Project Hyben:

Formalising Monitors for Distributed Deadlock Detection

Radosław Rowicki[1], Alceste Scalas[1], Adrian Francalanza[2]

[1] Technical University of Denmark
[2] University of Malta

INDEPENDENT RESEARCH FUND DENMARK

TYPES 2025

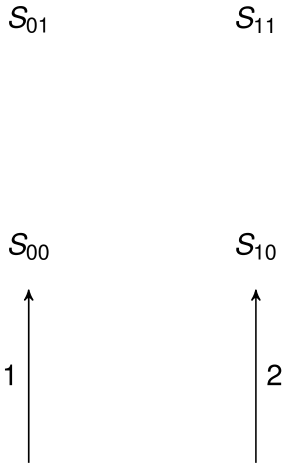
About the project

- Distributed deadlock detection via black-box monitors
- Inspiration: Mitchell / Chandy, Misra, Haas
- Soundness and completeness
- All mechanised in Coq

Deadlock

Deadlock

Deadlock

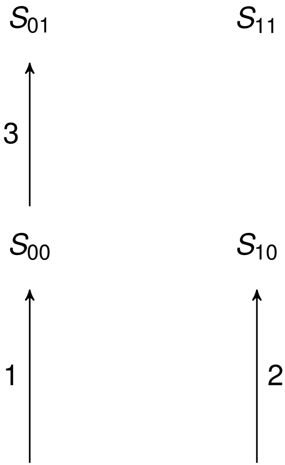


S_{01} S_{11}

S_{00} S_{10}

1 2

Deadlock



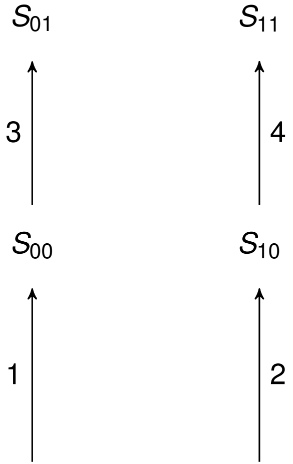
S_{01} S_{11}

3

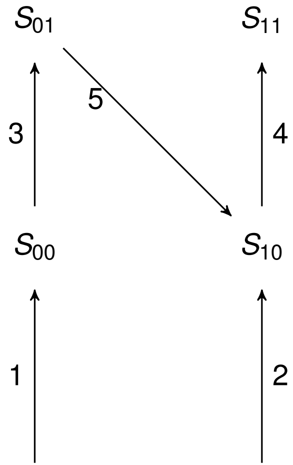
S_{00} S_{10}

1 2

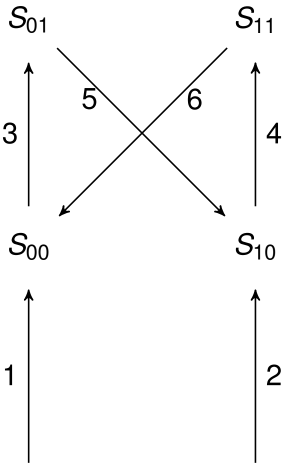
Deadlock



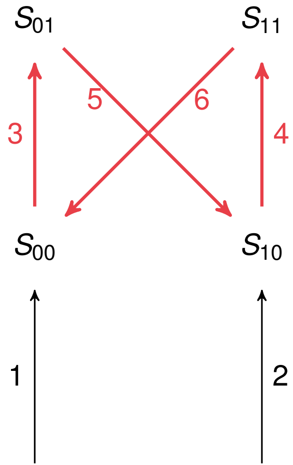
Deadlock



Deadlock



Deadlock



Services and monitored services

$$\texttt{Serv} = [Q_{in} \mid \texttt{Proc} \mid Q_{out}]$$

Services and monitored services

“In-code” state

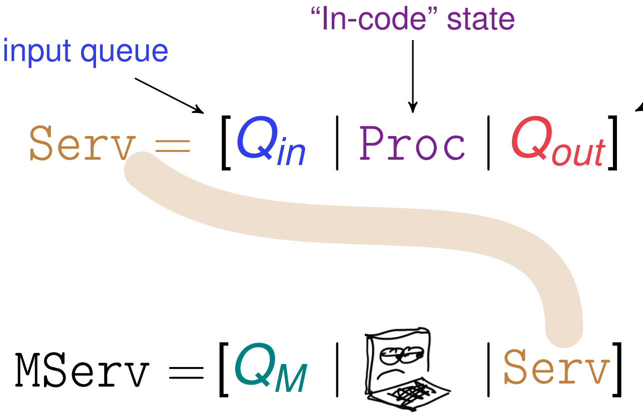
input queue

output queue

$$\mathtt{Serv} = [Q_{in} \mid \mathtt{Proc} \mid Q_{out}]$$

Our case: Erlang `gen_server`

Services and monitored services



“In-code” state

input queue

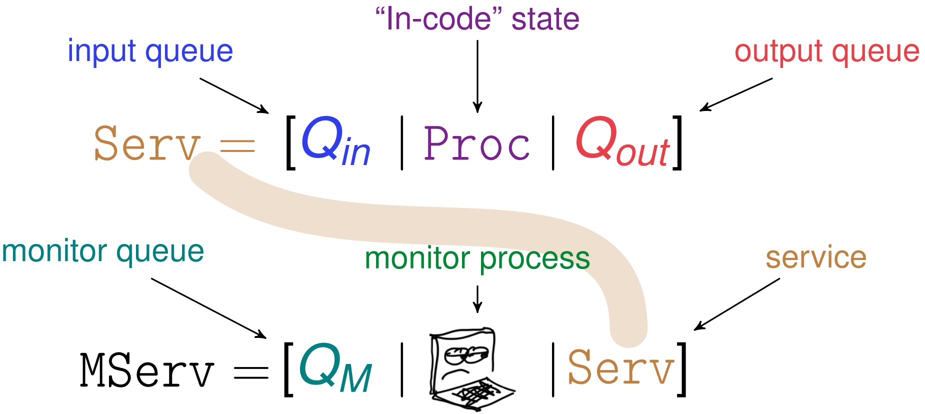
output queue

$$\mathtt{Serv} = [Q_{in} \mid \mathtt{Proc} \mid Q_{out}]$$

$$\mathtt{MServ} = [Q_{M} \mid \quad \mid \mathtt{Serv}]$$

$: \mathtt{state} \times (\mathtt{state} \times \mathtt{msg} \rightarrow \mathtt{state})$

Services and monitored services



“In-code” state

input queue

output queue

$$\mathtt{Serv} = [Q_{in} \mid \mathtt{Proc} \mid Q_{out}]$$

monitor queue

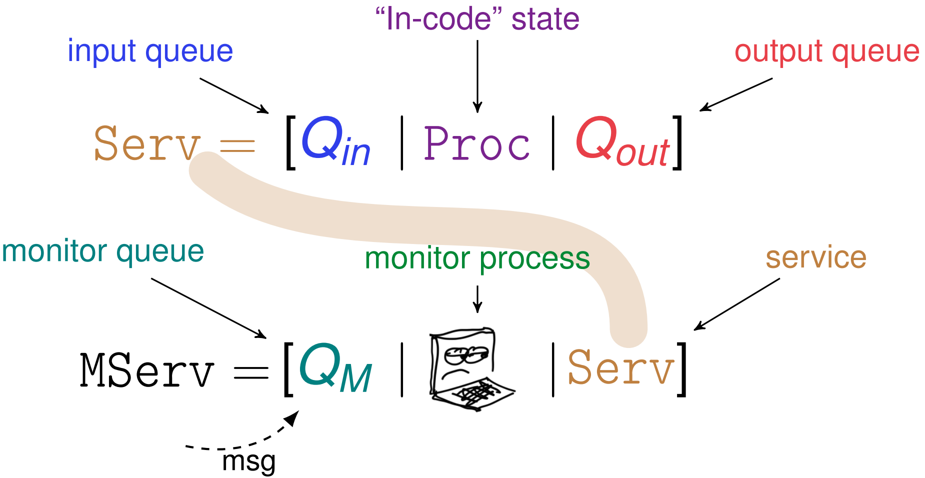
monitor process

service

$$\mathtt{MServ} = [Q_{M} \mid \quad \mid \mathtt{Serv}]$$

$: \mathtt{state} \times (\mathtt{state} \times \mathtt{msg} \to \mathtt{state})$

Services and monitored services



“In-code” state

input queue

output queue

$$\mathtt{Serv} = [Q_{in} \mid \mathtt{Proc} \mid Q_{out}]$$

monitor queue

monitor process

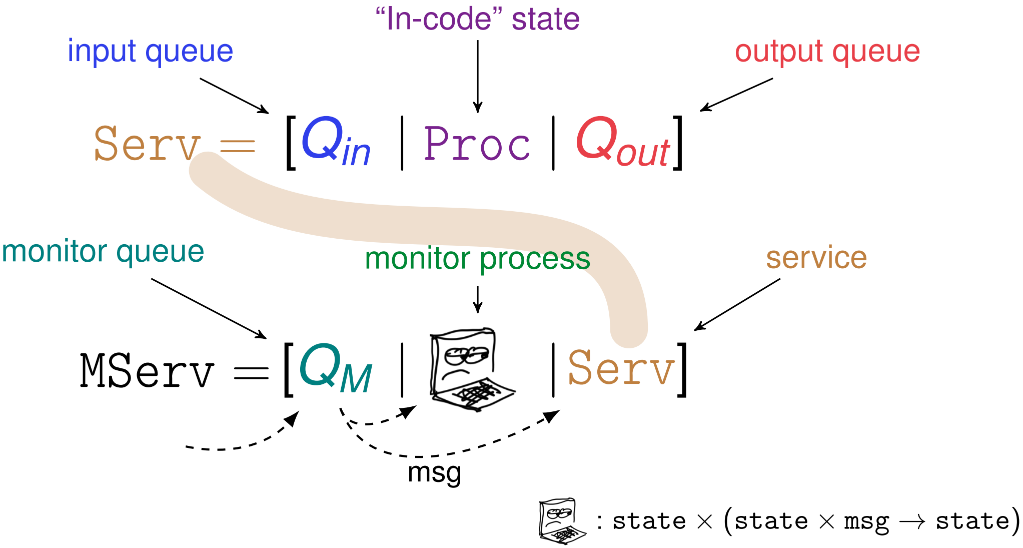
service

$$\mathtt{MServ} = [Q_M \mid \quad \mid \mathtt{Serv}]$$

msg

$: \mathtt{state} \times (\mathtt{state} \times \mathtt{msg} \to \mathtt{state})$

Services and monitored services



“In-code” state

input queue

output queue

$$\mathtt{Serv} = [Q_{in} \mid \mathtt{Proc} \mid Q_{out}]$$

monitor queue

monitor process

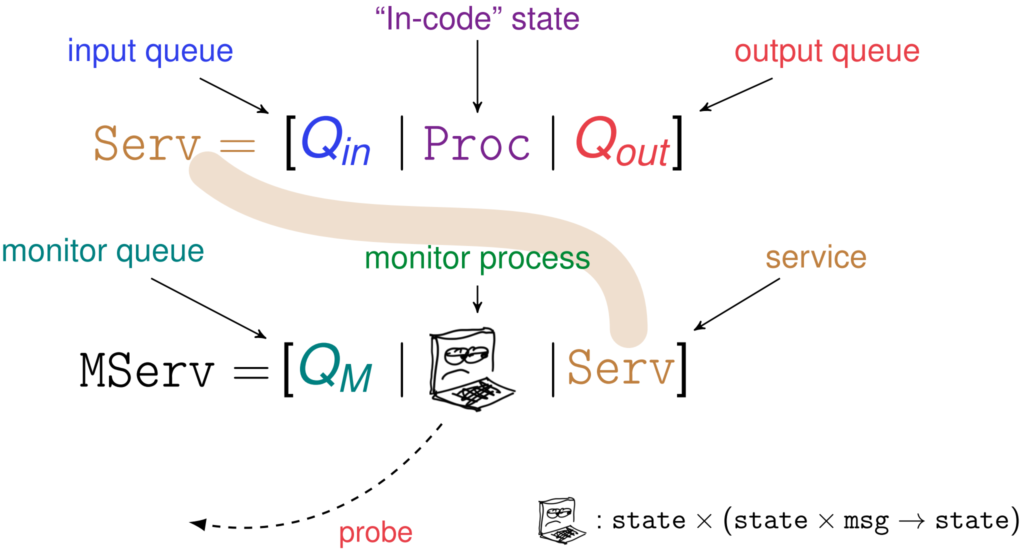
service

$$\mathtt{MServ} = [Q_M \mid \quad \mid \mathtt{Serv}]$$

msg

$: \mathtt{state} \times (\mathtt{state} \times \mathtt{msg} \to \mathtt{state})$

Services and monitored services



input queue

“In-code” state

output queue

$$\texttt{Serv} = [Q_{in} \mid \texttt{Proc} \mid Q_{out}]$$

monitor queue

monitor process

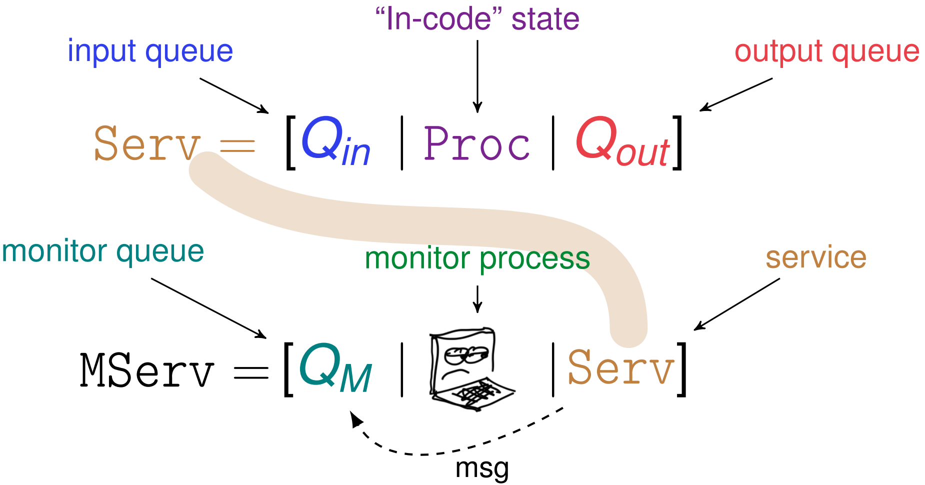
service

$$\texttt{MServ} = [Q_M \mid \quad \mid \texttt{Serv}]$$

probe

$: \texttt{state} \times (\texttt{state} \times \texttt{msg} \to \texttt{state})$

Services and monitored services



“In-code” state

input queue

output queue

$$\mathtt{Serv} = [Q_{in} \mid \mathtt{Proc} \mid Q_{out}]$$

monitor queue

monitor process

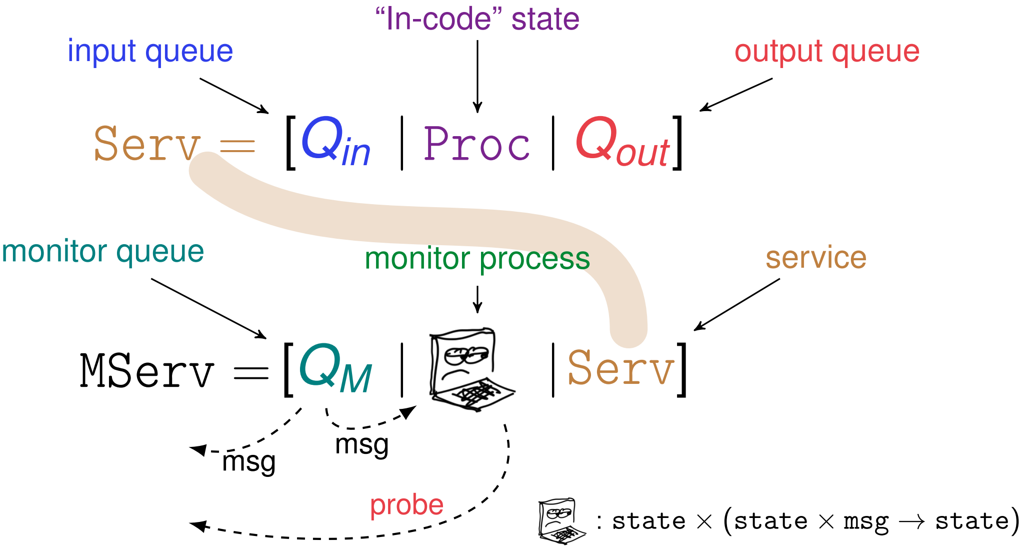
service

$$\mathtt{MServ} = [Q_M \mid \quad \mid \mathtt{Serv}]$$

msg

$: \mathtt{state} \times (\mathtt{state} \times \mathtt{msg} \rightarrow \mathtt{state})$

Services and monitored services



"In-code" state

input queue

output queue

$$\mathtt{Serv} = [Q_{in} \mid \mathtt{Proc} \mid Q_{out}]$$

monitor queue

monitor process

service

$$\mathtt{MServ} = [Q_M \mid \quad \mid \mathtt{Serv}]$$

$: \mathtt{state} \times (\mathtt{state} \times \mathtt{msg} \rightarrow \mathtt{state})$

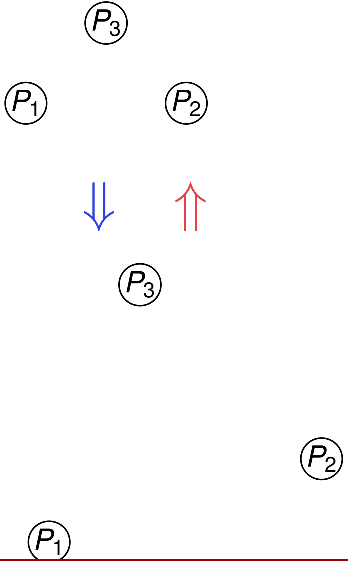
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

- $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \mathtt{mon}'(N_1)$

Soundness: If $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \mathtt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$



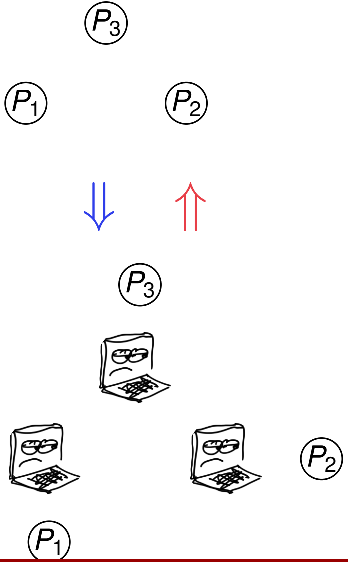
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

- $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \texttt{mon}'(N_1)$

Soundness: If $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \texttt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$



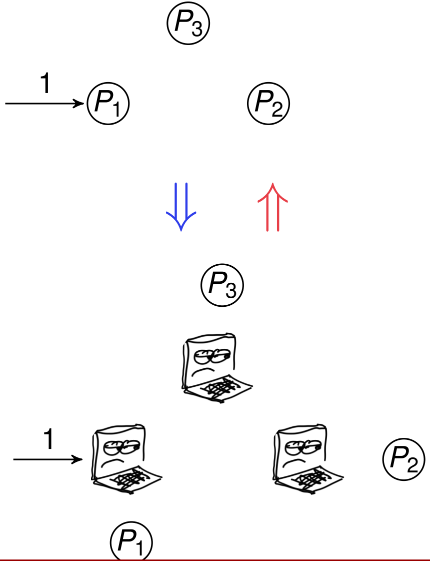
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

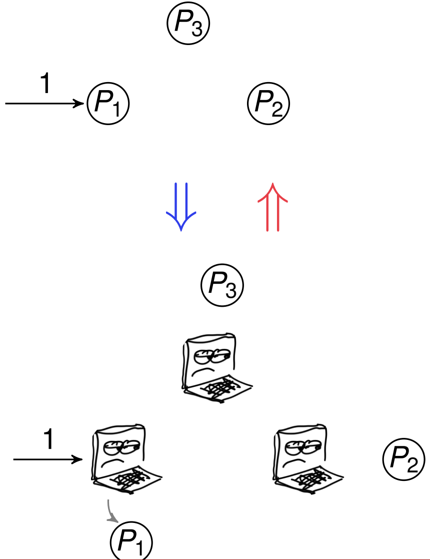
- $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \mathtt{mon}'(N_1)$

Soundness: If $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \mathtt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$



Monitoring: transparency



Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

- $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \mathtt{mon}'(N_1)$

Soundness: If $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \mathtt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$

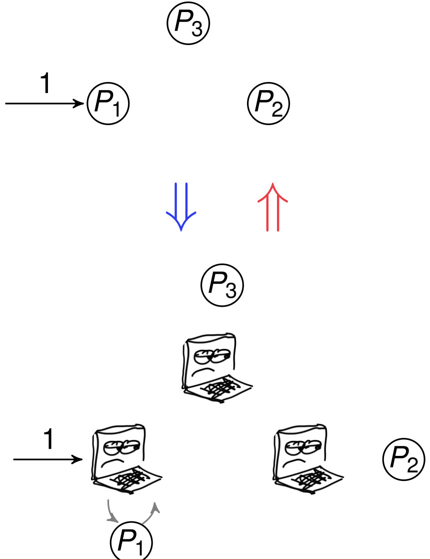
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

- $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \texttt{mon}'(N_1)$

Soundness: If $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \texttt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$



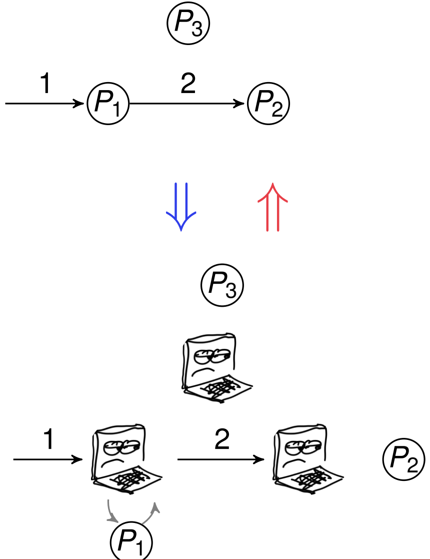
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$
then

- $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \texttt{mon}'(N_1)$

Soundness: If $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$
then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \texttt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{++} \sigma'} N_1$



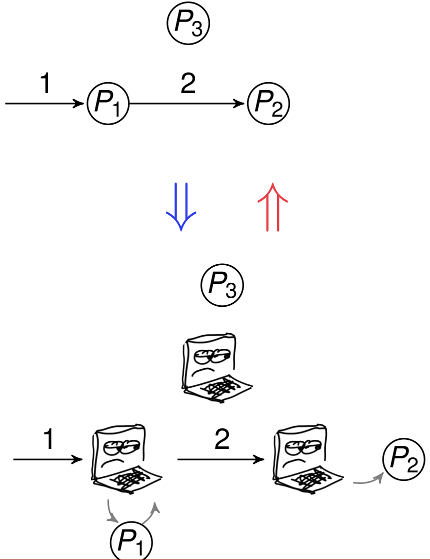
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

- $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \mathtt{mon}'(N_1)$

Soundness: If $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \mathtt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$



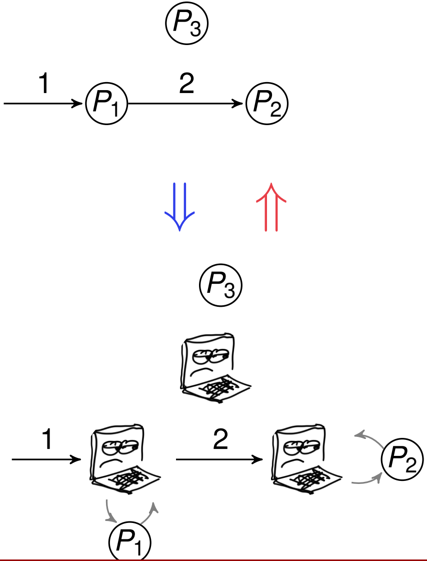
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

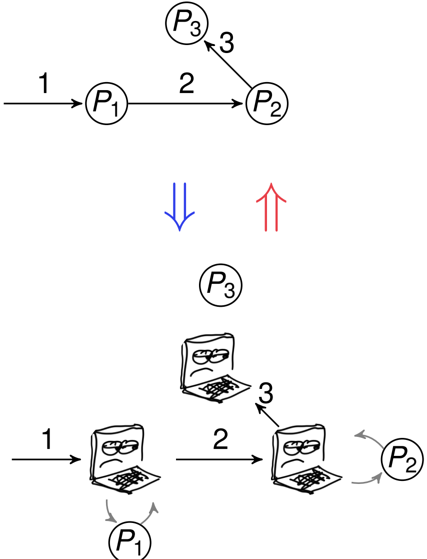
- $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \texttt{mon}'(N_1)$

Soundness: If $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \texttt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$



Monitoring: transparency



Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

- $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \mathtt{mon}'(N_1)$

Soundness: If $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \mathtt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$

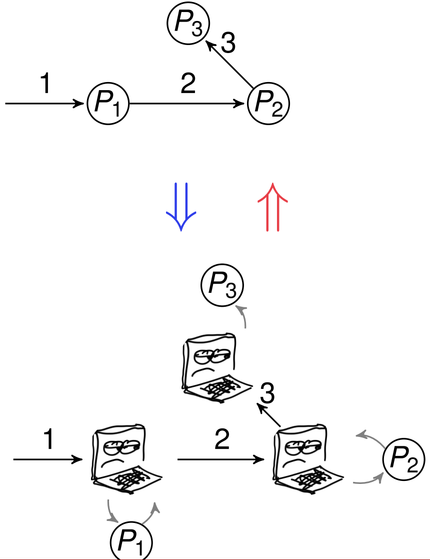
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

- $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \mathtt{mon}'(N_1)$

Soundness: If $\mathtt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \mathtt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$



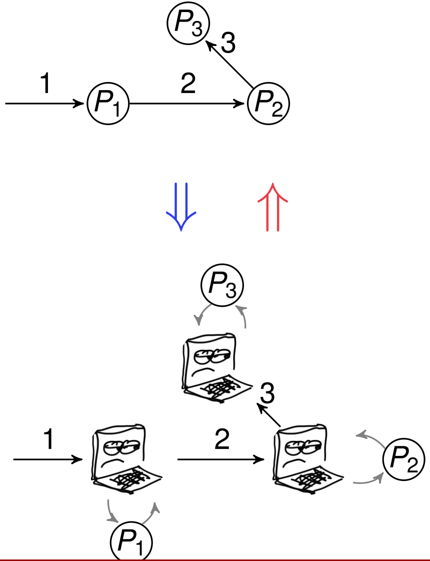
Monitoring: transparency

Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

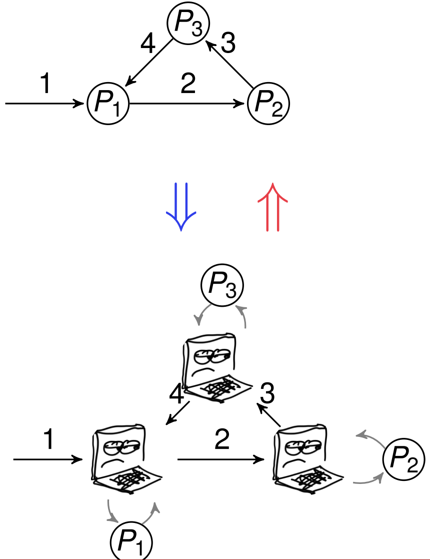
- $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \texttt{mon}'(N_1)$

Soundness: If $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \texttt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{+\!\!+} \sigma'} N_1$



Monitoring: transparency



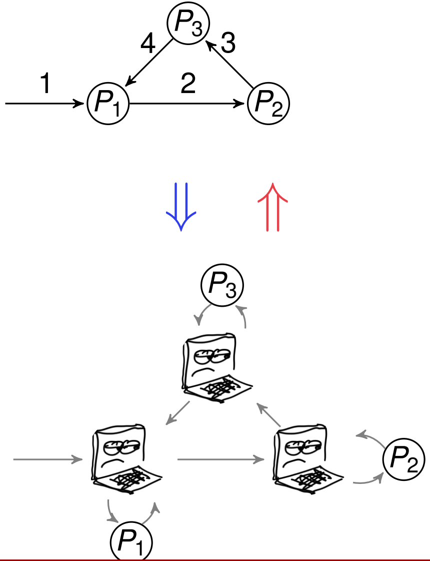
Completeness: If $N_0 \xrightarrow{\sigma} N_1$ then

- $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \texttt{mon}'(N_1)$

Soundness: If $\texttt{mon}(N_0) \xrightarrow{\hat{\sigma}} \hat{N}$ then

- $\hat{N} \xrightarrow{\hat{\sigma}'} \texttt{mon}'(N_1)$
- $N_0 \xrightarrow{\sigma \mathbin{++} \sigma'} N_1$

Monitoring: correctness



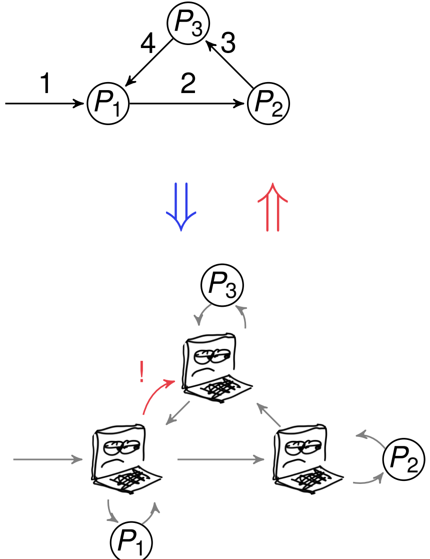
Completeness: All deadlocks are eventually reported

Soundness: All alarms indicate real deadlocks

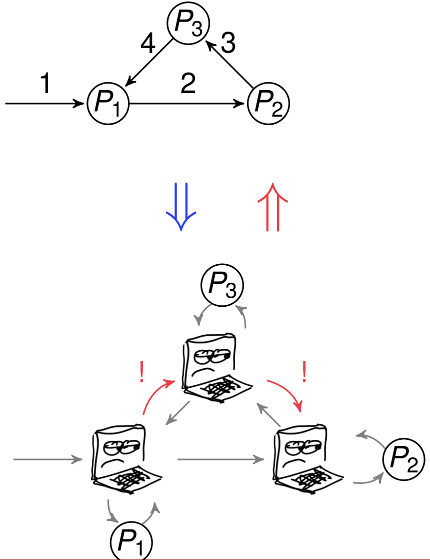
Monitoring: correctness

Completeness: All deadlocks are eventually reported

Soundness: All alarms indicate real deadlocks



Monitoring: correctness



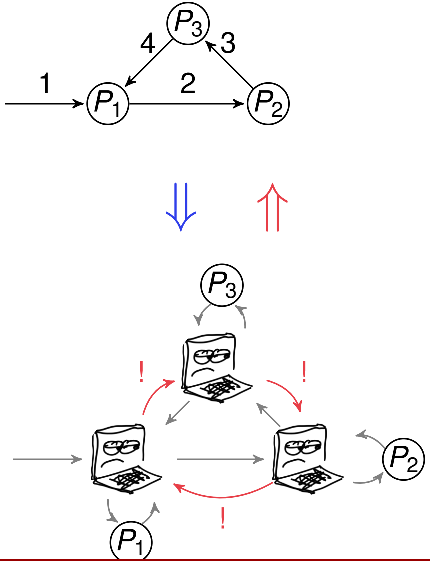
Completeness: All deadlocks are eventually reported

Soundness: All alarms indicate real deadlocks

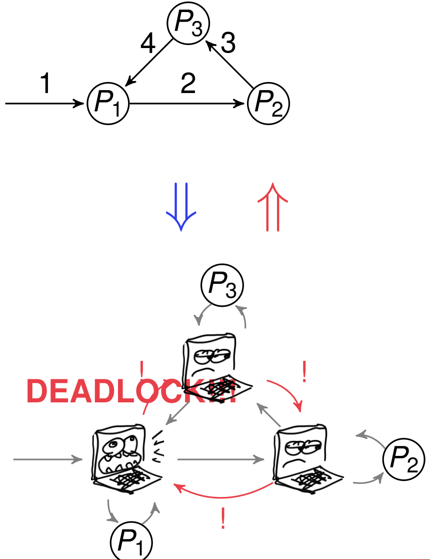
Monitoring: correctness

Completeness: All deadlocks are eventually reported

Soundness: All alarms indicate real deadlocks



Monitoring: correctness

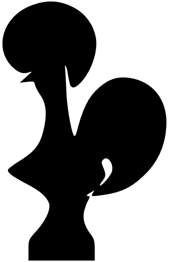


Completeness: All deadlocks are eventually reported

Soundness: All alarms indicate real deadlocks

Results

Everything mechanised and proven in Coq (over 25’000 lines)



Results

Everything mechanised and proven in Coq (over 25’000 lines)

```
Theorem transp_sound :
  ∀ (N₀ : Net) (i₀ : Instr) path' (MN₁ : MNet),
    (i₀ N₀ ⫤ path' ⤇ MN₁)  →

  ∃ path,
    (N₀ ⫤ path ⤇ deinstr MN₁).


Theorem transp_complete :
  ∀ (N₀ N₁ : Net) path (i₀ : Instr),
    (N₀ ⫤ path ⤇ N₁)  →

  ∃ path' (i₁ : Instr),
    (i₀ N₀ ⫤ path' ⤇ i₁ N₁).
```

```
Definition detect_sound (N₀ : Net) (i₀ : Instr) :=
  ∀ path' MN₁,
    (i₀ N₀ ⫤ path' ⤇ MN₁) ∧ reports_deadlock MN₁ →

  ∃ path,
    (N₀ ⫤ path ⤇ deinstr MN₁) ∧ has_deadlock (deinstr MN₁).


Definition detect_complete (N₀ : Net) (i₀ : Instr) :=
  ∀ path N₁,
    (N₀ ⫤ path ⤇ N₁) ∧ has_deadlock N₁ →

  ∃ path' (i₁ : Instr),
    (i₀ N₀ ⫤ path' ⤇ i₁ N₁) ∧ reports_deadlock (i₁ N₁).
```

Main challenge: coming up with invariants :)

Processes in Coq/Gallina

```
Parameters Name Tag Val : Set.

CoInductive Proc :=
| Tau (P : Proc)
| Send (to : Pid) (msg : Val) (P : Proc)
| Recv (select : Pid → Val → option Proc).
```

Selective receive:

- Processes can filter messages
- If message is accepted, the value yields a continuation
- Co-inductive functional syntax embeds Gallina for sequential features
- No issues with binders!

Processes in Coq/Gallina

```
Parameters Name Tag Val : Set.

CoInductive Proc :=
| Tau (P : Proc)
| Send (to : Pid) (msg : Val) (P : Proc)
| Recv (select : Pid → Val → option Proc).
```

Binders bad:

[1] Bengtson. *Formalising the pi-calculus using nominal logic*

[2] Accattoli. *Formalizing Functions as Processes*

[3] Carbone. *The Concurrent Calculi Formalisation Benchmark*

Selective receive:

- Processes can filter messages
- If message is accepted, the value yields a continuation
- Co-inductive functional syntax embeds Gallina for sequential features
- No issues with binders!

Processes in Coq/Gallina

```
Definition Msg := ℕ.

Definition fwd_service (target : string) :=
  {|
    (* State stores the count of forwarded messages *)
    state_t := ℕ;

    (* Initial count is 0 *)
    init := 0;

    (* [handle_call] handles calls *)
    handle_call (_from : Pid) (msg : Msg) (state : ℕ) :=
      match msg with
      | 0 ⇒
          (* Reply with the count *)
          reply c c
      | S msg' ⇒
          (* Query the target with the reduced value *)
          let? x := target ! msg' in
          (* Forward the reply and update the count *)
          reply x (c + 1)
      end |}.
```

```
-module(fwd_service).

-behaviour(gen_server).
-export([init/1, handle_call/3]).

init(Target) →
    register(target, Target),
    %% Initial count is 0
    {ok, 0}.

%% `handle_call` handles calls
handle_call(_From, Msg, State) →
    case Msg of
        0 →
            %% Reply with the count
            {reply, State, State};
        _ →
            %% Query the target with the reduced value
            X = gen_server:call(target, Msg - 1),
            %% Forward the reply and update the count
            {reply, X, State + 1}
    end.
```

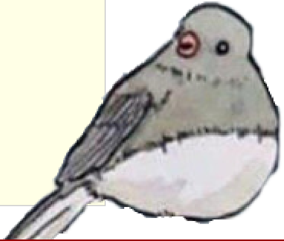
Lessons learned: prove reflexively

Lessons learned: prove reflexively

```
Theorem detection_completeness : ∀ (i0 : instr) N0 MN1 mpath0 DS,
    KIC (i0 N0) →
    (i0 N0 ⫤[ mpath0 ]⇒ MN1) →
    dead_set DS MN1 →
    ∃ mpath1 MN2 n, (MN1 ⫤[ mpath1 ]⇒ MN2) ∧ In n DS ∧ alarm (MN2 n) = true.

Proof.
  intros.

  consider (∃ n, In n DS ∧ dep_on MN1 n n)  by eauto using deadset_dep_self.

  consider (∃ n', dep_on MN1 n n' ∧ ac n' MN1).

  assert (dep_on MN1 n' n')  by eauto using dep_reloop with LTS.

  consider (∃ D mpath1 MN2, (MN1 ⫤[ mpath1 ]⇒ MN2)
                           ∧ dead_set D MN1
                           ∧ alarm (MN2 n') = true
           )
    by eauto using ac_to_alarm.

  ∃ mpath1, MN2, n'.
  now eauto with LTS.
Qed.
```

Lessons learned: prove reflexively

```
Theorem detection_completeness : ∀ (i0 : instr) N0 MN1 mpath0 DS,
    KIC (i0 N0) →
    (i0 N0 =[ mpath0 ]=> MN1) →
    dead_set DS MN1 →
    ∃ mpath1 MN2 n, (MN1 =[ mpath1 ]=> MN2) ∧ In n DS ∧ alarm (MN2 n) = true.

Proof.
  intros.

  consider (∃ n, In n DS ∧ dep_on MN1 n n)  by eauto using deadset_dep_self.

  consider (∃ n', dep_on MN1 n n' ∧ ac n' MN1).

  assert (dep_on MN1 n' n')  by eauto using dep_reloop with LTS.

  consider (∃ D mpath1 MN2, (MN1 =[ mpath1 ]=> MN2)
                             ∧ dead_set D MN1
                             ∧ alarm (MN2 n') = true
           )
    by eauto using ac_to_alarm.

  ∃ mpath1, MN2, n'.
  now eauto with LTS.
Qed.
```



Lessons learned: prove reflexively

```
    - destruct n.
      destruct s; destruct &t; simpl in *.
      + kill H0; hsimpl in *.
        * destruct MQ0; kill H7.
          hsimpl in *.
          econstructor 1; ieattac.
          specialize (H_I_hate_my_life v0). bs.
        * destruct MQ0; kill H7.
          hsimpl in *.
          (* TODO should use H_wtf7 here *)
          econstructor 2; destruct H_wtf6; ieattac.
          specialize (H v); bs.
          specialize (H v); bs.
      + destruct locked0 as [n0|].
        2: kill H0; bs.
        smash_eq n n0; hsimpl in ⊢ *.
        * destruct p, msg; hsimpl in *.
          smash_eq origin1 self0; hsimpl in *.
          -- destruct (PeanoNat.Nat.eqb lock_count0 lo
             ++ kill H0; hsimpl in *.
                --- destruct MQ0; kill H7; hsimpl in *; econstru
(* Leg space     *)     specialize (H v0); bs.
                --- destruct MQ0; kill H7; hsimpl in *; econstructor
                    specialize (H v); bs.
                    specialize (H v); bs.
```

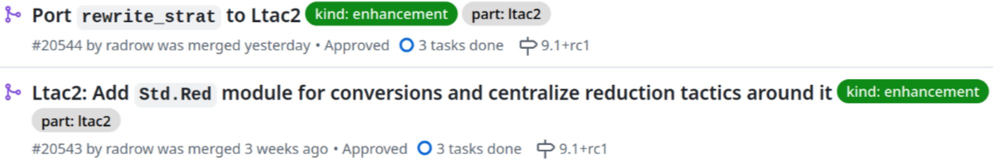
Lessons learned: use Ltac2

- Much better semantics compared to Ltac1
- Slightly uglier, but consistent
- Nicely typed
- Good interop with Ltac1

Lessons learned: use Ltac2

- Much better semantics compared to Ltac1
- Slightly uglier, but consistent
- Nicely typed
- Good interop with Ltac1

Missing a feature in Ltac2? You can contribute!



Port `rewrite_strat` to Ltac2 kind: enhancement part: ltac2

#20544 by radrow was merged yesterday • Approved 3 tasks done 9.1+rc1

Ltac2: Add `Std.Red` module for conversions and centralize reduction tactics around it kind: enhancement part: ltac2

#20543 by radrow was merged 3 weeks ago • Approved 3 tasks done 9.1+rc1

Summary

- Black-box monitors for distributed deadlock detection
- Soundness and completeness derived from syntax and semantics
- Rocq-solid, mechanised proofs