Generalized Alternation in Shared-Memory Systems

Michael E. Goldsby *

December 15, 2021

Abstract

This paper presents a new algorithm for generalized alternation in shared-memory systems. The algorithm is fully decentralized and contains no timed waits. Included are a proof sketch and a CSPM (machine-readable CSP) model of the algorithm suitable for use with the FDR refinement checker.

1 Introduction

1.1 Alternation

Hoare introduced the alternation construct in his first paper on communicating sequential processes (CSP) [Hoa78]. The alternation is a generalization of the `if..then..else` statement that allows the synchronization of processes and communication among them. It is based on Dijkstra's guarded commands [Dij75] but differs by allowing synchronization and communication as part of the guard.

Under development by Hoare and others, CSP went on to become a full-fledged process algebra [BHR84]. Here we follow the CSP scheme of representing a concurrent computation as a collection of processes that communicate by means of message-passing over synchronous channels. Each process

*michaelegoldsby@gmail.com

contains local variables that it alone may alter, and any global data must be invariant.

An alternation (or alt for short), like an `if..then..else` statement, consists of one or more branches. Each branch contains a *guard* and a *body*. The guard contains an optional boolean expression (the *condition*) and a *synchronization*. The condition may be absent, equivalent to its being true. The synchronization may be an *input*, an *output* or a *skip* (no synchronization).

The CSP algebra allows multi-way events and allows two-way data passage over the channels, but languges inspired by it (`occam, go`) typically allow only one-way communication between a sender and a receiver over a one-way synchronous channel. [1] Here also we follow the convention that a channel is a one-way point-to-point connection between a pair of processes and synchronization must involve one input (repesented by `?`) and one output (represented by `!`). The communication of data may be absent, making the interaction a pure synchronization.

The body contains executable statements and is the only part of the branch that may change the local variables of the process. It consists of assignment statements and procedure calls, and we may also refer to it as "the assignments". Note that the body is the only part of an alt that may change the values of the process's local variables.

An alternation completes when exactly one of its branches completes (and transfers the data if any). If more than one branch is capable of completion, the choice is nondeterministic.

Implementation of alternation is simpler if the guards may contain only inputs (or only outputs), that is, if one side of the synchronization/communication is committed. An alternation construct that allows either input or output in its guards is said to be generalized. Generalized alternation removes the non-essential asymmetry of input-only alternation and allows a simpler expression of certain algorithms.

In addition to channel guards and skip guards, a branch may have a timeout guard. For the sake of simplicity, we restrict ourselves to channel guards in this paper. Also for simplicity's sake, we assume that processes are nonterminating. For similar reasons we omit the details of data communication, as explained in section 2 below.

Figure 1 shows a symbolic representation of an alternation with two branches.

[1] An exception is `occam-pi`, with its barrier synchronization [BWS05].

```
(
   condition1 ; c1?var1 -> ..body1..
[] condition2 ; c2!expr2 -> ..body2..
)
```

Figure 1: An alternation with two branches.

1.2 Background

Alternation is a useful construct even outside the context of CSP. The current work grew out of projects that required writing software for embedded microcontrollers. The initial work used a single processor, serving processes in round-robin fashion and running their branches to completion without interruption, which imposed some delay on interrupt handling. In that setting, it was simple to implement generalized alternation.

When extending the work to multicore systems (and systems with more immediate interrupt handling), the author did not wish to give up the symmetry and convenience provided by generalized alternation, which led to the work presented here.

Section 2 describes the algorithm and section 3 presents a proof sketch for it. Section 4 comments briefly on performance, section 5 lists related work, and section 6 describes possible future work. The appendices contain a CSPM model that the author used with the FDR4 [Gib+14; Gib21; Gib+13] refinement checker to guide development of the algorithm.

2 Description

Figure 2 shows pseudo-code for the algorithm.

```


ProcState = { Alting, Completing, Quiescent, Clearing }

Process = {
        state: ProcState = Alting
        alt: Alternation
```

```
                mutex: Mutex
              }

   Channel = {
                mutex: Mutex
                first, second: Process
              }

   Alternation = {
                    nb: int
                    [nb] branch: Branch
                  }

   Branch = {
                c: Channel
                partner: Process
             }


run_branch(self: Process, partner: Process, b; int) :
                                           (fired: bool) =

   IF self.state = Alting THEN
      IF c.first = nil and c.second = nil THEN
          .. first
          c.first, fired := self, false

      ELSE IF c.first <> nil and c.first <> self THEN
                IF partner.state = Alting THEN
                    ..become second and complete
                    c.second, self.state, partner.state :=
                        self Completing, Quiescent
                    ..do xfr if any
                    fired := true
                    ..run assignments
                ELSE
                    ..this alt completed elsewhere
                    self.state, fired := Clearing, false
```

```
               END
           ELSE
             ..ignore anything else
             fired := false
           END
      END

   ELSE IF state = Quiescent THEN
           ..ignore everything
           fired := false
        END

   ELSE IF state = Completing THEN
           IF c.first = self and second <> nil THEN
               .. first completes
               c.first , c.secod := nil , nil
               self.state , partner , fired :=
                  Clearing , Clearing , true
               ..run assignments
           END
        END
      END

   ELSE ..state = Clearing
         IF c.first = self and c.second = nil THEN
             c.first , fired := nil , false

         ELSE ..ignore anything else
             fired := false
         END
    END

run_proc( self: Process ) =
     c := self.alt.c[b]
     partner := self.alt[b].partner

     proc1 , proc2 := IF self < partner
                          THEN self , partner
```

```
                        ELSE partner, self END

    nb := self.alt.nb
    count: int = 0
    WHILE true
        b: int = 0
        WHILE b < nb
            c.mutex.lock
            proc1.mutex.lock
            proc2.mutex.lock
            fired := run_branch(self, partner, b)
            proc2.mutex.unlock
            proc1.mutex.unlock
            c.mutex.unlock
            b, count := b+1,
                IF not fired THEN count+1 ELSE 0 END
        END
        ..b = nb
        IF self.state = Clearing THEN
            self.state := Alting END
    END
```

Figure 2. Pseudo-code for the generalized alternation algorithm.

We regard a stand-alone input or output to be a degenerate case of alternation, so all interprocess communication and synchronization is performed by alternation. [2] Thus the execution of a process involves the successive execution of a series of alts, possibly interspersed by non-alt, non-communication/non-synchronizaton code..

Also assume for clarity that each process runs on its own processor, whose scheduler calls the `run_proc` method, although nothing in the algoritm need change if several processes were (fairly [Fra86]) scheduled on the same processor.

We can remove the boolean conditions from consideration. The body of a branch can alter the values of the variables on which a guard depends,

[2] Interestingly, any CSP process has an implementation as a single iterated alt [ABC87]. We make no use of that fact in this paper.

changing its truth value. However, since by our assumption a process can install a different alt when it finishes its previous one, changing the values of the guards is equivalent to installing an alt lacking the branches with falsified conditions.

Synchronzation over a channel must involve exactly two processes, a sending process and a receiving process. We term the two processes *complementary* and say that one of the process is the *complementary process* or the *complement* of the other with respect to the channel. We may also simply refer to the complementary process when the channel is understood. We refer to the two complementary processes as *partners* wrt to the channel between them. If the alts of two processes share a channel, we call them *related* processes.

We say a process *selects* a channel if it returns `true` from the `run_branch` procedure when the process calls it for a branch containing the channel. A `true` return from `run_branch` means the branch will execute its body, possibly changing the values of the process's local variables.

The first and second fields of all channels are initially nil, and initially the state of all processes is `Alting`.

Suppose processes P and Q synchronize on a channel c. The sequence of events for such a synchronization is as follows:

- One of the processes, say P, finds c.first = nil. We say that P is first to the channel. P sets c.first = P and carries on executing its branches (see lines 30-33 of Fig. 2).

- P's partner Q discovers that c.first is not nil and is not itself (so it must be Q's partner wrt c) and also that c.second is nil. Provided P's state is `Alting`, it then stores itself in c.second and returns `true`, selecting c. We say that Q is second to the channel and has made an unmatched selection of c. (See lines 35-42 of Fig. 2.) If P's state is not `Alting`, the alt must have completed in another branch, and P sets its state to `Clearing` (see lines 43-45).

  If the branch includes data transfer as well as synchronization, Q carries it out before returning `true`. Both sender and receiver have at this point visited c and can have left sufficient information in c to allow the transfer to occur (source address, destination address and data length.) For simplicity's sake we do not show these details here.

Note that it does not matter which of P and Q is the sender and which is the receiver; except for the actual transfer, the algorithm is symmetric wrt to sender and receiver.

When it returns `true`, we say that Q has completed as second to the channel. Note that the second process to the channel completes first.

Before returning, Q sets its own state to `Quiescent` and P's state to `Completing`.

- Next, P discovers that c.first = itself and c.second is non-nil. In response, it will set both fields to nil and return `true`, selecting c. We say now that P completes as first, and that the selection matches the earlier (unmatched) selection of its partner. [3]

  Before returning, P sets both its own and its partner Q's state to `Clearing`. (See lines 58-66 of Fig. 2)

- In the `Clearing` state, a process visits each branch of its alt (by calling `run_branch`) and if it finds that it is first in the branch's channel, it resets the first field to nil (lines 69-71 of Fig. 2).

3 Proof sketch

The correctness criteria consist of a liveness property and a safety property. [4]

Note that during the execution of a branch, the executing process has exclusive access to itself, its partner and the channel the partners share (lines 91-97 of Fig. 2). Thus it is assured that the process's partner is not executing its `run_branch` procedure. Also note that we always lock the partners' mutexes in the same order to avoid an AB-BA deadlock [Han73] (lines 82-84 of Fig. 2).

[3] Note that an arbitrary interval of time may separate the second process's and the first process's completions, even though these two events together constitute the synchronization of the two processes. In practive, this interval would tend to be short.

[4] Any correctness condition is expressable as the conjunction of a liveness condiiton and a safety condition [Sch97].

3.1 Liveness

L1. If the system contains any complementary pair both of whose partners are in an alt, some alt will eventually complete; that is, for some complementary pair R, S, R will complete as second and S will complete as first.

Let P and Q be partners wrt channel c, and let c appear in both of their alts.

L0.1 If c.first = nil, then one of P, Q will become first to c unless P, Q or one of their partners selects some other channel.

L0.2 If c.first is neither P nor nil and c.second = nil, then P will complete as second to c unless P or Q completes on some other channel.

L0.3 If c.first = P and c.second is not nil, then P will eventually select c and complete as first to c. When it does, it resets c.first and c.second to nil values.

Since each process continually polls its branches (lines 88-100 of Figure 2), L0.1-L0.3 follow directly from the code, and L1 follows from L0.1-L0.3.

3.2 Safety

S1. If a process P completes as second wrt c, the next selection, if any, produced by its complementary process Q wrt c will be of c.

S2. Until Q performs that selection, P will perform no other selection.

That is, selections occur in matched pairs. [5]

3.3 Proof sketch for safety conditions

S1: When P completes as second, it puts its complementary process Q wrt c into the `Completing` state; it leaves that state only when it completes as first to c.

[5] But note that completions of unrelated processes may be interspersed with each other.

S2: When P completes as second, it enters the `Quiescent` state, which it leaves only when Q puts it into the `Clearing` state, which it does only when Q completes as first.

The above considerations, coupled with the observation that a process can only complete (as first or second) when in the Alting state, imply S1 and S2.

4 Performance

Assume that processes P and Q complete their alts for channel c, that P's alt has `m` branches and Q's has `n`, and that P is first to the channel. P must execute at most `m` branches to become first, after which Q must execute at most `n` branches to become second and complete, then again P must execute at most `m` branches to complete as first. Then process P must execute `m` branches and Q must execute `n` branches in order to clear their channels. Thus the algorithm executes at most `3m + 2n` branches to complete the alt. Note, however, that many of the branch executions of P and Q may occur concurrently. It is only when a branch contains a channel c' such that P and Q are partners wrt c' that P and Q cannot proceed concurrently.

The execution of the body of a branch may take any amount of time, but it takes place outside any locks and can therefore proceed concurrently with its partner's execution (and that of of every other process).

The biggest performance concern is the three locks, for c, P, and Q, that the program must acquire in order to run a branch. The author looks forward to making performance studies with an actual implementation (see Section 6.)

5 Related work

Most generalized alternation algorithms in the literature are for systems in which the processes communicate via message passing [BK84; Bag86; Ber80; BHR84; BS83; Dem98; BK92; Sch82; Sil79]. Fujimoto and Feng [FF87] present an algorithm for shared-memory systems with a proof of its correctness. It contains a timed delay and uses interprocessor signals in addition to locks.

It might be possible to use a message-passing algorithm as a model for a shared-memory algorithm by substituting procedure calls for messages. However, the patterns of mutual exclusion required to make the procedure calls atomic would tend to be costly, and most of the cited algorithms have other drawbacks, such as high message or time complexity, or a tendency to deadlock under certain messages patterns. The best candidate for such a conversion appears to be an efficient algorithm in [Dem98].

6 Future work

Presently, the partner processes synchronize at branch resolution. We might try to make the span of the synchronization smaller, possibly by use of the compare-and-swap instruction.

We could perhaps improve the algorithm's power usage by requiring it, after it has visited every branch of its alt, to wait for a signal from a partner before executing its branches again.

We might also try to lessen the number of branches that the first process to the channel must execute in order to complete by letting the second process specify the branch that the first is to execute, in effect "forcing" a branch on it.

We may undertake a formal proof of the algorithm.

We have the intention to make a `C` or `C++` implementation of the algorithm and study its performance.

We might investigate the possibility of using this shared-memory algorithm as a pattern for the development of a message-passing algorithm, although at a message per procedure call, the message complexity does not appear favorable.

References

[ABC87] K. R. Apt, Luc Bougé, and Ph. Clermont. "Two normal form theorems for CSP programs". In: *Information Processing Letters* 26.4 (1987), pp. 165–171.

[Bag86] R. Bagrodia. "A distributed algorithm to implement the generalized alternation command in CSP". In: *The 6th International Conference on Distributed Computeing Systems*. May 1986, pp. 422–427.

[Ber80] A. J. Bernstein. "Output guards and nondeterminism in communicating sequential processes". In: *ToPLaS* 2.2 (Apr. 1980), pp. 234–238.

[BHR84] Stephen Brookes, C. A. R. Hoare, and A. W. Roscoe. "A theory of communicating sequential processes". In: *JACM* 31.3 (July 1984), pp. 560–599.

[BK84] Eklund Back and Kurki-Suonio. *A fair and efficient implementaton of CSP with output guards*. Tech. rep. A-38. Departrment of Computer Science, University of Utah, 1984.

[BK92] Eklund Back and Kurki-Suonio. *A distributed protocol for channel-based communication with choice*. Tech. rep. ECRC-92-16. Muenchen: European computer-industry research center, 1992.

[BS83] G. N. Buckley and A. Silberschatz. "An effective implementaton for the genealized input–output construct of CSP". In: *ToPLaS* 5.2 (Apr. 1983), pp. 223–235.

[BWS05] Frederick R. M. Barnes, Peter H. Welch, and Adam T. Sampson. "Barrier synchronizaton for occam-pi". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2005, Volume 1*. Ed. by Hamid R. Arabnia. Las Vegas, NV, June 2005, pp. 173–179.

[Dem98] Erik D. Demaine. "Protocols for non-deterministic communication over synchronous channels". In: *12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98)*. Orlando, FL: IEEE Computer Society, 1998, pp. 24–30.

[Dij75] E. W. Dijkstra. "Guarded commands, nondeterminism and formal derivation of programs". In: *CACM* 18.8 (Aug. 1975), pp. 453–457.

[FF87] Richard M. Fujimoto and Hwa-chung Feng. *A shared memory algorithm and proof for the alternative construct in CSP*. Tech. rep. UUCS-87-021. Departrment of Computer Science, University of Utah, 1987.

[Fra86] Nissim Francez. *Fairnesss*. Springer Verlag, 1986.

[Gib+13] Thomas Gibson-Robinson et al. *Failures Divergences Refinement (FDR) Version 3*. 2013. URL: `https://www.cs.ox.ac.uk/projects/fdr/`.

[Gib+14] Thomas Gibson-Robinson et al. "FDR3 — A Modern Refinement Checker for CSP". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. 2014, pp. 187–201.

[Gib21] Thomas Gibson-Robinson. *FDR Manual*. 4.2.7. Accessed December, 2021, at https://www.cocotec.io/fdr/fdr-manual.pdf. University of Oxford. May 2021.

[Han73] Per Brinch Hansen. *Operating System Principles*. Englewood, NJ: Prentice-Hall, 1973.

[Hoa78] C. A. R. Hoare. "Communicating sequential processes". In: *CACM* 21.8 (Aug. 1978), pp. 666–677.

[Sch82] F. B. Schneider. "Synchronization in distributed programs". In: *ToPLaS* 4.2 (Apr. 1982), pp. 125–148.

[Sil79] A. Silberschatz. "Communication and synchronization in distributed systems". In: *IEEE Transactions of Software Engineering* SE-5.6 (Nov. 1979), pp. 542–546.

Appendix A

The FDR4 refinement checker allowed the author to find errors when developing the algorithm. In the author's judgement, it saved time and effort, though it is difficult to characterize how much, especially since it was necessary to climb a substantial learning curve in order to use the checker.

This appendix contains a CSPM model of the algorithm for use with FDR4. An included file contains functions that define the process network. The author exploits this separation in a somewhat crude but relatively effective way: the inclusion of a new network file in general causes a rash of trivial compile-time errors, which the author then corrects by hand. The specifications have to be rewritten with some care, however, for each new network configuration.

Since CSP has no notion of fairness, it is necessary in this model to include a scheduler (the SCHED process) to make sure that each CSP process gets a chance to execute.

```
--CHAN and PROC ids
NONE = 0

CHAN_C = 1
CHAN_D = 2
CHAN_E = 3

PROC_P = 1
PROC_Q = 2

--state
ALTING = 1
COMPLETING = 2
QUIESCENT = 3
CLEARING = 4


--CHAN and PROC ids
--channels bet. RUN_BRANCH and CHAN
channel pc, qc, pd, qd, pe, qe, chanerr: ChanOp
```

```
--channels bet. RUN_PROC or RUN_BRANCH and own PROC
channel p_p, q_q, procerr: ProcOp

--channels bet. RUN_BRANCH and partner's PROC
channel p_q, q_p: PartnerOp

--channels bet. RUN_PROC and RUN_BRANCH:
channel p_run, q_run, runerr: RunOp

--channel between SCHED and RUN_PROCs
channel schp, schq, scherr: SchedOp

nametype ProcId = {0..5}
nametype ChanId = {0..4}
nametype BranchNo = {0..3}
nametype ErrNo = {0..7}
nametype ProcState =
          { ALTING, COMPLETING, QUIESCENT, CLEARING }
nametype Position = {2..36}

datatype ChanOp = RD_1ST.ProcId | WR_1ST.ProcId |
                    RD_2ND.ProcId | WR_2ND.ProcId |
                      RD_BR.BranchNo | WR_BR.BranchNo |
                        LOCK_CHAN | UNLOCK_CHAN |
                          C_ERR.ErrNo
datatype ProcOp = WR_CURR_BR.BranchNo | RD_CURR_BR.BranchNo |
                    RD_NB.BranchNo | RD_STATE.ProcState |
                      WR_STATE.ProcState | WR_FIRED.Bool |
                        RD_FIRED.Bool |  LOCK_PROC.ProcId
                          | UNLOCK_PROC | P_ERR.ErrNo
subtype PartnerOp = WR_CURR_BR.BranchNo | WR_STATE.ProcState |
                      RD_STATE.ProcState | LOCK_PROC.ProcId |
                        UNLOCK_PROC
datatype RunOp = CALL.BranchNo | RET.Bool.ChanId
datatype SchedOp = GO | DONE

--topology-dependent functions
```

```
        include "2p3c.csp"

--runs a designated branch of the alternation
RUN_BRANCH(pid) =
(
  --  local variables
  --     pid   proc id of self

   let
      self = pid2proc(pid)
      run = pid2run(pid)
   within
    run.CALL?b

     -> let
           cid = br2cid(pid, b)
           c = ids2chan(pid, cid)
           partner = ids2partner(pid, cid)
         within

           self.RD_STATE?state
            -> c.RD_1ST?first
              -> c.RD_2ND?second

              -> if (state==ALTING) then

                  if ((first==NONE) and (second==NONE)) then
                    --first
                    c.WR_BR!b
                     -> c.WR_1ST!pid
                       -> run.RET!false!cid
                         -> RUN_BRANCH(pid)

                  else if ((first!=NONE) and (first!=pid)
                              and (second==NONE)) then
                    partner.RD_STATE?pstate
                     -> if (pstate==ALTING) then
                          --second completes
```

```
                c.WR_2ND!pid
                  -> c.RD_BR?b1
                    -> partner.WR_STATE!COMPLETING
                      --(do data transfer if any)
                      -> self.WR_STATE!QUIESCENT
                        -> run.RET!true!cid
                          -> RUN_BRANCH(pid)
              else
                --alt completed elsewhere
                self.WR_STATE!CLEARING
                  -> run.RET!false!cid
                    -> RUN_BRANCH(pid)

        else  --ignore anything else
          --procerr.P_ERR!1
            run.RET!false!cid
              -> RUN_BRANCH(pid)

      else if (state==QUIESCENT) then
        run.RET!false!cid
          -> RUN_BRANCH(pid)

      else if (state==COMPLETING) then

        if ((first==pid) and (second!=NONE)) then
          --first completes
          c.WR_1ST!NONE
            -> c.WR_2ND!NONE
              -> self.WR_STATE!CLEARING
                -> run.RET!true!cid
                  -> partner.WR_STATE!CLEARING
                    --(run assignments)
                    -> RUN_BRANCH(pid)

          else --ignore everytning else
            run.RET!false!cid
              -> RUN_BRANCH(pid)

```

```
             else --state = Clearing
               if ((first==pid) and (second==NONE)) then
                 c.WR_1ST!NONE
                  -> run.RET!false!cid
                   -> RUN_BRANCH(pid)

               --ignore any other combination
               else
                 run.RET!false!cid
                  -> RUN_BRANCH(pid)

)



RUN_PROC( pid, b, count, first, state ) =
(
    -- parameters:
    --    pid          proc id
    -- local variables:
    --    b          branch number in 0..nb-1
    --    count      progress counter for branch processing
    --    first      start of execution for a process
    --    state      state (ALTING, COMPLETING or CLEARING)
    if (first) then
      let
        sch = pid2sch(pid)
      within
        sch.GO
         -> RUN_PROC( pid, b, count, false, state )
    else
      let
        self = pid2proc(pid)
        cid = br2cid(pid,b)
        c = ids2chan(pid,cid)
        partid = ids2partid(pid,cid)
        partner = ids2partner(pid,cid)
        (proc1, proc2) = if pid < partid
```

```
                          then (self,partner)
                          else (partner,self)
         nb = pid2nb(pid)
         run = pid2run(pid)
       within
         c.LOCK_CHAN
          -> proc1.LOCK_PROC.pid
           -> proc2.LOCK_PROC.pid
             -> if count < nb then
                 run.CALL!b
                  --between CALL and RET, RUN_BRANCH
                  --has the use of x_y, x = pid,
                  --y in { y | y a partner of x } U {x}
                  -> run.RET?fired?cid
                     -> proc2.UNLOCK_PROC
                      -> proc1.UNLOCK_PROC
                       -> c.UNLOCK_CHAN
                         -> if (fired) then
                              --clear all branches
                              RUN_PROC( pid,
                                ((b+1)%nb), 0, first, state )
                            else
                             RUN_PROC( pid,
                                ((b+1)%nb), count+1, first, state )
                  else --count = nb
                    -- branches exhausted
                    self.RD_STATE?state
                     -> if state==CLEARING then
                           self.WR_STATE!ALTING
                            -> proc2.UNLOCK_PROC
                             -> proc1.UNLOCK_PROC
                              -> c.UNLOCK_CHAN
                               -> RUN_PROC( pid,
                                    b, 0, true, ALTING )
                        else --keep Alting
                           proc2.UNLOCK_PROC
                            -> proc1.UNLOCK_PROC
                             -> c.UNLOCK_CHAN
```

```
                                -> RUN_PROC( pid ,
                                     b, 0, true , state )

)

PROC( pid , nb, b, fired , state , locked ) =
(
    --parameters:
    --   pid      id of this process
    --   nb       # of branches in process's alt
    --local variables:
    -- b        branch number for completion
    --   state    Alting or Clearing
    --   fired    true if most recently-run branch fired
    --   locked   true if locked

  let
    p = pid2proc(pid)
    pp = pid2partners(pid)
  within

    (not locked) & p.LOCK_PROC?id
       -> PROC( pid , nb, b, fired , state , true )
  []
    (locked) & p.UNLOCK_PROC
      -> PROC( pid , nb, b, fired , state , false )
  []
    p.WR_CURR_BR?newb
      -> PROC( pid , nb, newb, fired , state , locked )
  []
    p.RD_CURR_BR!b
      -> PROC( pid , nb, b, fired , state , locked )
  []
    p.RD_NB!nb
      -> PROC( pid , nb, b, fired , state , locked )
  []
    p.WR_STATE?newstate
      -> PROC( pid , nb, b, fired , newstate , locked )
```

```
    []
      p.RD_STATE!state
        -> PROC( pid, nb, b, fired, state, locked )
    []
      p.WR_FIRED?newfired
        -> PROC( pid, nb, b, newfired, state, locked )
    []
      p.RD_FIRED!fired
        -> PROC( pid, nb, b, fired, state, locked )
    []
        [] c: pp @ c.WR_CURR_BR?newb
            -> PROC( pid, nb, newb, fired, state, locked )
    []
        [] c: pp @ (not locked) & c.LOCK_PROC?id
            -> PROC( pid, nb, b, fired, state, true )
    []
        [] c: pp @ (locked) & c.UNLOCK_PROC
            -> PROC( pid, nb, b, fired, state, false )
    []
        [] c: pp @ c.RD_STATE!state
            -> PROC( pid, nb, b, fired, state, locked )
    []
        [] c: pp @ c.WR_STATE?newstate
            -> PROC( pid, nb, b, fired, newstate, locked )


)

CHAN( cid, first, second, b, locked ) =
(
     --parameters:
     --    cid:          id of this channel
     --                  (CHAN_C or CHAN_D, eg)
     --local variables:
     --    first         first process to the channel
     --    second        second ...
     --    b             branch number for first
     --    locked        chan locked when true
```

```
let
  (c1,c2) = cid2procs(cid)
within

  (not locked) & c1.LOCK_CHAN
    -> CHAN( cid, first, second, b, true )
[]
  (locked) & c1.UNLOCK_CHAN
    -> CHAN( cid, first, second, b, false )
[]
  c1.WR_1ST?newfirst
    -> CHAN( cid, newfirst, second, b, locked )
[]
  c1.RD_1ST!first
    -> CHAN( cid, first, second, b, locked )
[]
  c1.WR_2ND?newsecond
    -> CHAN( cid, first, newsecond, b, locked )
[]
  c1.RD_2ND!second
    -> CHAN( cid, first, second, b, locked )
[]
  c1.WR_BR?newb
    -> CHAN( cid, first, second, newb, locked )
[]
  c1.RD_BR!b
    -> CHAN( cid, first, second, b, locked )

[]

  (not locked) & c2.LOCK_CHAN
    -> CHAN( cid, first, second, b, true )
[]
  (locked) & c2.UNLOCK_CHAN
    -> CHAN( cid, first, second, b, false )
[]
  c2.WR_1ST?newfirst
    -> CHAN( cid, newfirst, second, b, locked )
```

```
    []
      c2.RD_1ST!first
        -> CHAN( cid, first, second, b, locked )
    []
      c2.WR_2ND?newsecond
        -> CHAN( cid, first, newsecond, b, locked )
    []
      c2.RD_2ND!second
        -> CHAN( cid, first, second, b, locked )
    []
      c2.WR_BR?newb
        -> CHAN( cid, first, second, newb, locked )
    []
      c2.RD_BR!b
        -> CHAN( cid, first, second, b, locked )

  )
          {-
          --sequential scheduler:
          --let each process execute in turn until
          --it relinquishes
          SCHED( sch ) =
          (
            --parameters:
            --  sch       list of channels to RUN_PROC processes

              let
                ch = head(sch)
              within
                ch.GO
                 -> ch.DONE
                  -> SCHED( tail(sch)^<ch> )
          )
          -}

--let all process execute together
SCHED( schp, schq, pgo, qgo ) =
(
```

```
         pgo & schp.GO -> SCHED( schp, schq, false, true )
        []
         qgo & schq.GO -> SCHED( schp, schq, true, false )

)
--start all RUN_PROC processes and let each one execute
--according to the signals sent by the scheduler process

RUN() = ( ||| pid: all_pids() @
           ( RUN_PROC( pid, 0, 0, true, ALTING )
               [| run_events() |]
           RUN_BRANCH( pid ) ) )

PROCS() = ( ||| pid: all_pids() @
             PROC( pid,
               pid2nb(pid), 0, false, ALTING, false ) )

CHANS() = ( ||| cid: all_cids() @
               CHAN( cid, NONE, NONE, 0, false ) )

RUN_ALL()  = ( ( RUN() [| proc_events() |] PROCS() )
                 [| chan_events() |] CHANS()
              )
                 [| sched_events() |]
              SCHED( schp, schq, true, false )


SYSTEM = RUN_ALL()  \ Hidden
          ----



SPEC =
    (
        (
            ( p_run.RET.True.CHAN_C -> q_run.RET.True.CHAN_C
                -> SPEC )
               |~|                    --CHAN_C (in either order)
            ( q_run.RET.True.CHAN_C -> p_run.RET.True.CHAN_C
```

```
                    -> SPEC )
        )
        |~|
        (
            ( p_run.RET.True.CHAN_D -> q_run.RET.True.CHAN_D
                -> SPEC )
               |~|                  --CHAN_D (in either order)
            ( q_run.RET.True.CHAN_D -> p_run.RET.True.CHAN_D
                -> SPEC )
        )
        |~|
        (
            ( p_run.RET.True.CHAN_E -> q_run.RET.True.CHAN_E
               -> SPEC )
               |~|                  --CHAN_E (in either order)
            ( q_run.RET.True.CHAN_E -> p_run.RET.True.CHAN_E
               -> SPEC )
        )
    )

assert SYSTEM :[divergence-free]
  :[partial order reduce]
assert RUN_ALL() :[deadlock free [F]]
  :[partial order reduce]
assert SPEC [T= SYSTEM
          :[partial order reduce]
```

Appendix B

Below are the topology-defining functions for a simple network with two processes that share three channels.

File 2p3c.csp

```
{-          C
          /    \
        /        \
      /            \
   [P]---D---[Q]
      \            /
        \        /
          \    /
            E
-}


--topology-dependent functions
--
-- pid, cid  -> channel to CHAN
ids2chan(pid, cid) =
     if ((pid==PROC_P) and (cid==CHAN_C)) then pc else
     if ((pid==PROC_P) and (cid==CHAN_D)) then pd else
     if ((pid==PROC_P) and (cid==CHAN_E)) then pe else
     if ((pid==PROC_Q) and (cid==CHAN_C)) then qc else
     if ((pid==PROC_Q) and (cid==CHAN_D)) then qd else
     if ((pid==PROC_Q) and (cid==CHAN_E)) then qe else
     chanerr

-- pid -> channel between PROC and RUN_BRANCH
pid2proc(pid) = if (pid==PROC_P) then p_p else
                if (pid==PROC_Q) then q_q else
                procerr

--pid, br -> cid (id of channel
```

```
--                associated with a process's branch)
br2cid(pid, b) =
     if ((pid==PROC_P) and (b==0)) then CHAN_C else
     if ((pid==PROC_P) and (b==1)) then CHAN_D else
     if ((pid==PROC_P) and (b==2)) then CHAN_E else
     if ((pid==PROC_Q) and (b==0)) then CHAN_C else
     if ((pid==PROC_Q) and (b==1)) then CHAN_D else
     if ((pid==PROC_Q) and (b==2)) then CHAN_E else
     NONE

--cid to (channel,channel) = channels bet. CHAN and its PROCs
cid2procs(cid) = if (cid==CHAN_C) then (pc,qc) else
                 if (cid==CHAN_D) then (pd,qd) else
                 if (cid==CHAN_E) then (pe,qe) else
                 (chanerr,chanerr)

--pid -> set of channels to partners' PROCs
pid2partners(pid) = if (pid==PROC_P) then { q_p } else
                    if (pid==PROC_Q) then { p_q } else
                    { }

--pid, cid -> channel between RUN_BRANCH and partner's PROC
ids2partner(pid, cid) =
     if ((pid==PROC_P) and (cid==CHAN_C)) then p_q else
     if ((pid==PROC_P) and (cid==CHAN_D)) then p_q else
     if ((pid==PROC_P) and (cid==CHAN_E)) then p_q else
     if ((pid==PROC_Q) and (cid==CHAN_C)) then q_p else
     if ((pid==PROC_Q) and (cid==CHAN_D)) then q_p else
     if ((pid==PROC_Q) and (cid==CHAN_E)) then q_p else
     procerr

--pid, cid -> id of partner
ids2partid(pid, cid) =
     if ((pid==PROC_P) and (cid==CHAN_C)) then PROC_Q else
     if ((pid==PROC_P) and (cid==CHAN_D)) then PROC_Q else
     if ((pid==PROC_P) and (cid==CHAN_E)) then PROC_Q else
     if ((pid==PROC_Q) and (cid==CHAN_C)) then PROC_P else
     if ((pid==PROC_Q) and (cid==CHAN_D)) then PROC_P else
```

```
    if ((pid==PROC_Q) and (cid==CHAN_E)) then PROC_P else
    NONE

--pid -> nr of branches
pid2nb(pid) =
    if (pid==PROC_P) then 3 else
    if (pid==PROC_Q) then 3 else
    0

pid2run(pid) = if (pid==PROC_P) then p_run else
               if (pid==PROC_Q) then q_run else
               runerr


pid2sch(pid) =  if (pid==PROC_P) then schp else
                if (pid==PROC_Q) then schq else
                scherr

all_pids()  = { PROC_P, PROC_Q }

all_cids() = { CHAN_C, CHAN_D, CHAN_E }

chan_events() = {| pc, qc, pd, qd, pe, qe |}

proc_events() = {| p_p, q_q, p_q, q_p |}

run_events() = {| p_run, q_run |}

sched_list() = < schp, schq >

sched_events() = {| schp, schq |}

----

Events = {|
            pc, qc, pd, qd, pe, qe,
              p_p, q_q, p_q, q_p, p_run, q_run,
                schp, schq
```

```
            |}

Hidden = diff( union(Events, {| report |} ),
                  {| p_run.RET.true, q_run.RET.true |} )
```