Spectra: a Framework for Decomposing Logical Atomicity of LTS Composition

Gregory Malecha, Jonas K. Hinrichsen, Gordon Stewart,
Abhishek Anand, Paolo Giarrusso, Yoichi Hirai, **Hai Dang**

BedRock Systems (soon to be BlueRock.io)

Iris Workshop
June 3, 2024

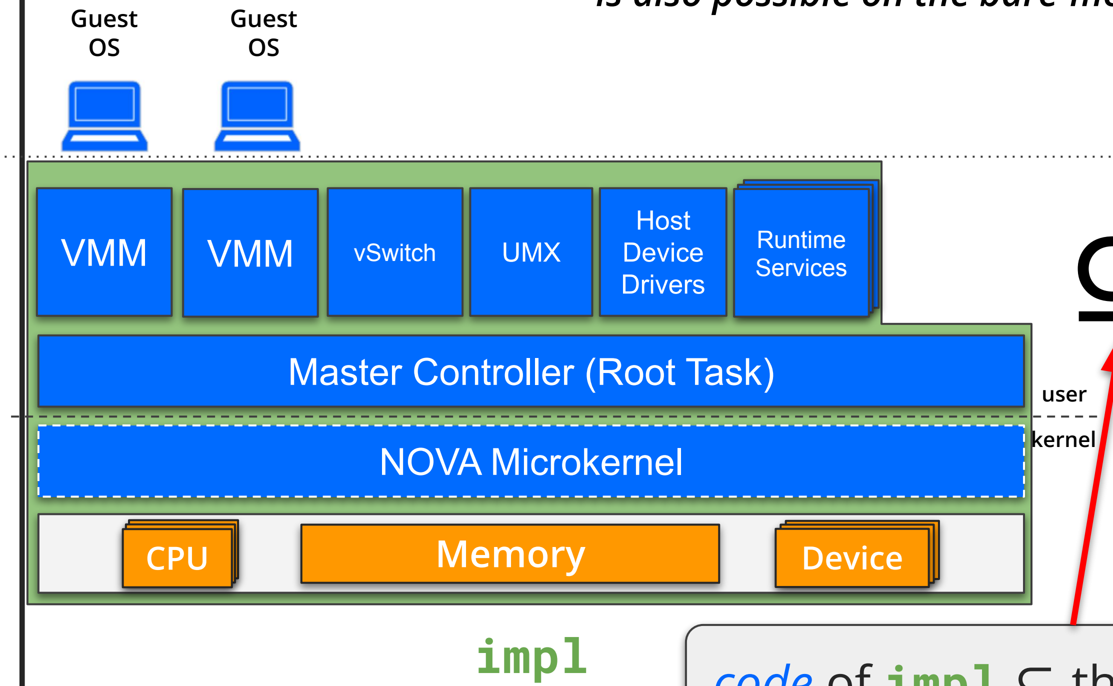
BlueRock's Spectra Framework

- v1 (2020) by G. Malecha and J.K. Hinrichsen
- v2 (2021) by G. Stewart
- further developments by A. Anand, P. Giarrusso, Y. Hirai, H. Dang and others

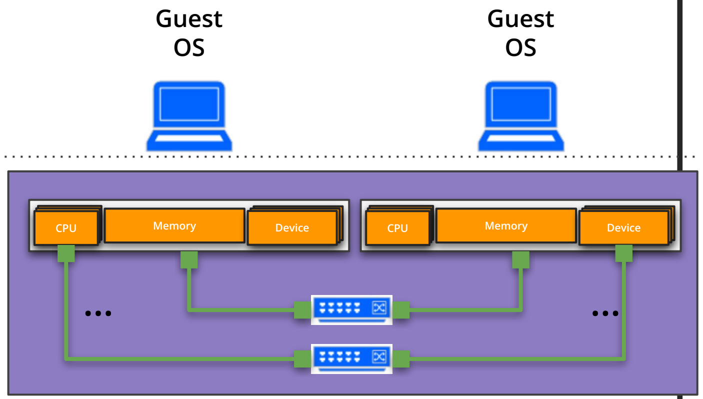
BEDROCK Systems

To prove: refinement

The Bare-metal Property: “any guest behavior possible on the BedRock virtualization stack is also possible on the bare-metal”



$\subseteq$



Physically separate machines with network communications + …

spec

code of impl ⊆ the formal spec

BedRock Systems

Horizontal decomposition of impl ⊆ spec

specification models

this talk focuses on horizontal decomposition for specification models in separation logic

big_spec

=

machine || switch || uart

⊆ ⊆ ⊆

VMM ⊕ vSwitch ⊕ UMX

(C++) (C++) (C++)

=

big_impl

Q: impl ⊆ spec in separation logic, how?

CaReSL-style
[Turon et al. 2013]

```
{ (p,s) * relate (s,σ) }
    (ρ,σ)
{ (ρ',σ'). ∃ (p,s) -> (p',s') *
      (p',s') * relate (s',σ') }
```

triple of impl program

spec program as ghost

⇒ Logic's adequacy gives impl ⊆ spec

ReLoC-TADA-style
[Frumin et al. 2021]

```
⟨ (p,s) * relate (s,σ) ⟩
    (ρ,σ)
⟨ (ρ',σ'). ∃ … * (p',s') * relate (s',σ') ⟩
```

logically atomic triple

BedRock Systems

Decomposing Specification Models

To decompose *atomic updates* (logical atomicity) of specification state

$$\mathbf{AU}\ \langle\ \boxed{\texttt{big\_s}} * \ \dots\ \rangle\ \langle\ \boxed{\texttt{big\_s'}} * \ \dots\ \rangle \qquad \vdash \texttt{wp}\ \texttt{big\_}\sigma$$

into
(this talk)

into
(last year's talk)

$$\mathbf{AU}\ \langle\ \boxed{\texttt{s1}} * \ \dots\ \rangle\ \langle\ \boxed{\texttt{s1'}} * \ \dots\ \rangle \qquad \vdash \texttt{wp}\ \texttt{i1}$$

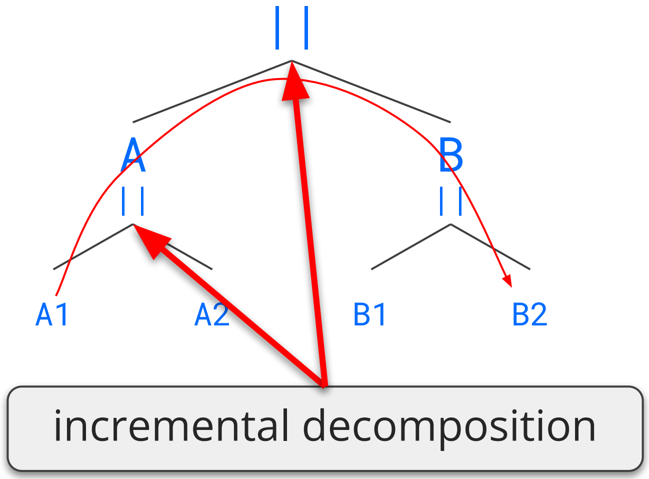
$$\mathbf{AU}\ \langle\ \boxed{\texttt{s2}} * \ \dots\ \rangle\ \langle\ \boxed{\texttt{s2'}} * \ \dots\ \rangle \qquad \vdash \texttt{wp}\ \texttt{i2}$$

$$\mathbf{AU}\ \langle\ \boxed{\texttt{s3}} * \ \dots\ \rangle\ \langle\ \boxed{\texttt{s3'}} * \ \dots\ \rangle \qquad \vdash \texttt{wp}\ \texttt{i3}$$

BedRock Systems

Spec Model Decomposition: A Modular Interface

- Decomposition of not just state, but also steps, of *communicating* specification models
- Hierarchical decomposition, incrementally as needed
- Local specs for communications:
  - one needs not know who is on the other side
  - A1 can talk to B2 without worrying about traversing the tree of decomposition invariants



BedRock Systems

BlueRock's Spectra Framework

- An Iris library to provide separation logic analogs of standard process calculi reasoning principles, on ghost state.
- With separating conjunction and *logically atomic updates*, we can decompose and prove, e.g. parallel composition, communication, and simulation.
- We have applied the framework in several large ongoing verification efforts, including a virtual machine monitor and a virtual network switch.
- Publicly available as part of https://github.com/bedrocksystems/BRiCk

Parallel Composition: State

- `Cs = || Cs[n]` — Cs is an LTS, parallel composed of LTSes `Cs[n]`
- `n : Name` — set of names for each constituent LTS `Cs[n]`
- `Cs.state :=` $\bigtimes_{n}$ `Cs[n].state`
- State (de)composition: $\times$ can be encoded as separating conjunction

$$\boxed{\texttt{Cs.state}} \vdash |\!\Rightarrow_{E}\; \mathop{\ast}_{n} \boxed{\texttt{Cs[n].state}}$$

`Decompose_inv`

Parallel Composition: Semantics

```
                Cs.state[n] ~{τ}~> s’
tau step   ─────────────────────────────────────
          Cs.state ~{τ}~> Cs.state[n := s’]
```

- A step is reflected as an AU

```
AU ⟨ ∃Cs, Cs * … ⟩  ⟨ ∀Cs’, Cs ~{τ}~> Cs’ * Cs’ * … ⟩
⊢
Decompose_inv    -*
AU ⟨ ∃s, s^n * … ⟩  ⟨ ∀s’, s ~{τ}~> s’ * s’^n * … ⟩
```

the direction is due to contravariance

```
(AU s -* WP σ) ⊢ (AU Cs -* WP Cσ)
```

BedRock Systems

Parallel Composition: Communication

- Communications are through events (labels) with input & output arguments
- External communications produce externally visible events

external comm step

```
      Cs.state[n] ~{e}~> s'        is_external(e,ex)
─────────────────────────────────────────────────────────
           Cs.state ~{ex}~> Cs.state[n := s']
```

External Communication as AU entailment

$$\text{external comm step}\quad \frac{\texttt{Cs.state[n] \textasciitilde\{e\}\textasciitilde> s'} \qquad \texttt{is\_external(e,ex)}}{\texttt{Cs.state \textasciitilde\{ex\}\textasciitilde> Cs.state[n := s']}}$$

- A step is reflected as an AU with event

$$\mathbf{AU}\ \langle\ \exists \texttt{Cs},\ \boxed{\texttt{Cs}} * \ldots\ \rangle\quad \langle\ \forall \texttt{Cs'},\ \texttt{Cs} \sim\{\texttt{ex}\}\sim> \texttt{Cs'} * \boxed{\texttt{Cs'}} * \ldots\ \rangle$$

$$\vdash$$

$$\ulcorner\ \texttt{is\_external(e,ex)}\ \urcorner \quad -\!*$$

$$\boxed{\texttt{Decompose\_inv}} \quad -\!*$$

$$\mathbf{AU}\ \langle\ \exists \texttt{s},\ \boxed{\texttt{s}}^{\texttt{n}} * \ldots\ \rangle\quad \langle\ \forall \texttt{s'},\ \texttt{s} \sim\{\texttt{e}\}\sim> \texttt{s'} * \boxed{\texttt{s'}}^{\texttt{n}} * \ldots\ \rangle$$

BedRock Systems

Parallel Composition: Internal Communication

- Internal communications among constituent LTSes result in silent τ events

```
                 Cs.state[n1] ~{e1}~> s1'   Cs.state[n2] ~{e2}~> s2'
                      (e1,e2) is matching (request,response)
 internal    ---------------------------------------------------------------
comm step          Cs.state ~{τ}~> Cs.state[n1 := s1'][n2 := s2']
```

Internal Communications with 2 AUs

```
                 Cs.state[n1] ~{e1}~> s1'      Cs.state[n2] ~{e2}~> s2'
internal              (e1,e2) is matching (request,response)
comm step    ----------------------------------------------------------------
                       Cs ~{τ}~> Cs[n1 := s1'][n2 := s2']
```

- This requires 2 AUs, for e1 (n1) and e2 (n2), that are ***matching*** and ***atomic***

```
AU ⟨ ∃Cs, [Cs] * … ⟩ ⟨ ∀Cs', Cs ~{τ}~> Cs' * [Cs'] * … ⟩
⊢
 ⌜ (e1,e2) matching ⌝    -*
 [Decompose_inv]         -*          ???
AU ⟨ [s1]^n1 * … ⟩ ⟨ … ⟩ * AU ⟨ [s2]^n2 * … ⟩ ⟨ … ⟩
```

BedRock Systems

Internal Communications with 2 AUs

```
                Cs.state[n1] ~{e1}~> s1'        Cs.state[n2] ~{e2}~> s2'
 internal             (e1,e2) is matching (request,response)
 comm step    ─────────────────────────────────────────────────────────────
                         Cs ~{τ}~> Cs[n1 := s1'][n2 := s2']
```

- **Matching** is asymmetric: the *caller* n1 first makes the request (with e1) and then, the *callee* n2 makes the corresponding response (with e2)
- AU of callee ***depends on*** and ***is atomic*** with AU of caller

⇒ AU of callee ***helps*** committing AU of caller

BedRock Systems

Internal Communications as helping AUs

```
AU ⟨ ∃Cs, Cs  * … ⟩  ⟨ ∀Cs', Cs ~{τ}~> Cs' * Cs' * … ⟩
⊢
 ⌜ (e1,e2) matching ⌝     -*
 Decompose_inv            -*
AU ⟨ s1 ^n1 * … ⟩ ⟨ … ⟩   -*
AU ⟨ s2 ^n2 * … ⟩ ⟨ … ⟩
```

Reading this “transport” lemma:

- The caller needs to show AU e1, i.e. shows how it wants to step n1;
- The lemma, by the decomposition, transforms AU e1 to AU e2;
- The callee consumes AU e2, and commits e1 and e2 together.

Parallel Composition (Cs = || Cs[n])

tau step

```
        Cs.state[n] ~{τ}~> s'
----------------------------------------
Cs.state ~{τ}~> Cs.state[n := s']
```

$AU^{Cs}\ \tau \vdash AU^{Cs[n]}\ \tau$

external comm step

```
 Cs.state[n] ~{e}~> s'    is_externa
----------------------------------------
     Cs.state ~{ex}~> Cs.state[n := s']
```

$AU^{Cs}\ ex \vdash AU^{Cs[n]}\ e$

internal comm step

$AU^{Cs}\ \tau \vdash AU^{Cs[n1]}\ e1\ -\!\!*\ AU^{Cs[n2]}\ e2$

```
Cs.state[n1] ~{e1}~> s1'      Cs.state[n2] ~{e2}~> s2'
         (e1,e2) is matching (request,response)
------------------------------------------------------------
          Cs ~{τ}~> Cs[n1 := s1'][n2 := s2']
```

BEDROCK Systems

Actually …

not all AUs are created equal

Due to the asymmetry in communication, and the se

- Updaters: update for τ steps

  ○s ⇛ ○{s'|s ~{τ}-> s'}

- Requesters: AU for request side of communication

  AU ⟨ ○s ∗ … ⟩ ⟨ ○{s'|s ~{e}->

- Committers: AU for response side of communications

  AU ⟨ ●s ∗ … ⟩ ⟨ ●{s'|s ~{e}-> s'} ∗ … ⟩

The decomposition uses the AUTH construction:

- fragmentary elements are given to clients,
- while authoritative ones are maintained by the decomposition.

BEDROCK Systems

Parallel Composition (Cs = || Cs[n])

$\texttt{Committer}^{\texttt{Cs}}\ \texttt{ex} \vdash \texttt{Committer}^{\texttt{Cs[n]}}\ \texttt{e}$
$\texttt{Requester}^{\texttt{Cs[n]}}\ \texttt{e} \vdash \texttt{Requester}^{\texttt{Cs}}\ \texttt{ex}$

external comm step

```
Cs.state[n] ~{e}~> s'      is_e            (e,ex)
---------------------------------------------------
        Cs.state ~{ex}~> Cs.state[n := s']
```

$\texttt{Updater}^{\texttt{Cs}} \vdash \texttt{Updater}^{\texttt{Cs[n]}}$

tau step

```
        Cs.state[n] ~{τ}~> s'
------------------------------------------
Cs.state ~{τ}~> Cs.state[n := s']
```

$\texttt{Updater}^{\texttt{Cs}} \vdash \texttt{Requester}^{\texttt{Cs[n1]}}\ \texttt{e1}\ \texttt{-*}\ \texttt{Committer}^{\texttt{Cs[n2]}}\ \texttt{e2}$

internal comm step

```
Cs.state[n1] ~{e1}~> s1'      Cs.state[n2] ~{e2}~> s2'
       (e1,e2) is matching (request,response)
-------------------------------------------------------------
          Cs ~{τ}~> Cs[n1 := s1'][n2 := s2']
```

BedRock Systems

The Decomposition Theorem

State decomposed

τ steps decomposed

```
○ Cs.state * □ Updater^Cs
⊢ |={E}=>
    [* list] n ∈ Name, ○ Cs.state[n] * □ Updater^Cs[n]
  * □ (∀ n1 ≠ n2, e1, e2. (e1,e2) matching ->
          Requester^Cs[n1] e1 -* Committer^Cs[n2] e2)
  * □ (∀ n, e, ex. is_external(e,ex) ->
          Requester^Cs[n] e -* Requester^Cs ex)
  * □ (∀ n, e, ex. is_external(e,ex) ->
          Committer^Cs ex -* Committer^Cs[n] e)
```

internal comm steps

external request steps

external response steps

BedRock Systems

Together with Refinement Adequacy

$$\begin{array}{l}\texttt{refine\_inv(R,TR)} := \\ \quad \exists\ s\ tr,\ \boxed{\bullet\, s_{init}} * s_{init} \sim\{tr\}\sim> s\ * \\ \quad \texttt{Forall2 R (no\_}\tau\texttt{ TR) (no\_}\tau\texttt{ tr)}\end{array}$$

- instantiate Iris with the **big_impl** LTS
- encode **big_spec** as ghost state
- define a refinement inv that relates traces
- prove WP σ while maintaining the refinement inv

decomposable with this framework

decomposable with WPio

$$\boxed{\circ\, s_{init}} * \texttt{impl\_resources} \vdash \mathrm{WP}\ \sigma_{init}\ \{\ \lambda\_,\ \texttt{False}\ \}$$

Enters separation logic

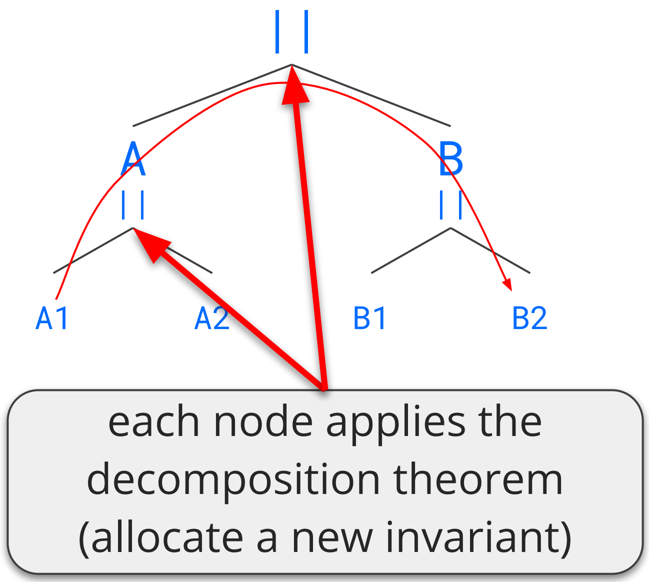
Iris adequacy

$$\texttt{big\_impl}\ \texttt{trace\_refines}^{R}\ \texttt{big\_spec}$$

$$\forall\ \sigma'\ TR,\ \sigma_{init} \sim\{TR\}\sim>^{*} \sigma' \Rightarrow \exists\ s'\ tr,\ s_{init} \sim\{tr\}\sim> s' \wedge \texttt{Forall2 R (no\_}\tau\texttt{ TR) (no\_}\tau\texttt{ tr)}$$

Spectra: A Modular Interface

- Decomposition of state and steps (as AUs)
- Decomposition theorem can be applied multiple times as needed
- Local specs for comm: with Requesters & Committers, one needs not know the other side
    - A CPU issuing a request doesn't know who can serve it (main memory or devices)
- Seamless comm across levels of decomposition
    - A1 can talk to B2 without worrying about traversing the tree of decomposition invariants


||
A
||
B
||
A1
A2
B1
B2
each node applies the decomposition theorem (allocate a new invariant)

BedRock Systems

Conclusions

- A BI-general library to reflect LTS (de)composition into AUs on ghost state
- Publicly available as part of https://github.com/bedrocksystems/BRiCk
- Technical points not mentioned in this talk:
  - hiding
  - masks
  - the decomposition invariant
  - non-determinism (set of states)
  - simulation
- Limitations:
  - Dynamic topology
  - Stateful communication
  - Multi-party sync communication

Updaters

- Updates to make any τ steps
- Proven by clients, consumed by the decomposition

```
Updater lts γ : PROP :=
  □ ∀ (sset : propset lts.State)
      (_ : ∃ s, s ∈ sset)
      (_ : tau_safe lts sset),
      ○ sset γ ={E}=∗
      ○{[s'|∃ s, s ∈ sset ∧ s ~{τ}~>lts s']} γ .
```

sset is non-empty

sset can make τ steps

set of states
τ-reachable from sset

Requesters

- Updates to initiate external communication request steps
- Proven by clients, transported by the decomposition

```
Requester lts γ (EV : propset lts.(Event)) (Q : lts.(Event) -> PROP) : PROP :=
  AC ⟨ ∃ sset, ○ sset^γ ∗ [| ∃s, s ∈ sset |] ∗
                [| ∀s ∈ sset, e ∈ STEP, ∃s', s ~{e}~>lts s' |] ⟩
     Eo∖E, Ei
     ⟨ ∀ e ∈ STEP, ○ {[s'|∃s ∈ sset,s ~{e}~>lts s']}^γ ,
      COMM Q e ⟩
```

sset is non-empty

sset can make EV-steps

set of states
e-reachable from sset

local (private)
post-condition

BedRock Systems

Committers

- Updates to make external communication response steps
- Transported by the decomposition from requesters, consumed by clients

```
Committer lts γ (EV : propset lts.(Event)) (Q : lts.(Event) -> PROP) : PROP :=
  AU ⟨ ∃ sset, ● sset γ ⟩
    Eo, Eo∖E
    ⟨ ∀ sset', e ∈ STEP, [| ∃s, s ∈ sset' |] ∗
      [| ∀ s' ∈ sset', ∃s ∈ sset, s ~{e}~>lts s' |] ∗ ● sset' γ ,
     COMM Q e ⟩
```

set of states
e-reachable from sset

Requesters & Committers

caller picks a set STEP of events to step

```
Requester :=
  AC ⟨ ∃sset, ┊○sset┊γ  ∗ [| ∃s, s ∈ sset |] ∗
                [| ∀s ∈ sset, e ∈ STEP, ∃s', s ~{e}~>lts s' |] ⟩
     Eo∖E, Ei
    ⟨ ∀e ∈ STEP,┊○{[s'|∃s ∈ sset,s ~{e}~>lts s']}┊γ ,
      COMM Q e ⟩
```

callee picks a concrete event e in the set STEP

```
Committer :=
  AU ⟨ ∃sset, ┊● sset┊γ  ⟩
     Eo, Eo∖E
    ⟨ ∀sset', e ∈ STEP, [| ∃s, s ∈ sset' |] ∗
       [| ∀s' ∈ sset', ∃s ∈ sset, s ~{e}~>lts s' |] ∗ ┊● sset'┊γ  ,
      COMM Q e ⟩
```

BedRock Systems

Requesters & Committers

```
Requester :=
  AC 〈 ∃sset, ○sset^γ  * [| ∃s, s ∈ sset |] *
                [| ∀s ∈ sset, e ∈ STEP, ∃s', s ~{e
     Eo∖E, Ei
     〈 ∀e ∈ STEP, ○{[s'|∃s ∈ sset, s ~{e}~>_lts s']}  ,
      COMM Q e 〉

Committer :=
  AU 〈 ∃sset, ●sset^γ 〉
     Eo, Eo∖E
     〈 ∀sset', e ∈ STEP, [| ∃s, s ∈ sset' |] *
       [| ∀s' ∈ sset', ∃s ∈ sset, s ~{e}~>_lts s' |] *● sset'^γ    ,
       COMM Q e 〉
```

caller provides an update with caller's frag

the decomposition ties that update to caller's auth

the decomposition ties caller's auth to callee's auth

callee consumes this update using callee's frag

When callee commits, all updates for caller, callee, and the composed are committed

BedRock Systems

The Decomposition Invariant

Decompose_inv (γ : gname) (γs : Name -> gname) : PROP :=

  ∃ sset,

    [∗list] n ∈ Name, $\boxed{\bullet\ \texttt{sset n}}^{(\gamma s\ n)}$

  ∗ $\boxed{\circ\ \texttt{sset}}^{\gamma}$

  ∗ [| ∃s, s ∈ sset |].

Recall the decomposition theorem: the decomposition

- consumes the frag of the whole (γ)
- produces the frags of the components (γs n)

BEDROCK Systems

Simulation also as AU entailment

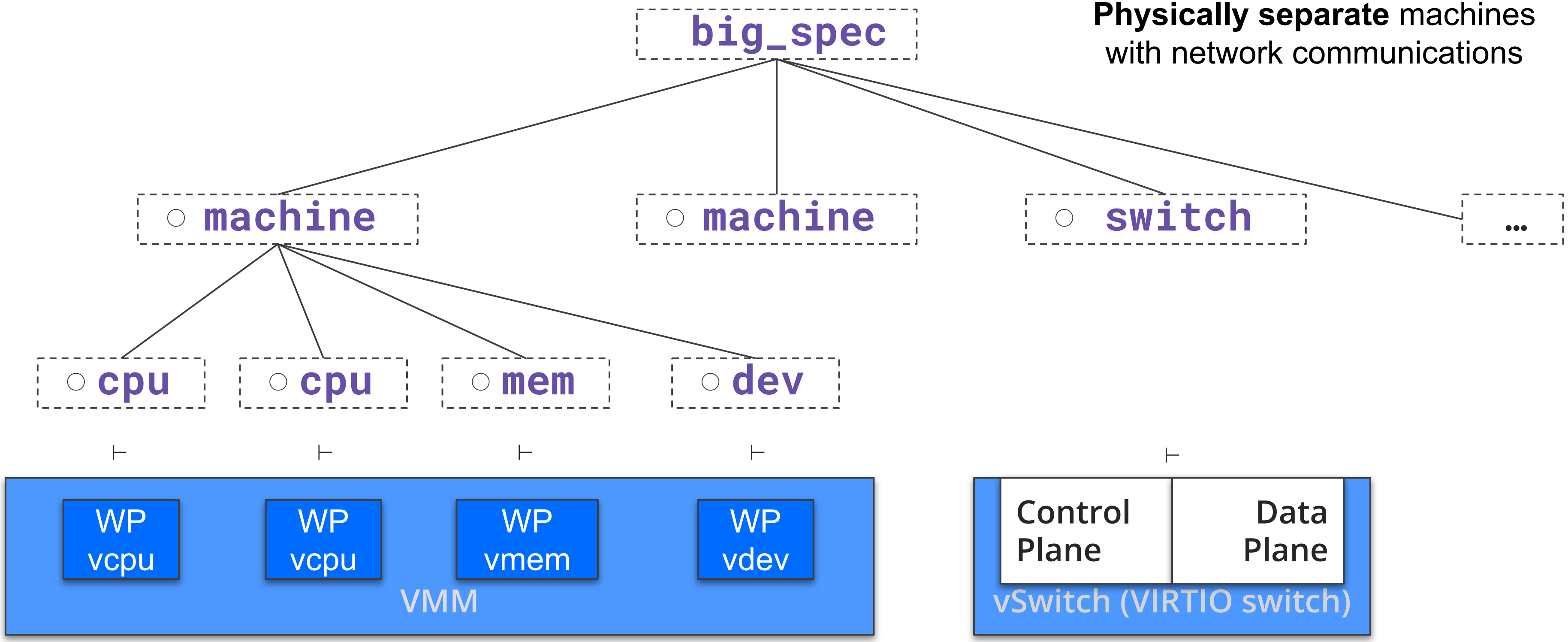
$$\textbf{AU}\ \langle\ \exists \texttt{Cs}.\ \boxed{\texttt{Cs}}\ *\ \ldots\ \rangle\ \texttt{ex}\ \langle\ \forall \texttt{Cs}'.\ \texttt{Cs}\ \texttt{\textasciitilde\{ex\}\textasciitilde>}\ \texttt{Cs}'\ *\ \boxed{\texttt{Cs}'}\ *\ \ldots\ \rangle$$

$$\vdash$$

$$\textbf{AU}\ \langle\ \exists \texttt{s}.\ \boxed{\texttt{s}}^{n}\ *\ \ldots\ \rangle\ \texttt{e}\ \langle\ \forall \texttt{s}'.\ \texttt{s}\ \texttt{\textasciitilde\{e\}\textasciitilde>}\ \texttt{s}'\ *\ \boxed{\texttt{s}'}^{n}\ *\ \ldots\ \rangle$$

Application at BedRock

Physically separate machines
with network communications



BedRock Systems