Decomposing end-to-end refinement proofs

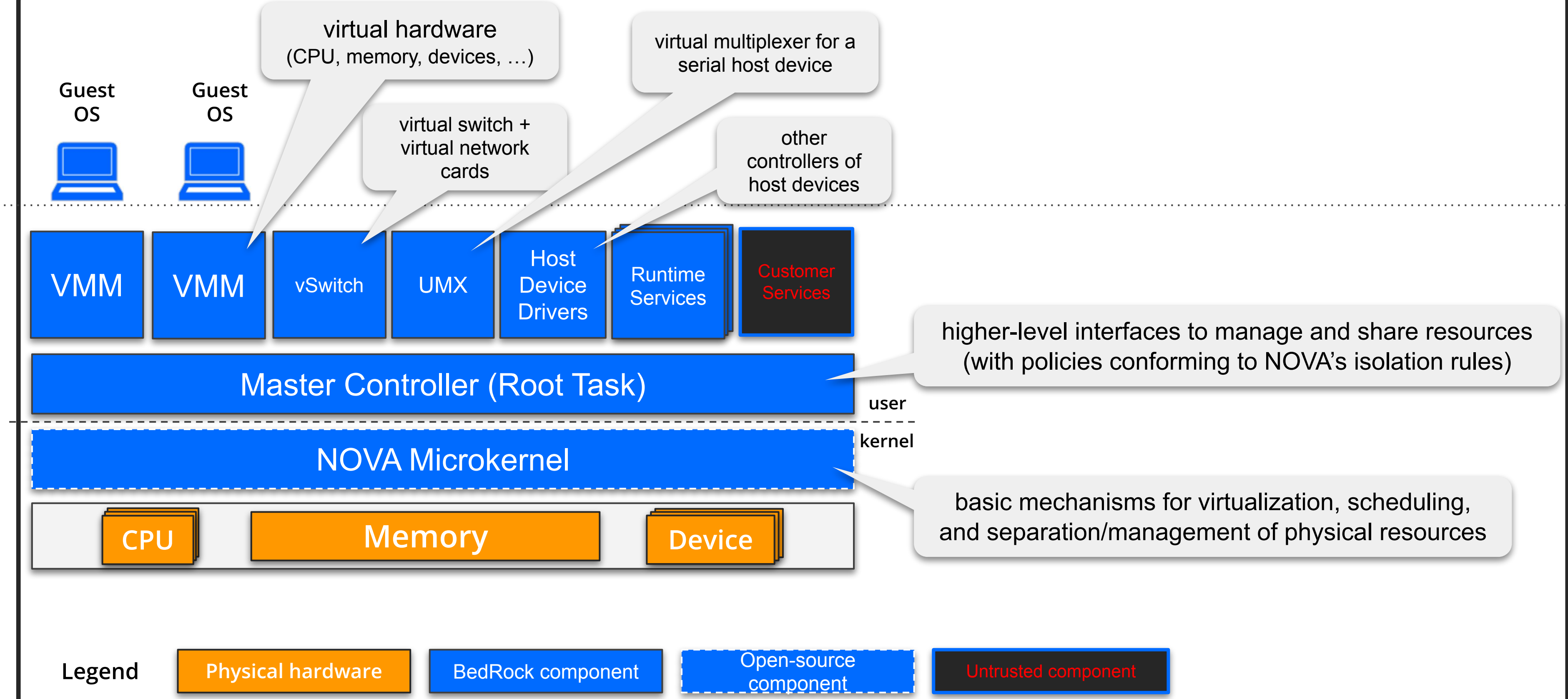
across multiple semantics within separation logics

Hai Dang, David Swasey, Gregory Malecha, Gordon Stewart, Abhishek Anand, and others

Iris Workshop
May 24, 2023

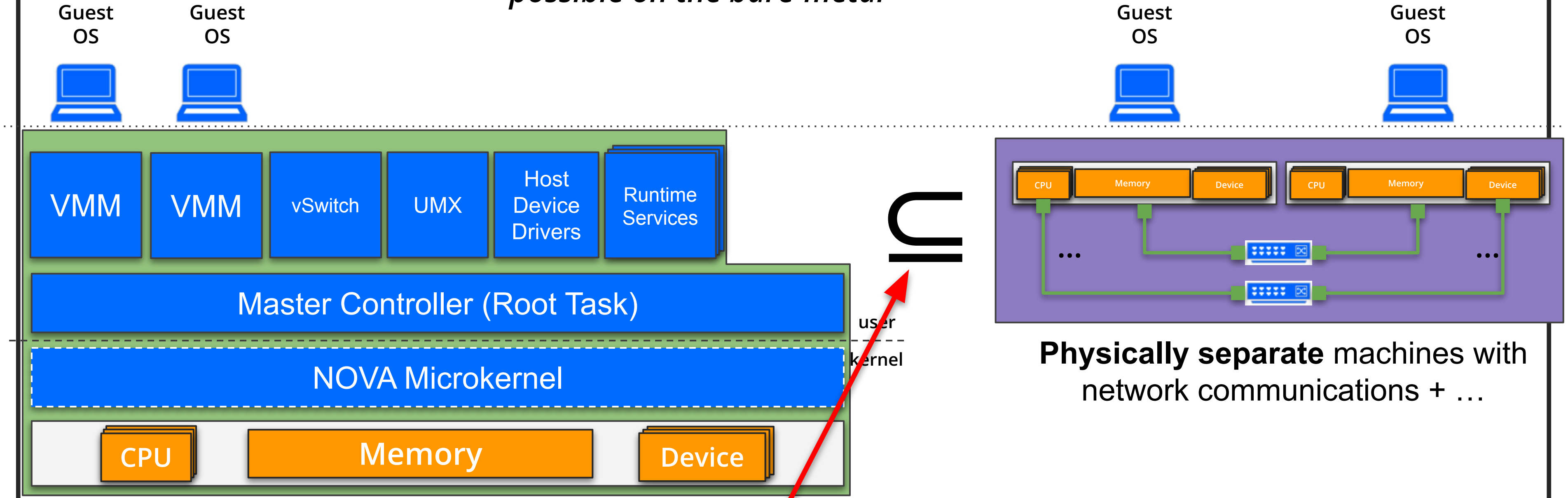
BedRock Systems

BedRock virtualization stack


Guest OS
Guest OS
virtual hardware (CPU, memory, devices, …)
virtual switch + virtual network cards
virtual multiplexer for a serial host device
other controllers of host devices
VMM
VMM
vSwitch
UMX
Host Device Drivers
Runtime Services
Customer Services
higher-level interfaces to manage and share resources (with policies conforming to NOVA's isolation rules)
Master Controller (Root Task)
user
kernel
NOVA Microkernel
basic mechanisms for virtualization, scheduling, and separation/management of physical resources
CPU
Memory
Device
Legend
Physical hardware
BedRock component
Open-source component
Untrusted component

To prove: end-to-end refinement

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



Physically separate machines with network communications + …

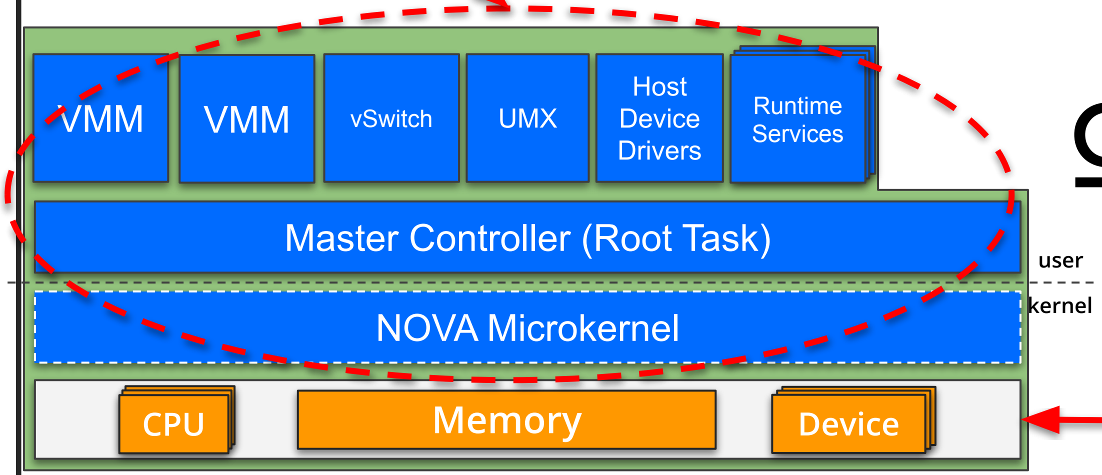
end-to-end: *compiled binary code* of `impl` ⊆ the formal `spec`

BEDROCK Systems

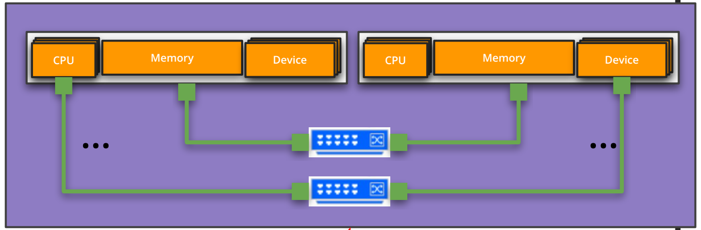
Challenge: heterogeneous semantics

multiple language abstractions

`machine <-> ASM <-> C++`



$\subseteq$



reconfigurable multiple components

each with its own semantics

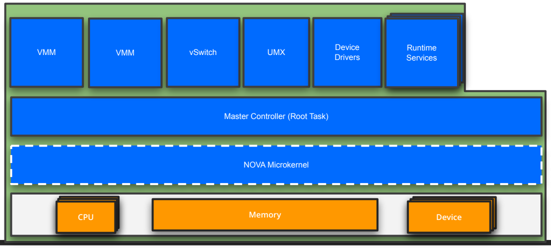
`impl`

`spec`

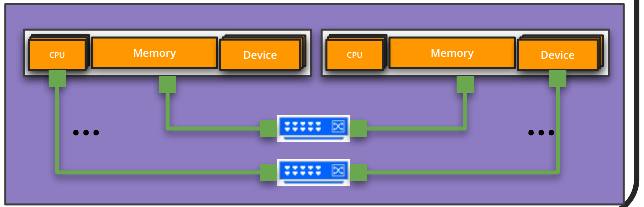
BedRock Systems

Decomposing refinement in separation logic

- Horizontally across multiple component semantics (in both `impl` and `spec`)
  - linking multiple (modularly developed) separation logics into one
- Vertically across multiple language abstractions (compiler correctness + language interoperability)
  - building language abstraction layers with resources
- All in separation logic?
  - PRO: avoid intermediate operational semantics, more expressive with resources
  - CONS: complex logic (later, fancy update modality, etc.)



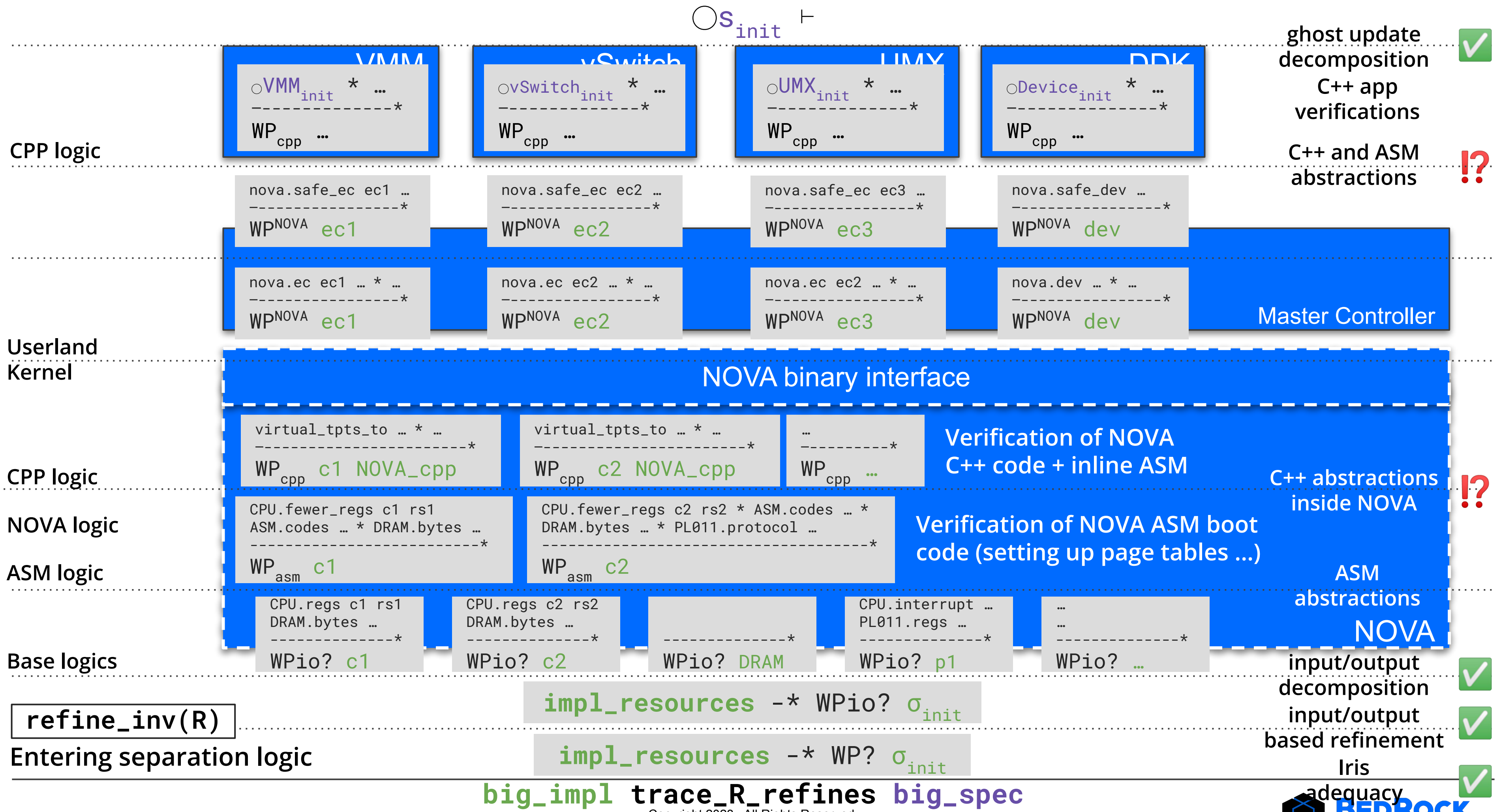
$\subseteq$



BedRock Systems

A path towards incremental end-to-end refinement in separation logics

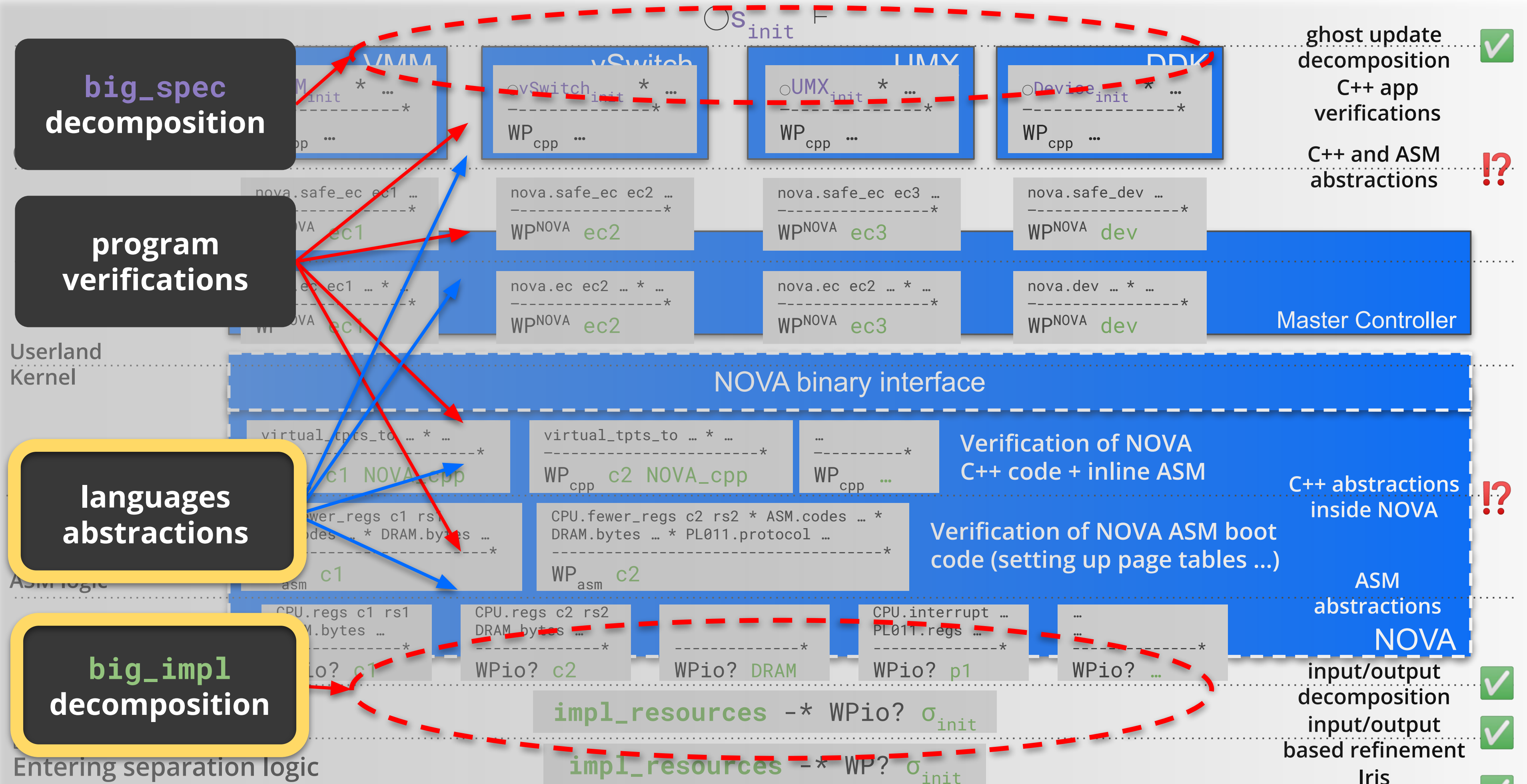
- We develop general frameworks to decompose multiple semantics in both the `implementation` and the `specification`
- We develop a demo setup of (simplified x86) CPUs + memory + I/O devices
  - decompose a refinement proof using the frameworks
  - construct abstractions *from machine logics to an ASM logic*
- We outline a path towards incremental end-to-end refinement within separation logic, with sketches on the abstractions from ASM to C++



○S_init ⊢
VMM
vSwitch
UMX
DDK
CPP logic
Userland
Kernel
Master Controller
NOVA binary interface
Verification of NOVA C++ code + inline ASM
Verification of NOVA ASM boot code (setting up page tables …)
CPP logic
NOVA logic
ASM logic
Base logics
NOVA
ghost update decomposition ✅
C++ app verifications
C++ and ASM abstractions ⁉️
C++ abstractions inside NOVA ⁉️
ASM abstractions
input/output decomposition ✅
input/output based refinement ✅
Iris adequacy ✅
Entering separation logic

```
○VMM_init * …
---------------*
WP_cpp …

○vSwitch_init * …
---------------*
WP_cpp …

○UMX_init * …
---------------*
WP_cpp …

○Device_init * …
---------------*
WP_cpp …

nova.safe_ec ec1 …
----------------*
WP^NOVA ec1

nova.safe_ec ec2 …
----------------*
WP^NOVA ec2

nova.safe_ec ec3 …
----------------*
WP^NOVA ec3

nova.safe_dev …
----------------*
WP^NOVA dev

nova.ec ec1 … * …
----------------*
WP^NOVA ec1

nova.ec ec2 … * …
----------------*
WP^NOVA ec2

nova.ec ec2 … * …
----------------*
WP^NOVA ec3

nova.dev … * …
----------------*
WP^NOVA dev

virtual_tpts_to … * …
------------------------*
WP_cpp c1 NOVA_cpp

virtual_tpts_to … * …
------------------------*
WP_cpp c2 NOVA_cpp

…
----------*
WP_cpp …

CPU.fewer_regs c1 rs1
ASM.codes … * DRAM.bytes …
-------------------------*
WP_asm c1

CPU.fewer_regs c2 rs2 * ASM.codes … *
DRAM.bytes … * PL011.protocol …
-----------------------------------*
WP_asm c2

CPU.regs c1 rs1
DRAM.bytes …
--------------*
WPio? c1

CPU.regs c2 rs2
DRAM.bytes …
--------------*
WPio? c2

--------------*
WPio? DRAM

CPU.interrupt …
PL011.regs …
--------------*
WPio? p1

…
…
--------------*
WPio? …

impl_resources -* WPio? σ_init

refine_inv(R)

impl_resources -* WP? σ_init

big_impl trace_R_refines big_spec
```

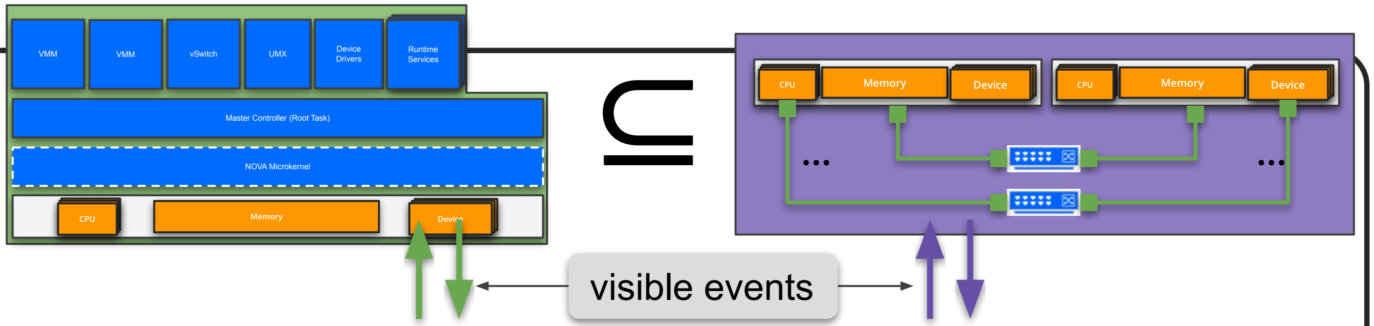
BEDROCK Systems



big_impl trace_R_refines big_spec

BEDROCK Systems

Warm up 1

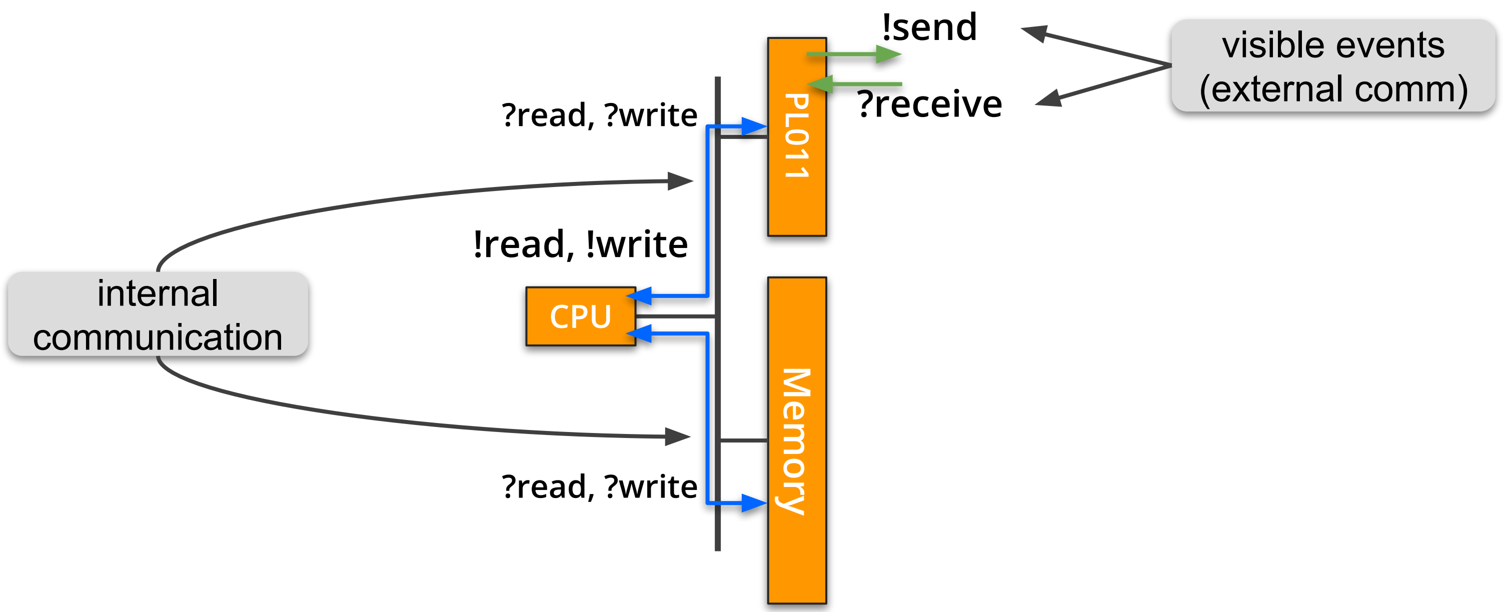


<u>An open-world setup for multiple semantics:</u>

Hardware components (in both `impl` and `spec`) are modeled as *processes communicating through request/response (output/input) events*

- **Events allows for flexible and dynamic communications among components**
  - A CPU read or write is a request event that can be responded by a device other than the RAM memory
- **All visible events are from physical I/O devices**

CPU || Memory || PL011 (I/O Device)



LTS Composition ($Cs = ||_{LTS}\ Cs[i]$)

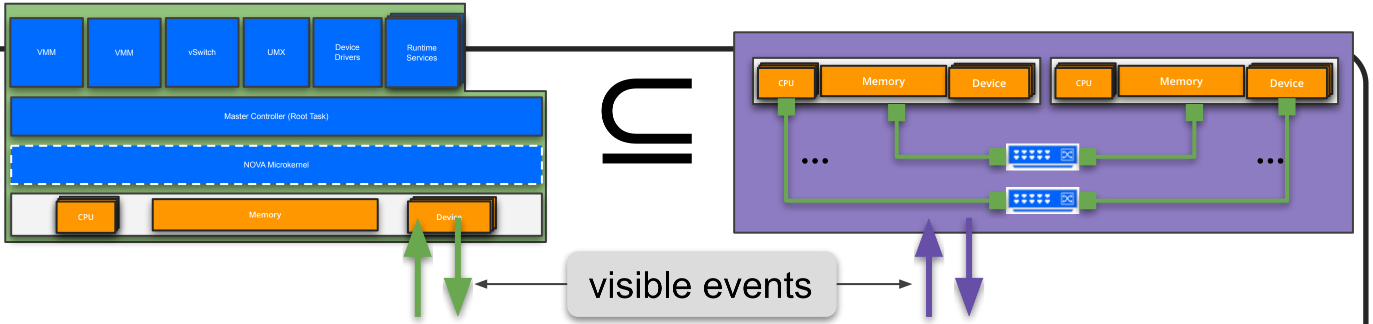
$$\text{tau step}\quad \frac{\texttt{Cs[i] \~{}\{τ\}\~{}> c'}}{\texttt{Cs \~{}\{τ\}\~{}> Cs[i := c']}}$$

$$\text{external communication step}\quad \frac{\texttt{Cs[i] \~{}\{e\}\~{}> c'} \qquad \texttt{e request/response <-> (Fext e) request/response}}{\texttt{Cs \~{}\{Fext e\}\~{}> Cs[i := c']}}$$

$$\text{internal comm step}\quad \frac{\texttt{Cs[i1] \~{}\{e1\}\~{}> c1'} \qquad \texttt{Cs[i2] \~{}\{e2\}\~{}> c2'} \qquad \texttt{(e2,e1) is (request,response)}}{\texttt{Cs \~{}\{τ\}\~{}> Cs[i1 := c1'][i2 := c2']}}$$

- Like a threadpool but each thread has its own semantics
- τ are internal steps
- External communications produce externally visible events
- Internal communications are matching request/response pairs (***atomic*** step)

Warm up 2



Formally defining the notion of ⊆ as ***trace refinement***

```
big_impl trace_R_refines big_spec :=
    ∀ σ' TR, σ_init ~{TR}~>* σ' ⇒
    ∃ s' tr, s_init ~{tr}~> s' ∧ Forall2 R (no_τ TR) (no_τ
tr)
```

Warm up 2: a lightweight refinement setup in Iris

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

- 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

$$\dfrac{\bigcirc s_{init} * \texttt{impl\_resources} \vdash WP\ \sigma_{init}\ \{\ \lambda\_,\ \texttt{False}\ \}}{\texttt{big\_impl trace\_R\_refines big\_spec}}$$

Enters separation logic

Iris adequacy

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

Decomposing multiple semantics of big_impl

◯s_{init} ⊢

big_impl decomposition

```
CPU.regs c1 rs1      CPU.regs c2 rs2                           CPU.interrupt …     …
DRAM.bytes …         DRAM.bytes …                              PL011.regs …        …
---------------*     ---------------*     ---------------*     ---------------*    ---------------*
WPio c1              WPio c2              WPio DRAM            WPio p1             WPio …
```

Base logics

input/output decomposition

`impl_resources -* WPio` σ_{init}

refine_inv(R)

input/output based refinement

`impl_resources -* WP` σ_{init}

Entering separation logic

Iris adequacy

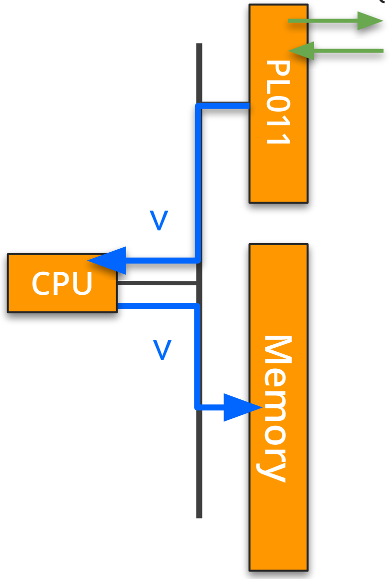
`big_impl trace_R_refines big_spec`

∀ σ' TR, σ_{init} ~{TR}~>* σ' ⇒ ∃ s' tr, s_{init} ~{tr}~> s' ∧ Forall2 R (no_τ TR) (no_τ tr)

BedRock Systems

Building a logic with multiple semantics

Consider a monolithic logic of a CPU + DRAM + a PL011 (I/O device)



```
{ cpu.reg r2 v' * cpu.reg r1 pl011.data_reg * pl011.reg data_reg [v] }
```

MOV r2 [r1] **read from PL011 ([r1]) to r2**

```
{ cpu.reg r2 v * cpu.reg r1 pl011.data_reg * pl011.reg data_reg [] }
```

```
{ cpu.reg r2 v * cpu.reg r3 l * l ↦ v' }
```

MOV [r3] r2 **write r2 to DRAM ([r3])**

```
{ cpu.reg r2 v * cpu.reg r3 l * l ↦ v }
```

BedRock Systems

Decomposing a logic for multiple semantics

The monolithic logic considers steps of the whole monolithic machine

$$\{\ \texttt{Pre}\ \}\ \sigma_{\texttt{CPU+DRAM+PL011}}\ \texttt{\~{}\~{}>}\ \sigma'_{\texttt{CPU+DRAM+PL011}}\ \{\ \texttt{Post}\ \}$$

Instead, we want to consider each component's steps modularly.

$$\{\ \dots\ \}\ \sigma_{\texttt{CPU}}\ \texttt{\~{}\{!cpu.Read l v\}\~{}>}\ \sigma'_{\texttt{CPU}}\ \{\ \dots\ \}$$
$$\{\ \dots\ \}\ \sigma_{\texttt{CPU}}\ \texttt{\~{}\{!cpu.Write l v\}\~{}>}\ \sigma'_{\texttt{CPU}}\ \{\ \dots\ \}$$

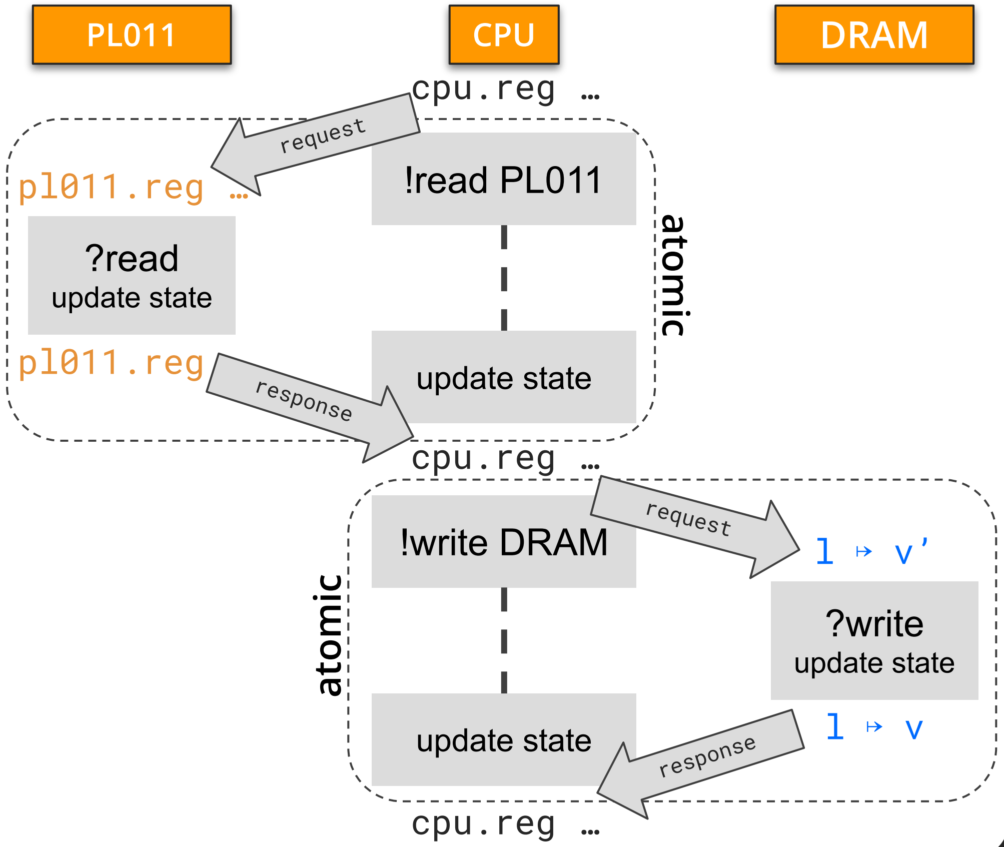
$$\{\ \dots\ \}\ \sigma_{\texttt{DRAM}}\ \texttt{\~{}\{?dram.Write l v\}\~{}>}\ \sigma'_{\texttt{DRAM}}\ \{\ \dots\ \}$$

$$\{\ \dots\ \}\ \sigma_{\texttt{PL011}}\ \texttt{\~{}\{?pl011.DataRead l v\}\~{}>}\ \sigma'_{\texttt{PL011}}\ \{\ \dots\ \}$$

Decomposing multiple semantics with WPio

WPio captures extra ***rely-guarantee*** assumptions/obligations one may have when interacting with the environment.

- the requester not only shows that it can make its own step, but also ***helps the responder making the matching step, and vice versa***
- request/response conditions capture the atomic spec of the responder, often as AUs



BedRock Systems

Linking logics for multiple semantics with `WPio`

```
{ cpu.reg r1 l * cpu.reg r2 v’ } !cpu.Read l v { cpu.reg r1 l * cpu.reg r2 v }
{ cpu.reg r1 l * cpu.reg r2 v } !cpu.Write l v { cpu.reg r1 l * cpu.reg r2 v }
```

```
{ pl011.reg data_reg [v] } ?pl011.DataRead l v’ { v’ = v * pl011.reg data_reg [] }
```

```
{ l ↦ v’ } ?dram.Write l v { l ↦ v }
```

logics more modularly developed

logics linked through internal comm (matching request/response) events

```
cpu.reg r1 pl011.data_reg * cpu.reg r2 v’ * pl011.reg data_reg [v] ⊢
  cpu.WPio (MOV r2 [r1], !cpu.Read pl011.data_reg v)
{ cpu.reg r1 pl011.data_reg * cpu.reg r2 v * pl011.reg data_reg [] }
```

```
cpu.reg r1 l * cpu.reg r2 v * l ↦ v’ ⊢
  cpu.WPio (MOV [r1] r2, !cpu.Write l v) { cpu.reg r1 l * cpu.reg r2 v * l ↦ v }
```

```
emp ⊢ dram.WPio (?dram.Read l v) { emp }
```

Decomposing multiple semantics of big_impl

$\bigcirc$s$_{\text{init}}$ ⊢

Theorem [WPio to WP]: `WPio σ ⊢ WP σ`

Linking Theorem [Decomposition of WPio's]:

`[*set]c ∈ Cs, WPio c ⊢ WPio (||`$_{\text{LTS}}$` Cs)`

```
CPU.regs c1 rs1     CPU.regs c2 rs2                     CPU.interrupt …     …
DRAM.bytes …        DRAM.bytes …                        PL011.regs …        …
-----------*        -----------*        -----------*    -----------*        -----------*
WPio c1             WPio c2             WPio DRAM       WPio p1             WPio …
```

Base logics

input/output decomposition

`impl_resources -* WPio σ`$_{\text{init}}$

input/output based refinement

refine_inv(R)

Entering separation logic

`impl_resources -* WP σ`$_{\text{init}}$

Iris adequacy

`big_impl trace_R_refines big_spec`

`∀ σ' TR, σ`$_{\text{init}}$` ~{TR}~>* σ' ⇒ ∃ s' tr, s`$_{\text{init}}$` ~{tr}~> s' ∧ Forall2 R (no_τ TR) (no_τ tr)`

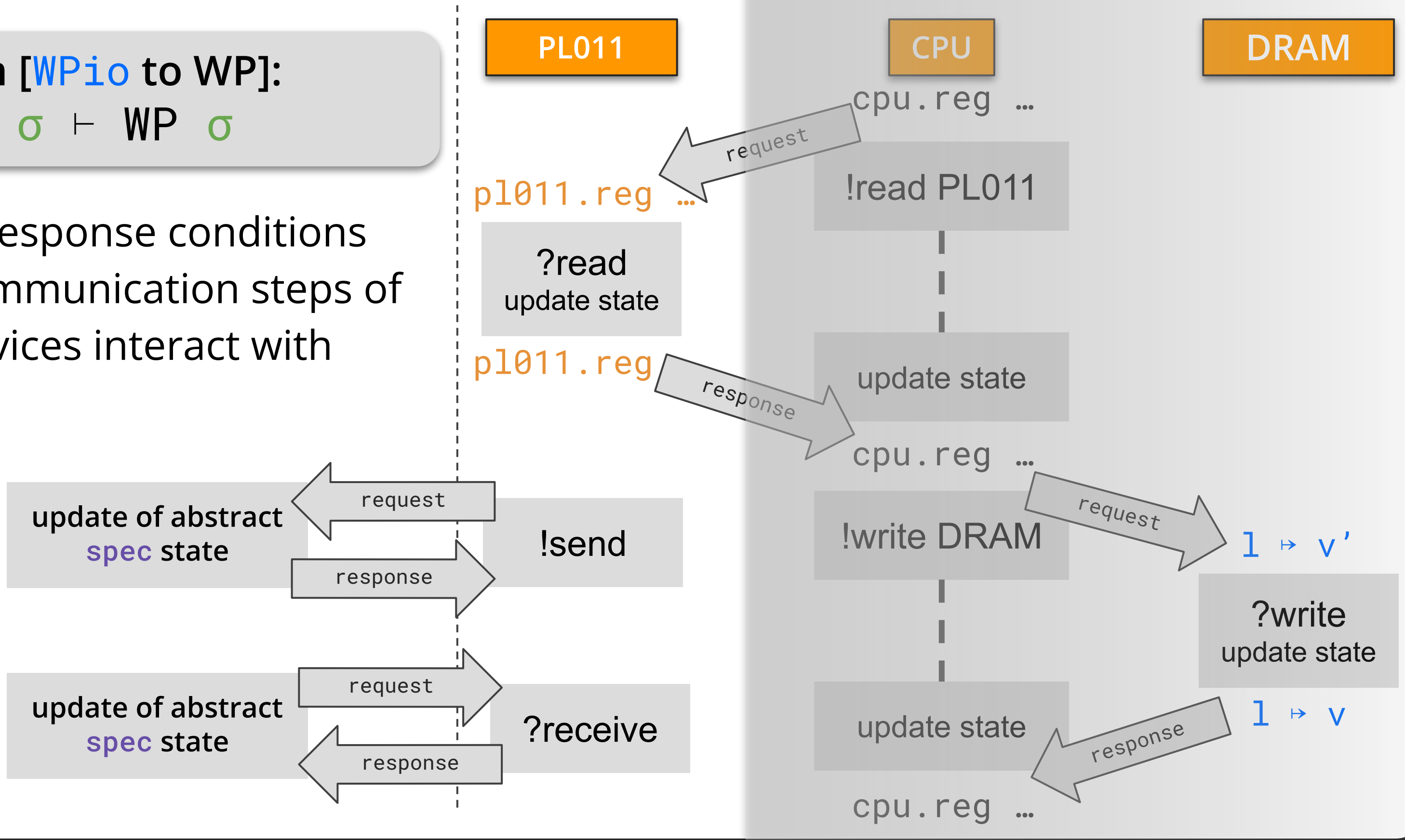
BEDROCK Systems

Refinement and WPio

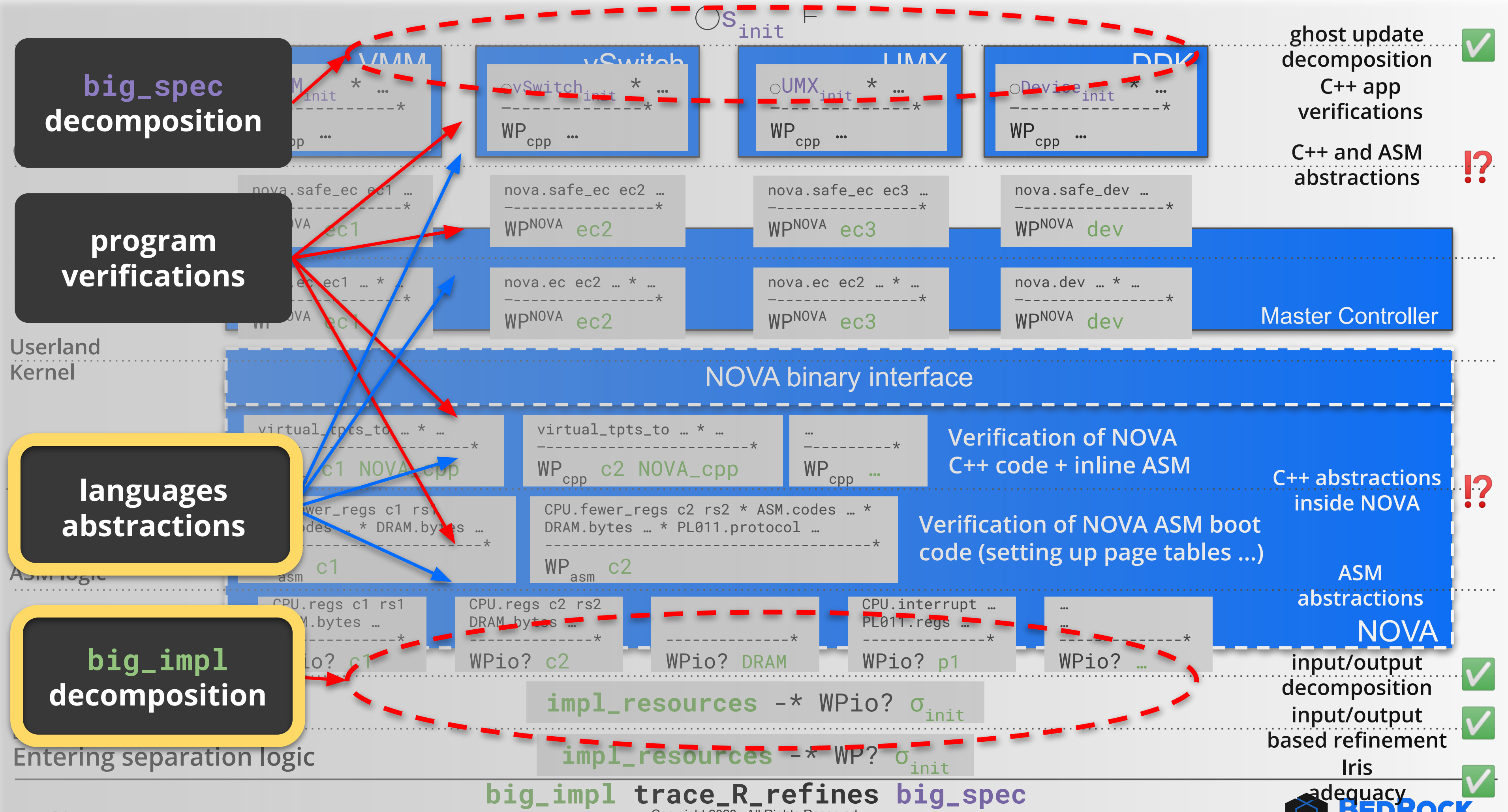
Theorem [WPio to WP]:

$\texttt{WPio}\ \sigma \vdash \texttt{WP}\ \sigma$

WPio request/response conditions for external communication steps of physical I/O devices interact with ***refinement***.



BedRock Systems



BEDROCK Systems

ASM abstractions

- **Going from machine logics to ASM logic (“compiler correctness” of assembler)**
- **Reusing existing logics looks promising [1,2]**
- **We develop a demo ASM logic *without stating an operational semantics for ASM***
  - After the `WPio` decomposition, abstract from `WPio` of CPU to $\texttt{WP}_{\texttt{asm}}$
  - Intuition: to get to ASM, give up resources needed for the ASM abstractions
    - instruction fetch and decoding (code stored in memory)
    - jump labels

[1] High-Level Separation Logic for Low-Level Code. POPL’13

[2] Islaris: Verification of Machine Code Against Authoritative ISA Semantics. PLDI’22

Building the ASM abstractions

```
Axiom abstract_asm :
    WP_asm c1 asm_prog ⊢
    ∀ pc q, cpu.pc c1 pc -* dram.bytes l q (assemble asm_prog) -* WPio c1.
```

Solution :

```
Definition WP_asm c1 l prog :=
    ∀ pc q prog', cpu.pc c1 pc -* dram.bytes l q (assemble prog') -*
    prog' @ pc = prog -* … -* WPio c1.
```

Machine	`cpu.pc c1 pc * dram.byte pc q (op_encode MOV [r1] r2) *`
	`cpu.reg r1 l * cpu.reg r2 v * l ↦ v ⊢`
	`cpu.WPio c1`
	`  { cpu.pc c1 pc+1 * dram.byte pc q (op_encode MOV [r1] r2) * … }`

ASM	`cpu.reg r1 l * cpu.reg r2 v * l ↦ v' ⊢`
	`WP_asm c1 (MOV [r1] r2) { cpu.reg r1 l * cpu.reg r2 v * l ↦ v }`

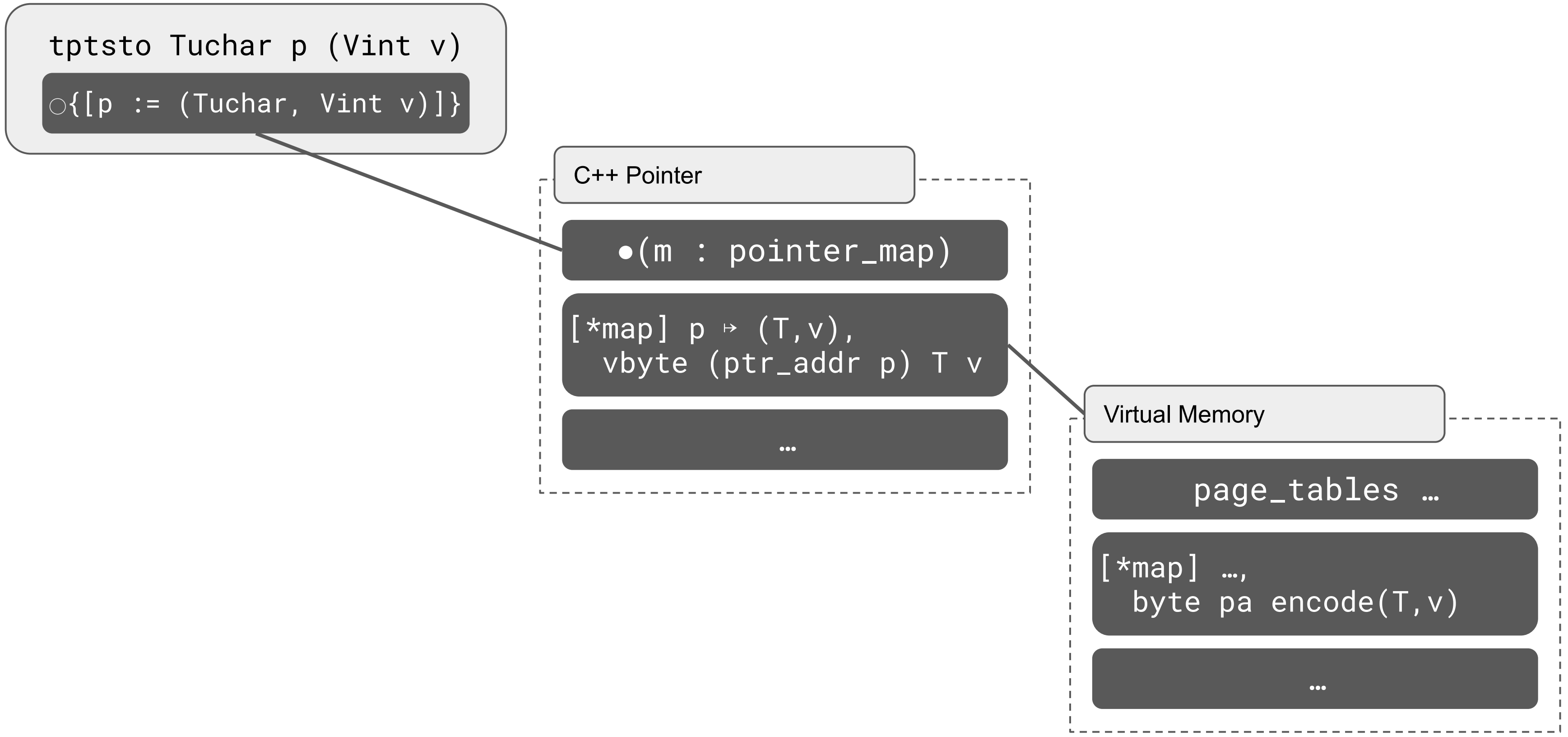
BedRock Systems

C++ abstractions ⁉️

- *Compiler correctness as separation logic specifications?*
- **This would require building multiple layers of abstractions for C++ in the logic**
- **We need to allow breaking abstractions, consider inline ASM**
- **Goal: to get to C++, give up resources needed for the C++ abstraction**

```
Axiom abstract_wp_cpp :
    ([*list] tpts_to … -* WP_cpp (c1, cpp_prog)) ⊢
    stack_frames … -* page_tables … -* bytes … -* … -* WP_asm (c1, asm_prog).
```

Sketching a C++ tpts_to



```
tptsto Tuchar p (Vint v)
○{[p := (Tuchar, Vint v)]}
```

```
●(m : pointer_map)
```

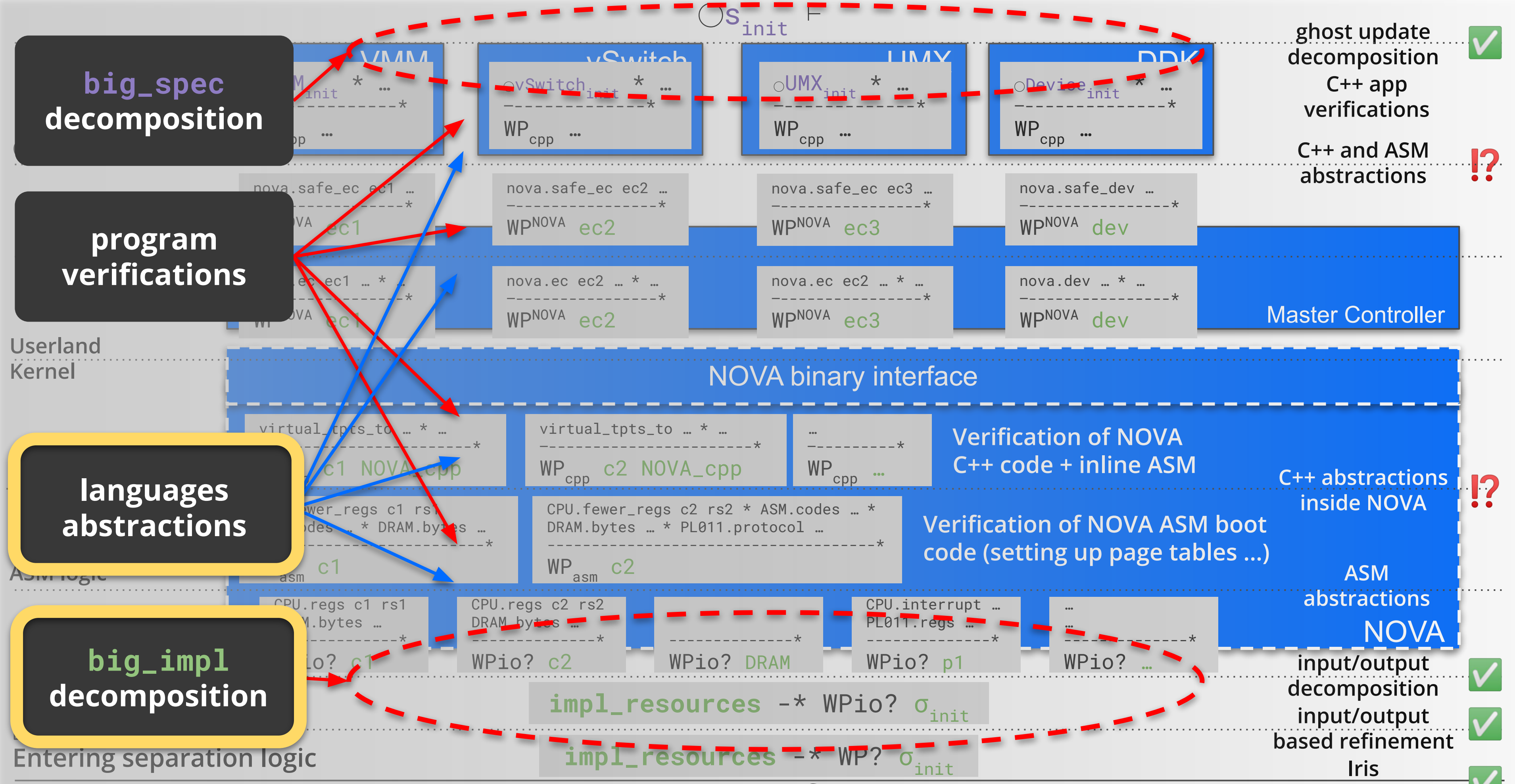
```
[*map] p ↦ (T,v),
  vbyte (ptr_addr p) T v
```

```
…
```

```
page_tables …
```

```
[*map] …,
  byte pa encode(T,v)
```

```
…
```



big_spec decomposition

program verifications

languages abstractions

big_impl decomposition

big_impl trace_R_refines big_spec

BEDROCK Systems