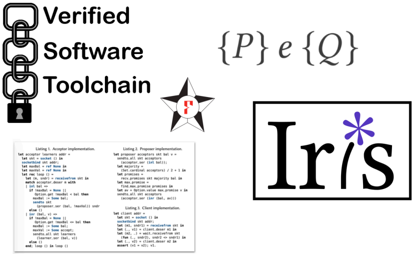
Trillium: Intensional Refinement in Higher-Order Separation Logic

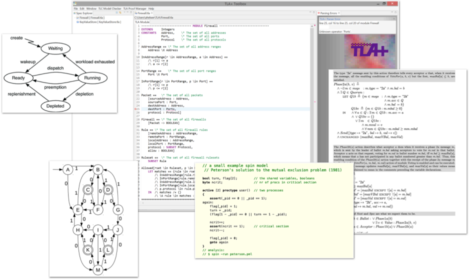
Simon Oddershede Gregersen

joint work with Amin Timany[1], Léo Stefanesco[3], Jonas Kastberg Hinrichsen[1], Léon Gondelman[2], Abel Nieto[1], and Lars Birkedal[1].

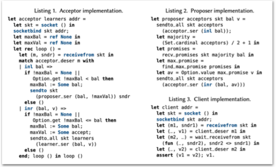
[1]Aarhus University [2]Aalborg University [3]MPI-SWS



Implementations



Models



Listing 1. Acceptor implementation.

```
let acceptor learners addr =
  let skt = socket () in
  socketbind skt addr;
  let maxBal = ref None in
  let maxVal = ref None in
  let rec loop () =
    let (m, sndr) = receivefrom skt in
    match acceptor_deser m with
    | inl bal =>
      if !maxBal = None ||
         Option.get !maxBal < bal then
        maxBal := Some bal;
        sendto skt
          (proposer_ser (bal, !maxVal)) sndr
      else ()
    | inr (bal, v) =>
      if !maxBal = None ||
         Option.get !maxBal <= bal then
        maxBal := Some bal;
        maxVal := Some accept;
        sendto_all skt learners
          (learner_ser (bal, v))
      else ()
    end; loop () in loop ()
```

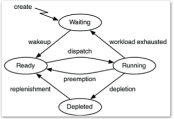
Listing 2. Proposer implementation.

```
let proposer acceptors skt bal v =
  sendto_all skt acceptors
    (acceptor_ser (inl bal));
  let majority =
    (Set.cardinal acceptors) / 2 + 1 in
  let promises =
    recv_promises skt majority bal in
  let max_promise =
    find_max_promise promises in
  let av = Option.value max_promise v in
  sendto_all skt acceptors
    (acceptor_ser (inr (bal, av)))
```

Listing 3. Client implementation.

```
let client addr =
  let skt = socket () in
  socketbind skt addr;
  let (m1, sndr1) = receivefrom skt in
  let (_, v1) = client_deser m1 in
  let (m2, _) = wait_receivefrom skt
    (fun (_, sndr2), sndr2 <> sndr1) in
  let (_, v2) = client_deser m2 in
  assert (v1 = v2); v1.
```

? $\preceq$



How do we connect **realistic implementations** to more abstract **models**?

- Fork-based (node-local) concurrency
- Socket-based communication with serialization
- Higher-order functions, higher-order state, …

This work

Trillium A higher-order separation logic framework for showing different notions of trace refinement between programs and models.

We consider two instantiations of the framework:

This work

Trillium A higher-order separation logic framework for showing different notions of trace refinement between programs and models.

We consider two instantiations of the framework:

Aneris for reasoning about safety properties of implementations of distributed systems communicating over an unreliable network.

This work

Trillium A higher-order separation logic framework for showing different notions of trace refinement between programs and models.

We consider two instantiations of the framework:

Aneris for reasoning about safety properties of implementations of distributed systems communicating over an unreliable network.

Fairis for reasoning about termination of fine-grained concurrent programs under fair scheduling assumptions.

Trillium

A language-generic framework for showing **lockstep simulation**, built on top of the Iris separation logic framework and mechanized in the Coq proof assistant.

Trillium

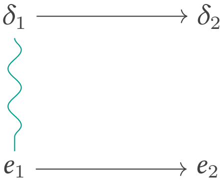
A language-generic framework for showing **lockstep simulation**, built on top of the Iris separation logic framework and mechanized in the Coq proof assistant.

Trillium

A language-generic framework for showing **lockstep simulation**, built on top of the Iris separation logic framework and mechanized in the Coq proof assistant.

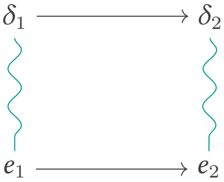
Trillium

A language-generic framework for showing **lockstep simulation**, built on top of the Iris separation logic framework and mechanized in the Coq proof assistant.



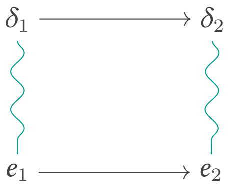
Trillium

A language-generic framework for showing **lockstep simulation**, built on top of the Iris separation logic framework and mechanized in the Coq proof assistant.



Trillium

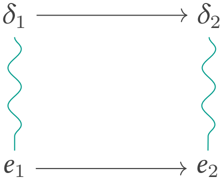
A language-generic framework for showing **lockstep simulation**, built on top of the Iris separation logic framework and mechanized in the Coq proof assistant.



We will weaken lockstep simulation through model constructions.

Key Ideas

1. Use a program logic $\{P\}\, e\, \{Q\}$ to reason about the program.
2. Use a separation logic resource Model(δ) to embed the current model state in the logic and restrict its progression to preserve properties of interest.
3. Encode the refinement mapping using Iris invariant assertions $\boxed{P}$.



Example

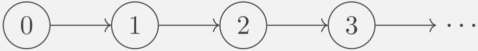
To show that $e \triangleq$ **while** true **do** $\ell \leftarrow\, !\ell + 1$ **end** refines the state-transition system

one shows a specification of the shape

$$\left\{\boxed{\exists n.\, \ell \mapsto n * \mathsf{Model}(n)}\right\}\, e\, \{Q\}.$$

Example

To show that $e \triangleq$ **while** true **do** $\ell \leftarrow\ !\,\ell + 1$ **end** refines the state-transition system



one shows a specification of the shape

$$\left\{\boxed{\exists n.\, \ell \mapsto n * \mathsf{Model}(n)}\right\}\, e\, \{Q\}.$$

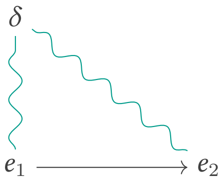
But lockstep simulation—while sound—is much too restrictive, *e.g.*,

$$\frac{\delta \rightharpoonup \delta'}{\{\mathsf{Model}(\delta)\}\; n + m\; \{v.\, v = (n + m) * \mathsf{Model}(\delta')\}}$$

Safety Properties

Models are (often) simpler than implementations so stuttering is necessary.

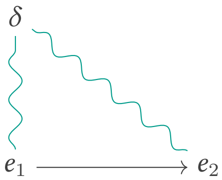
To preserve safety properties, it is sound to allow **unrestricted stuttering**.



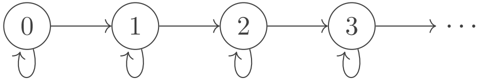
Safety Properties

Models are (often) simpler than implementations so stuttering is necessary.

To preserve safety properties, it is sound to allow **unrestricted stuttering**.



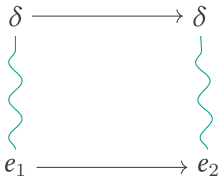
That is, lockstep simulation of the **reflexive closure** of the model.



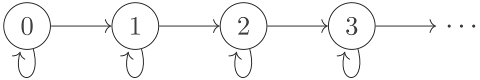
Safety Properties

Models are (often) simpler than implementations so stuttering is necessary.

To preserve safety properties, it is sound to allow **unrestricted stuttering**.



That is, lockstep simulation of the **reflexive closure** of the model.



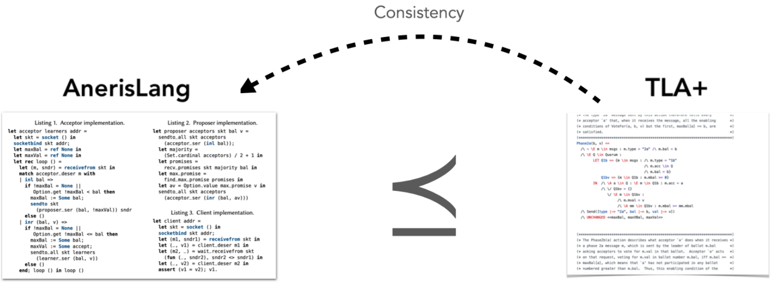
Aneris

We "bake in" the reflexive closure, instantiate Trillium with AnerisLang—an ML-like language with UDP communication primitives—and recover *Aneris* [ESOP'20], a **distributed separation logic**.

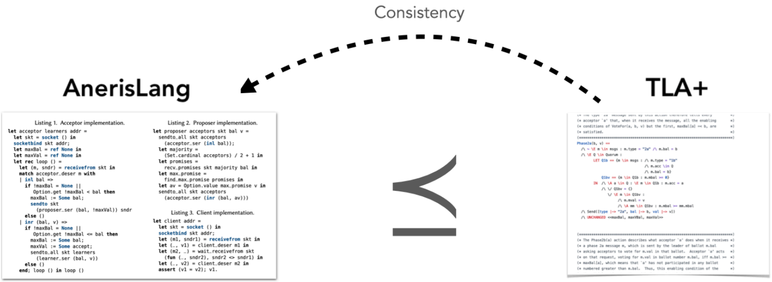
All existing specifications and reasoning principles still hold, with the addition of just **one rule** for progressing the model.

$$\frac{\{P\}\, e\, \{Q\} \qquad \delta \rightharpoonup \delta' \qquad \mathsf{Atomic}(e)}{\{P * \mathsf{Model}(\delta)\}\, e\, \{Q * \mathsf{Model}(\delta')\}}$$

Single-Decree Paxos by Refinement



Single-Decree Paxos by Refinement



Theorem (Consistency)

If $\delta_{\mathit{init}} \rightharpoonup^{*} \delta'_{\mathsf{SDP}}$ *and both* $\mathrm{Chosen}(\delta'_{\mathsf{SDP}}, v_1)$ *and* $\mathrm{Chosen}(\delta'_{\mathsf{SDP}}, v_2)$ *then* $v_1 = v_2$.

…and show **node- and role-local specifications**

$$\{\boxed{I_{\mathsf{SDP}}} * \ldots\}\ \texttt{acceptor}\ L\ a\ \{\ldots\}$$

$$\{\boxed{I_{\mathsf{SDP}}} * \ldots\}\ \texttt{proposer}\ A\ s\ b\ v\ \{\ldots\}$$

$$\{\boxed{I_{\mathsf{SDP}}} * \ldots\}\ \texttt{learner}\ s\ a\ \{\ldots\}$$

where

…and show **node- and role-local specifications**

$$\{\boxed{I_{\mathsf{SDP}}} * \ldots\}\, \texttt{acceptor}\ L\ a\ \{\ldots\}$$

$$\{\boxed{I_{\mathsf{SDP}}} * \ldots\}\, \texttt{proposer}\ A\ s\ b\ v\ \{\ldots\}$$

$$\{\boxed{I_{\mathsf{SDP}}} * \ldots\}\, \texttt{learner}\ s\ a\ \{\ldots\}$$

where

$$I_{\mathsf{SDP}} \triangleq \exists \delta_{\mathsf{SDP}}.\, \mathsf{Model}(\delta_{\mathsf{SDP}}) * \mathsf{PaxosRes}_{\bullet}(\delta_{\mathsf{SDP}}) * \mathsf{BallotCoh}(\delta_{\mathsf{SDP}})$$

…and show **node- and role-local specifications**

$$\left\{\boxed{I_{\mathsf{SDP}}} * \ldots\right\} \texttt{acceptor}\ L\ a\ \{\ldots\}$$

$$\left\{\boxed{I_{\mathsf{SDP}}} * \ldots\right\} \texttt{proposer}\ A\ s\ b\ v\ \{\ldots\}$$

$$\left\{\boxed{I_{\mathsf{SDP}}} * \ldots\right\} \texttt{learner}\ s\ a\ \{\ldots\}$$

where

$$I_{\mathsf{SDP}} \triangleq \exists \delta_{\mathsf{SDP}}.\ \boxed{\mathsf{Model}(\delta_{\mathsf{SDP}})} * \mathsf{PaxosRes}_{\bullet}(\delta_{\mathsf{SDP}}) * \mathsf{BallotCoh}(\delta_{\mathsf{SDP}})$$

...and show **node- and role-local specifications**

$$\left\{\boxed{I_{\mathsf{SDP}}} * \boxed{\mathsf{PaxosRes}_{\circ}(\ldots)} * \ldots\right\} \texttt{acceptor}\ L\ a\ \{\ldots\}$$

$$\left\{\boxed{I_{\mathsf{SDP}}} * \ldots\right\} \texttt{proposer}\ A\ s\ b\ v\ \{\ldots\}$$

$$\left\{\boxed{I_{\mathsf{SDP}}} * \ldots\right\} \texttt{learner}\ s\ a\ \{\ldots\}$$

where

$$I_{\mathsf{SDP}} \triangleq \exists \delta_{\mathsf{SDP}}.\, \mathsf{Model}(\delta_{\mathsf{SDP}}) * \boxed{\mathsf{PaxosRes}_{\bullet}(\delta_{\mathsf{SDP}})} * \mathsf{BallotCoh}(\delta_{\mathsf{SDP}})$$

…and show **node- and role-local specifications**

$$\{\boxed{I_{\mathsf{SDP}}} * \mathsf{PaxosRes}_{\circ}(\ldots) * \ldots\}\ \texttt{acceptor}\ L\ a\ \{\ldots\}$$

$$\{\boxed{I_{\mathsf{SDP}}} * \boxed{\mathsf{pending}(b)} * \ldots\}\ \texttt{proposer}\ A\ s\ b\ v\ \{\ldots\}$$

$$\{\boxed{I_{\mathsf{SDP}}} * \ldots\}\ \texttt{learner}\ s\ a\ \{\ldots\}$$

where

$$I_{\mathsf{SDP}} \triangleq \exists \delta_{\mathsf{SDP}}.\ \mathsf{Model}(\delta_{\mathsf{SDP}}) * \mathsf{PaxosRes}_{\bullet}(\delta_{\mathsf{SDP}}) * \boxed{\mathsf{BallotCoh}(\delta_{\mathsf{SDP}})}$$

…and show **node- and role-local specifications**

$$\left\{\boxed{I_{\mathsf{SDP}}} * \mathsf{PaxosRes}_{\circ}\,(\ldots) * \ldots\right\} \texttt{acceptor}\ L\ a\ \{\ldots\}$$

$$\left\{\boxed{I_{\mathsf{SDP}}} * \mathsf{pending}(b) * \ldots\right\} \texttt{proposer}\ A\ s\ b\ v\ \{\ldots\}$$

$$\left\{\boxed{I_{\mathsf{SDP}}} * \ldots\right\} \texttt{learner}\ s\ a\ \{\ldots\}$$

where

$$I_{\mathsf{SDP}} \triangleq \exists \delta_{\mathsf{SDP}}.\, \mathsf{Model}(\delta_{\mathsf{SDP}}) * \mathsf{PaxosRes}_{\bullet}\,(\delta_{\mathsf{SDP}}) * \mathsf{BallotCoh}(\delta_{\mathsf{SDP}})$$

Takeaway: the invariant is quite simple and **only** concerned with refinement!

…and show **node- and role-local specifications**

$$\{\boxed{I_{\mathsf{SDP}}} * \mathsf{PaxosRes}_{\circ}(\ldots) * \ldots\}\ \texttt{acceptor}\ L\ a\ \{\ldots\}$$

$$\{\boxed{I_{\mathsf{SDP}}} * \mathsf{pending}(b) * \ldots\}\ \texttt{proposer}\ A\ s\ b\ v\ \{\ldots\}$$

$$\{\boxed{I_{\mathsf{SDP}}} * \ldots\}\ \texttt{learner}\ s\ a\ \{\ldots\}$$

where

$$I_{\mathsf{SDP}} \triangleq \exists \delta_{\mathsf{SDP}}.\ \mathsf{Model}(\delta_{\mathsf{SDP}}) * \mathsf{PaxosRes}_{\bullet}(\delta_{\mathsf{SDP}}) * \mathsf{BallotCoh}(\delta_{\mathsf{SDP}})$$

Takeaway: the invariant is quite simple and **only** concerned with refinement!

Putting everything together gives us **consistency for all program traces**.

Benefits

1. No need to come up with a new consensus proof.

Benefits

1. No need to come up with a new consensus proof.
2. The refinement proof requires almost **no advanced ghost state usage**.

Benefits

1. No need to come up with a new consensus proof.
2. The refinement proof requires almost **no advanced ghost state usage**.
3. As the model is embedded as a resource in the logic, we can **internalize** properties of the model while proving specifications

$$\frac{\{P * v_1 = v_2\}\ e\ \{Q\}}{\{P * \mathsf{Chosen}(v_1) * \mathsf{Chosen}(v_2)\}\ e\ \{Q\}}$$

which allows us to verify clients, *e.g.*,

```
let client addr =
  // ...
  let v1 = client_deser m1 in
  let v2 = client_deser m2 in
  assert (v1 == v2); v1.
```

Liveness Properties

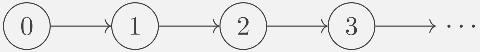
To preserve liveness properties, unrestricted stuttering is **unsound**.

Liveness Properties

To preserve liveness properties, unrestricted stuttering is **unsound**.

Example

The program `while true do skip end` refines (using unrestricted stuttering)



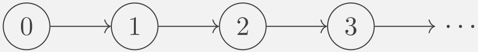
but “the value of the counter is eventually 3” is obviously not preserved.

Liveness Properties

To preserve liveness properties, unrestricted stuttering is **unsound**.

Example

The program `while true do skip end` refines (using unrestricted stuttering)

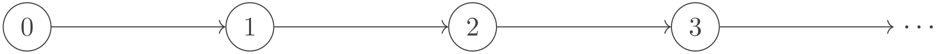


but “the value of the counter is eventually 3” is obviously not preserved.

We can only permit **finite stuttering**.

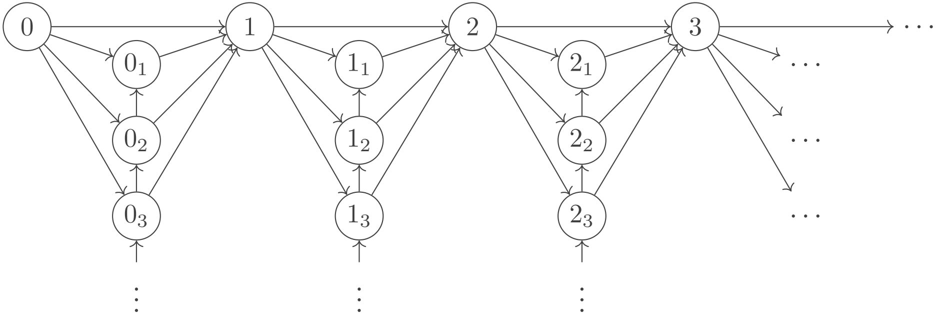
Liveness Properties

Rather than adding self-loops, we allow **finite stuttering** through what essentially corresponds to lockstep simulation with finite unrollings of self-loops.



Liveness Properties

Rather than adding self-loops, we allow **finite stuttering** through what essentially corresponds to lockstep simulation with finite unrollings of self-loops.



Fair Termination

To talk about fairness of model traces, we consider **labeled transitions systems** where labels denote abstract roles.

Fair Termination

To talk about fairness of model traces, we consider **labeled transitions systems** where labels denote abstract roles.

To preserve fair termination, the simulation relation also has to **preserve fairness**.

$$\mathsf{Fair}(\tau_{\mathsf{model}}) \Longrightarrow \mathsf{Term}(\tau_{\mathsf{model}})$$

$$\preceq$$

$$\mathsf{Fair}(\tau_{\mathsf{prog}}) \Longrightarrow \mathsf{Term}(\tau_{\mathsf{prog}})$$

Fair Termination

To talk about fairness of model traces, we consider **labeled transitions systems** where labels denote abstract roles.

To preserve fair termination, the simulation relation also has to **preserve fairness**.

$$\begin{array}{ccc} \mathsf{Fair}(\tau_{\mathsf{model}}) & \Longrightarrow & \mathsf{Term}(\tau_{\mathsf{model}}) \\ \Uparrow & \preceq & \\ \mathsf{Fair}(\tau_{\mathsf{prog}}) & \Longrightarrow & \mathsf{Term}(\tau_{\mathsf{prog}}) \end{array}$$

Fair Termination

To talk about fairness of model traces, we consider **labeled transitions systems** where labels denote abstract roles.

To preserve fair termination, the simulation relation also has to **preserve fairness**.

$$
\begin{array}{ccc}
\mathsf{Fair}(\tau_{\mathsf{model}}) & \Longrightarrow & \mathsf{Term}(\tau_{\mathsf{model}}) \\
\Uparrow & \preceq & \Downarrow \\
\mathsf{Fair}(\tau_{\mathsf{prog}}) & \Longrightarrow & \mathsf{Term}(\tau_{\mathsf{prog}})
\end{array}
$$

Fair Termination

To talk about fairness of model traces, we consider **labeled transitions systems** where labels denote abstract roles.

To preserve fair termination, the simulation relation also has to **preserve fairness**.

$$
\begin{array}{ccc}
\mathsf{Fair}(\tau_{\mathsf{model}}) & \Longrightarrow & \mathsf{Term}(\tau_{\mathsf{model}}) \\
\Uparrow & \preceq & \Downarrow \\
\mathsf{Fair}(\tau_{\mathsf{prog}}) & \Longrightarrow & \mathsf{Term}(\tau_{\mathsf{prog}})
\end{array}
$$

This is achieved by making sure roles **do not get “starved”**.

Fairis

Given an (LTS) model $\mathcal{M}$, the Fairis logic exploits a Fuel$(\mathcal{M})$ construction that enforces **finite stuttering for all roles**.

Fairis

Given an (LTS) model $\mathcal{M}$, the Fairis logic exploits a Fuel$(\mathcal{M})$ construction that enforces **finite stuttering for all roles**.

- Each thread id is associated with a set of roles, each with an amount of “fuel”,

Fairis

Given an (LTS) model $\mathcal{M}$, the Fairis logic exploits a Fuel($\mathcal{M}$) construction that enforces **finite stuttering for all roles**.

- Each thread id is associated with a set of roles, each with an amount of “fuel”,
- If a thread stutters, the fuel of all its roles are decremented, and

Fairis

Given an (LTS) model $\mathcal{M}$, the Fairis logic exploits a Fuel$(\mathcal{M})$ construction that enforces **finite stuttering for all roles**.

- Each thread id is associated with a set of roles, each with an amount of “fuel”,
- If a thread stutters, the fuel of all its roles are decremented, and
- If a thread takes a step in $\mathcal{M}$ for role ρ, then ρ is refueled.

Fairis

Given an (LTS) model $\mathcal{M}$, the Fairis logic exploits a Fuel$(\mathcal{M})$ construction that enforces **finite stuttering for all roles**.

- Each thread id is associated with a set of roles, each with an amount of "fuel",
- If a thread stutters, the fuel of all its roles are decremented, and
- If a thread takes a step in $\mathcal{M}$ for role ρ, then ρ is refueled.

The **Fairis logic** manages the complexity using a resource

$$\text{tid} \Mapsto \{\rho_1 \mapsto \mathrm{f}_1, \ldots, \rho_n \mapsto \mathrm{f}_n\}$$

together with the Model(δ) resource for the user-chosen model $\mathcal{M}$.

Summary

Trillium A higher-order separation logic framework for showing trace refinement between programs and models.

Aneris An instantiation of Trillium for reasoning about distributed systems.

- Single-decree Paxos refines its TLA+ model.

Fairis An instantiation of Trillium for proving termination of fine-grained concurrent programs under fair scheduling assumptions.

Thank you!

NYU

Future Work

- Fairis applies to (non-distributed) concurrent programs—fairness of distributed systems traces is a bit more subtle.
- Explore more constructions at the model level to allow for more modularity.
- More high-level reasoning principles for liveness reasoning.

Remark

- Logics (like Iris) based on step indexing fundamentally cannot prove liveness properties—at least directly.
- The Fairis approach sidesteps this issue entirely.
- **No (entirely) free lunch**: we have a “relative image-finiteness requirement” for the simulation relation. In practice, it has not (yet?) been an obstacle, but the restriction can be lifted with transfinite step indexing.