A Concurrency Model for seL4

IFIP WG 2.3 Providence

Gerwin Klein (for the TS concurrency team)

Trustworthy Systems @ Data61

May 2018

www.ts.data61.csiro.au

Aim

Aim

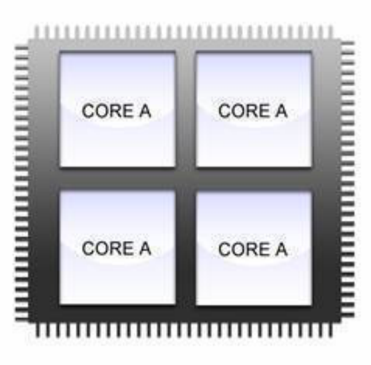
seL4:

- formally verified microkernel
- unicore only
- even embedded devices, phones, etc, are now multicore

How do we get a verified multicore seL4 without redoing everything?

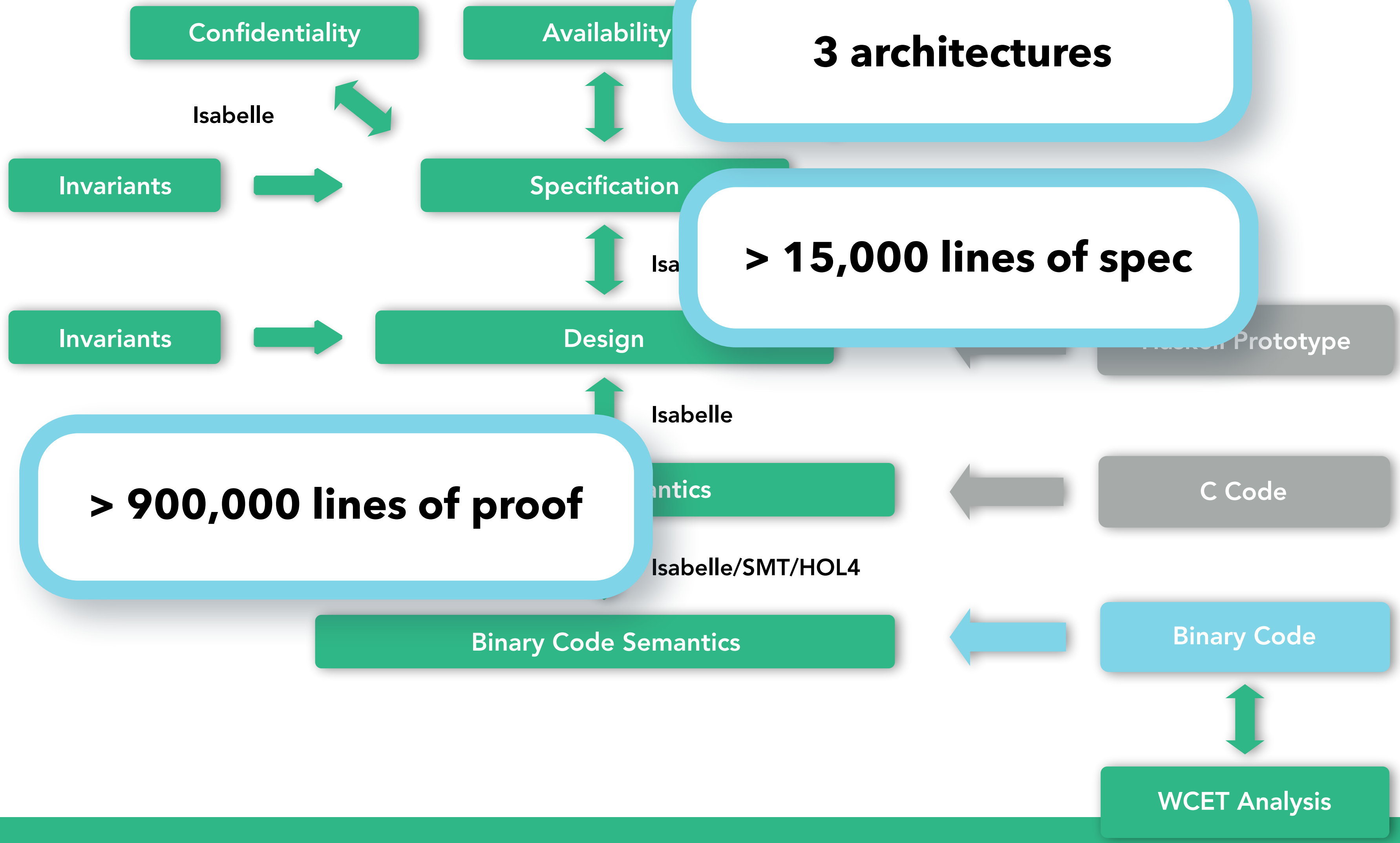
Multicore sel4

- **seL4 multicore design**
  - “mostly” big-lock kernel
  - comparable to Linux fine-grained lock
  - good performance



- **relatively small code changes**
  - per-core data structures
  - lock/synchronisation
  - small number of new syscalls

Current seL4 verification



3 architectures

> 15,000 lines of spec

> 900,000 lines of proof

What about multicore?

- Strongly prefer shallow/algebraic style on abstract levels
  - **better automation, more flexibility**

- Would like to **re-use**:
  - **specification text**
    - lots of existing work, properties, and validation
  - **invariant proof text**
  - **refinement proof text where behaviour is essentially sequential**

A Concurrency Model

Aczel Traces

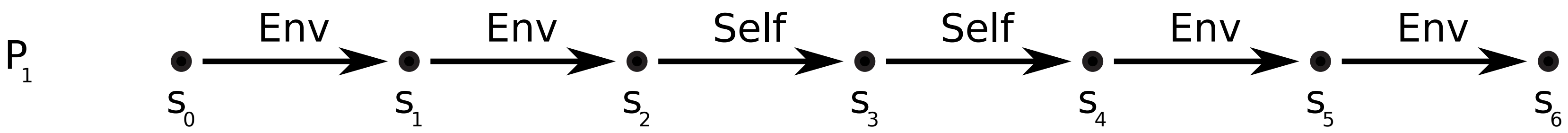
- Popular semantic concurrency model:

process = set of traces

- Aczel traces:

trace = list of state x actor x state
actor = Self | Env

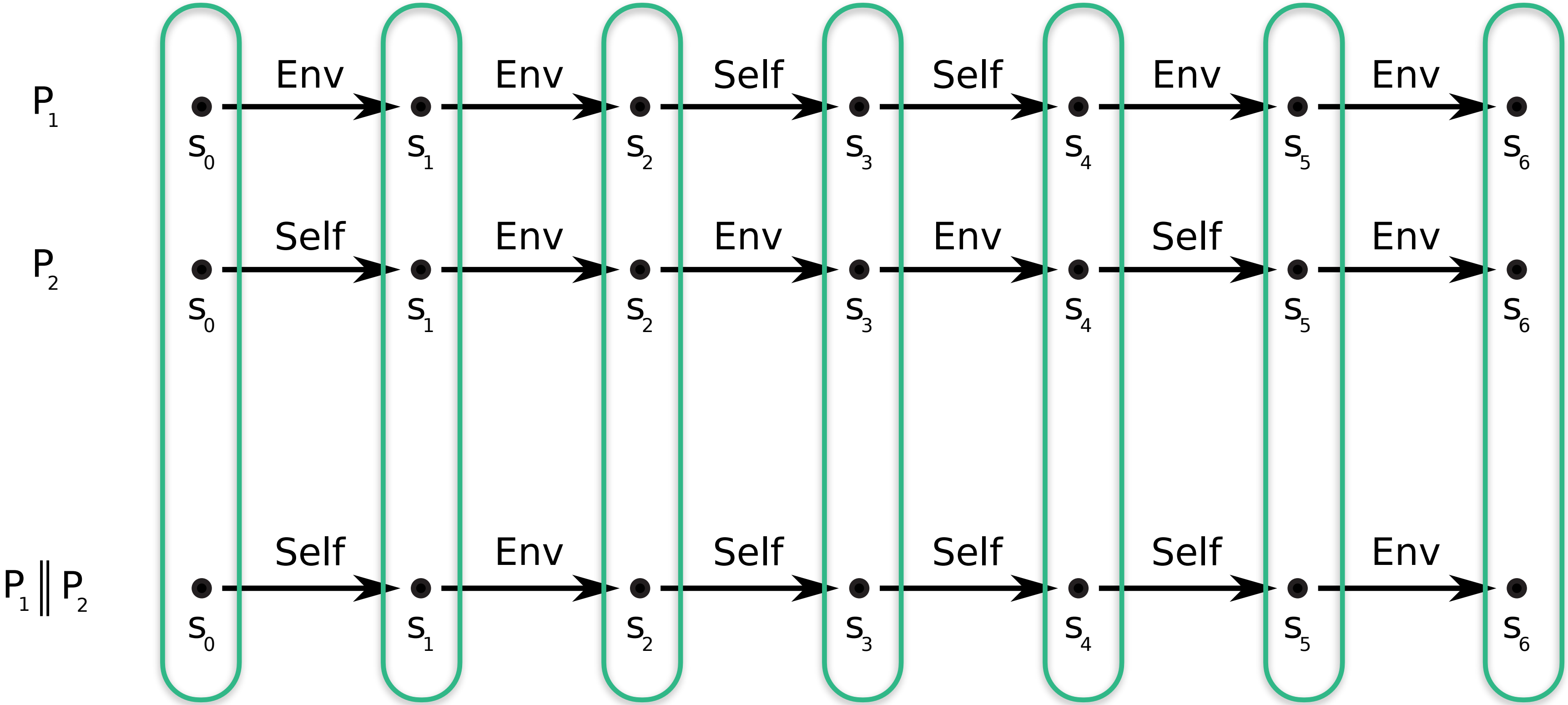
$P_1 \| P_2$



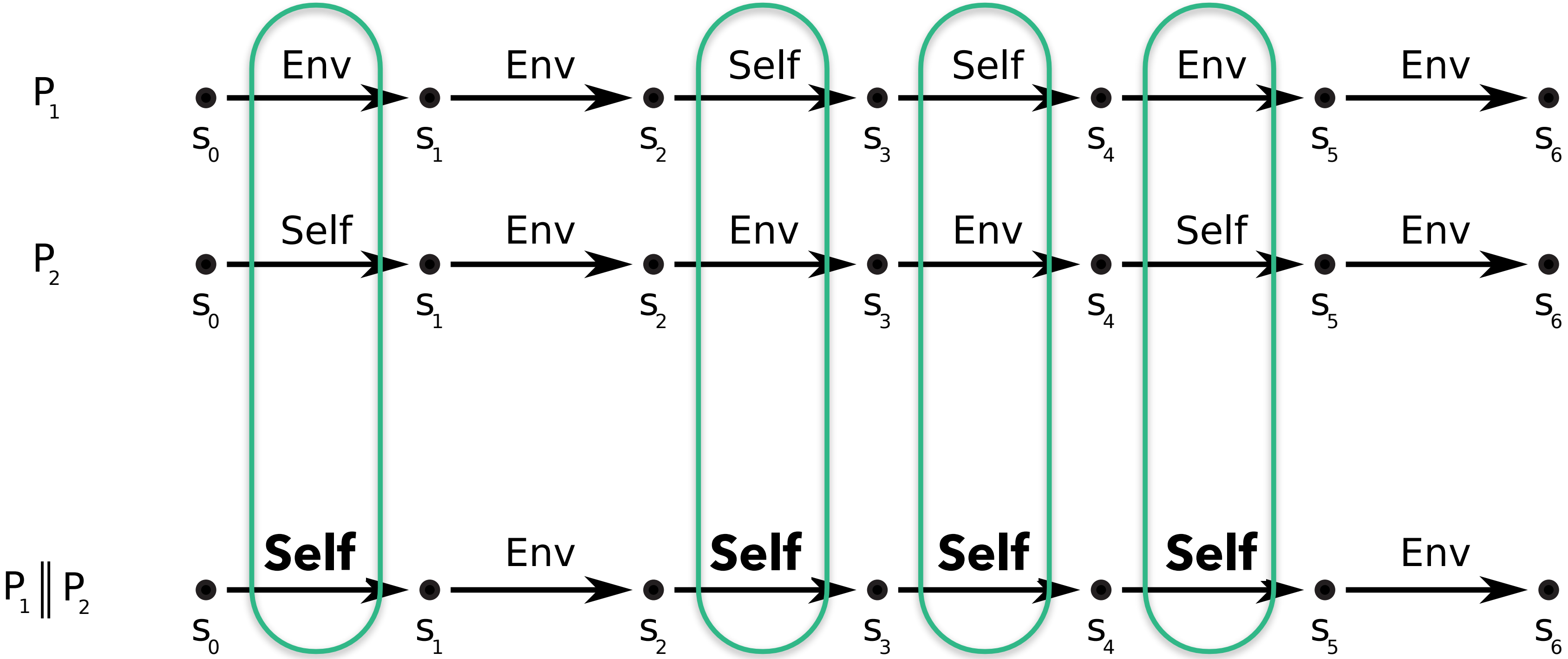
P_1

s_0 Env s_1 Env s_2 Self s_3 Self s_4 Env s_5 Env s_6

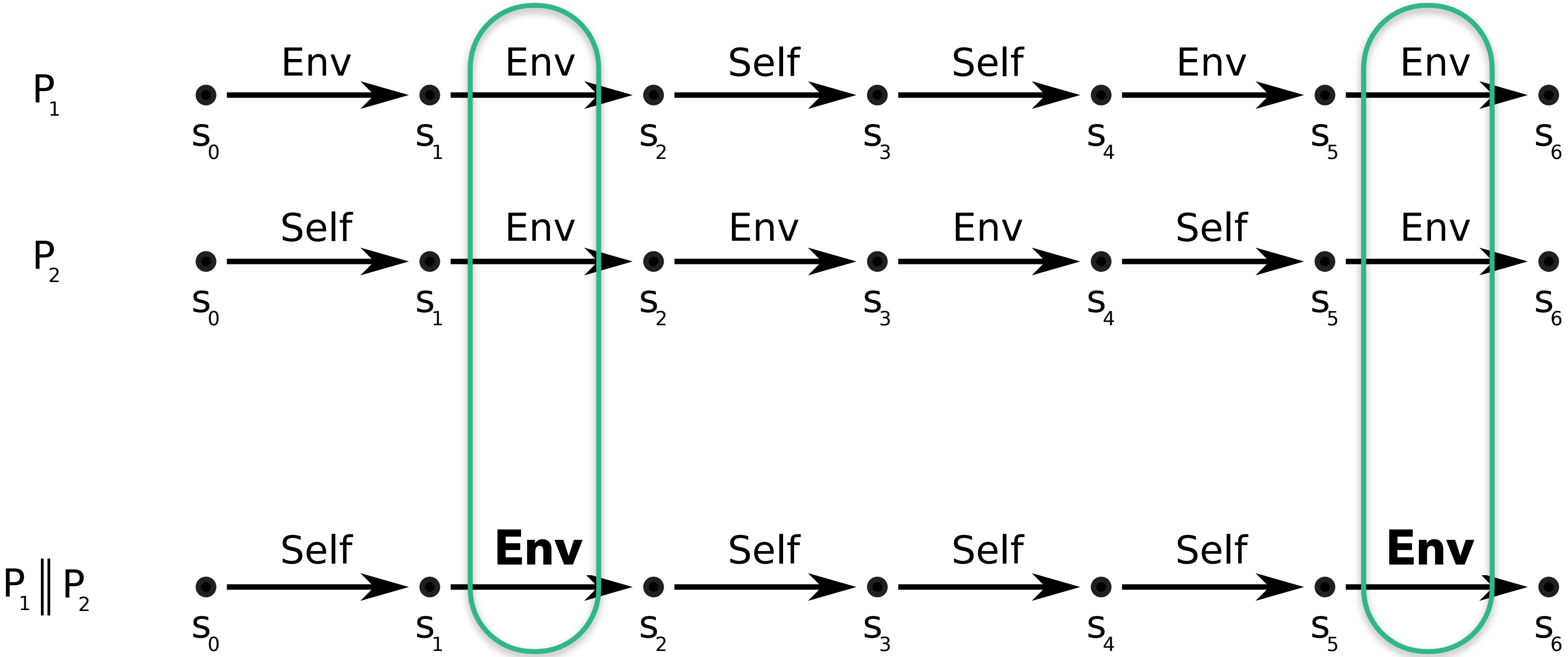
$P_1 \parallel P_2$


P_1
s_0 Env s_1 Env s_2 Self s_3 Self s_4 Env s_5 Env s_6
P_2
s_0 Self s_1 Env s_2 Env s_3 Env s_4 Self s_5 Env s_6
$P_1 \parallel P_2$
s_0 Self s_1 Env s_2 Self s_3 Self s_4 Self s_5 Env s_6

$P_1 \parallel P_2$



$P_1 \parallel P_2$



P_1
Env
Env
Self
Self
Env
Env
s_0 s_1 s_2 s_3 s_4 s_5 s_6
P_2
Self
Env
Env
Env
Self
Env
s_0 s_1 s_2 s_3 s_4 s_5 s_6
$P_1 \parallel P_2$
Self
Env
Self
Self
Self
Env
s_0 s_1 s_2 s_3 s_4 s_5 s_6

seL4 Monad

Sequential
Nondeterministic State Monad with Failure

type (‘a,’s) nondet-monad = ’s ⇒ (‘a x ’s) set x bool

Trace Monad

```
datatype actor =
   Self | Env

datatype ('a,'s) result =
                                        ('a x 's)

type ('a,'s) trace-monad =
  's ⇒ ((actor x 's) list x ('a,'s) result) set
```

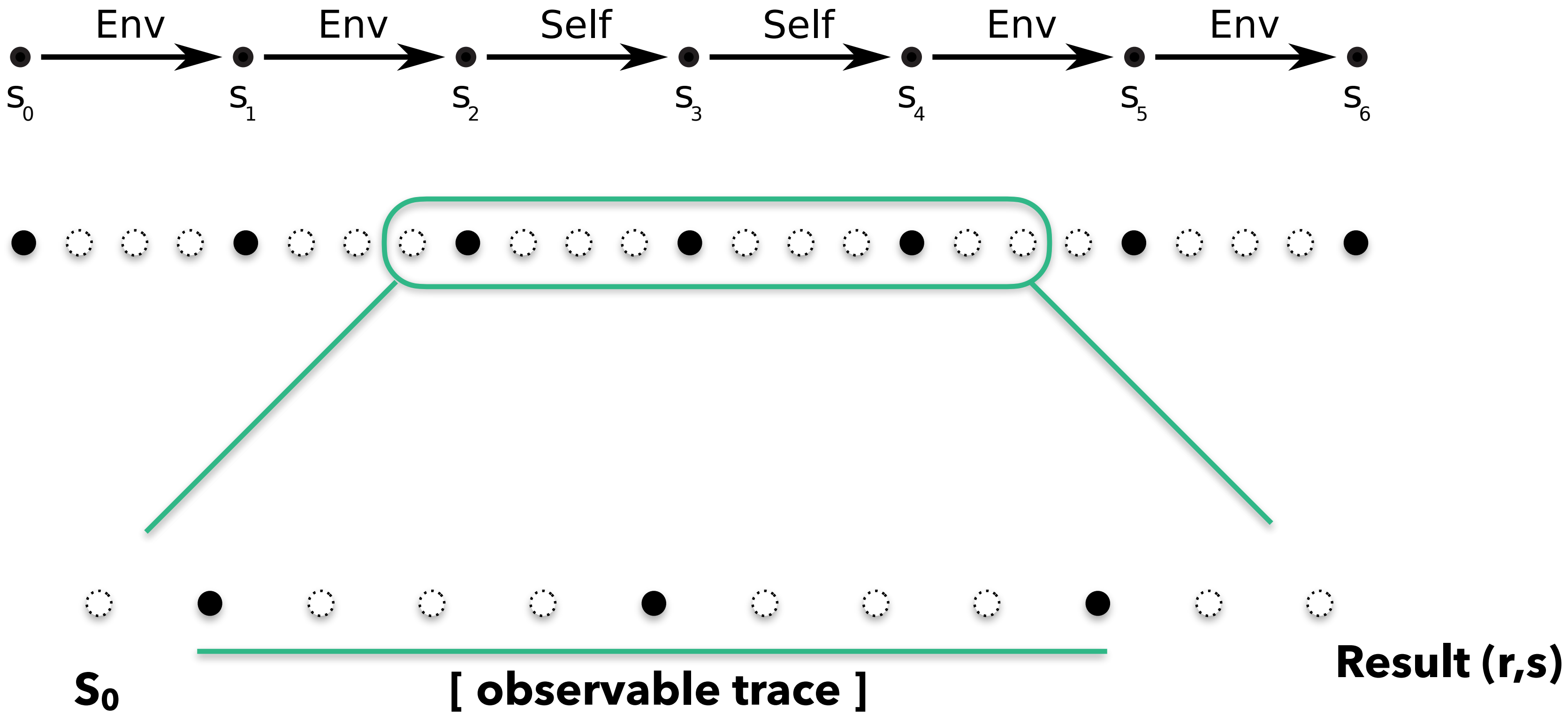
trace of observable actions

set of possible outcomes

start state

return value + final state
(or incomplete/fail)

Example


Env
Env
Self
Self
Env
Env
s_0
s_1
s_2
s_3
s_4
s_5
s_6
s_0
[observable trace]
Result (r,s)

Trace Monad

return :: 'a ⇒ ('a,'s) monad
return a = λs. { ([], Result (a, s)) }

bind :: ('a,'s) monad ⇒ ('a ⇒ ('b,'s) monad) ⇒ ('b,'s) monad
bind f g = λs. ⋃ (xs, r) ∈ f s. case r of
    Failed ⇒ {(xs, Failed)}
  | Incomplete ⇒ {(xs, Incomplete)}
  | Result (rv, s') ⇒ {(ys @ xs, r') | ys r'. (ys, r') ∈ g rv s'}

Trace Monad

The trace monad is a monad:

Lemma

$$\begin{aligned} \mathbf{return}\ x \mathbin{>\!\!>\!\!=} f &= f\ x \\ f \mathbin{>\!\!>\!\!=} \mathbf{return} &= f \\ (m \mathbin{>\!\!>\!\!=} f) \mathbin{>\!\!>\!\!=} g &= m \mathbin{>\!\!>\!\!=} (\lambda x.\ f\ x \mathbin{>\!\!>\!\!=} g) \end{aligned}$$

(**bind f g** written as **f >>= g**)

$P_1 \parallel P_2$

Parallel composition as in Aczel traces:

compatible xs ys =

$|xs| = |ys| \wedge$

$\forall((x, s), (y, s')) \in \text{zip xs ys}.\ s = s' \wedge (x = \text{Env} \vee y = \text{Env})$

merge (x,s) (y,s) =

(if x = Env then y else Self, s)

f || g =

$\lambda s.\ \{\ (zs, rv).\ \exists xs\ ys.\ (xs, rv) \in f\ s \wedge (ys, rv) \in g\ s \wedge$

$\text{compatible xs ys} \wedge zs = \text{map merge (zip xs ys)}\ \}$

State Operations

get :: (‘s,’s) monad
get = λs. { ([], Result (s, s)) }

put :: ‘s ⇒ (unit,’s) monad
put s = λ_. { ([], Result ((), s)) }

fail :: (unit,’s) monad
fail = λs. { ([], Failed) }

assert :: bool ⇒ (unit,’s) monad
assert P = if P then return () else fail

select :: ‘a set ⇒ (‘a,’s) monad
select S = λs. { ([], Result (a, s)) | a. a∈A }

State Laws

do-notation:

do { x ← f; g x } = bind f (λx. g x)

The usual State Monad laws hold:

do { **put** s; **put** s′ } = **put** s′
do { s ← **get**; **put** s } = **return** ()
do { **put** s; **get** } = **do** { **put** s; **return** s }
do { s ← **get**; s′ ← **get**; f s s′ } = **do** { s ← **get**; f s s }

do { **assert** P; **assert** P′ } = **assert** (P ∧ P′)
do { **assert** False; f } = **do** { **assert** False; g }

Trace Steps

Adding observable steps to the trace:

trace x =
  λs. { ([x], Result ((), s)), ([], Incomplete)}

commit =
  do { s ← get; trace (Self, s) }
env-step =
  do { s ← select UNIV; trace (Env, s) }

interference =
  do { commit; star env-step }
interferences =
  star interference

repeat f 0 = return ()
repeat f (Suc n) = do { f; repeat f n }

star f =
  do { n ← select UNIV; repeat f n }

Example

A "normal" concurrent

```
do {
  interference;
  a;
  interference;
  b;
  interference;
  ect..
  interference
}
```

Usual pattern in seL4 spec

```
f y = do {
  interference;
  a;
  x ← b;
  g x;
  a
}
```

```
g x = do {
      c;
      d;
      interference;
      read_shared_mem;
      b;
      c
  }
```

sparse interference

So far

- We can now:

    1. **Replace sequential nondet monad with trace monad**
    2. **Add interference points where necessary**
    3. **Add entry/exit code outside lock**
    4. **Add new kernel features**
    5. **…**
    6. **Profit!**

Reasoning

Rely/Guarantee

rely R ts = env-steps ts ⊆ R

guar G ts = self-steps ts ⊆ G

a-trace s [] = []

a-trace s ((a, s')#ts) = (s, a, s') # a-trace ts

prefix_closed f =
∀s xs. (_ # xs, _) ∈ f s ⟶ (xs, Incomplete) ∈ f s

{P} {R} f {G} {Q} =
prefix-closed f ∧
∀s_0 s. **P** s_0 s ⟶ ∀ts res. (ts, res) ∈ f s ⟶
(rely **R** (a-trace s_0 ts) ⟶ guar **G** (a-trace s_0 ts)) ∧
(∀rv s'. res = Result (rv, s') ⟶ **Q** rv (last s_0 ts) s')

Compositionality

$$\frac{\{P\}\{R\}\ f\ \{G\}\{Q'\} \qquad \forall x.\ \{Q'\ x\}\{R\}\ g\ x\ \{G\}\{Q\}}{\{P\}\{R\}\ f\ {>}{>}{=}\ g\ \{G\}\{Q\}}$$

$$\frac{\{P\}\{R_f\}\ f\ \{G_f\}\{Q\} \qquad \{P\}\{R_g\}\ g\ \{G_g\}\{Q\} \qquad G_g \subseteq R_f \qquad G_f \subseteq R_g}{\{P\}\{R_f \cap R_g\}\ f \,||\, g\ \{G_g \cup G_g\}\{Q\}}$$

Interference

Weakest Precondition Rule:

$$\{\lambda s_0\ s.\ (\forall s'.\ R^*\ s\ s' \longrightarrow Q\ ()\ s'\ s') \wedge G\ s_0\ s\}\{R\}$$
$$\text{interference}$$
$$\{G\}\{Q\}$$

Refinement

Trace subset refinement extends to monad:

$$f \sqsubseteq g \;=\; \forall s.\ g\ s \subseteq f\ s$$

$$\frac{f \sqsubseteq f' \qquad \forall x.\ g\ x \sqsubseteq g'\ x}{f \ \texttt{>>=}\ g \ \sqsubseteq\ f' \ \texttt{>>=}\ g'}$$

$$\frac{f \sqsubseteq f' \qquad g \sqsubseteq g'}{f \parallel g \ \sqsubseteq\ f' \parallel g'}$$

Contextual Data Refinement

Contextual data refinement

$$\{P\}\{R\}\ f\ \sqsubseteq_{I,O,rv}\ \{P'\}\{R'\}\ g$$

- if precondition P and Rely R hold on abstract state,
- and P′ and R′ hold on concrete state,
- and initial states satisfy the internal state relation I,
- then each trace in g matches a trace in f such that:
  - trace states are in observable state relation O
  - result states are in the result relation rv

+ closure conditions (env-closed, enabled, prefix-closed)

Data Refinement

$$\frac{\begin{array}{cc} \{P\}\{R\}\, f \sqsubseteq_{I,O,rv'} \{P'\}\{R'\}\, f' & \forall (x, x') \in rv'.\ \{Q\ x\}\{R\}\, g\ x \sqsubseteq_{I,O,rv} \{Q'\ x'\}\{R'\}\, g'\ x \\ \{P\}\{R\}\ f\ \{UNIV\}\{Q\} & \{P'\}\{R'\}\ g\ \{UNIV\}\{Q'\} \end{array}}{\{P\}\{R\}\, f\ >\!\!>\!= g \quad \sqsubseteq_{I,O,rv} \quad \{P'\}\{R'\}\ f'\ >\!\!>\!= g'}$$

$$\frac{\begin{array}{cc} \{P\}\{R\}\, f \sqsubseteq_{I,O,rv'} \{P'\}\{R'\}\, f' & \{P\}\{R\}\, g\ x \sqsubseteq_{I,O,rv} \{P'\}\{R'\}\, g' \\ \{P\}\{R \cup G_g\}\ f\ \{G_f\}\{UNIV\} & \{P\}\{R \cup G_f\}\ g\ \{G_g\}\{UNIV\} \\ \{P'\}\{R' \cup G_{g'}\}\ f'\ \{G_{f'}\}\{UNIV\} & \{P'\}\{R' \cup G_{f'}\}\ g'\ \{G_{g'}\}\{UNIV\} \end{array}}{\{P\}\{R\}\, f \parallel g \quad \sqsubseteq_{I,O,rv} \quad \{P'\}\{R'\}\ f' \parallel g'}$$

Summary

Summary

- Interference Trace Monad
- Rely/Guarantee
- Refinement
- Compositional in seq + ||
- Contextual in pre + rely

Possible to reuse most of the sequential seL4 proofs.

Probably.

Thank You

Trustworthy Systems
Gerwin Klein

t +61 2 8306 0578
e gerwin.klein@data61.csiro.au
w ts.data61.csiro.au

data61.csiro.au