pc
```

where the return address in `%o7` is incremented in the delay slot to skip over the call and the increment instruction itself.

For debugging, some identifiable value must be placed in the workspace while the communication is pending, and removed when it has completed. Using the workspace pointer itself, in the `pw.Link` word of the process header, is quick and simple:

```
        st      %l3,[%l3-8]     !+ set flag
        call    $$out
        inc     12,%o7          !* skip magic
```

```
        .word   0f
        .data                   !* debugging record
        .align  4
0:      .word   0x10300001,18,40,81,LP0,LF0,LS0,LD0
        .text
        clr     [%l3-8]         !+ clear flag
```

Here the return address is incremented to skip over the pointer to the debugging record, at no extra cost.

The code word contains:

- The type of the routine, as above;
- Two digits for the instruction (in this case `out`);
- The remaining digits for the byte count, if known (this is not currently used).

In this case all the strings are the same as before, so they are not repeated.

The cost in time is two instructions, setting and clearing the flag in the workspace, for each communication. (Actually, clearing this location is unnecessary, as the current scheduler always changes it, but other versions might not.)

When `octran` encounters the names of new routines or files, it updates the debugging records:

```
        deccc   %l1
        bvc     9f

        ...

0:      .word   0x12001000,27,62,121,LP1,LF0,LS0,LD0
LP1:    .asciz  "out_ch"
```

It would have been possible to place the debugging records in the `.text` segment, but then even more of the code displayed in the post-mortem dump would have been concerned with debugging, rather than with the user's program.

2.5.2. *Post-mortem routines*

As shown above, when a run-time error occurs, an assembly-language routine, `$$debug`, is called. This in turn calls, in sequence, a number of C functions to print information about the error. (This is simpler than calling a single routine, since passing a few parameters at a time in registers avoids the complexity of the SPARC method of passing them in memory.)

When deadlock is detected, the kernel calls the post-mortem routine `deadlock`. If debugging is in force, this routine searches the workspace for occurrences of `Wptr[pw.Link] = Wptr`. Of course, since pointers to local variables are common, some of these may not represent genuine debugging 'magic', so further checks are made to reduce the probability of spurious 'hits'. For each occurrence, the routine inspects `Wptr[pw.Iptr]`, which should contain the return address from the communication routine concerned, checking that it is a properly aligned pointer into the code area, and that it points back to the correct sequence of instructions.

2.6. *Usage*

Post-mortem debugging is enabled by a flag to the `kroc` command.

```
kroc -D prog.occ
```

gives the full debugging report; `-d` instead of `-D` omits the dump of registers and memory, which is unlikely to be useful to ordinary users.

2.6.1. *Separate compilation*

If a program is made up from separately compiled components, error reports will be given only for those parts compiled with the debugging flag. Whether the long or short form is given depends on the final compilation of the main program.

2.7. *Examples*

The following is a typical error report, actually caused by an uninitialized variable.

```
KRoC:   Run-time error

        RANGE ERROR in PROC "fair_alt_phil";
        in occam file "q7.occ" near line 170
        ("q7.kt8" line 938; "q7.s" line 1638)

        Areg = #00000005, Breg = #00000000
        Wptr = #EFFFF5C8, Iptr = #000033B0

        Workspace  Code

                -5 #00003724  #E004E008
                -4 #EFFFF5E8  #E204E030
                -3 #80000003  #80A40011
                -2 #00003704  #0A800004
                -1 #00000000  #11000039
                 0 #0000374C  #40000B1C
                 1 #00000000  #9012214C
                 2 #00000005  #E204E02C
                 3 #00000001  #A12C2002
                 4 #00000000  #A0040011
                 5 #00000005  #E0040000

        Fptr = #EFFFF65C -> #00000008
        Bptr = #EFFFF65C
        Tptr = #EFFFF9AC -> #EFFFF924 -> #80000000
```

The compiler does not provide numbers for every line; hence the word ‘near’, though the line numbers are normally exact.

Many occam programs contain processes with `WHILE TRUE` loops. In this case, deadlock is the normal form of termination. Here is a classic example, with the short form of report:

```
KRoC:   Deadlock

        Instruction "out" in PROC "id";
        in occam file "cycles.lib" near line 9
        ("nos.kt8" line 320; "nos.s" line 662)

        Instruction "in" in PROC "succ";
        in occam file "cycles.lib" near line 18
        ("nos.kt8" line 336; "nos.s" line 691)

        Instruction "out" in PROC "delta";
        in occam file "cycles.lib" near line 43
        ("nos.kt8" line 386; "nos.s" line 799)
```

3. Linux version

The implementation of post-mortem debugging in the i386/Linux port of KRoC differs significantly from the SPARC/Solaris version, largely because of the different translation mechanism used.

3.1. *Errors detected*

Debugging code is placed around the following operations:

- Blocking kernel calls (inputs, outputs, and `ALT`s), to report process states if deadlock occurs
- Arithmetic operations, to check for overflow
- Integer division, to check for division by zero
- Range checks
- Application-level errors (`seterr`)

The translator used in KRoC/Linux, `tranpc`, converts from extended transputer code (ETC) directly into native i386 ELF object files [5]. This means that the debugging code has to be generated as in-line machine code, complicating things somewhat.

As for the SPARC, the mechanism for post-mortem debugging is split into two parts: the generation of additional debugging code during translation, and extra code in the occam kernel to make use of it.

3.2. *Translation*

During translation, `tranpc` maintains a note of the current procedure names, obtained from their entry points, and file names and line numbers, from ETC specials in the instruction stream. The line number is just an integer, while procedure and file names must be kept in arrays indexed by integers.

As the translation proceeds, `tranpc` looks for points where debugging information is required, and inserts the relevant code. This involves five different generation sequences for each of the five debugging points mentioned above.

3.2.1. *Deadlock*

The first, and perhaps the most complicated, is the deadlock debugging information. The debugging code is placed immediately before and after kernel calls which could result in deadlock (input, output, and ALTs). As with the SPARC, just before the kernel call, the Wptr of the current process is placed in its link field: `Wptr[Link] := Wptr`. The following code is placed immediately after the call:

```
# return from call here
jump to Lxx
# debug record
    [LINE_LOW, LINE_HIGH, KENTRY, DLOP]
    [PROC_LOW, PROC_HIGH, FILE_LOW, FILE_HIGH]
    [#DE, #AD, #BE, #EF]
    jump to procfile.setup.label
Lxx:
# normal execution continues here
```

The two bytes LINE_HIGH:LINE_LOW together form the 16-bit current line number, PROC_HIGH:PROC_LOW similarly form an index into the procedure-name array, and FILE_HIGH:FILE_LOW form an index into the file-name array. These six bytes are common to all debugging records, as they specify the current point in the source file.

KENTRY is the kernel entry-point which was called, and DLOP is the operation which KENTRY handles. This will be one of the following constants:

DLOP_INVALID	invalid debug record
DLOP_IN	process was blocked on an input
DLOP_OUT	process was blocked on an output
DLOP_OUTBYTE	process was blocked outputting a byte
DLOP_OUTWORD	process was blocked outputting a word
DLOP_ALTWT	process was blocked on an ALT
DLOP_TALTWT	process was blocked on a timer ALT

The target for `procfile.setup.label` is given in § 3.3.

3.2.2. *Arithmetic overflows*

After each arithmetic operation which might overflow (addition, subtraction, division, multiplication, and remainder), the following code is placed:

```
# arithmetic operation performed here
jump to Lxx if overflow flag not set
# debug record
    set EDX register to [LINE_LOW, LINE_HIGH, #00, OPCODE]
    set ECX register to [PROC_LOW, PROC_HIGH, FILE_LOW, FILE_HIGH]
    jump to overflow.label
Lxx:
# normal execution continues here
```

OPCODE specifies the operation which overflowed, and will be one of the following constants:

OOP_INVALID	invalid debug record
OOP_ADD	addition overflow
OOP_SUB	subtraction overflow
OOP_MUL	multiplication overflow
OOP_DIV	division overflow
OOP_REM	modulus overflow
OOP_LADD	long (INT64) addition overflow
OOP_LSUB	long (INT64) subtraction overflow
OOP_ADC	add-constant overflow (transputer ADC instruction)

The target for overflow.label is given in § 3.3.

3.2.3. *Division by zero*

In addition to the overflow debugging code placed after division operations, a check is made before the division to ensure that the divisor is not zero. This is implemented by the following code:

```
compare divisor with zero, and jump to Lxx if zero flag not set
# debug record
    set EDX register to [LINE_LOW, LINE_HIGH, #00, #00]
    set ECX register to [PROC_LOW, PROC_HIGH, FILE_LOW, FILE_HIGH]
    jump to zerodiv.label
Lxx:
# division performed here
```

The target for zerodiv.label is given in § 3.3.

3.2.4. *Range errors*

Range errors occur when a run-time check (shift left or right, CSNGL, CSUB0, and CCNT1) fails. The most common cause of errors of this type are array index out-of-bounds errors. At the point where the check fails, the following code is placed:

```
push [LINE_LOW, LINE_HIGH, #FF, OPCODE] on kernel stack
push [PROC_LOW, PROC_HIGH, FILE_LOW, FILE_HIGH]
jump to range.entry.label
```

OPCODE specifies the operation which caused the range error. This will be REOP_SHIFT for shift errors, or one of REOP_CSNGL, REOP_CSUB0, or REOP_CSUB1 for the corresponding transputer operations. The target for range.entry.label is given in § 3.3.

3.2.5. *Application-level errors*

Application-level errors are generated by such things as the STOP process, the compiler library function CAUSEERROR, and ASSERT statements that evaluate to FALSE, all of which correspond to the transputer seterr instruction.

At the point where the error occurs, the following code is generated:

```
push [LINE_LOW, LINE_HIGH, #00, #FB] on kernel stack
push [PROC_LOW, PROC_HIGH, FILE_LOW, FILE_HIGH]
push address of filename.label
push address of procedure.label
call K.SETERR kernel entry point
```

This is the simplest of the five, as the entry to the occam kernel is made directly, whereas the other four jump somewhere else first. This is because we do not expect to generate this very often, so space is less of a constraint. The other four could be generated quite frequently, and to stop the executable size exploding too much, they must be kept small.

3.3. *Coding the rest*

After the input has finished being translated, additional debugging code is placed in the output. The first item to be generated is the procedure names array. The data is organized thus:

```
procedure.label:
    .word   <number of procedure names, N>
    .word   <offset of procname(0) from procedure.label>
            ...
    .word   <offset of procname(N-1) from procedure.label>
    .bytes  <null-terminated procname(0)>
            ...
    .bytes  <null-terminated procname(N-1)>
```

Each of the names is adjusted to be a multiple of four bytes in length, being padded by at least one NULL character (BYTE 0) at the end. The NULL characters ensure that the strings can be printed directly from the C world. The PROC_HIGH:PROC_LOW debugging records provide the index of the procedure name in this structure.

After the procedure names have been written, the filenames are written, in much the same way:

```
filename.label:
    .word   <number of filenames, N>
    .word   <offset of filename(0) from filename.label>
            ...
    .word   <offset of filename(N-1) from filename.label>
    .bytes  <null-terminated filename(0)>
            ...
    .bytes  <null-terminated filename(N-1)>
```

FILE_HIGH:FILE_LOW provides the index into the array in this case. Both arrays are packed fairly tightly into the output, to try to keep the size of the executable down.

Following these two arrays are the entry points for procfile.setup.label, overflow.label, zerodiv.label, and range.entry.label. The function of these is to provide any final setup before jumping into the occam kernel to report the error to the user. The code at procfile.setup.label (jumped to from deadlock debugging records) is slightly different. The reasons are explained in § 3.5.

```
procfile.setup.label:
     set EAX register to address of filename.label
     set EBX register to address of procedure.label
     RET
overflow.label:
     push address of filename.label on kernel stack
     push address of procedure.label
     push EDX ([LINE_LOW, LINE_HIGH, #00, OPCODE])
     push ECX ([PROC_LOW, PROC_HIGH, FILE_LOW, FILE_HIGH])
     call K.OVERFLOW kernel entry point
zerodiv.label:
     push address of filename.label
     push address of procedure.label
     push EDX ([LINE_LOW, LINE_HIGH, #00, #00])
     push ECX ([PROC_LOW, PROC_HIGH, FILE_LOW, FILE_HIGH])
     call K.ZERODIV kernel entry point
range.entry.label:
     push address of filename.label
     push address of procedure.label
     call K.RANGERR kernel entry point
```

When many source files are combined to create a single executable, each object file generated from the translation process will contain its own file name and procedure name arrays. For this reason, the debugging code and information for a particular source file must remain fully within the corresponding output file.

3.4. *The* occam *kernel*

The occam kernel provides four entry points for the different types of run-time error. These are referenced by `K.OVERFLOW`, `K.ZERODIV`, `K.RANGERR`, and `K.SETERR`. For each of these, the parameters passed on the kernel stack provide information about where the error occurred, and what specifically the error was. In each case, the values are checked for sanity (indices in range, valid opcodes, etc.), the error is reported to the user, and the program is terminated.

3.5. *Deadlock*

When the occam kernel detects deadlock, the debugging code attempts to locate processes blocked on inputs, outputs, and `ALT`s. This is done by scanning the workspace looking for instances where `Wptr[Link] = Wptr`, which would have been set just before the kernel call. This condition could occur quite frequently in the workspace, so a series of checks are made to refine the probability that it is a kernel call.

Firstly, `Wptr[Iptr]` is checked to see if it points at a valid address. If it does, two bytes are deferenced and checked. These bytes should be the jump placed immediately after the kernel call. If the jump instruction looks good, the next twelve bytes should be the debugging record. Only a small portion of the first eight bytes can be checked, as no information about the procedure or file name arrays is available. The remaining four bytes are checked for the magic word `#DEADBEEF`*.

* From the Linux kernel spin-lock debugging code.

If everything looks good up to this point, the thirteenth byte is checked for being a valid jump opcode, and, if it is, its address is cast into a pointer to a function, which is then executed. If what we have found is not actually a blocked process, then the program will probably crash, but the probability of this occurring is extremely small. If what was found was indeed a blocked kernel call, then the `function` called will return with pointers to the relevant procedure and file name arrays. This is where the code in `procfile.setup.label` (§ 3.3) differs from the other cases, as it must return to the deadlock debugging code, not to an occam kernel entry point. After return, the rest of the debugging record can be checked, and if everything still looks good, the position where the process deadlocked and the associated operation are reported to the user.

3.6. *The costs of debugging*

The cost of post-mortem debugging on KRoC/Linux can be defined as the amount of extra code executed when an operation does not generate an error, together with the increase in code size when debugging is present. The costs incurred per debugging fragment generated are:

Debug operation	Instructions executed	Bytes used
deadlock	2	21
overflow	1	17
divide by zero	2	20
range checks	0	12
`seterr`	0	25

Range checks and `seterr` do not generate any additional code on the non-error path, as the checks were already there before. The following table shows the increase in code size for various occam programs:

Program	Debug records	% increase
`tranetcp.occ`	772	30
`philfred.occ`	728	49
`beer_punters.occ`	192	56
`commstime.occ`	8	11

The cost at run-time is relatively small; three fifths of the instructions in the non-failure cases are short forward jumps; the remaining are a comparison and a move. Two of these are conditional jumps, in which following the jump (non-failure case) is the predicted course.

The following table shows the time taken for two versions of `tranpc`, one with debugging on, the other with debugging off, to translate `tranetcp.occ` in two ways. The first translation is with debugging disabled, the second is with debugging enabled. The execution times are given in milliseconds:

Translated	`tranpc` without debug	`tranpc` with debug
without debug	66	70
with debug	70	74

4. Further work

When it detects deadlock, this system assumes that all the processes concerned are blocked on communications. It does not yet know anything about semaphores or other synchronization primitives [6].

It also expects all relevant information to be in the normal workspace, so it does not work properly with the experimental recursive version of KRoC [7]. It correctly reports running out of dynamic memory, but cannot handle deadlock in recursive routines.

A possible extension would be for the debugger, after locating the error, to interact with the user to inspect the contents of the workspace. Since `octran` does not keep track of occam variable names (and it would not be easy to make it do so), this would require the user to refer to the assembly-language files.

There are trivial inconsistencies between the two versions that should be eliminated.

The i386 version does not yet deal with floating-point errors.

5. Conclusions

This work has shown that useful post-mortem debugging information can be provided in KRoC at little cost in speed and size of code. We have already found it very helpful in locating errors in students' programs.

6. References

[1] David C. Wood and Peter H. Welch. The Kent Retargetable occam Compiler. *Proceedings of WoTUG-19: Parallel Processing Developments*, edited by Brian C. O'Neill. IOS Press, 1996. ISBN 90–5199– 261–0.

[2] Michael D. Poole. Occam for all – two approaches to retargeting the INMOS compiler. *Proceedings of WoTUG-19: Parallel Processing Developments*, edited by Brian C. O'Neill. IOS Press, 1996. ISBN 90–5199–261–0.

[3] Sun microsystems. SPARCworks 3.0x Debugger Interface. 1994.

[4] SPARC International. The SPARC Architecture Manual. Prentice Hall, 1992. ISBN 0–13– 825001–4.

[5] Michael D. Poole. Extended Transputer Code – A Target-Independent Representation of Parallel Programs. *Proceedings of WoTUG-21: Architectures, Languages and Patterns*, edited by P. H. Welch and A. W. P. Bakkers. IOS Press, 1998. ISBN 90–5199–391–9.

[6] David C. Wood and Peter H. Welch. Higher Levels of Process Synchronisation. *Proceedings of WoTUG-20: Parallel Programming and Java*, edited by A. W. P. Bakkers. IOS Press, 1997. ISBN 90–5199–336–6.

[7] David C. Wood. An Experiment with Recursion in occam. *Proceedings of WoTUG-23: Communicating Process Architectures 2000*, edited by P. H. Welch and A. W. P. Bakkers. IOS Press, 2000.