Policy-synchronised Deterministic Memory: Reconciling Synchrony & Asynchrony

M. Mendler
University of Bamberg

Synchron 2019
Aussois, 25.11.-28.11.2019

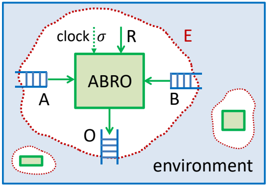
Based on joint work with
J. Aguado, M. Pouzet, P. Roop, R. von Hanxleden [ESOP'18]

Prologue

Synchr. Interfaces - Memory or Signal?

```
module ABRO
inout E : Interface;
input R : Signal;
var x,y : int;
loop
  abort [
     while E.A.empty do pause
    ||
     while E.B.empty do pause
    ];
    repeat
      x = E.A.get;
      y = E.B.get;
      E.O.put = x+y
    until
      (E.A.empty or E.B.empty)
  when R.present
end loop
end module
```

What is the status of *E*?



The interface *E* is both

- memory (*E* is written after being read) and
- signal (*E* is shared with concurrent environment)

Trust Your Semicolon

In traditional synchronous/functional languages shared data structures can only be

- either memory (hence not synchronised)
- or signal (hence no destructive update)

but not both.

When sequential program order is taken prescriptively the shared interface becomes a first-class citizen:

- Sequential Constructiveness (SCL, SCEst, SCCharts) [von Hanxleden et.al. TECS'14, TECS'17]
- Policy-synchronised Memory [Aguado et. al. ESOP'18]

In This Talk ...

... we revisit & extend (a bit) the [ESOP'18] theory of

Policy-synchronised Memory (PSM)

to illustrate how it

- ... permits shared data structures
- ... leads to fixed-point semantics
- ... naturally accommodates asynchrony.

Overview

1. A Simple Core Language & Nondeterministic Semantics
2. Policy Interfaces
3. Policy-conformant Scheduling
4. Macro-step Determinacy
5. Predictions & Fixed points
6. Towards GALS
7. Conclusion

A Simple Core Language

Synchronous Concurrent Language SCoL

P	$::=$	$x = a\ ;\ P$	data action on store
	$\mid$	$\texttt{pause}\ \sigma\ ;\ P$	clock action (wait for tick of σ)
	$\mid$	$\texttt{return}$	termination
	$\mid$	$[P]\sigma$	ignore clock[1]
	$\mid$	$P \parallel P$	parallel composition
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P$	conditional branching
	$\mid$	$\mathrm{rec}\, p.\, P$	recursive closure
	$\mid$	p	process variable

where x is a value variable, a access to shared data (method call), σ a clock, e side-effect-free value expression.

[1] $[P]\sigma$ is the same as $P{\uparrow}\sigma$ in PMC [Andersen & Mendler ESOP'94] and CSA [Cleaveland et.al. CONCUR'97]

Action Structure

$\mathbb{D}$ fixed universal domain of values.

An action structure $\mathcal{S} = (\mathbb{S}, \mathbb{A}, \mathbb{C}, \centerdot, \odot)$ over $\mathbb{D}$ consists of

- global store $\Sigma \in \mathbb{S}$
- data actions $a \in \mathbb{A}$
- clock actions $\sigma \in \mathbb{C}$
- return value $\Sigma \centerdot a \in \mathbb{D}$ of data action $a \in \mathbb{A}$ in store $\Sigma \in \mathbb{S}$
- memory update $\Sigma \odot \alpha \in \mathbb{S}$ for action $\alpha \in \mathbb{A} \cup \mathbb{C}$ and $\Sigma \in \mathbb{S}$.

"Free" Nondeterministic Micro-step Execution:

$$\Sigma \vdash P \xrightarrow{a} \Sigma' \vdash P'$$

defined by the following inductive rules ...

"Free" Nondeterministic Scheduling

$$\frac{\Sigma' = \Sigma \odot a \qquad v = \Sigma \mathbin{.} a}{\Sigma \vdash x = a; P \xrightarrow{a} \Sigma' \vdash P\{v/x\}} \qquad \frac{\Sigma \vdash P \xrightarrow{a} \Sigma' \vdash P'}{\Sigma \vdash [P]\sigma \xrightarrow{a} \Sigma' \vdash [P']\sigma}$$

$$\frac{\Sigma \vdash P \xrightarrow{a} \Sigma' \vdash P'}{\Sigma \vdash P \parallel Q \xrightarrow{a} \Sigma' \vdash P' \parallel Q} \qquad \frac{\Sigma \vdash Q \xrightarrow{a} \Sigma' \vdash Q'}{\Sigma \vdash P \parallel Q \xrightarrow{a} \Sigma' \vdash P \parallel Q'}$$

$$\frac{\Sigma' = \Sigma \odot \sigma}{\Sigma \vdash \mathtt{pause}\, \sigma; P \xrightarrow{\sigma} \Sigma' \vdash P} \qquad \frac{\Sigma' = \Sigma \odot \sigma}{\Sigma \vdash [P]\sigma \xrightarrow{\sigma} \Sigma' \vdash [P]\sigma}$$

$$\frac{\Sigma \vdash P \xrightarrow{\sigma} \Sigma' \vdash P' \qquad \Sigma \vdash Q \xrightarrow{\sigma} \Sigma' \vdash Q'}{\Sigma \vdash P \parallel Q \xrightarrow{\sigma} \Sigma' \vdash P' \parallel Q'}$$

(...omitting the rules for `return`, `rec` $p.\,P$, `if` e `then` P `else` Q)

Policy Interfaces

The Policy Contract

Every shared object is protected by a policy constraining the admissibility and ordering of concurrent accesses to its methods.

- Assumption on Environment: The scheduler is policy conformant. I.e., all executions must satifsy the policy.
- Guarantee by Object: The object evaluation semantics is policy coherent. I.e., concurrent methods are confluent.

Policy Constructiveness (Static Analysis):

- Object implementations are coherent
- Program admits deadlock-free conformant schedule.

Theorem: Objects are coherent + schedule conformant $\Rightarrow$ program execution globally confluent.

Policy-synchronised Memory

Let $\mathcal{S} = (\mathbb{S}, \mathbb{A}, \mathbb{C}, \centerdot, \odot)$ be action structure.

Policy

A policy $\Vdash$ for $\mathcal{S}$ is given by a pair $(\downarrow, \dashrightarrow)$ consisting of

- an admissibility predicate $\Sigma \Vdash \downarrow \alpha$
- a precedence relation $\Sigma \Vdash \alpha \dashrightarrow \beta$ ("α blocks β")

for $\Sigma \in \mathbb{S}$ and $\alpha, \beta \in \mathbb{A} \cup \mathbb{C}$ such that

- $\Sigma \Vdash \alpha \dashrightarrow \beta$ implies $\Sigma \Vdash \downarrow \alpha$ and $\Sigma \Vdash \downarrow \beta$.

Intuition: The policy protects determinacy of $\mathcal{S}$ under concurrent admissible actions, subject to precedence constraints.

Concurrent Independence & Coherence

Concurrent Independence: Actions α, β are concurrently independent, written

$$\Sigma \Vdash \alpha \diamond \beta$$

if both α and β are admissible ($\Sigma \Vdash {\downarrow}\alpha$, $\Sigma \Vdash {\downarrow}\beta$) and none blocks the other ($\Sigma \nVdash \alpha \dashrightarrow \beta$, $\Sigma \nVdash \beta \dashrightarrow \alpha$).

Coherence (Confluence):

The action structure $\mathcal{S}$ is policy coherent if $\Sigma \Vdash \alpha \diamond \beta$ implies

1. $\Sigma \odot \beta \Vdash {\downarrow}\alpha$ and $\Sigma \odot \alpha \Vdash {\downarrow}\beta$
2. $\Sigma \,\centerdot\, \alpha = (\Sigma \odot \beta) \,\centerdot\, \alpha$ and $\Sigma \,\centerdot\, \beta = (\Sigma \odot \alpha) \,\centerdot\, \beta$
3. $\Sigma \odot \alpha \odot \beta = \Sigma \odot \beta \odot \alpha$.

with the last two conditions (2), (3) only for data actions $\alpha, \beta \in \mathbb{A}$.

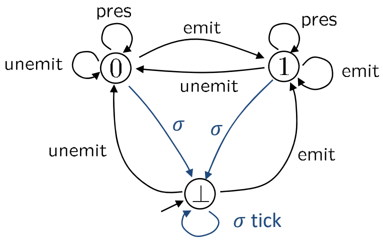
Example – SCEsterel[2] Pure Signals (PSig)

$x \in \text{PSig}$
$\mathbb{S} = \{\bot, 0, 1\}$
$\mathbb{A} = \{\texttt{x.pres}, \texttt{x.emit}, \texttt{x.unemit}\}$
$\mathbb{C} = \{\texttt{x.tick}\}$

```
host class PSig {
   bool pres()
   void emit()
   void unemit()
   void tick()
   policy ⊩psig
}
```

$\Sigma \quad \Vdash_{\text{psig}} \quad \downarrow\alpha \text{ iff } \alpha \neq \texttt{x.pres}$

$\Sigma \quad \Vdash_{\text{psig}} \quad \downarrow\texttt{x.pres} \text{ iff } \Sigma \neq \bot$



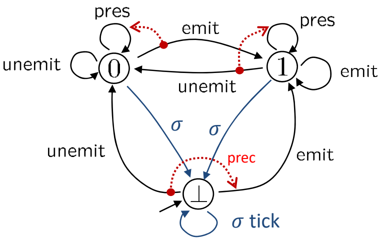
[2] R. v. Hanxleden et. al [TECS'17]

Example – SCEsterel[2] Pure Signals (PSig)

$x \in \text{PSig}$
$\mathbb{S} = \{\bot, 0, 1\}$
$\mathbb{A} = \{\texttt{x.pres}, \texttt{x.emit}, \texttt{x.unemit}\}$
$\mathbb{C} = \{\texttt{x.tick}\}$

```
host class PSig {
   bool pres()
   void emit()
   void unemit()
   void tick()
   policy ⊩psig
}
```

$\Sigma \quad \Vdash_{\text{psig}} \quad \downarrow\alpha \text{ iff } \alpha \neq \texttt{x.pres}$
$\Sigma \quad \Vdash_{\text{psig}} \quad \downarrow\texttt{x.pres} \text{ iff } \Sigma \neq \bot$
$\bot \quad \Vdash_{\text{psig}} \quad \texttt{x.unemit} \dashrightarrow \texttt{x.emit}$
$0 \quad \Vdash_{\text{psig}} \quad \texttt{x.emit} \dashrightarrow \texttt{x.pres}$
$1 \quad \Vdash_{\text{psig}} \quad \texttt{x.unemit} \dashrightarrow \texttt{x.pres}$



[2] R. v. Hanxleden et. al [TECS'17]

Example – SCEsterel[2] Pure Signals (PSig)

$x \in \text{PSig}$
$\mathbb{S} = \{\bot, 0, 1\}$
$\mathbb{A} = \{\texttt{x.pres}, \texttt{x.emit}, \texttt{x.unemit}\}$
$\mathbb{C} = \{\texttt{x.tick}\}$

```
host class PSig {
    bool pres()
    void emit()
    void unemit()
    void tick()
    policy ⊩psig
}
```

$\Sigma \Vdash_{\text{psig}} \downarrow\alpha$ iff $\alpha \neq \texttt{x.pres}$

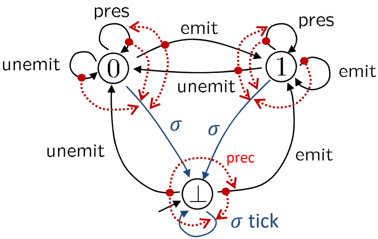
$\Sigma \Vdash_{\text{psig}} \downarrow\texttt{x.pres}$ iff $\Sigma \neq \bot$

$\bot \Vdash_{\text{psig}} \texttt{x.unemit} \dashrightarrow \texttt{x.emit}$

$0 \Vdash_{\text{psig}} \texttt{x.emit} \dashrightarrow \texttt{x.pres}$

$1 \Vdash_{\text{psig}} \texttt{x.unemit} \dashrightarrow \texttt{x.pres}$

$\Sigma \Vdash_{\text{psig}} \alpha \dashrightarrow \texttt{x.tick}$



[2] R. v. Hanxleden et. al [TECS'17]

Policy-conformant Scheduling

$$\Sigma; E \Vdash P \xrightarrow{a} \Sigma' \Vdash P'$$

Enabling (Stability)

Informal: An action α is enabled in store Σ for an environment E, written

$$\Sigma; E \Vdash \downarrow \alpha,$$

if α is admissible in Σ and remains admissible and unblocked along all concurrently independent and admissible actions of E.

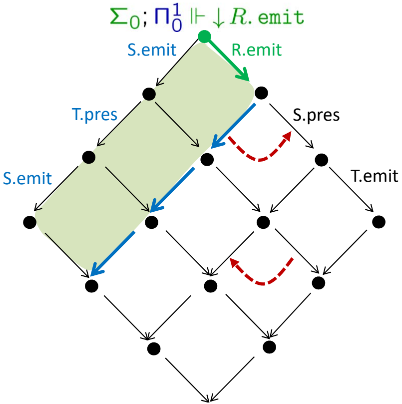
Definition

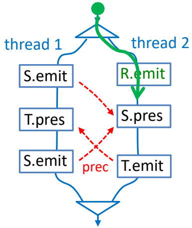
Enabling $\Sigma; E \Vdash \downarrow \alpha$ is the largest relation such that

- $\Sigma \Vdash \downarrow \alpha$

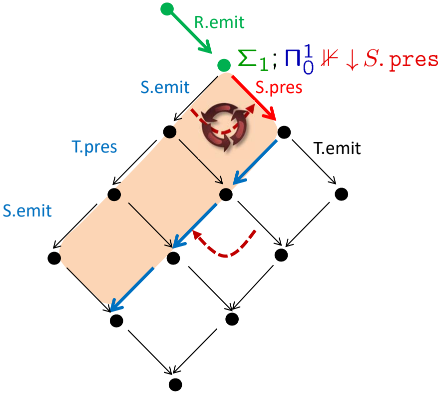
  and for all $\Sigma \vdash E \xrightarrow{\beta} \Sigma' \vdash E'$ and $\Sigma \Vdash \downarrow \beta$ we have
- $\Sigma \nVdash \beta \dashrightarrow \alpha$ and
- if $\Sigma \nVdash \alpha \dashrightarrow \beta$ then $\Sigma'; E' \Vdash \downarrow \alpha$.

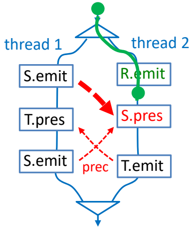
Enabling & Predictive Scheduling



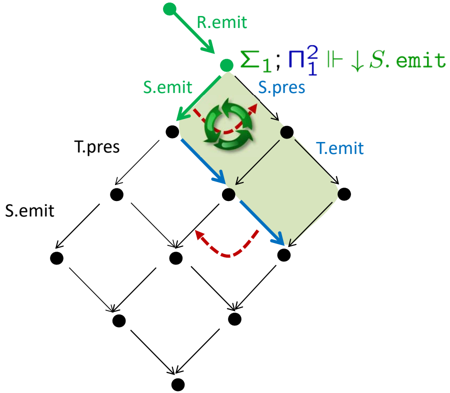


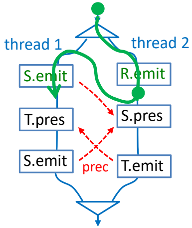
Enabling & Predictive Scheduling



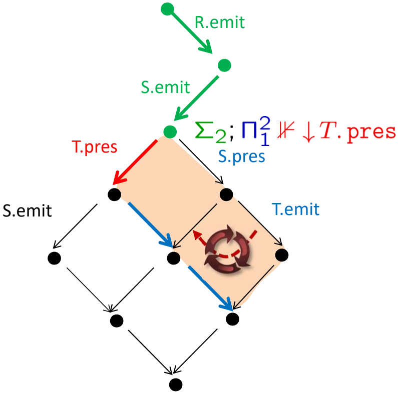


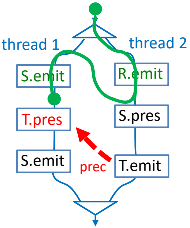
Enabling & Predictive Scheduling



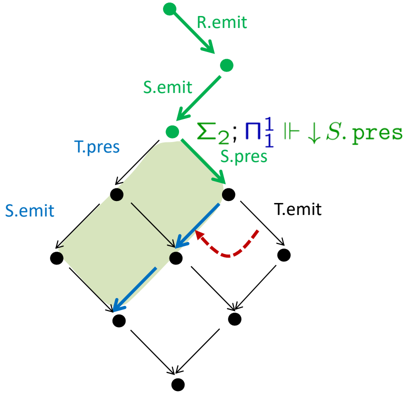


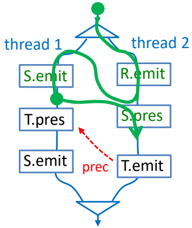
Enabling & Predictive Scheduling



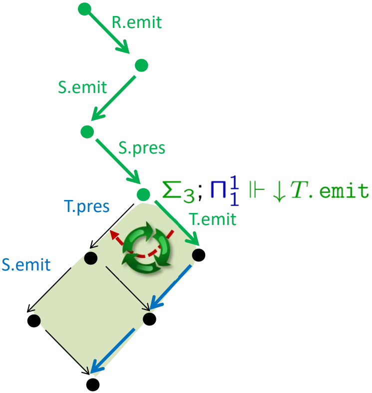


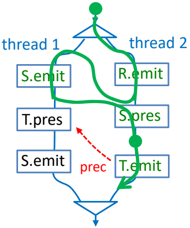
Enabling & Predictive Scheduling



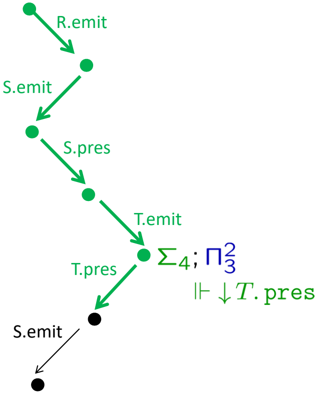


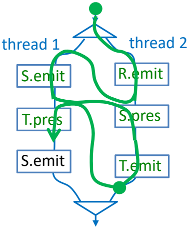
Enabling & Predictive Scheduling



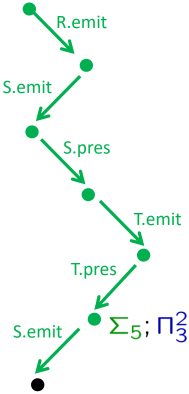


Enabling & Predictive Scheduling

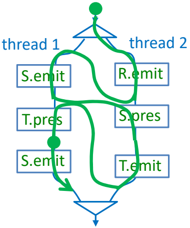




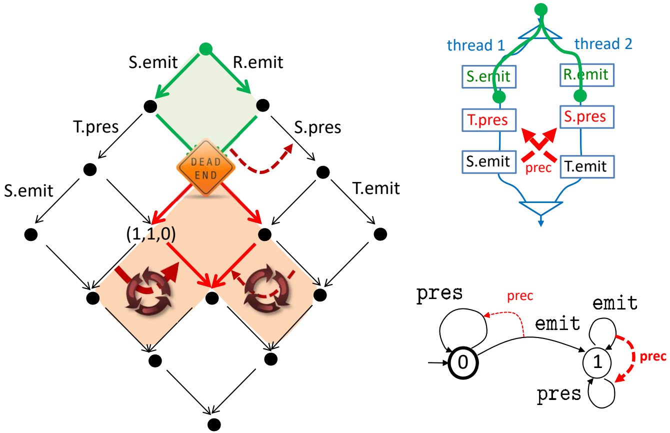
Enabling & Predictive Scheduling



$$\Sigma_5; \Pi_3^2 \Vdash \downarrow S.\mathtt{emit}$$



Enabling & Predictive Scheduling



Macro-step Determinacy

Macro Step

We write $\Sigma \Vdash P \xrightarrow{\alpha} \Sigma' \Vdash P'$ for $\Sigma, \texttt{return} \Vdash P \xrightarrow{\alpha} \Sigma' \Vdash P'$.

(Closed Big-step) Reduction: $\Sigma \Vdash P \Rightarrow \Sigma' \Vdash P'$ is the smallest reflexive relation on configurations such that for $a \in \mathbb{A}$

$$\frac{\Sigma_1 \Vdash P_1 \xrightarrow{a} \Sigma_2 \Vdash P_2 \qquad \Sigma_2 \Vdash P_2 \Rightarrow \Sigma_3 \Vdash P_3}{\Sigma \Vdash P \Rightarrow \Sigma_3 \Vdash P_3}$$

Note: Reductions "$\Rightarrow$" iterate only data actions, no clocks.

Macro Step

A σ-macro step

$$\Sigma \Vdash P \Rrightarrow \Sigma' \Vdash_\sigma P'$$

is a reduction $\Sigma \Vdash P \Rightarrow \Sigma' \Vdash P'$ where the configuration $\Sigma' \Vdash P'$ is σ-pausing, i.e., $\Sigma' \Vdash P' \xrightarrow{\sigma} \Sigma'' \Vdash P''$.

Synchrony

Synchrony:

- An action α is called σ-synchronous (in scope of σ, σ-urgent) if for all Σ, if $\Sigma \Vdash \downarrow \alpha$ and $\Sigma \Vdash \downarrow \sigma$ then $\Sigma \Vdash \alpha \dashrightarrow \sigma$.
- A process P is σ-synchronous if all actions of P are σ-synchronous.
- An object in the store Σ is σ-synchronous if it is accessible only through σ-synchronous actions.

Macro-step Determinacy

Macro Step Determinacy [ESOP'18]

Let P be a σ-synchronous process with

- $\Sigma \Vdash P \Rrightarrow \Sigma_1 \vdash_\sigma P_1$ and $\Sigma \Vdash P \Rrightarrow \Sigma_2 \vdash_\sigma P_2$.

Then $\Sigma_1 = \Sigma_2$ and $P_1 = P_2$.

Macro Step Confluence (Synchron'19 . Conjecture)

Let P be an arbitrary process with

- $\Sigma \Vdash P \Rrightarrow \Sigma_1 \vdash_\sigma P_1$ and $\Sigma \Vdash P \Rrightarrow \Sigma_2 \vdash_\sigma P_2$.

Then

- there exist Σ' and P' such that $\Sigma_i \Vdash P_i \Rrightarrow \Sigma' \Vdash_\sigma P'$
- Each σ-synchronous object has identical state in Σ_1 and Σ_2.

Predictions & Fixed Points

Multi-set Predictions

The σ-prediction counts for each action α how often it is possibly executed, from a given configuration $\Sigma; E$ in the macro step.

Definition

The σ-prediction $can_\sigma(\Sigma; E) : \mathsf{A} \to \mathbb{N}_\infty$ is the smallest multiset such that for all $\alpha \neq \sigma$,

- if $\Sigma \Vdash \downarrow\alpha$ and $\Sigma \vdash E \xrightarrow{\alpha} \Sigma' \vdash E'$,
- then $can_\sigma(\Sigma; E) \geq \alpha \oplus can_\sigma(\Sigma'; E')$.

Note: The counting of actions terminates at the clock tick σ.

Fixed-Point Semantics for SCEsterel PSigs

The policy $\Vdash_{\mathtt{psig}}$ induces the following 4-valued semantics $[\![\Sigma; E]\!] : \mathrm{PSig} \to \{\bot_{0,1} \sqsubseteq \bot_0 \sqsubseteq 0, \bot_{0,1} \sqsubseteq \bot_1 \sqsubseteq 1\}$:

$$[\![\Sigma; E]\!](x) =_{df} \begin{cases} 1 & \text{if } \Sigma\,.\,x = 1 \wedge can_\sigma(\Sigma; E)(\mathtt{x.unemit}) = 0 \\ 0 & \text{if } \Sigma\,.\,x = 0 \wedge can_\sigma(\Sigma; E)(\mathtt{x.emit}) = 0 \\ \bot_0 & \text{if } \Sigma\,.\,x \neq 1 \wedge can_\sigma(\Sigma; E)(\mathtt{x.emit}) = 0 \\ \bot_1 & \text{if } \Sigma\,.\,x \neq 0 \wedge can_\sigma(\Sigma; E)(\mathtt{x.unemit}) = 0 \\ \bot_{0,1} & \text{otherwise} \end{cases}$$

$$\begin{array}{llll} \Sigma; E & \Vdash_{\mathsf{psig}} & \downarrow x.\mathtt{unemit} & \textit{iff} \quad \bot_0 \sqsubseteq [\![\Sigma; E]\!](x) \\ \Sigma; E & \Vdash_{\mathsf{psig}} & \downarrow x.\mathtt{emit} & \textit{iff} \quad \bot_1 \sqsubseteq [\![\Sigma; E]\!](x) \\ \Sigma; E & \Vdash_{\mathsf{psig}} & \downarrow x.\mathtt{pres} & \textit{iff} \quad 0 \sqsubseteq [\![\Sigma; E]\!](x) \text{ or } 1 \sqsubseteq [\![\Sigma; E]\!](x). \end{array}$$

Fixed-Point Semantics for SCEsterel PSigs

Reduction is Inflationary

$\Sigma \Vdash P \Rightarrow \Sigma' \Vdash P'$ implies $[\![\Sigma; P]\!] \sqsubseteq [\![\Sigma'; P']\!]$.

From the initial store $\Sigma_v \centerdot x = v$ for $x \in$ PSig and $v \in \{\bot, 0, 1\}$,

$$\Sigma_v \Vdash P \Rightarrow \Sigma_1 \Vdash P_1 \Rightarrow \Sigma_2 \Vdash P_2 \cdots \Rightarrow \Sigma_n \Vdash P_n$$

converges to $[\![P]\!]_v =_{df} \bigsqcup_i [\![\Sigma_i; P_i]\!]$.

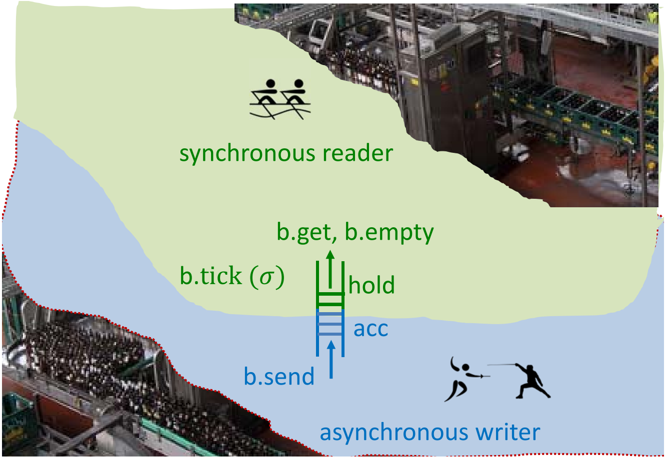
Esterel Semantics

Suppose P does not use x.`unemit` or x.`emit` sequentially after x.`pres`. Then the fixed point $[\![P]\!]_0$ corresponds to the constructive three-valued fixed point semantics of [Berry 2002].

A Policy-view of GALS

What We Are Really Interested In...

The SYNC Buffer

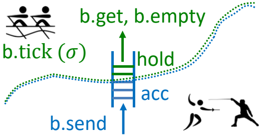


The SYNC Buffer

```
class sync {
private [int] acc, hold = []

bool empty(){ return (hold == []) }

void send(x:int){ acc = acc ++ [x] }

int get(){
 int x;
 if !hold == [] {
   x = head hold;
   hold = tail hold}
 return x }

void tick(){ hold = hold ++ acc }

policy ⊩_sync
}
```

The SYNC Buffer

$C = \{\mathtt{b.tick}\} = \{\sigma\}$
$A = \{\mathtt{b.empty}, \mathtt{b.get}, \mathtt{b.send}\}$
$\Sigma.b = (\mathtt{acc}, \mathtt{hold})$



The policy $\Vdash_{\mathtt{sync}}$ induces the following enabling conditions:

$$\Sigma; E \Vdash_{\mathtt{sync}} \downarrow\mathtt{b.send} \text{ iff always}$$

$$\Sigma; E \Vdash_{\mathtt{sync}} \downarrow\mathtt{b.get} \text{ iff } \mathtt{length}(\Sigma.b.\mathtt{hold}) > 0 \text{ and } can_\sigma(\Sigma; E)(\mathtt{b.get}) = 0$$

$$\Sigma; E \Vdash_{\mathtt{sync}} \downarrow\mathtt{b.empty} \text{ iff } \mathtt{length}(\Sigma.b.\mathtt{hold}) = 0 \text{ or } can_\sigma(\Sigma; E)(\mathtt{b.get}) < \mathtt{length}(\Sigma.b.\mathtt{hold})$$

Note: If `b.empty`, `b.get` are executed by the same thread and non-emptiness is tested before `b.get`, then SYNC is wait-free.

Conclusion

Policy-synchronised Memory

- race-free, determinate sharing of abstract data structures
- synchronous & asynchronous memory accesses for GALS

Towards Practical Application

- Complete experimental implementation of policy-based run-time scheduling in Haskell.
- Develop schedulability analysis and deadlock detection (as in von Hanxleden et al. PLDI'14, Haller et al. SCALA'16).

Extending Theory of Policies

- Develop a (algebraic, logical) policy specification language.
- Extend language to define objects and induce policies from program code (e.g., for SCCharts, Blech).

Related Work

- P. Caspi et al.: *Synchronous Objects with Scheduling Policies: Introducing Safe Shared Memory in Lustre.* LCTES'09.
- R. von Hanxleden et al.: *Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation.* ACM TECS 2014. Deterministic Concurrency: A Clock-Synchronised Shared Memory Approach
- J. Aguado et al.: *Grounding Synchronous Deterministic Concurrency in Sequential Programming*. ESOP'14.
- L. Kuper et al: *Freeze after writing: Quasideterministic parallel programming with LVars.* POPL'14.
- P. Haller et. al.: *Reactive Async: Expressive deterministic concurrency.* SCALA'16.
- J. Aguado et al.: *Deterministic Concurrency: A Clock-Synchronised Shared Memory Approach.* ESOP'18.