Checking Process-Oriented Operating System Behaviour using CSP and Refinement

Frederick R. M. Barnes and Carl G. Ritson
School of Computing, University of Kent
Canterbury, Kent, CT2 7NF, United Kingdom
frmb@kent.ac.uk, cgr@kent.ac.uk

ABSTRACT

Process orientation is an approach to concurrency that uses concepts of processes and message-passing communication, with whole systems constructed from layered and dynamically evolving networks of communicating processes. The work described in this paper relates to the automatic model generation and verification of systems developed in process-oriented languages. We discuss some early applications of this technique to our experimental operating system, RMoX, as a means to giving a guarantee of correct system behaviour at a range of levels.

Keywords

occam-pi, RMoX, CSP, FDR, refinement

1. INTRODUCTION

Process orientation is an approach to concurrency using processes and communication, with whole systems built from layered networks of communicating processes. The occam-π programming language [17] is an enhanced version of the original 'occam 2.1' language, previously used for programming *Transputer* based systems in the 1980s and 1990s. Ideas underlying the Transputer programming model have prevailed, and have been adapted to take advantage of modern shared-memory processors. Lightweight process scheduling and efficient communication allow concurrency to be used freely, without the concern of significant performance overheads [14].

We have used the occam-π language to develop an experimental operating system, "RMoX", for x86 based hardware [1]. Within RMoX are hundreds to thousands of concurrent processes, organised into layered networks, communicating and synchronising through channel communication (message passing). In addition to communicating data with *copying* semantics, data and *mobile channel-ends* may be communicated with *movement* semantics, transferring the 'ownership' of them and incurring only a small fixed overhead on shared-memory systems (communicating a pointer).

Use of the occam-π language itself eliminates many potential sources of programmer error, such as those arising from dereferencing null or undefined pointers, as well as the more serious issue of race-hazards on shared data and uncontrolled aliasing. Whilst we do support various forms of low-level manipulation in the language (necessary for hardware interaction), these are clearly identified. The static checks employed by the compiler give some guarantee of correct component operation (e.g. freedom from aliasing and race-hazard errors), but cannot guarantee that inter-process interactions (by channel communication) do not lead to deadlock. This is one of the issues investigated here.

The formal models considered use the *Communicating Sequential Processes* (CSP) process algebra [10, 15], that allows for reasoning about the behaviour of concurrent systems. The Transputer hardware and original occam language were developed with CSP in mind [8], providing both with a strong formal underpinning. Leveraging this formal basis, a variety of tools were developed throughout the 1980s and 1990s to aid in the development of provably correct Transputer systems [12, 19, 3].

The work presented here takes this relationship 'full-circle', with the generation of CSP formal models from occam-π implementations, independent of any surrounding software development environments and tools (that may be domain specific or impose particular design constraints). Checking and verifying the generated models requires specifications and an environment in which to check them. For the latter, we use the 'FDR' tool (*Failures-Divergence Refinement*) from Formal Systems [6], that accepts a machine-readable form of the algebra, CSP$_M$. This paper focuses on the model checking of components within the RMoX OS, as a means to guarantee our own and third-party implementations — specifically that a process behaves correctly with respect to the surrounding system.

Section 2 provides background on the technologies involved. New ideas for the automatic modelling and checking of occam-π programs are presented in Section 3, alongside initial results for RMoX components. Section 4 provides some concluding remarks, including benefits and limitations, and outlines our plans for future work.

2. BACKGROUND

The RMoX OS is constructed as a network of communicating processes, the top-level of which is shown in Figure 1. Some components such as the '`idle.task`' are single sequential processes. Whereas others such as the '`driver.core`', are networks of sub-processes, as shown in Figure 2, a layering

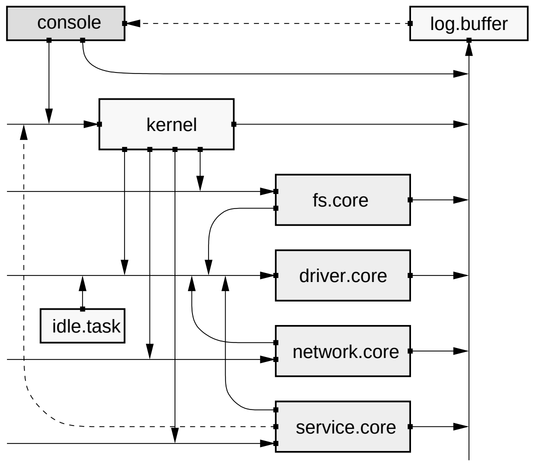

Figure 1: RMoX top-level process network.

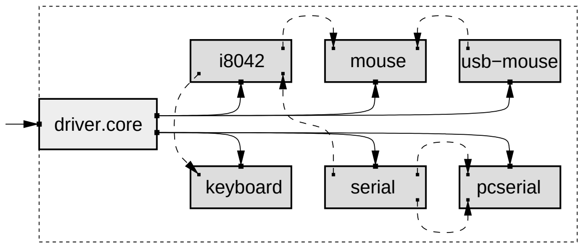

Figure 2: Driver processes and interconnects.

that continues downwards as appropriate. Instead of routing messages around the static infrastructure of the system, direct communication links between components are dynamically established — with the existing infrastructure routing these *mobile channel-bundles*. As a result, the connectivity of components changes at run-time, consequently making traditional 'top-down' static analysis difficult or impossible.

The connecting lines in Figure 1 and Figure 2 represent *mobile channel-bundles*; unbroken lines show the static layout, dashed lines the dynamic. Each of these bundles contains one or more individual *channels*, that are synchronous directed point-to-point communication links. A common usage pattern for these is *client-server*, and as a result, we refer to the two *ends* as 'client' and 'server', the latter indicated by the arrowhead. These bundles may also be shared, at either end, where processes compete for access through a mutual exclusion mechanism (*claiming*). The client-server *design pattern* is used often, partly because it is appropriate for this type of system, but also because we have an understanding of how this pattern can be used to build deadlock free systems [19]. However, this can only guide the programmer — if errors are made when implementing, the result will likely be a deadlocked system, or worse, a system that fails in specific circumstances, hard to recreate in testing. By automatically generating formal models of actual implementations, and checking these against design specifications, we aim to avoid such errors.

2.1 The occam-pi language

The occam-π language extends classical occam with features that provide *dynamics* and *mobility*. Some of the underlying ideas, such as the ability to communicate channels

```
PROC driver (CT.UPORT? link, SHARED LOG! log)
  INT info:
  SEQ
    CLAIM log!
      out.string ("driver starting..*n", log[b]!)
    ...  local initialisation
    CHAN INT notify:
    PAR
      watchdog (notify!, log!)

      WHILE TRUE
        INT port, dir, start, len, val:
        PRI ALT
          link[in] ? CASE
            get.info
              link[out] ! info; info
            set.dir; port; dir
              ...  set port direction
            rd.dig; start; len
              IF
                (start + len) > max.ports
                  link[out] ! error
                TRUE
                  SEQ
                    ...  read port values
                    link[out] ! dig; val
            wr.dig; start; len; val
              ...  write port values
          INT v:
          notify ? v
            ...  handle notification
:
```

Figure 3: Example occam-pi source code.

over channels, stem from the π-calculus [13]. Although we do not use the π-calculus directly, we do have a CSP model of the mobile channel mechanism present in the occam-π language [18].

Figure 3 shows an example of the occam-π code within RMoX. This partial implementation of a 'user-port' driver handles requests on the '`in`' channel inside its '`link`' channel-bundle, and responds on the '`out`' channel in that same bundle. The code shown also services a local '`notify`' channel by selecting between inputs from this and the '`link[in]`' channel, with priority given to the '`notify`' channel ('`PRI ALT`' construct). Scope in occam-π is denoted using indentation; both parallel and sequential code must be explicitly identified, by '`PAR`' and '`SEQ`' respectively.

The individual channels within channel-bundles are referred to by name, using an array subscript syntax. The ends of these bundles have distinct types, e.g. '`CT.UPORT?`' for the server-end and '`CT.UPORT!`' for the client. As shown in Figure 3, before the channels within a *shared* end ('`log!`') may be used, they must be claimed. This adds opportunities for deadlock arising from improperly ordered claims, and from communication on other channels within these blocks.

2.2 CSP and FDR

The CSP algebra describes concurrent processes and their interactions [10, 15]. At the simplest level, processes *engage* (or *synchronise*) on individual *events*. This synchronisation can extend to more than two processes in CSP, however, occam-π channel communication always involves exactly two processes (sender and receiver).

To give a flavour of the algebra, equation (1) shows a simplified model of the `WHILE` loop from Figure 3. The internal choice operator, $\sqcap$, is used to model the branches of the `IF`. External choice, $\square$, models the '`ALT`' construct which *chooses* between available events. The event-prefix operator,

$\rightarrow$ ('then'), engages on the event on its left, then behaves as the process on its right. Sequential composition is expressed with ';' and *SKIP* is a primitive process that does nothing except terminates successfully.

$$P = \big((l.get \rightarrow l.inf \rightarrow SKIP) \,\Box\, (l.set \rightarrow SKIP) \,\Box\, (l.rd \rightarrow ((l.err \rightarrow SKIP) \sqcap (l.rd \rightarrow SKIP))) \,\Box\, (l.wr \rightarrow SKIP) \,\Box\, (n \rightarrow SKIP)\big);\ P \quad (1)$$

Those developing occam-π components for RMoX (and other systems) do not need to be CSP experts to gain benefit from the work described here. While behavioural specifications, for example describing legitimate patterns of communication over occam-π channel bundles, are needed for verifying implementations, these can be automatically generated from occam-π source code. However, an appreciation of certain aspects of CSP and related model-checking is necessary to understand the validity of our approach.

2.2.1 Traces, failures, divergence and refinement

For any CSP process, there are three behavioural models: *traces*, *failures* and *divergences.* Traces describe what *sequences* of events a process *might* perform, roughly equivalent to tracing out all possible paths of execution. The six obvious traces for the process P in (1) are:

$$\langle l.get, l.inf\rangle, \langle l.set\rangle, \langle l.rd, l.err\rangle, \langle l.rd, l.dig\rangle, \langle l.wr\rangle, \langle n\rangle$$

Because the process P is recursive, the above shows only *partial* traces (from one cycle of its execution). The traces here also indicate that P is *non-deterministic* — after accepting $l.rd$, it responds with either $l.err$ or $l.dig$.

Failures describe what sequences of actions must have been performed (observed behaviour) and what events may next be offered (or not offered) to cause deadlock. This is represented as a set of pairs of traces and event sets. For example:

$$\big\{(\langle\rangle, \{\}), (\langle l.get\rangle, \{l.get, l.set\}), (\langle l.get, l.inf\rangle, \{l.dig\})\big\}$$

The first is straightforward — if the process has not performed any actions, and none are offered, it will deadlock. The second states that if the process has accepted the $l.get$ event, and only $l.get$ and $l.set$ are offered, then it will deadlock — it can be seen in (1) that the process will only engage with $l.inf$ once it has accepted $l.get$.

Divergences are similar to failures, except that these describe under what conditions the process will livelock. The process P, as it stands, is *non-divergent.*

A primary use of CSP is for *refinement* checking. Based on traces, failures and divergences are *refinement relations*, effectively subset relations on these models. For traces, these indicate that one process can perform at least the same sequences of events as another; for failures and divergences, these indicate whether one process is more *deterministic* than another. In practice, refinement checks are used to verify that a particular implementation meets a given specification, and to ensure that these are deadlock and livelock free. For this we use the FDR tool, whose input is a CSP$_M$ script.

3. MODEL CHECKING OS COMPONENTS

We have modified the occam-π compiler to generate a CSP-like representation of a program's behaviour. Within the compiler itself, a simplified semantic model is constructed, using CSP-style operators, ultimately resulting in an XML file containing models for the various source-level procedures. This is further processed using an XML stylesheet to generate CSP$_M$ for FDR. The intermediate XML can additionally be used to produce input for other tools that check different properties of the system (that are not CSP and unsuitable for FDR). An assortment of compiler flags give specific control over model generation from occam-π source.

A complete 'top-down' model-check of the whole RMoX system is unrealistic given the state-space size involved (arising from dynamic extensions). Checks are instead limited to individual components and parallel compositions of a few. As our primary goal is to verify the correct behaviour of *individual* components with respect to the rest of the system, this 'bottom-up' approach is sufficient, and is a necessary step towards verifying the correct behaviour of the *entire* system (future work).

3.1 Model generation

The RMoX OS makes extensive use of *variant* protocols and channel-bundle types, so our early work concentrates on these. At the language level, protocol declarations are essentially separate, and are combined only within individual channel-bundle type declarations. The formal model pulls these together, generating single *data-types* (an FDR abstraction) that represent all the variant actions that can be performed. For example, from the simplified declarations:

```
PROTOCOL P.UPORT.IN              -- protocol definition
  CASE
    get.info
    set.dir; INT; INT
    rd.dig; INT; INT
    wr.dig; INT; INT; INT
:
PROTOCOL P.UPORT.OUT             -- protocol definition
  CASE
    info; INT
    dig; INT
    error
:
CHAN TYPE CT.UPORT               -- channel-bundle type
  MOBILE RECORD
    CHAN P.UPORT.IN in?:
    CHAN P.UPORT.OUT out!:
:
```

we (automatically) produce the following FDR definition:

```
datatype CTPROT_CT_UPORT = InPUpInGetInf |
  InPUpInSetDir | InPUpInRdDig | InPUpInWrDig |
  OutPUpOutInf | OutPUpOutDig | OutPUpOutErr |
  DoClaimCtUp | DoReleaseCtUp
```

This models communication events on the channels within a CT.UPORT bundle in terms of the individual *variant* names. Also included are events for the claiming and releasing of shared channel-ends.

We also construct a CSP model for the behaviour of a process using this channel-type (in occam-π, converted to CSP$_M$). For the above protocol this model is:

```
SPEC_CT_UPORT(s) =
  (((s.InPUpInGetInf -> s.OutPUpOutInf -> SKIP) []
   (s.InPUpInSetDir -> SKIP) []
   (s.InPUpInRdDig -> ((s.OutPUpOutErr -> SKIP) |~|
    (s.OutPUpOutDig -> SKIP))) []
   (s.InPUpInWrDig -> SKIP));
  SPEC_CT_UPORT(s))
```

The SPEC_CT_UPORT specification is how we expect a server implementation to behave (e.g. the code in Figure 3). That is, loop continuously processing requests from a client. The majority of choices in the above specification are *external* ('[]'), i.e. made by the 'client' process. The single internal choice ('|~|') is made by the server, between data or error output.

The corresponding behaviour of a 'client' process is similarly described, against which particular implementations are checked. For the user-port protocol 'CT.UPORT', implementations will typically be application-level processes or other (higher level) drivers and services:

```
SPEC_CLI_CT_UPORT(c) =
  (((c.InPUpInGetInf -> c.OutPUpOutInf -> SKIP) |~|
   (c.InPUpInSetDir -> SKIP) |~|
   (c.InPUpInRdDig -> ((c.OutPUpOutDig -> SKIP) []
    (c.OutPUpOutErr -> SKIP))) |~|
   (c.InPUpInWrDig -> SKIP));
  SPEC_CLI_CT_UPORT(c))
```

In contrast with the server model, the majority of choices here are internal, i.e. the client decides what action it performs on the server. The single external choice indicates that the client must be prepared to accept either of the specified responses from the server. Although not encountered here, both the client and the server can make *external* choices over the same events safely. The same is not true where both make *internal* choices over the same events (the client could choose one event, whilst the server chooses another, leading to deadlock). Algebraically, external choice is a refinement of internal choice, because it is more deterministic.

3.2 Model checks

To ensure that behavioural models for particular channel-types are consistent, the respective client and server models are composed in parallel. This must currently be done by hand, as it requires some additional knowledge regarding connectivity between the models:

```
channel c : CTPROT_CT_UPORT
SYSTEM =
  (SPEC_CLI_CT_UPORT(c) [|{|c|}|] SPEC_CT_UPORT(c))

assert SYSTEM :[deadlock free]
```

The definition of 'SYSTEM' is where the two models are composed in parallel, synchronising on the set of events referred to by 'c'. The last line in this CSP$_M$ script is intended for *batch mode* checking, instructing FDR to check that the parallel composition of these processes is deadlock free. FDR correctly reports that this system is both deadlock and livelock free. When run interactively, FDR can be used to examine specific examples of deadlock, livelock and non-determinism to determine where faults lie (if any). Deadlock or livelock reported for these composed client-server models would normally indicate an error in the specification.

3.3 Component checks

To ensure the correctness of particular client and server implementations, refinement checks are used. For example, from the following sample client implementation:

```
WHILE TRUE
  SEQ i = 0 FOR nports
    SEQ
      cli[in] ! rd.dig; i; 1      -- read port 'i'
      cli[out] ? CASE
        INT v:
        dig; v                    -- incoming value
          ...  process data
        error                     -- read error
          SKIP
```

the following model is generated:

```
PMYCLIENT_L0(c) =
  (SKIP |~|
   ((c.InPUpInRdDig -> ((c.OutPUpOutDig -> SKIP) []
     (c.OutPUpOutErr -> SKIP))); PMYCLIENT_L0(c)))

PMYCLIENT(c) =
  (PMYCLIENT_L0(c); PMYCLIENT(c))
```

With this model of the implementation, and the previous specification, we create the following refinement checks:

```
channel d : CTPROT_CT_UPORT
THESPEC = SPEC_CLI_CT_UPORT(d)
THEIMPL = PMYCLIENT(d)

assert THESPEC [T= THEIMPL
assert THESPEC [F= THEIMPL
assert THESPEC [FD= THEIMPL
```

The first two assertions, *traces* and *failures* refinements, are reported as correct. The third *failures-divergences* refinement check fails as expected — if the value of 'nports' happened to be zero, the implementation would behave as livelock. In such cases, the counter-examples generated by FDR can be examined to determine where the error, if it is indeed an error, lies. An incorrectly programmed client implementation, e.g. one that does not accept the 'error' response from the server, does not pass the *failures* refinement check, clearly indicated in the counter-examples produced.

By independently checking client and server implementations against the given specifications, and checking those specifications against each other, we can guarantee that the interaction of these particular components does not lead to deadlock or livelock.

4. CONCLUSIONS

We have shown how formal models of occam-π processes may be constructed and used to perform correctness checks. These currently include checking that a process behaves correctly with respect to its environment (not leading to deadlock or livelock) and that our models of interprocess interaction (over strongly typed *channel-bundles*) are themselves correct. In the context of the RMoX OS, this work provides a method for guaranteeing the correct operation of components with respect to the rest of the system, without necessarily having full knowledge about that system. Although this paper shows a trivial example, we have successfully verified properties for several actual implementations.

The benefits of this work to RMoX include providing guarantees about the correct operation of our own and *third-party* components (currently to the extent of their interaction with the rest of the system). While this does not guarantee the overall correct behaviour of the system, including global deadlock freedom, it is an important first step towards this. Although still at an early stage, this work has already proved useful for finding previously undiscovered bugs in existing components (caused by incorrect channel I/O in seldom executed error handling paths). Also, this approach cannot be used to guarantee the correctness of the run-time system [14], currently around 10,000 lines of C code — this run-time has been throughly tested, however.

While more work is required to transparently bring the benefits of these techniques to occam-π developers in general, and RMoX component authors in particular, we have established that the approach is both valid and feasible. The approach is not without its limitations, however. Firstly, to make the checking process feasible, computation-only statements are modelled as *SKIP*. However, run-time errors can (and do) occur in practice (e.g. arithmetic overflow, array-bounds, etc.), which behave as *STOP* (local deadlock). Rather than model each computation as $SKIP \sqcap STOP$ (*might* deadlock), we are investigating language-level *exception handling* constructs. Here, computations would either behave as *SKIP* or the appropriate error-handler. Second, to make the size of the generated models manageable, we do not model the communication of *mobile* channel bundles (i.e. the restructuring of the process network) beyond the fact that a communication occurs. Instead, all possible mobile channel interactions are captured — each distinct channel-end variable is hoisted into the model's parameter list. This is not a particular issue for individual components, but it does limit global checking (deadlock freedom in particular). We are currently investigating other formal analyses that can model mobile channel movement more easily than in CSP [2].

Lastly, the limited data and control flow analyses performed by the current occam-π compiler leads to less deterministic models than we might otherwise like. This is being addressed in the ongoing development of new compilers.

4.1 Related work

Guaranteeing the correct behaviour of software systems is very much an area of active research (and industrial) interest. Within the field of concurrency, approaches such as those employed by the 'Singularity' OS use language extensions to describe *sequences* of interactions between processes [5] (OS components), in a way not dissimilar to the *traces* described here. A mixture of compile-time static checks and run-time state-machine based checks are used. Other language and type-system based approaches such as *session-types* use related ideas [11] (programmatic descriptions of interactions). A number of other systems take similar approaches to OS construction (from processes and communication), partly to take advantage of modern multi-core processors, e.g. Corey [4], Barrelfish [16] and K42/Tornado [7]. Some of the ideas discussed here could potentially be applied to these systems.

For mission-critical systems, where software failure is intolerable, validation and certification through the whole software stack is necessary. This is present in industrial systems such as Integrity RTOS [9]. Although we are a long way from such validation, we are steadily moving in that direction.

5. ACKNOWLEDGEMENTS

This work was funded by EPSRC grants EP/D061822/1 and EP/E049419/1. We would also like to thank the anonymous reviewers for their constructive feedback.

6. REFERENCES

[1] F. Barnes, C. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In J. Broenink and G. Hilderink, editors, *Proceedings of Communicating Process Architectures 2003*. IOS Press, Sept. 2003.

[2] F. R. M. Barnes. Mobile escape analysis for occam-pi. In *CPA 2009*. IOS Press. To Appear.

[3] D. Beckett and P. Welch. A Strict occam Design Tool. In *Proceedings of UK Parallel '96*. Springer-Verlag, July 1996.

[4] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. D. Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[5] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of EuroSys 2006*, Leuven, Belgium, Apr. 2006.

[6] Formal Systems (Europe) Ltd., 3, Alfred Street, Oxford. OX1 4EH, UK. *FDR2 User Manual*, May 2000.

[7] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, Feb. 1999. USENIX.

[8] M. Goldsmith, A. Roscoe, and B. Scott. Denotational Semantics for occam2, Part 2. In *Transputer Communications*, volume 2 (1), pages 25–67. Wiley and Sons Ltd., UK, Mar. 1994.

[9] Green Hills Software Inc. Integrity RTOS. URL: http://www.ghs.com/.

[10] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[11] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP'98*, volume 1381/1998 of *Lecture Notes in Computer Science*. Springer, 1998.

[12] INMOS Limited. *Transputer development system (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[13] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999. ISBN: 0-52165-869-1.

[14] C. G. Ritson, A. T. Sampson, and F. R. M. Barnes. Multicore scheduling for lightweight communicating processes. In *Proceedings of COORDINATION 2009*, volume 5521 of *LNCS*, pages 163–183. Springer, June 2009.

[15] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[16] A. Schüpback, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS) 2008*. ACM, June 2008.

[17] P. Welch and F. Barnes. Communicating mobile processes: introducing occam-pi. In A. Abdallah, C. Jones, and J. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, Apr. 2005.

[18] P. Welch and F. Barnes. A CSP model for mobile channels. In *Proceedings of Communicating Process Architectures 2008*. IOS Press, Sept. 2008.

[19] P. Welch, G. Justo, and C. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In *Proceedings of the 1993 World Transputer Congress*. IOS Press, Netherlands, September 1993.