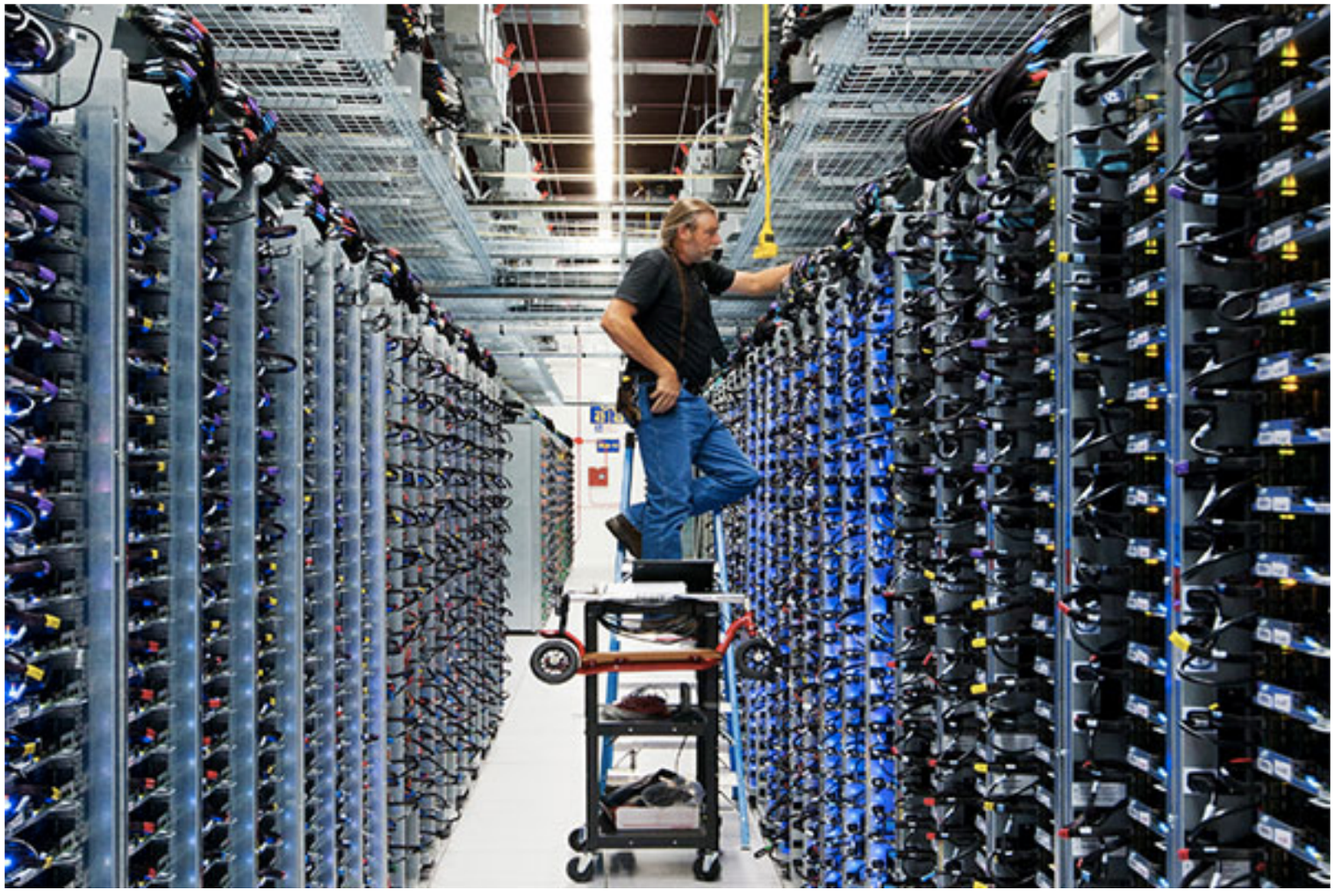
Verifying distributed systems with Verdi

Ryan Doenges
with Zachary Tatlock, James Wilcox,
Doug Woos, and Karl Palmskog

PLSE

Distributed systems: reliability and scalability



Verifying systems in Verdi

Previously:

- Raft consensus protocol (50k LOC)

Presently:

- Chord distributed hash table
- a data aggregation protocol

Verdi workflow

1. Write your system as *event handlers*
2. Verify it using our *network semantics*
3. Run it with the corresponding *shim*

Handlers change local state and send messages.

```
Definition result :=
  state * list (addr * msg).
```

new state

what to send

where to send it

Existing event: delivery

```
Definition result :=
  state * list (addr * msg).

Definition recv_handler
  (dst : addr)
  (st  : state)
  (src : addr)
  (m   : msg)
  : result := ...
```

New event: node start-up

```
Definition result :=
  state * list (addr * msg).

Definition init_handler
  (h       : addr)
  (knowns : list addr)
  : result := ...
```

Semantics: fixed networks

```
Record net :=
  {| failed_nodes : list addr;
     packets : addr -> addr -> list msg;
     state : addr -> state |}.

Inductive step : net -> net -> Prop :=
| Step_deliver : ...
| Step_fail : ...
```

what is a network?

when can one network turn into another?

Semantics: fixed networks

```
Record net :=
  {| failed_nodes : list addr;
     packets : addr -> addr -> list msg;
     state : addr -> state |}.
```

```
Inductive step : net -> net -> Prop :=
| Step_deliver : 📬...
| Step_fail : ☠️...
```

Semantics: fixed networks

probably finite

```
Record net :=
  {| failed_nodes : list addr;
     packets : addr -> addr -> list msg;
     state : addr -> state |}.
```

```
Inductive step : net -> net -> Prop :=
| Step_deliver : 📪...
| Step_fail : ☠...
```

Semantics with churn

```
Record net :=
  {| failed_nodes : list addr;
     nodes : list addr;
     packets : addr -> addr -> list msg;
     state : addr -> option state |}.

Inductive step : net -> net -> Prop :=
| Step_deliver : 📬...
| Step_fail : ☠...
| Step_init : 🐣...
```

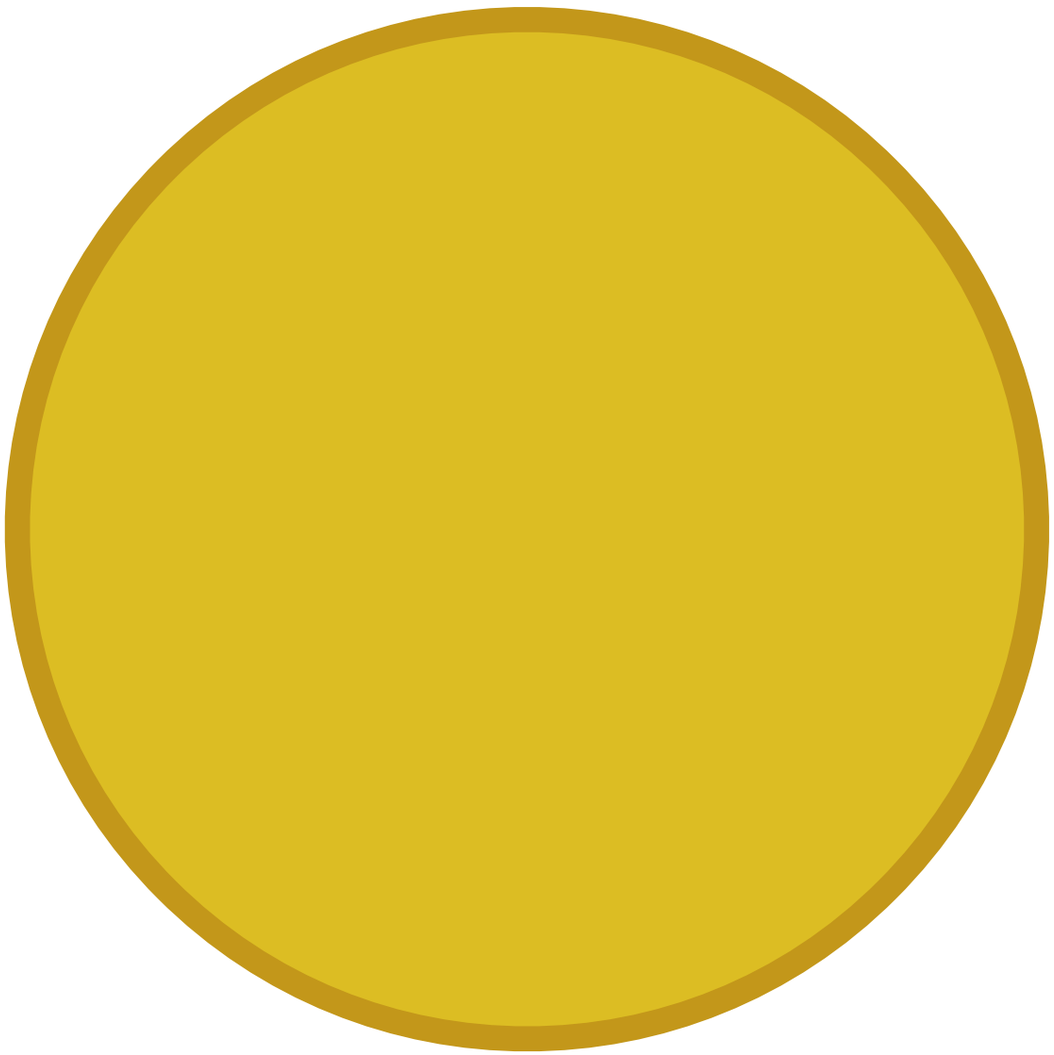
Verification of safety (bad things don't happen)

1. State a safety property `P : net -> Prop`.

2. Try to prove that for all initial states `init` and all networks `net`, we have `star (step init) net -> P net`.

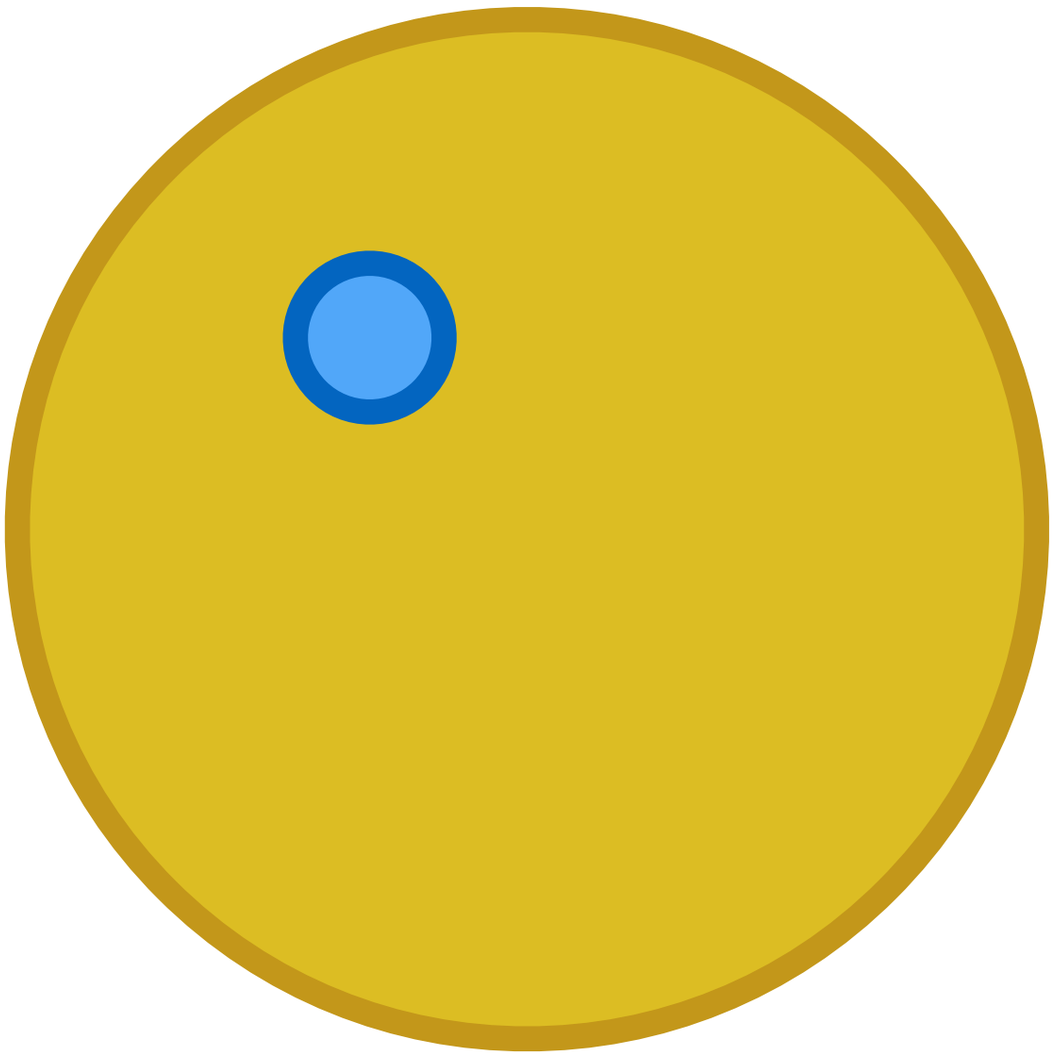
3. Fail to prove this by induction on `star`, find a stronger property `P'` which is inductive and implies `P`.

We can prove safety properties with inductive invariants



A predicate P on states is an inductive invariant when

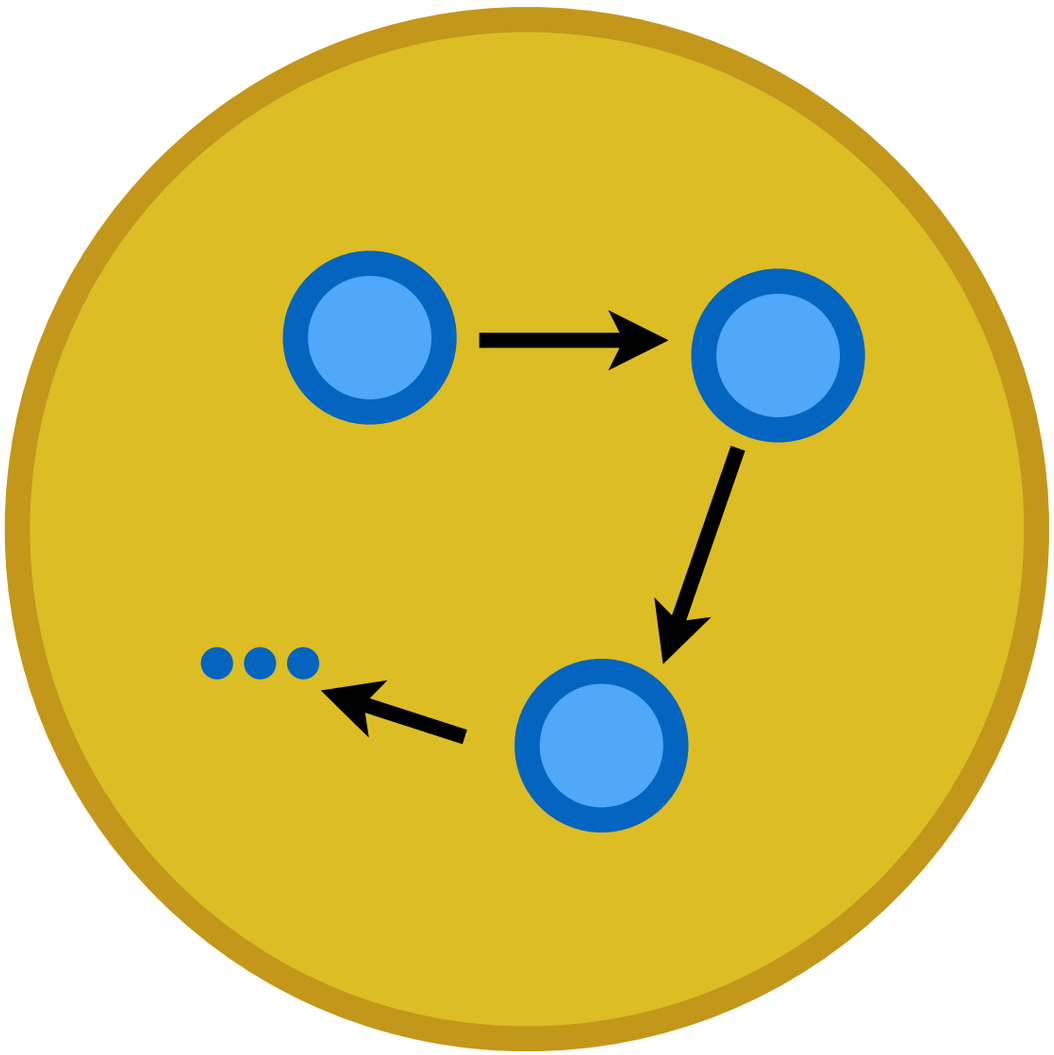
Inductive invariants



A predicate P on states is an inductive invariant when

- P holds for the initial state

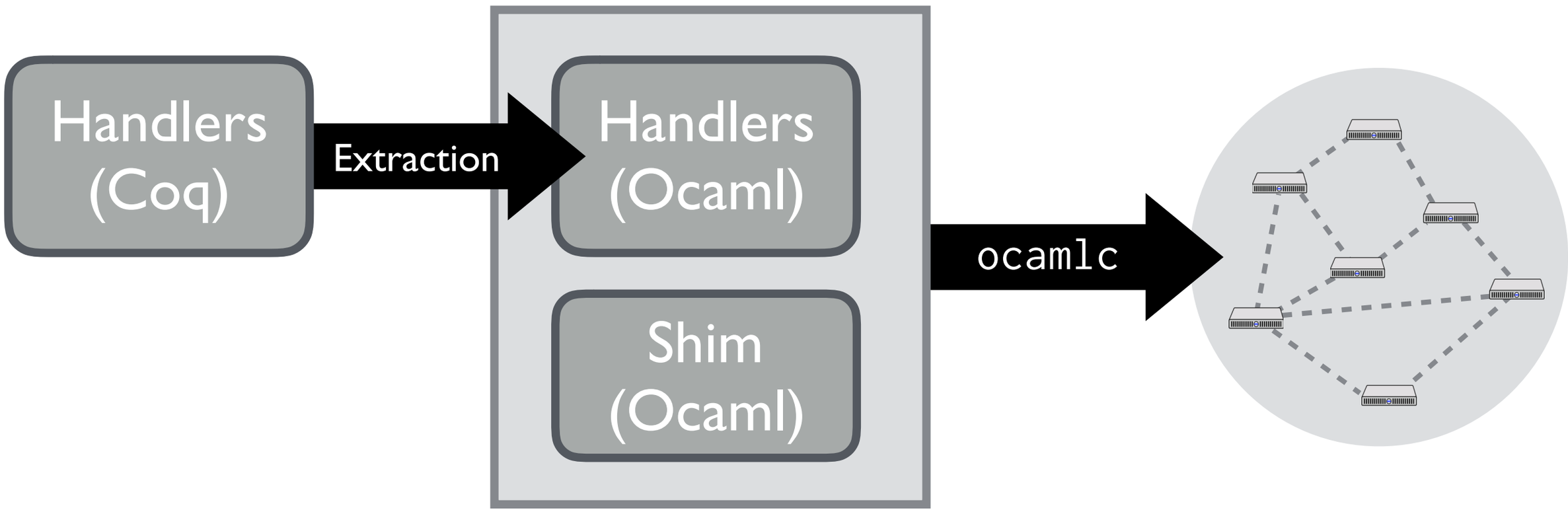
Inductive invariants



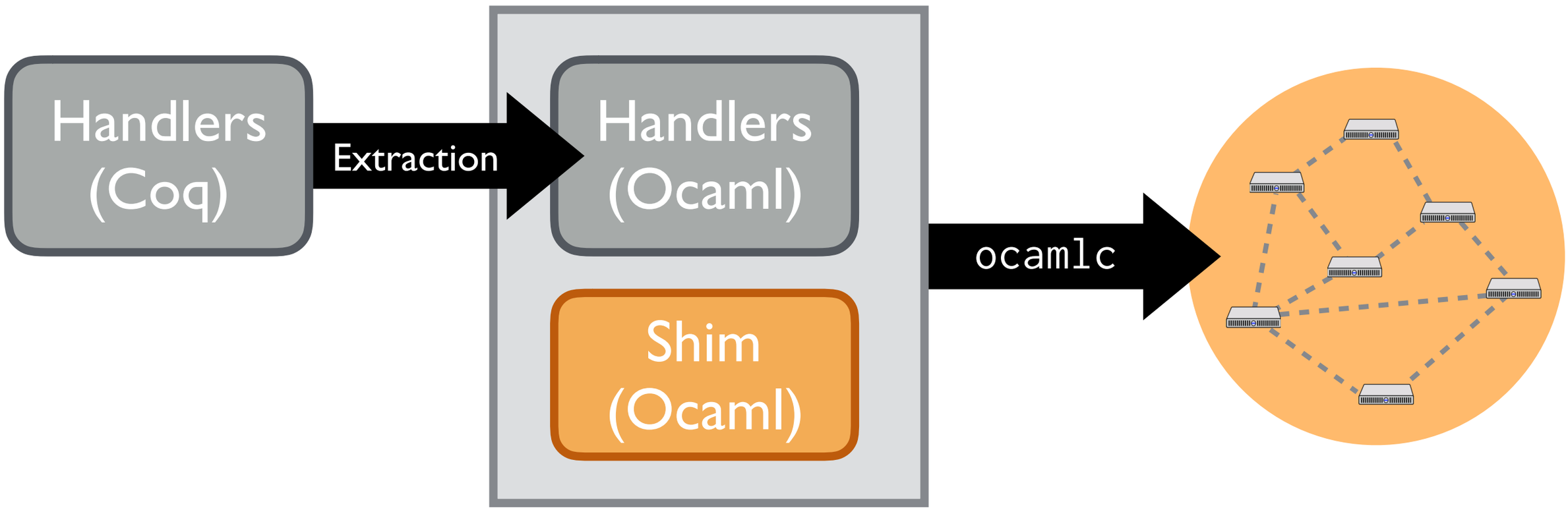
A predicate P on states is an inductive invariant when

- P holds for the initial state
- P is preserved by the step

The shim runs handlers on a real network.

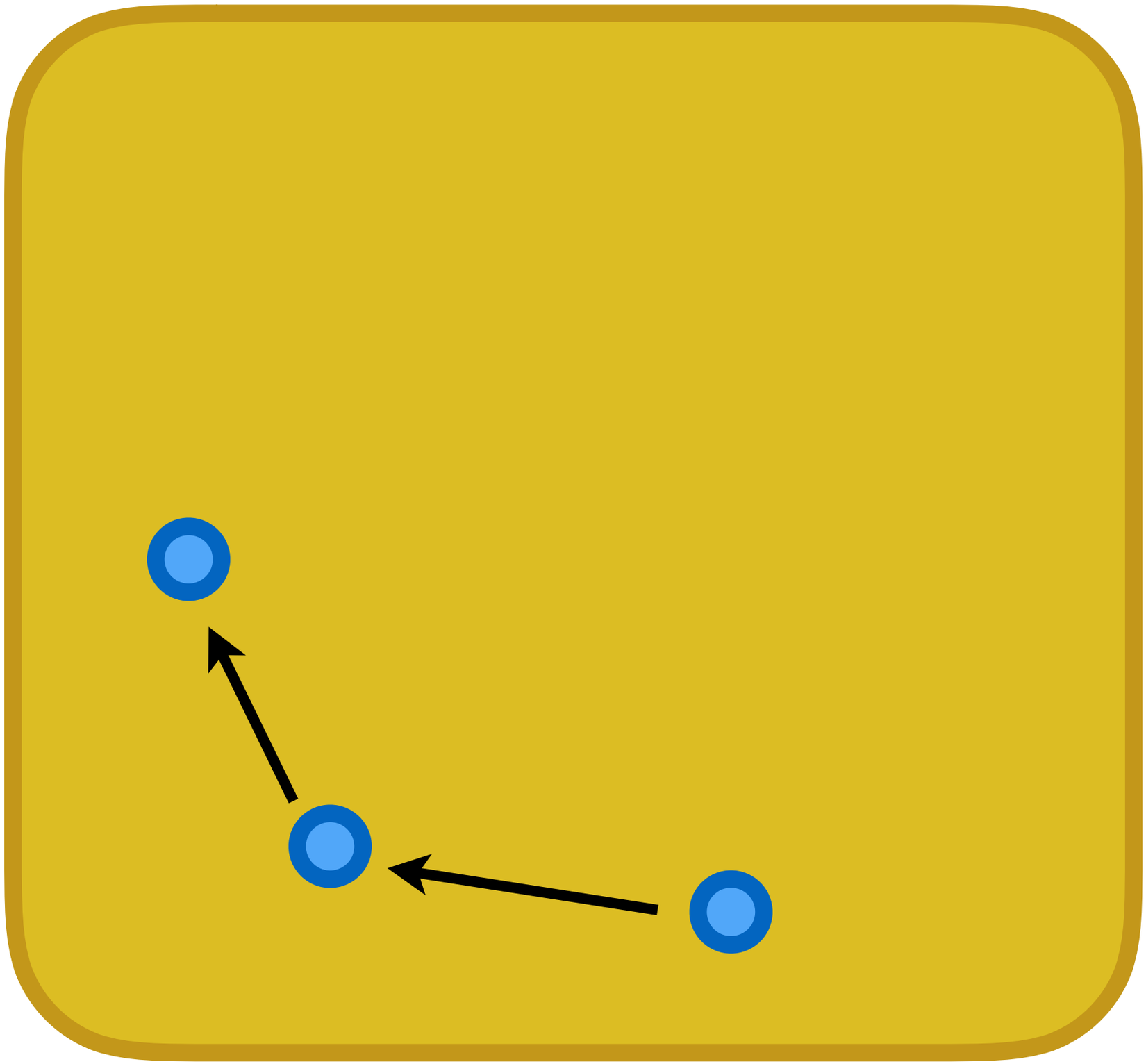


We trust that the semantics describe the behavior of the shim and the network.



Thanks!

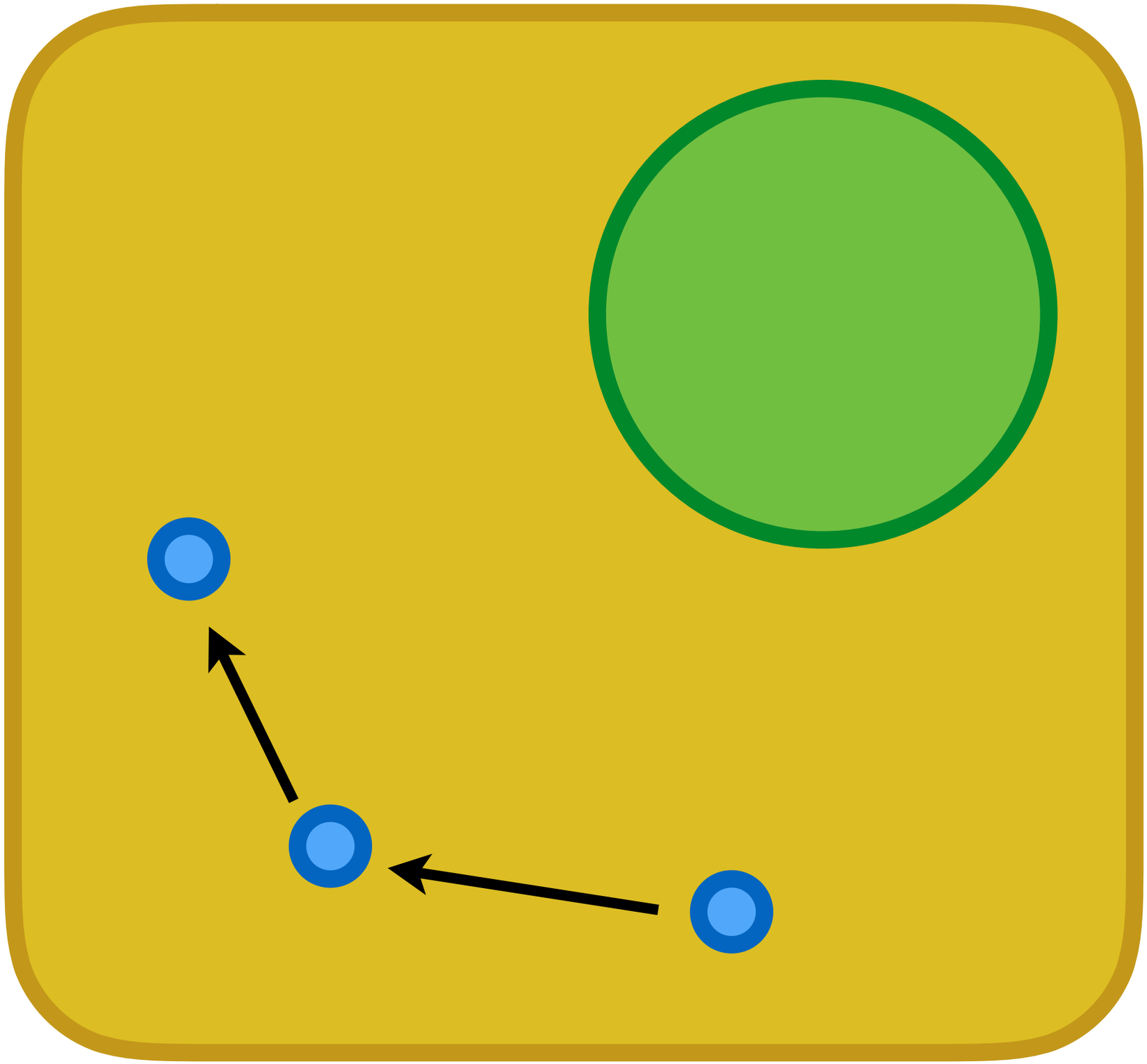
Punctuated safety properties



Reachable

under churn

Punctuated safety properties



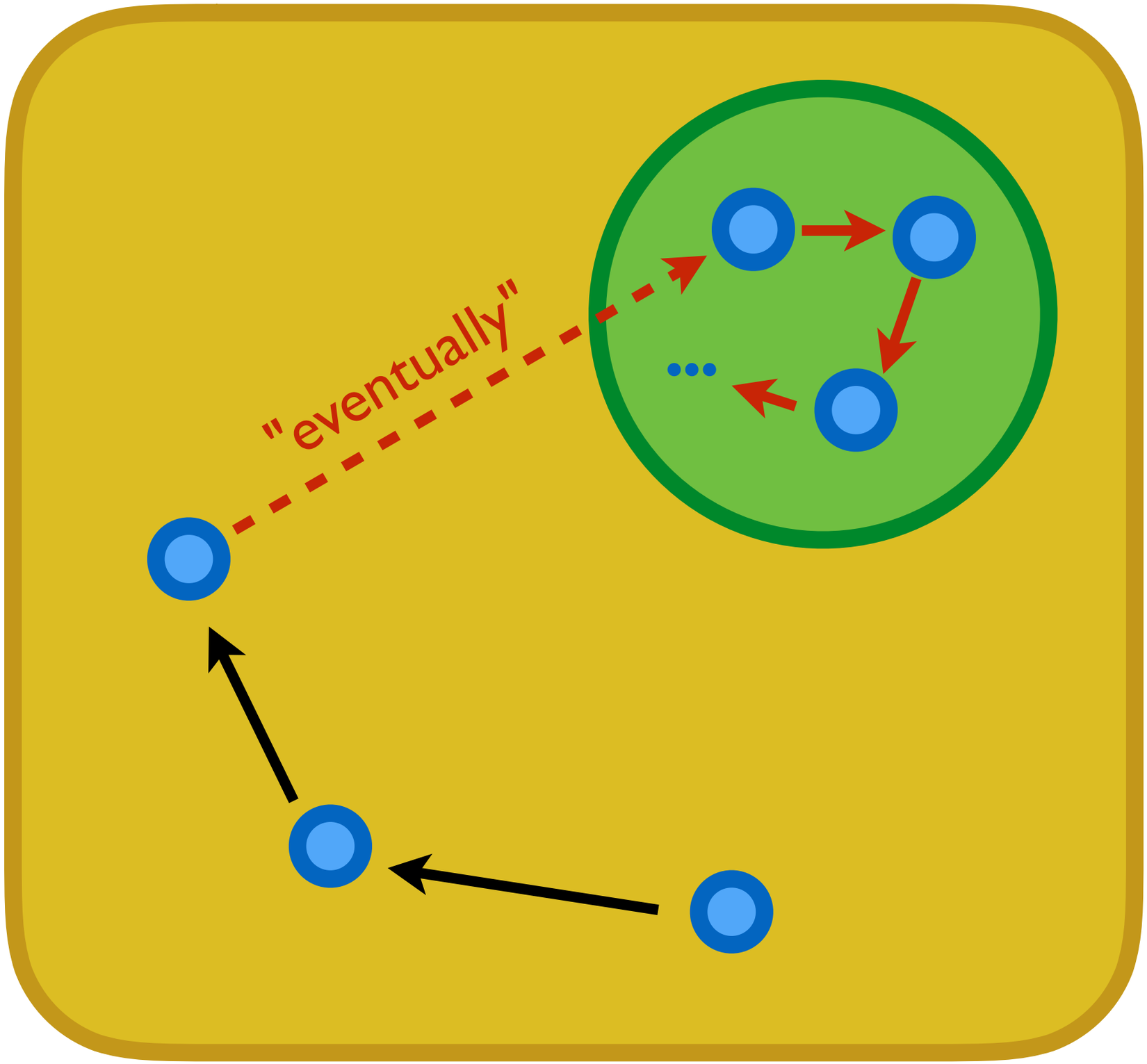
Reachable

under churn

Safety

after churn stops

Punctuated safety properties



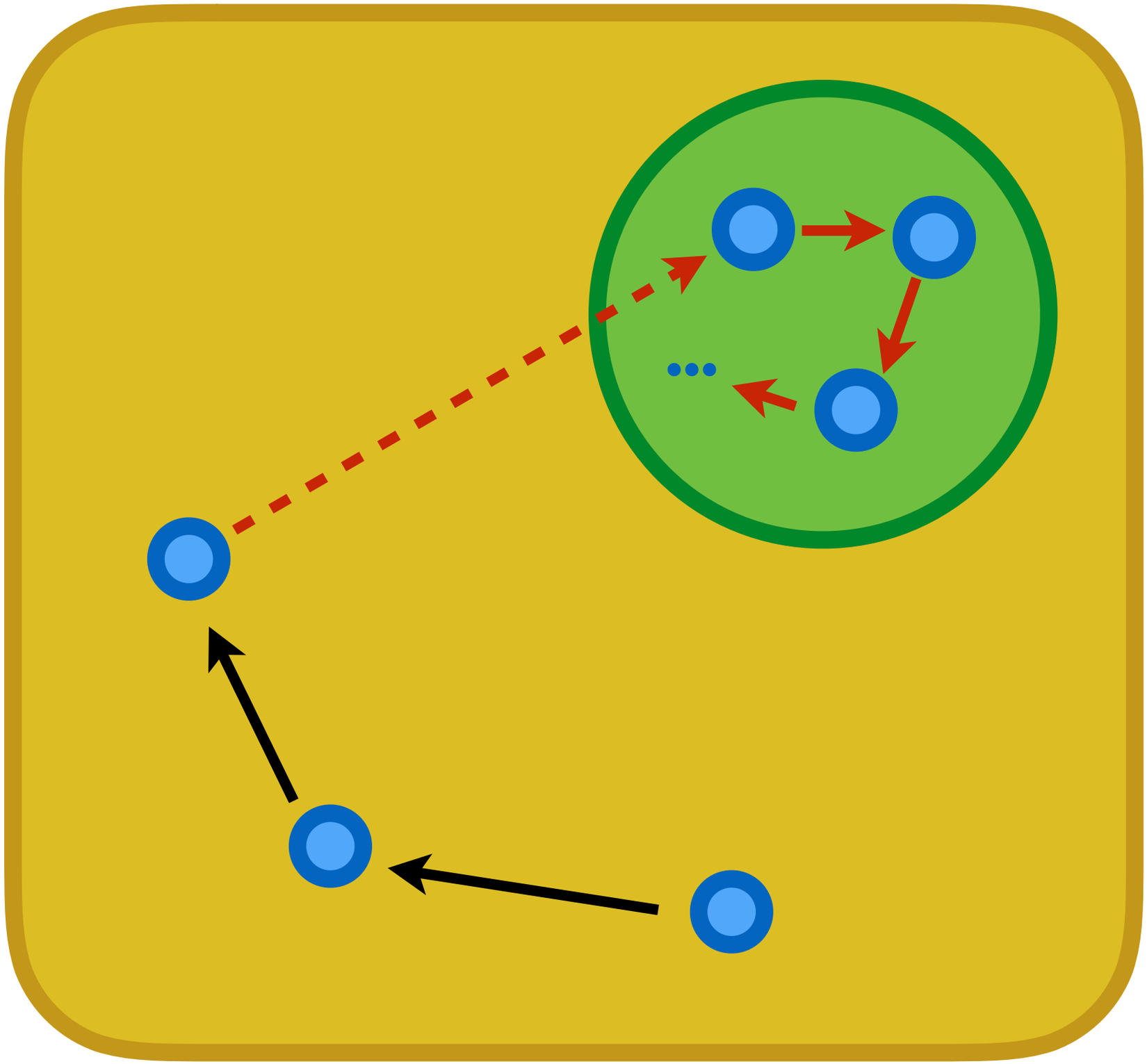
Reachable

under churn (→)

Safety

after churn stops (→)

Punctuated safety properties



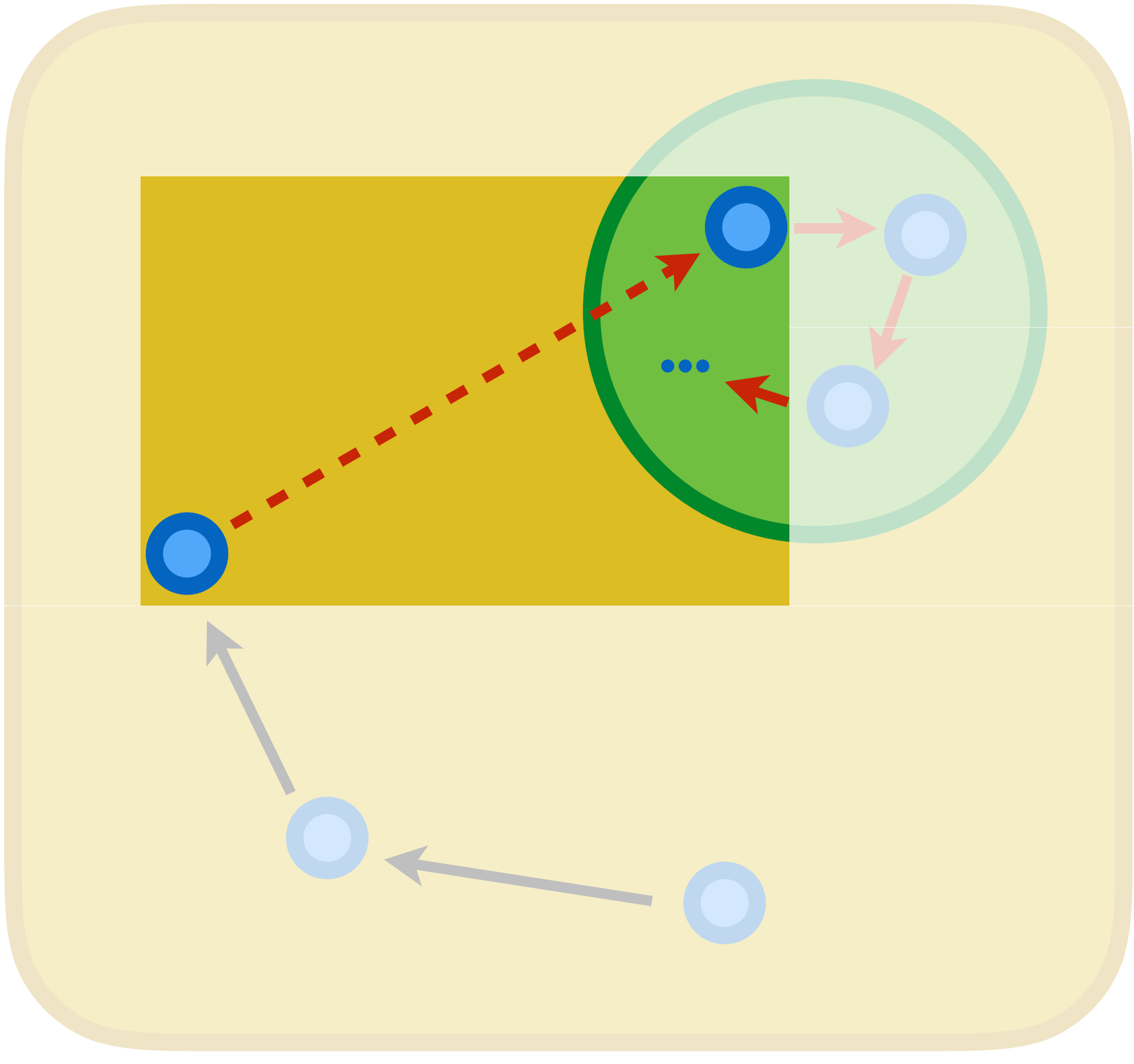
Reachable

under churn (→)

Safety

after churn stops (→)

We don't know how to prove this yet



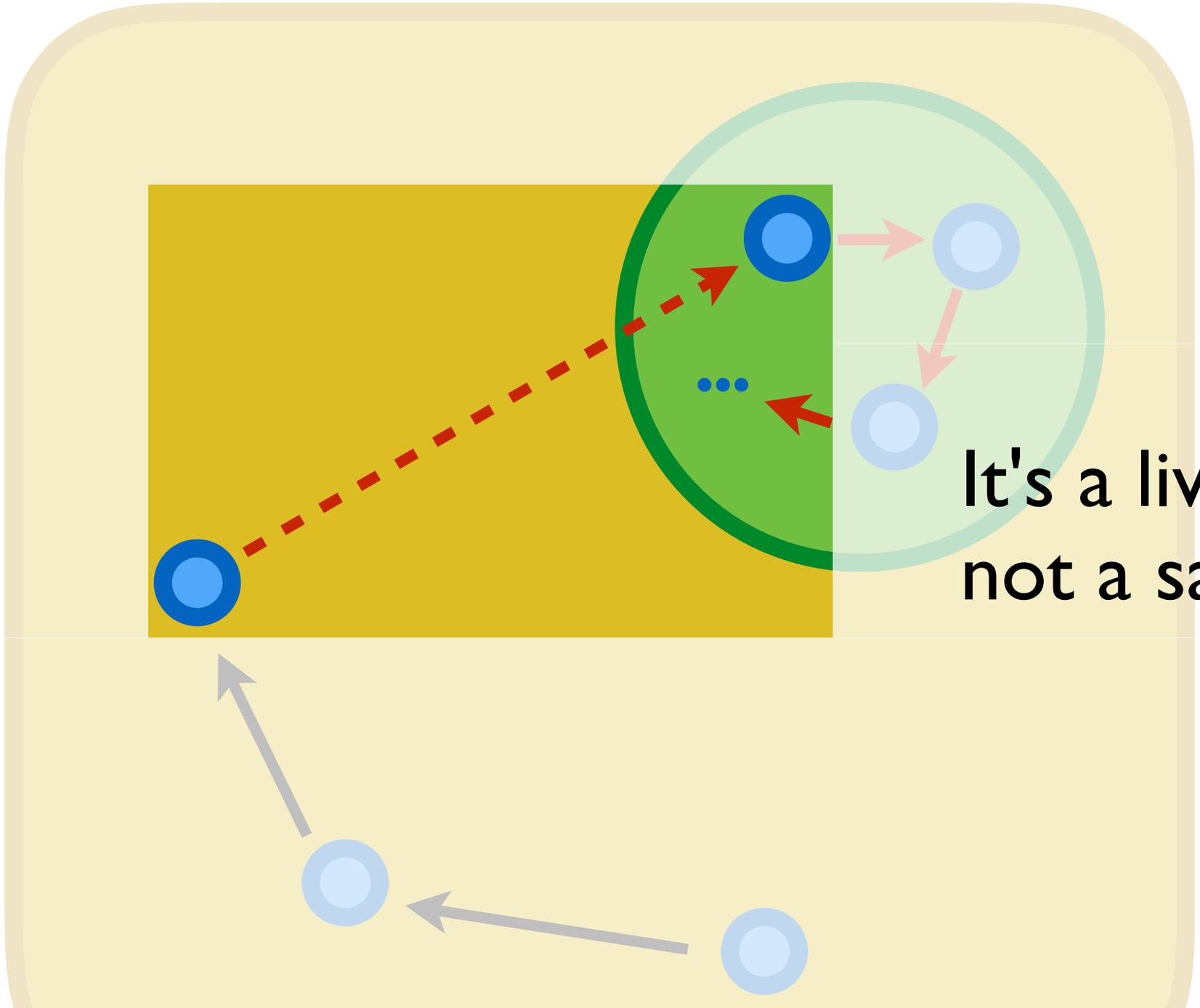
Reachable

under churn (→)

Safety

after churn stops (→)

We don't know how to prove this yet



Reachable

under churn (→)

Safety

after churn stops (→)

It's a liveness argument,
not a safety argument

We need a way to talk about infinite executions: liveness can't be proved with only finite traces.

Representing infinite executions in Coq

```
(* Infinite stream of terms in T *)
CoInductive infseq (T : Type) :=
  Cons : T -> infseq -> infseq.
```

```
(* Stream of system states connected by step *)
CoInductive execution
: infseq (net * label) -> Prop :=
  Cons_exec : forall n n',
    step n n' ->
    execution (Cons n' s) ->
    lb_execution (Cons n (Cons n' s)).
```

Reasoning about executions: linear temporal logic (LTL)

Next P

Always P

Eventually P

...and much, much more!

LTL in Coq

```
Inductive eventually P : infseq T -> Prop :=
 | E0 : forall s,
     P s -> eventually P s
 | E_next : forall x s,
     eventually P s ->
     eventually P (Cons x s).

CoInductive always P : infseq T -> Prop :=
 | Always : forall s,
     P s ->
     always P (tl s) ->
     always P s.
```

InfSeqExt: LTL in Coq

- Extensions to a library by Deng & Monin for doing LTL over infinite (coinductive) streams of events

- Coq source code is on GitHub at `DistributedComponents/InfSeqExt`