A problem in concurrency [25]

Problem Definition

Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa (otherwise, he may **never** get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise ... This could also explain the quickness in their delivering of presents, since the reindeer can't wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

A Solution

The solution that has worked best over the years, and also appears to be the simplest, is written using C statements and pseudo-code. (Constants are also used in case the number of reindeer were to change, or if the group size of "solution-seeking" elves is modified.) Basically, the reindeer arrive, update the count of how many have arrived, and the last one wakes up Santa. An elf, upon discovering a problem, attempts to modify the count for the number of elves with a problem and either: waits outside Santa's shop if he/she is the first or second such elf; knocks on the door and wakes up Santa if that elf is the third one; or waits in the elves' shop until the elves currently with Santa start coming back. (The code for this solution can be found in the Appendix.)

The solution with semaphores takes about 2 pages of C code [25]!

Smarter solutions to Santa's problem

Better with Erlang...yet, 43 LoC

```
-module(santa).
-author('ok@cs.otago.ac.nz'). % Richard A. O'Keefe
-export([start/0]).

worker(Secretary, Message) ->
    receive after random:uniform(1000) -> ok end,  % random delay
    Secretary ! self(),                            % send my PID to the secretary
    Gate_Keeper = receive X -> X end,              % await permission to enter
    io:put_chars(Message),                         % do my action
    Gate_Keeper ! {leave,self()},                  % tell the gate-keeper I'm done
    worker(Secretary, Message).                    % do it all again

secretary(Santa, Species, Count) ->
    secretary_loop(Count, [], {Santa,Species,Count}).

secretary_loop(0, Group, {Santa,Species,Count}) ->
    Santa ! {Species,Group},
    secretary(Santa, Species, Count);
secretary_loop(N, Group, State) ->
    receive PID ->
        secretary_loop(N-1, [PID|Group], State)
    end.

santa() ->
    {Species,Group} =
        receive                                    % first pick up a reindeer group
            {reindeer,G} -> {reindeer,G}           % if there is one, otherwise
        after 0 ->
                receive                            % wait for reindeer or elves,
                    {reindeer,G} -> {reindeer,G}
                  ; {elves,G}    -> {elves,G}
                end                                % whichever turns up first.
        end,
    case Species
      of reindeer -> io:put_chars("Ho, ho, ho!  Let's deliver toys!\n")
       ; elves    -> io:put_chars("Ho, ho, ho!  Let's meet in the study!\n")
    end,
    [PID ! self() || PID <- Group],        % tell them all to enter
    [receive {leave,PID} -> ok end         % wait for each of them to leave
        || PID <- Group],
    santa().

spawn_worker(Secretary, Before, I, After) ->
    Message = Before ++ integer_to_list(I) ++ After,
    spawn(fun () -> worker(Secretary, Message) end).

start() ->
    Santa = spawn(fun () -> santa() end),
    Robin = spawn(fun () -> secretary(Santa, reindeer, 9) end),
    Edna  = spawn(fun () -> secretary(Santa, elves,    3) end),
    [spawn_worker(Robin, "Reindeer ", I, " delivering toys.\n")
     || I <- lists:seq(1, 9)],
```

Smarter solutions to Santa's problem

Better with Erlang...yet, 43 LoC

```
-module(santa).
-author('ok@cs.otago.ac.nz'). % Richard A. O'Keefe
-export([start/0]).

worker(Secretary, Message) ->
    receive after random:uniform(1000) -> ok end,  % random delay
    Secretary ! self(),                            % send my PID to the secretary
    Gate_Keeper = receive X -> X end,              % await permission to enter
    io:put_chars(Message),                         % do my action
    Gate_Keeper ! {leave,self()},                  % tell the gate-keeper I'm done
    worker(Secretary, Message).                    % do it all again

secretary(Santa, Species, Count) ->
    secretary_loop(Count, [], {Santa,Species,Count}).

secretary_loop(0, Group, {Santa,Species,Count}) ->
    Santa ! {Species,Group},
    secretary(Santa, Species, Count);
secretary_loop(N, Group, State) ->
    receive PID ->
        secretary_loop(N-1, [PID|Group], State)
    end.

santa() ->
    {Species,Group} =
        receive                                    % first pick up a reindeer group
            {reindeer,G} -> {reindeer,G}           % if there is one, otherwise
        after 0 ->
                receive                            % wait for reindeer or elves,
                    {reindeer,G} -> {reindeer,G}
                  ; {elves,G}    -> {elves,G}
                end                                % whichever turns up first.
        end,
    case Species
      of reindeer -> io:put_chars("Ho, ho, ho!  Let's deliver toys!\n")
       ; elves    -> io:put_chars("Ho, ho, ho!  Let's meet in the study!\n")
    end,
    [PID ! self() || PID <- Group],        % tell them all to enter
    [receive {leave,PID} -> ok end         % wait for each of them to leave
        || PID <- Group],
    santa().

spawn_worker(Secretary, Before, I, After) ->
    Message = Before ++ integer_to_list(I) ++ After,
    spawn(fun () -> worker(Secretary, Message) end).

start() ->
    Santa = spawn(fun () -> santa() end),
    Robin = spawn(fun () -> secretary(Santa, reindeer, 9) end),
    Edna  = spawn(fun () -> secretary(Santa, elves,    3) end),
    [spawn_worker(Robin, "Reindeer ", I, " delivering toys.\n")
     || I <- lists:seq(1, 9)],
```

```
%   This is an Erlang solution to "The Santa Claus problem",
%   as discussed by Simon Peyton Jones (with a Haskell solution using
%   Software Transactional Memory) in "Beautiful code".
%   He quotes J.A.Trono "A new exercise in concurrency", SIGCSE 26:8-10, 199
  ...
%   The Haskell version is 77 SLOC of complex code.
%   The Erlang  version is 43 SLOC of straightforward code.
```

Smarter solutions to Santa's problem

Better with Erlang...yet, 43 LoC

```
-module(santa).
-author('ok@cs.otago.ac.nz'). % Richard A. O'Keefe
-export([start/0]).

worker(Secretary, Message) ->
    receive after random:uniform(1000) -> ok end,  % random delay
    Secretary ! self(),                            % send my PID to the secretary
    Gate_Keeper = receive X -> X end,              % await permission to enter
    io:put_chars(Message),                         % do my action
    Gate_Keeper ! {leave,self()},                  % tell the gate-keeper I'm done
    worker(Secretary, Message).                    % do it all again

secretary(Santa, Species, Count) ->
    secretary_loop(Count, [], {Santa,Species,Count}).

secretary_loop(0, Group, {Santa,Species,Count}) ->
    Santa ! {Species,Group},
    secretary(Santa, Species, Count);
secretary_loop(N, Group, State) ->
    receive PID ->
        secretary_loop(N-1, [PID|Group], State)
    end.

santa() ->
    {Species,Group} =
        receive                                    % first pick up a reindeer group
            {reindeer,G} -> {reindeer,G}           % if there is one, otherwise
        after 0 ->
                receive                            % wait for reindeer or elves,
                    {reindeer,G} -> {reindeer,G}
                  ; {elves,G}    -> {elves,G}
                end                                % whichever turns up first.
        end,
    case Species
      of reindeer -> io:put_chars("Ho, ho, ho!  Let's deliver toys!\n")
       ; elves    -> io:put_chars("Ho, ho, ho!  Let's meet in the study!\n")
    end,
    [PID ! self() || PID <- Group],        % tell them all to enter
    [receive {leave,PID} -> ok end         % wait for each of them to leave
        || PID <- Group],
    santa().

spawn_worker(Secretary, Before, I, After) ->
    Message = Before ++ integer_to_list(I) ++ After,
    spawn(fun () -> worker(Secretary, Message) end).

start() ->
    Santa = spawn(fun () -> santa() end),
    Robin = spawn(fun () -> secretary(Santa, reindeer, 9) end),
    Edna  = spawn(fun () -> secretary(Santa, elves,    3) end),
    [spawn_worker(Robin, "Reindeer ", I, " delivering toys.\n")
     || I <- lists:seq(1, 9)],
```

```
%   This is an Erlang solution to "The Santa Claus problem",
%   as discussed by Simon Peyton Jones (with a Haskell solution using
%   Software Transactional Memory) in "Beautiful code".
%   He quotes J.A.Trono "A new exercise in concurrency", SIGCSE 26:8-10, 199
  ...
%   The Haskell version is 77 SLOC of complex code.
%   The Erlang  version is 43 SLOC of straightforward code.
```

Even more straighforward

```
receive
        {reindeer, Pid1} and {reindeer, Pid2} and {reindeer, Pid3} and
        {reindeer, Pid4} and {reindeer, Pid5} and {reindeer, Pid6} and
        {reindeer, Pid7} and {reindeer, Pid8} and {reindeer, Pid9} ->
                io:format("Ho, ho, ho! Let's deliver presents!~n"),
                [Pid1, Pid2, Pid3, Pid4, Pid5, Pid6, Pid7, Pid8, Pid9];
        {elf, Pid1} and {elf, Pid2} and {elf, Pid3} ->
                io:format("Ho, ho, ho! Let's discuss R&D possibilities!~n"),
                [Pid1, Pid2, Pid3]
end
```

with JErlang [22]

A running example

```
def monitor() = Actor[Event, Unit] {                        // gets Event and (possibly) returns Unit
    receive { (self: ActorRef[Event]) => {
        case ( Fault(_, id1, _, ts1),
               Fix(_, id2, ts2) )                           // Acca/Pekko cannot handle multiple msgs
                  if id1 == id2 =>
            updateMaintenanceStats(ts1, ts2)
            Continue
        case ( Fault(mid, id1, descr, ts1),
               Fault(_, id2, _, ts2),
               Fix(_, id3, ts3) )
                  if id2 == id3 && ts2 - ts1 > T =>         // assume T > 0
            updateMaintenanceStats(ts2, ts3)
            log(s"...")
            self ! DelayedFault(mid, id1, descr, ts1)       // For later processing
            Continue
        case ( DelayedFault(_, id1, _, ts1),
               Fix(_, id2, ts2) )
                  if id1 == id2 =>
            updateMaintenanceStats(ts1, ts2)
            Continue
    }
  }
}
```

Matching join patterns

```
case ( Fault(_, id1, _, ts1),
       Fix(_, id2, ts2) )
                  if id1 == id2 => ...

case ( Fault(mid, id1, descr, ts1),
       Fault(_, id2, _, ts2),
       Fix(_, id3, ts3) )
         if id2 == id3 && ts2 - ts1 > T => ...
```

Let the mailbox of the monitor actor get

Fault(_, 1, _, 10:35)	Fault(_, 2, _, 10:40)	Fault(_, 3, _, 10:55)	Fix(_, 3, _, 11:00)	···

Exercise

Which are the matching sequences?

Choosing patterns

Choosing patterns

If more matchings are possible, how do we pick one?

Choosing patterns

If more matchings are possible, how do we pick one?
Usual solutions

- non-deterministic selection
- longest matching sequence

Fair join patterns [13, 14, 16]: take the "oldest" sequence

- using a length-biased lexicographic order on positions in the mailbox
- avoid leaving messages indefinitely in the mailbox

Choosing patterns

If more matchings are possible, how do we pick one?
Usual solutions

- non-deterministic selection
- longest matching sequence

Fair join patterns [13, 14, 16]: take the "oldest" sequence

- using a length-biased lexicographic order on positions in the mailbox
- avoid leaving messages indefinitely in the mailbox

Exercise

Find the fair join pattern for

```
case ( Fault(id1, ts1), Fix(id2) ) if id1 == id2 => ...
```

when the mailbox becomes

```
Fix(3)
```

Choosing patterns

If more matchings are possible, how do we pick one?
Usual solutions

- non-deterministic selection
- longest matching sequence

Fair join patterns [13, 14, 16]: take the "oldest" s

- using a length-biased lexicographic order on positi
- avoid leaving messages indefinitely in the mailbox

a single message doesn't match any patterns

so we've to wait for a new message

Note: messages arrive one at a time and search must be restarted each time!

Exercise

Find the fair join pattern for

```
case ( Fault(id1, ts1), Fix(id2) ) if id1 == id2 => ...
```

when the mailbox becomes

`Fix`(3)	`Fault`(1, 10:35)

Choosing patterns

If more matchings are possible, how do we pick one?
Usual solutions

- non-deterministic selection
- longest matching sequence

the next message doesn't satisfy the condition

Fair join patterns [13, 14, 16]: take the "oldest" sequence

- using a length-biased lexicographic order on positions in the mailbox
- avoid leaving messages indefinitely in the mailbox

Exercise

Find the fair join pattern for

```
case ( Fault(id1, ts1), Fix(id2) ) if id1 == id2 => ...
```

when the mailbox becomes

`Fix`(3)	`Fault`(1, 10:35)	`Fault`(2, 10:36)

Choosing patterns

If more matchings are possible, how do we pick one?
Usual solutions

- non-deterministic selection
- longest matching sequence

the next message doesn't satisfy the condition

Fair join patterns [13, 14, 16]: take the "oldest" sequence

- using a length-biased lexicographic order on positions in the mailbox
- avoid leaving messages indefinitely in the mailbox

Exercise

Find the fair join pattern for

```
case ( Fault(id1, ts1), Fix(id2) ) if id1 == id2 => ...
```

when the mailbox becomes

`Fix`(3)	`Fault`(1, 10:35)	`Fault`(2, 10:36)	`Fault`(3, 10:37)

Choosing patterns

If more matchings are possible, how do we pick one?
Usual solutions

- non-deterministic selection
- longest matching sequence

finally we've a match

Fair join patterns [13, 14, 16]: take the "oldest" sequence

- using a length-biased lexicographic order on positions in the mailbox
- avoid leaving messages indefinitely in the mailbox

Exercise

Find the fair join pattern for

```
case ( Fault(id1, ts1), Fix(id2) ) if id1 == id2 => ...
```

when the mailbox becomes

`Fix`(3)	`Fault`(1, 10:35)	`Fault`(2, 10:36)	`Fault`(3, 10:37)

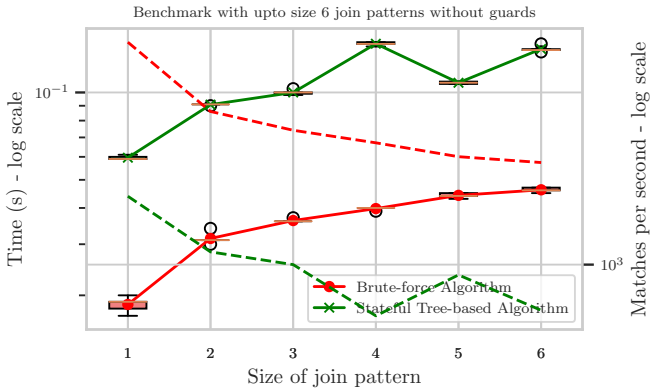
Go to Ayman's slides

Evaluating the JoinActor library

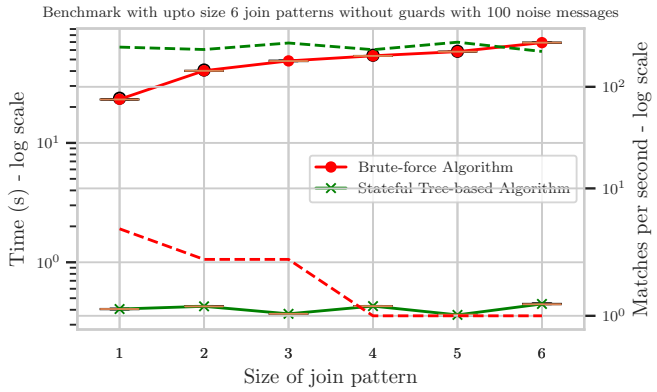
Evaluating the JoinActor library

- Benchmarking pattern size without guards

no noise



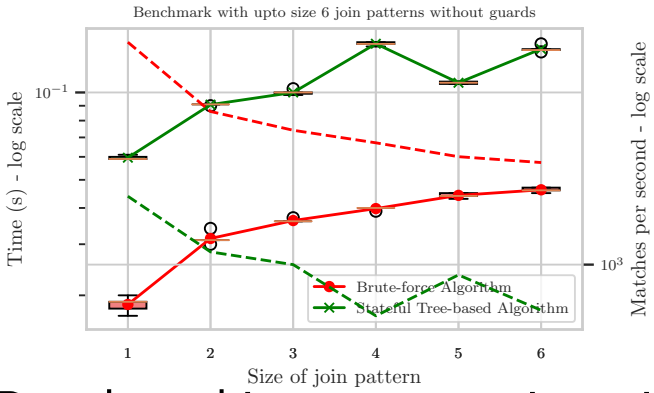
noise



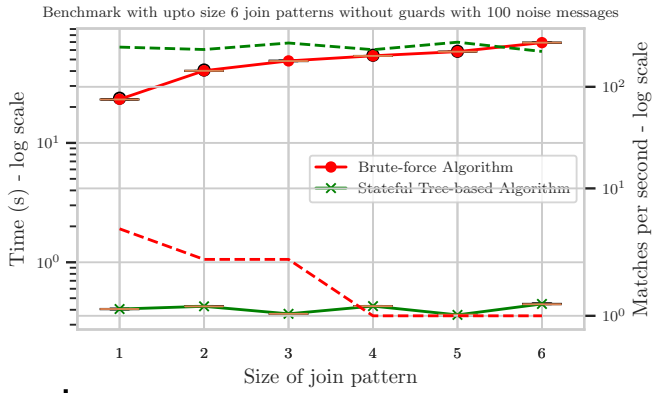
Evaluating the JoinActor library

- Benchmarking pattern size without guards

no noise

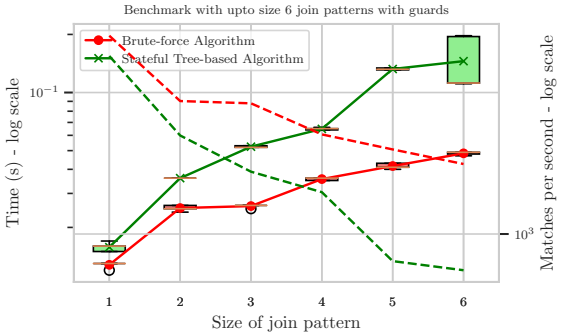


noise

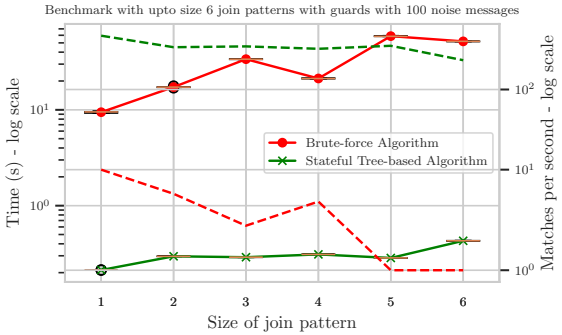


- Benchmarking pattern size with guards

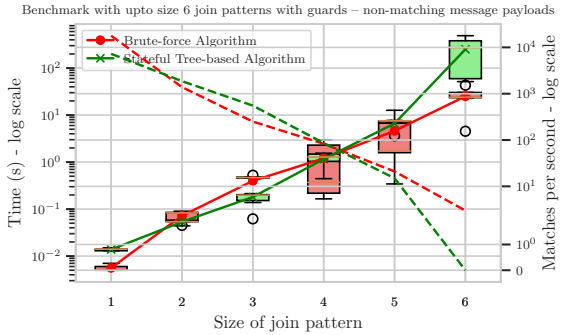
no noise



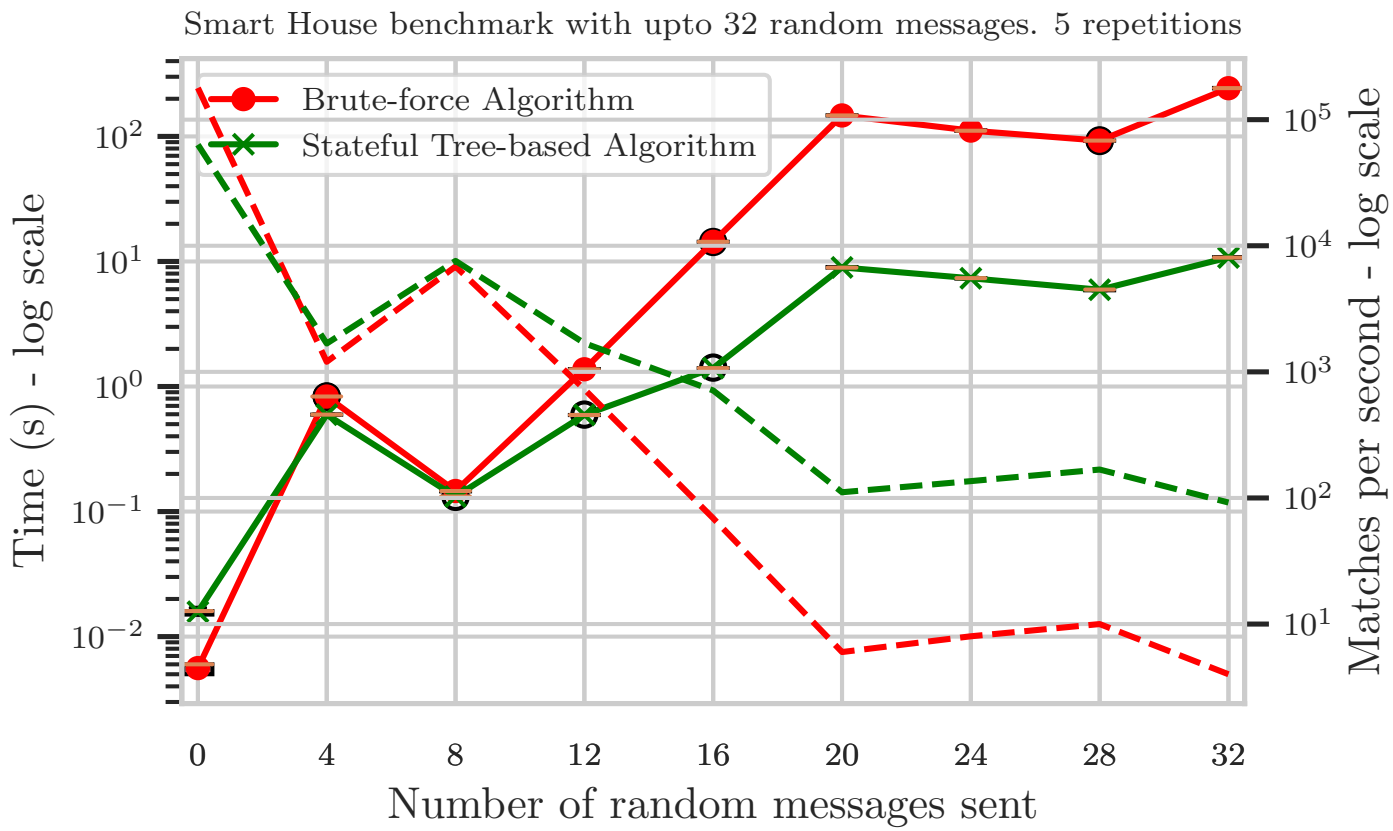
some matching noise



non-matching noise



A non-synthetic benchmark



References I

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*.
MIT Press, Cambridge, MA, USA, 1986.

[2] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines.
JACM, 30(2):323–342, 1983.

[3] Alex Coto, Roberto Guanciale, and Emilio Tuosto. An abstract framework for choreographic testing.
In *ICE*, volume abs/2009.07990, 2020.

[4] Alex Coto, Roberto Guanciale, and Emilio Tuosto. Choreographic development of message-passing applications - A tutorial.
In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2020.

[5] Alex Coto, Roberto Guanciale, and Emilio Tuosto. An abstract framework for choreographic testing.
Journal of Logic and Algebraic Methods in Programming, 123:100712, 2021. Extended version of [3].

[6] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata.
In *ESOP 2012*, pages 194–213, 2012.

[7] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process.
J. ACM, 32(2):374–382, apr 1985.

[8] Cedric Fournet and George Gonthier. The reflexive CHAM and the join-calculus.
In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 1996.

References II

[9] Leonardo Frittelli, Facundo Maldonado, C. Hernán Melgratti, and Emilio Tuosto. A Choreography-Driven Approach to APIs: the OpenDXL Case Study.
In *COORDINATION*, volume 12134 of *LNCS*. Springer, June 2020.

[10] Roberto Guanciale and Emilio Tuosto. An abstract semantics of the global view of choreographies.
In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.*, pages 67–82, 2016.

[11] Roberto Guanciale and Emilio Tuosto. Realisability of pomsets.
Journal of Logic and Algebraic Methods in Programming, 108:69–89, 2019.

[12] Roberto Guanciale and Emilio Tuosto. Pomcho: a tool chain for choreographic design.
Science of Computer Programming, 202:102535, 2021.

[13] Philipp Haller, Ayman Hussein, Hernán C. Melgratti, Alceste Scalas, and Emilio Tuosto. Fair join pattern matching for actors.
In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming, ECOOP 2024, Vienna, Austria, September 16-20, 2024*, volume 313 of *LIPIcs*, pages 17:1–17:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[14] Philipp Haller, Ayman Hussein, Hernán C. Melgratti, Alceste Scalas, and Emilio Tuosto. Fair join pattern matching for actors (artifact).
Dagstuhl Artifacts Ser., 10(2):8:1–8:3, 2024.

[15] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence.
In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.

[16] Ayman Hussein, Philipp Haller, Ioannis Karras, Hernán C. Melgratti, Alceste Scalas, and Emilio Tuosto. Joinactors: A modular library for actors with join patterns.
Art Sci. Eng. Program., 11(1), 2026.

References III

[17] Nickolas Kavantzas, Davide Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0.
http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217. Working Draft 17 December 2004.

[18] Roland Kuhn and Alan Darmasaputra. Behaviorally typed state machines in typescript for heterogeneous swarms.
In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1475–1478. ACM, 2023.

[19] Roland Kuhn, Hernán C. Melgratti, and Emilio Tuosto. Behavioural types for local-first software.
In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[20] Roland Kuhn, Hernán C. Melgratti, and Emilio Tuosto. Behavioural types for local-first software (artifact).
Dagstuhl Artifacts Ser., 9(2):14:1–14:5, 2023.

[21] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From Communicating Machines to Graphical Choreographies.
In *POPL15*, pages 221–232, 2015.

[22] Hubert Plociniczak and Susan Eisenbach. Jerlang: Erlang with joins.
In *Coordination Models and Languages: 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings 12*, pages 61–75. Springer, 2010.

[23] Carlos G. Lopez Pombo, Agustín E. Martinez Suñé, and Emilio Tuosto. A dynamic temporal logic for quality of service in choreographic models.
In Erika Ábrahám, Clemens Dubslaff, and Silvia Lizeth Tapia Tarifa, editors, *Theoretical Aspects of Computing – ICTAC 2023*, pages 119–138. Springer, 2023.

References IV

[24] Carlos López Pombo, Agustín E. Martinez Suñé, and Emilio Tuosto. A dynamic temporal logic for quality of service in choreographic models.
TCS, 1043:115247, 2025.

[25] John A. Trono. A new exercise in concurrency.
SIGCSE Bull., 26(3):8–10, September 1994.

[26] Emilio Tuosto and Roberto Guanciale. Semantics of global view of choreographies.
Journal of Logic and Algebraic Methods in Programming, 95:17–40, 2018. Revised and extended version of [10]. available at http://www.cs.le.ac.uk/people/et52/jlamp-with-proofs.pdf.