Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 515–519
DOI: 10.15439/2022F212
ISSN 2300-5963 ACSIS, Vol. 30

Channel-Less Process Communication

Tomas Plachetka
Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics
Email: plachetka@fmph.uniba.sk

***Abstract*—A channel is an abstract data structure which allows for passing messages from one process to another one. We propose several variants of OCCAM, a minimalistic programming language in which a program consists only of processes and channels. The variants differ in how channels are accessed by processes. We prove that all these variants are equally expressive, i.e. an arbitrary OCCAM program can be simulated in any of the variants and the other way around. A particularly interesting variant is to assign exactly one channel to each parallel process. This makes the concept of channels redundant, provided that the parallel processes are named. The simulation techniques can be applied to a variety of abstract models and practical systems.**

I. Introduction

CHANNELS are widely used in abstract models of communicating parallel processes [1], [2], [3], [4], as well as in computer programming languages [5], [6], [7] [8], [9], [10], and hardware descriptions [11], [12], [13]. In operating systems, Unix pipes [14] directly correspond to channels.

The main contribution of this paper is showing a transformation of a channel-based programming language to an equally expressive programming language in which channels and processes become a single entity. We illustrate this transformation on OCCAM [5], [15], which is is based on the synchronous abstract model CSP (Concurrent Sequential Processes) [1] and belongs to practical programming languages whose semantics has been formally defined [16], [17], [18], [19], [20]. Although OCCAM is "pure and small", a sharper Ockham's razor trims it even more. Along with channels, we also eliminate nesting of parallel processes and the `ALT` constructor from OCCAM.

The motivation of this work is not solely theoretical. Many programming languages build on a message passing paradigm without channels, e.g. MPI [21], Erlang [22] and Akka (JVM) [23]. An important question is whether the absence of channels is restraining. This paper suggests a negative answer. A consequence is that channel-based (OCCAM-like) programming languages are intrinsically redundant.

The paper is organised as follows. Section II presents a relevant subset of the OCCAM language (leaving out unnecessary details, occasionally using an abbreviated syntax). In Section III, it is shown that nesting of parallel processes can be replaced by a flat process structure (a variant OCCAM-1PAR). In Section IV, the directional graph interconnection of processes and channels is replaced with a hypergraph interconnection which uses shared channels (OCCAM-SH). In Section V, a concrete hypergraph structure is proposed which leads to a unification of process and channel identifiers, i.e. to a channel-less model (OCCAM-CL). All the variants OCCAM, OCCAM-1PAR, OCCAM-SH, and OCCAM-CL are equally expressive. Section VI concludes the paper.

This research has been supported by the grant 1/0601/20 of the Slovak Scientific Grant Agency VEGA.

II. OCCAM

OCCAM was targeted to Transputers [24], single-chip computers specifically designed to support parallel programming. Transputers could be easily connected to form a network, process scheduling and communication were implemented in the hardware. Although Transputers were discontinued in 1990's, they could in some parameters compete with contemporary computers (e.g. context switch below 1 μsec still belongs to the fastest ever). OCCAM found its followers, e.g. OCCAM-π [7] and Rain [8].

An OCCAM program consists of a finite number of processes and a finite number of channels; these numbers are known before the program starts and do not change in runtime. A process in OCCAM is either an atomic process (`:=`, assignment; `?`, input from a channel; `!`, output to a channel; `SKIP`, which does nothing and terminates), or a compound process. Constructors of compound processes (`SEQ`, `PAR`, `IF`, `WHILE`, `ALT`) combine processes into a single one. Processes in the `SEQ` constructor are executed sequentially in the given order, i.e. when a process terminates, the following one becomes active. `SEQ` terminates when its last mentioned process terminates. The constructors `IF`, `WHILE` and the assignment process behave in a common way, except for `IF`, in which the conditions are always tested in the order they are written, and (only) the process under the first satisfied condition becomes active (`IF` terminates when this process terminates).

The replicator `FOR` can be used with constructors. Replication works as a macro expansion. For example, `SEQ i=0 FOR 2 p` expands to `SEQ p p`, with $i = 0$ in the first replica of the process p, and $i = 1$ in the second one. This corresponds to a sequential repetition of the process p in a for-loop.

The scope of a variable is limited to the process following the variable's declaration (e.g. `INT v:`), including processes nested in that process (variables used in replicators are not declared). Usual primitive types are available. The only compound type is an array, indexed from 0.

Processes of the `PAR` constructor run in parallel and have read-only access to variables which are shared in their scopes. A process is allowed to change (using `:=` or `?`) only the

variables which it does not share with another parallel process. The only way how parallel processes may influence one another is via channels which are declared as variables of type `CHAN`. `PAR` terminates when all its processes terminate.

Channels in OCCAM are unidirectional and unbuffered. Each channel connects exactly two parallel processes—one reads from the channel, the other one writes to the channel. Parallel processes can be depicted as vertices and channels as edges of a directed graph (an edge points from the writer to the reader). Reading from a channel (`?`) blocks until a `!` process writes to the channel; and conversely, writing to a channel (`!`) blocks until a `?` process reads from the channel). The corresponding `?` and `!` both terminate after the message has been transferred across the channel from the `!` process to the `?` process. A message is a finite sequence of values (of primitive types or arrays).[1]

The `ALT` constructor multiplexes reading from several input channels.[2] While reading from each single input channel would block, the `ALT` constructor blocks. When reading at least one input channel would terminate (we say that such a channel is readable), a message is read from one readable channel and a process in the corresponding branch takes over. When this process terminates, then `ALT` terminates.

The choice of the readable channel inside the `ALT` constructs and the scheduling of the parallel processes is not under program's control. The scheduler executing the program is unpredictable (nondeterministic) in making these choices. This does not mean that it has to be "random". For example, it is allowed to (but does not have to) prioritise the readable channels in the order they are mentioned in `ALT` constructs. Similarly, it is allowed to (but does not have to) prioritise the execution of active parallel processes in the order they are mentioned in `PAR` constructs. A correct program must guarantee its intended behaviour for all possible schedules.

Processes and channels are static, i.e. they are constructed before the execution of a program. During the execution, each process is in one of the following states: active, passive, blocked. The program evolves according to the rules described above (see [18] for details). At the beginning of the execution of a program, the first process is active, all other processes are passive. A termination of an active process does not mean that the process ceases to exist—it just changes its state to passive. The program execution terminates when there are no active processes. (We can distinguish between a "correct termination" where all processes are inactive, and a deadlock where at least one process is blocked.)

[1]Although OCCAM requires a declaration of types of messages which are passed over channels (so-called protocols, e.g. `CHAN OF INT; BOOL ch:`, we consistently use type-less channel declarations throughout the paper. These correspond to `CHAN OF ANY` declarations in OCCAM, where the programmer is responsible for ensuring that the sequences of values in messages transferred over a channel from a `!` process are of the same types as the corresponding variables in a `?` process (so that an incoming message can be stored to the variables in the `?` process).

[2]For the sake of simplicity, we do not consider `ALT` with boolean expressions attached to the channel inputs (so-called guarded `ALT`).

Fig. 1 shows two OCCAM programs used as running examples throughout the paper. The programs simulate each other. For an arbitrary schedule, both programs terminate and the variable m attains values 0, 1, 2, 3 in one of the following orderings: [0, 1, 2, 3], [0, 1, 3, 2], [1, 0, 2, 3], [1, 0, 3, 2]. The actually observed ordering depends on a concrete schedule. For an arbitrary schedule of P_1, a schedule of P_2 exists such that the orderings match, and vice versa. Moreover, there is a bijective correspondence of processes which run in parallel in P_1 and P_2, regardless of their nesting in `PAR` constructs (we refer to line numbers in P_1 and P_2): [5, 5], [11, 12], [12, 13]. In this sense, P_1 and P_2 are equivalent.

Program P_1

```
 1: CHAN ch1, ch2:
 2: SEQ i = 0 FOR 2
 3:     PAR
 4:         INT m:
 5:         SEQ j = 0 FOR 2
 6:             ALT
 7:                 ch1 ? m
 8:                   SKIP
 9:                 ch2 ? m
10:                   SKIP
11:         ch1 ! 2 * i
12:         ch2 ! (2 * i) + 1
```

Program P_2

```
 1: CHAN ch1, ch2:
 2: SEQ i = 0 FOR 2
 3:     PAR
 4:         INT m:
 5:         SEQ j = 0 FOR 2
 6:             ALT
 7:                 ch1 ? m
 8:                   SKIP
 9:                 ch2 ? m
10:                   SKIP
11:         PAR
12:             ch1 ! 2 * i
13:             ch2 ! (2 * i) + 1
```

Fig. 1. Two OCCAM programs (running examples)

In the sequel, we only deal with one-sided simulations in which P' is constructed from P by replacing its fragments of code. In all proofs, we give a construction of P' which preserves parallelism of P (as the simulations are one-sided, we only insist on an injective mapping of parallel processes of P to P'; e.g. additional parallel processes can be added to P' in a simulation. A programming language L is *at least as expressive* as a programming language L', if there is an algorithm (compiler) which translates an arbitrary program P' in L' to a program P in L which simulates P'. Two programming languages L and L' are *equally expressive*, if L is at least as expressive as L', and vice versa.

III. OCCAM WITH A SINGLE TOP-LEVEL `PAR` CONSTRUCTOR (OCCAM-1PAR)

Theorem 1: OCCAM with a single top-level `PAR` constructor (OCCAM-1PAR) is as expressive as OCCAM.

Proof: We need to show that an arbitrary OCCAM program can be simulated by an OCCAM program which uses exactly one `PAR` which is the top-level process. Consider an arbitrary OCCAM program. All channel declarations are moved to the top-level `PAR` (collisions of channel names are resolved by using fresh names). We will refer to processes of `PAR`s in the OCCAM program as child processes. Each child process is moved to the top-level `PAR`, together with declarations of all variables in its scope (including shared variables used in replicators). A new local integer variable `terminate` is declared in the child process; and two new channels are declared in the top-level `PAR`, we will call

them $ch.s$ (start) and $ch.e$ (end). A child process is started when it receives a message beginning with `FALSE` from the channel $ch.s$. This message also contains values of all the variables shared for reading between the parent and the child. When the child finishes its original program, it sends an acknowledgement to the channel $ch.e$. The parent (the sequence which replaces the `PAR`) starts its children and waits for their acknowledgements. Just before its own termination, the parent terminates its children. ■

Fig. 2 shows the translation of the program P_1 (Fig. 1) to an OCCAM-1PAR program P_3 with a single top-level `PAR`.

IV. OCCAM WITH SHARED CHANNELS (OCCAM-SH)

Recall that a channel in OCCAM connects two parallel processes. OCCAM-SH is an interesting variant in which channels are shared, i.e. a channel can be simultaneously accessed by arbitrarily many parallel processes for both reading and writing. When several `!` processes simultaneously write to a channel, then they are blocked until a `?` process reads from the channel. When several `?` processes simultaneously read from the same channel, they are blocked until a `!` process writes to the channel. When one or more `?` processes read from a channel and one or more `!` processes write to the channel, then eventually one of the `?` processes and one of the `!` processes are chosen for communication. This choice is made arbitrarily (i.e. the scheduler can freely decide which processes it chooses for communication). Then the message is passed from the chosen `!` process to the chosen `?` process and then these two processes terminate. Unlike in OCCAM, there is no `ALT` constructor in OCCAM-SH. Furthermore, OCCAM-SH programs use only one top-level `PAR` constructor.

It turns out that in spite of the absent `ALT` constructor, OCCAM-SH is a generalisation of OCCAM. We will show how an arbitrary OCCAM-1PAR program can be translated to an OCCAM-SH program which simulates the former one. This translation requires that the parallel processes and channels have identifiers. The order in which a parallel process appears in the single `PAR` constructor) will serve as its identifier. Analogously, channels are numbered in the order they are declared (to keep the notation simple, a channel identifier will serve as the channel's number). Let the numbering start with 0, let N denote the number of processes.

Theorem 2: OCCAM-SH is at least as expressive as OCCAM.

Proof: We have already proved that an arbitrary OCCAM program can be simulated by an OCCAM-1PAR program. It remains to show how the `ALT` constructors of OCCAM-1PAR are simulated in OCCAM-SH. Consider an OCCAM-1PAR program. For each parallel process p, merge all the channels from which the process reads to a single channel $ch.in_p$. This shared channel will be the only one which the process p will read, and p will be its only reader (there may be more than one writer, though). Declare arrays `INT pending.ch[N]` and `MSG pending.msg[N]` in the scope of each parallel process, where `MSG` is the type of messages transferred in the OCCAM-1PAR program. Initialise

Program P_3

```
 1: CHAN ch1, ch2:
 2: CHAN ch.s1, ch.e1, ch.s2, ch.e2, ch.s3, ch.e3:
 3: PAR
 4:     SEQ -- original top-level code
 5:         SEQ i = 0 FOR 2
 6:             BOOL ack:
 7:             SEQ -- replacement of parent PAR
 8:                 ch.s1 ! FALSE; i -- start child 1
 9:                 ch.s2 ! FALSE; i -- start child 2
10:                 ch.s3 ! FALSE; i -- start child 3
11:                 ch.e1 ? ack -- wait for end of child 1
12:                 ch.e2 ? ack -- wait for end of child 2
13:                 ch.e3 ? ack -- wait for end of child 3
14:         ch.s1 ! TRUE; 0 -- terminate child 1
15:         ch.s2 ! TRUE; 0 -- terminate child 2
16:         ch.s3 ! TRUE; 0 -- terminate child 3
17:     BOOL terminate:
18:     INT i:
19:     SEQ -- child 1 of PAR
20:         ch.s1 ? terminate; i -- wait for parent
21:         WHILE NOT terminate
22:             SEQ
23:                 INT m:
24:                 SEQ j = 0 FOR 2
25:                     ALT
26:                         ch1 ? m
27:                             SKIP
28:                         ch2 ? m
29:                             SKIP
30:                 ch.e1 ! TRUE -- acknowledge parent
31:                 ch.s1 ? terminate; i -- wait for parent
32:     BOOL terminate:
33:     INT i:
34:     SEQ -- child 2 of PAR
35:         ch.s2 ? terminate; i -- wait for parent
36:         WHILE NOT terminate
37:             SEQ
38:                 ch1 ! 2 * i
39:                 ch.e2 ! TRUE -- acknowledge parent
40:                 ch.s2 ? terminate; i -- wait for parent
41:     BOOL terminate:
42:     INT i:
43:     SEQ -- child 3 of PAR
44:         ch.s3 ? terminate; i -- wait for parent
45:         WHILE NOT terminate
46:             SEQ
47:                 ch2 ! (2 * i) + 1
48:                 ch.e3 ! TRUE -- acknowledge parent
49:                 ch.s3 ? terminate; i -- wait for parent
```

Fig. 2. OCCAM → OCCAM-1PAR ($P_1 \to P_3$)

the array `pending.ch[N]` with values -1, i.e. insert the following sequence after the initial `SEQ` in each process:

```
SEQ i = 1 FOR NP
    pending.ch[i] := -1
```

Replace each `ch ! m` of the parallel process w with

```
BOOL ack:
SEQ
    ch.in_r ! w; ch; m
    ch.in_w ? ack
```

where r is the identifier of the process which reads the channel ch in the original OCCAM program.

Replace each

```
ALT
    ch0 ? m
      b0
    ...
    chk ? m
      bk
```

of the parallel process r with the sequence

```
INT w, ch:
BOOL consumable:
SEQ
  consumable := FALSE
  WHILE NOT consumable
    SEQ
      w := 0 -- look for a consumable message
      WHILE (w < NP) AND (pending.ch[w] ≠ ch0) ...
        AND (pending.ch[w] ≠ chk)
        w := w + 1
      IF
        w < NP
          SEQ -- found a consumable message
            m := pending.msg[w]
            ch := pending.ch[w]
            consumable := TRUE
        TRUE -- otherwise save another message
          SEQ
            ch.inr ? w; ch; m
            pending.ch[w] := ch
            pending.msg[w] := m
  IF -- execute the corresponding branch of ALT
    ch = ch0
      SEQ
        ch.inw ! TRUE
        pending.ch[w] := -1
        b0
    ...
    ch = chk
      SEQ
        ch.inw ! TRUE
        pending.ch[w] := -1
        bk
```

Treat each `ch ? m` as

```
ALT
    ch ? m
      SKIP
```

and use the translation above.

Hence, each `!` subsequently blocks until it is acknowledged by the `ALT`ing process. The `ALT`ing process reads all incoming messages, but acknowledges only those which have been consumed by the `ALT` of the OCCAM-1PAR program. ■

Theorem 3: OCCAM is at least as expressive as OCCAM-SH.

Proof: We present a construction which replaces shared channels with directed channels which connect exactly two processes. Let N denote the number of parallel processes in the OCCAM-SH program. For each shared channel ch, an additional parallel process $sh.ch$ is created which relays the communication on the shared channel using OCCAM channels. These $sh.ch$ processes terminate when all the other parallel processes terminate. A process $sh.ch$ is connected via three channels with each parallel process p ($p = 0, \ldots, N-1$) of the OCCAM-SH program: $ch.r_{sh.ch}[p]$, $ch.w_{sh.ch}[p]$ and $ch.d_{sh.ch}[p]$. All these channels are declared as global. The first two channels are oriented towards $sh.ch$, the third one towards the parallel process p,

The program of a process $sh.ch$ is:

```
BOOL terminate:
INT out, t.count:
MSG m:
SEQ
  t.count := 0
  WHILE t.count < N
    SEQ
      ALT -- pick a reader
        ch.rsh.ch[0] ? terminate
          out := 0
        ...
        ch.rsh.ch[N - 1] ? terminate
          out := N − 1
      IF
        terminate
          t.count := t.count + 1
        TRUE -- otherwise
          ALT -- pick a writer, deliver m to the reader
            ch.wsh.ch[0] ? m
              ch.dsh.ch[out] ! m
            ...
            ch.wsh.ch[N - 1] ? m
              ch.dsh.ch[out] ! m
```

Replace each `ch ? m` of the parallel process r in the OCCAM-SH program with the sequence

```
SEQ
  ch.rsh.ch[r] ! FALSE
  ch.dsh.ch[r] ? m
```

Replace each `ch ! m` of the parallel process w with `ch.`$w_{sh.ch}$`[w] ! m`.

Insert `ch.`$r_{sh.ch}$`[p] ! TRUE` at the end of each parallel process p of the OCCAM program. ■

V. CHANNEL-LESS OCCAM (OCCAM-CL)

OCCAM-SH1 is a stricter variant of OCCAM-SH in which each parallel process p is allowed to read only from one channel $ch.in_p$, whereby p is the only process which reads from the channel $ch.in_p$. We call this variant channel-less, because the identifiers of channels unify with the identifiers of processes. The processes can be numbered in the order they appear in the single `PAR` constructor.

Theorem 4: OCCAM-SH1 and OCCAM-SH are equally expressive.

Proof: An OCCAM-SH1 program is also an OCCAM-SH program. Conversely, consider an arbitrary OCCAM-SH program. Translate it to OCCAM and then back to OCCAM-SH using compilers from the proofs of Theorem 3 and Theorem 2. This yields an OCCAM-SH1 program. ■

OCCAM-CL syntactically removes the redundancy related to channels from OCCAM-SH1. An arbitrary OCCAM-SH1 program can be rewritten to OCCAM-CL as follows:

- Remove all channel declarations.
- Replace each `ch ? m` with `? m`.
- Replace each `ch ! m` with `p ! m`, where p is the identifier of the process which reads the channel ch.

Figure 3 illustrates the correspondence between OCCAM-SH1 and OCCAM-CL.

```
1: Program P4
2: CHAN ch0, ch1, ch2:
3: PAR
4:     INT m:
5:     SEQ -- process 0
6:         SEQ i = 0 FOR 2
7:             SEQ j = 0 FOR 2
8:                 ch0 ? m
9:             ch11 ! TRUE
10:            ch2 ! TRUE
11:    BOOL ack:
12:    SEQ -- process 1
13:        SEQ i = 0 FOR 2
14:            SEQ
15:                ch0 ! 2 * i
16:                ch1 ? ack
17:    BOOL ack:
18:    SEQ -- process 2
19:        SEQ i = 0 FOR 2
20:            SEQ
21:                ch0 ! (2 * i) + 1
22:                ch2 ? ack
```

```
1: Program P5
2:
3: PAR
4:     INT m:
5:     SEQ -- process 0
6:         SEQ i = 0 FOR 2
7:             SEQ j = 0 FOR 2
8:                 ? m
9:             1 ! TRUE
10:            2 ! TRUE
11:    BOOL ack:
12:    SEQ -- process 1
13:        SEQ i = 0 FOR 2
14:            SEQ
15:                0 ! 2 * i
16:                ? ack
17:    BOOL ack:
18:    SEQ -- process 2
19:        SEQ i = 0 FOR 2
20:            SEQ
21:                0 ! (2 * i) + 1
22:                ? ack
```

Fig. 3. Hand-made translations of the OCCAM program P_1 (Fig 1) to OCCAM-SH1 (left) and OCCAM-CL (right)

VI. Conclusions

We proposed a programming language OCCAM-CL which differs from OCCAM (only) in having no nested `PAR` processes, no `ALT` constructors, and—most importantly—no channels. In spite of this, OCCAM-CL is as expressive as OCCAM. We proved this using several OCCAM variants and compilers which translate programs from one variant to any other one. These compilers preserve parallelism of programs as well as their message complexity (up to a constant multiplicative factor). A similar result was published in [25] for a lambda calculus with typed asynchronous channels and a lambda calculus with typed actors.

When it comes to writing an actual compiler, the choice between OCCAM-CL and OCCAM is not just a matter of taste. Apparently, writing a parser for OCCAM-CL is easier, but there are more subtle reasons for favouring OCCAM-CL. For example, OCCAM requires that each channel connects exactly two parallel processes (one reader, one writer). However, its syntax does not prevent the programmer from violating this requirement. It is up to the compiler or a run-time system to detect such a violation.

The channel-less approach can be found in actor models [26] as well as in contemporary programming languages, e.g. Erlang and Akka. A subsequent extension of Akka with the channel concept is in the light of our results a backward step. It does not increase expressiveness, unnecessarily increases the complexity of the language and the compiler, and increases the structural complexity of programs which mix the channel and channel-less paradigms.

References

[1] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[2] R. Milner, "Elements of interaction," *Communications of the ACM*, vol. 36, no. 1, pp. 70–89, 1993, Turing Award lecture.

[3] M. Ahuja, A. D. Kshemkalyani, and T. Carlson, "A basic unit of computation in distributed systems," in *International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 1990. doi: 10.1109/ICDCS.1990.89327 pp. 12–19.

[4] J. Biernacki, "Alvis models of safety critical systems state-base verification with nuXmv," in *FedCSIS*, ser. Annals of Computer Science and Information Systems, vol. 8. IEEE, 2016. doi: 0.15439/2016F264 pp. 1701–1708.

[5] SGS Thomson Ltd., *Occam 2.1 Reference Manual*. Prentice Hall, 1988.

[6] A. Ripke, A. A. Allen, R. Alastair, and Y. Feng, "Distributed computing using channel communications in Java," in *Communicating Process Architectures 2000*. IOS Press, 2000, pp. 49–62.

[7] P. H. Welch and F. R. M. Barnes, "Communicating mobile processes: Introducing OCCAM-π," in *Communicating Sequential Processes*, ser. LNCS. Springer, 2005, vol. 3525, pp. 712–713.

[8] N. C. C. Brown, "Rain: A new concurrent process-oriented programming language," in *Communicating Process Architectures*. IOS Press, 2006, pp. 237–251.

[9] R. Loogen, "Eden—parallel functional programming with Haskell," in *Central European Functional Programming School, (CEFP), Budapest, Hungary*, ser. LNCS, vol. 7241. Springer, 2011. doi: 10.1007/978-3-642-32096-5_4 pp. 142–206.

[10] G. D'Angelo, S. Ferretti, and M. Marzolla, "Time warp on the Go," in *Proc. of the International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTOOLS '12. ICST, Brussels, Belgium, 2012. doi: 10.5555/2263019.2263057 pp. 242–248.

[11] M. W. Heath, W. P. Burleson, and I. G. Harris, "Synchro-tokens: A deterministic GALS methodology for chip-level debug and test," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1532–1546, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2005.203

[12] M. A. Rahimian, S. Mohammadi, and M. Fattah, "A high-throughput, metastability-free GALS channel based on pausible clock method," in *Asia Symposium on Quality Electronic Design*. IEEE, 2010. doi: 10.1109/ASQED.2010.5548259 pp. 294–300.

[13] P. Hajder, L. Rauch, M. Nycz, and M. Hajder, "A heterogeneous parallel processing system based on virtual multi-bus connection network," in *FedCSIS (Position Papers)*, ser. Annals of Computer Science and Information Systems, vol. 19, 2019. doi: 10.15439/2019F356 pp. 9–17.

[14] *ISO/IEC 9945-1: 1990 Information Technology. Portable Operating System Interface (POSIX), Part 1: System Application Program Interface*.

[15] J. Galletly, *OCCAM 2. Including OCCAM 2.1*. UCL Press, 1996.

[16] A. W. Roscoe, "Denotational semantics for OCCAM," in *Seminar on Concurrency, Carnegie-Mellon University*. London, UK: Springer, 1985, pp. 306–329.

[17] A. Eliëns, "Semantics for OCCAM," Centre for Mathematics and Computer Science (CWI), Amsterdam, Tech. Rep. 6255, 1986.

[18] Y. Gurevich and L. S. Moss, "Algebraic operational semantics and OCCAM," in *Proceedings of the 3rd Workshop on Computater Science Logic*, ser. CSL '89. London, UK: Springer, 1990. doi: 10.1007/3-540-52753-2_39 pp. 176–192.

[19] A. W. Roscoe, M. H. Goldsmith, and B. G. O. Scott, "Denotational semantics for OCCAM 2, part 1," *Transputer Communications*, vol. 1, pp. 65–91, 1994.

[20] ——, "Denotational semantics for OCCAM 2, part 2," *Transputer Communications*, vol. 2, pp. 25–67, 1994.

[21] MPI Forum, *MPI-4.0*. HLRS, 2021.

[22] F. Cesarini and S. Thompson, *ERLANG Programming*. O'Reilly, 2009.

[23] D. Wyatt, *Akka Concurrency*. Artima Incorporation, 2013.

[24] I. Graham and T. King, *The Transputer Handbook*. Prentice Hall, 1990.

[25] S. Fowler, S. Lindley, and P. Wadler, "Mixing metaphors: Actors as channels and channels as actors," in *31st European Conference on Object-Oriented Programming, ECOOP 2017, Barcelona, Spain*, ser. LIPIcs, vol. 74. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.11 pp. 11:1–11:28.

[26] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. Morgan Kaufmann, 1973, pp. 235–245.