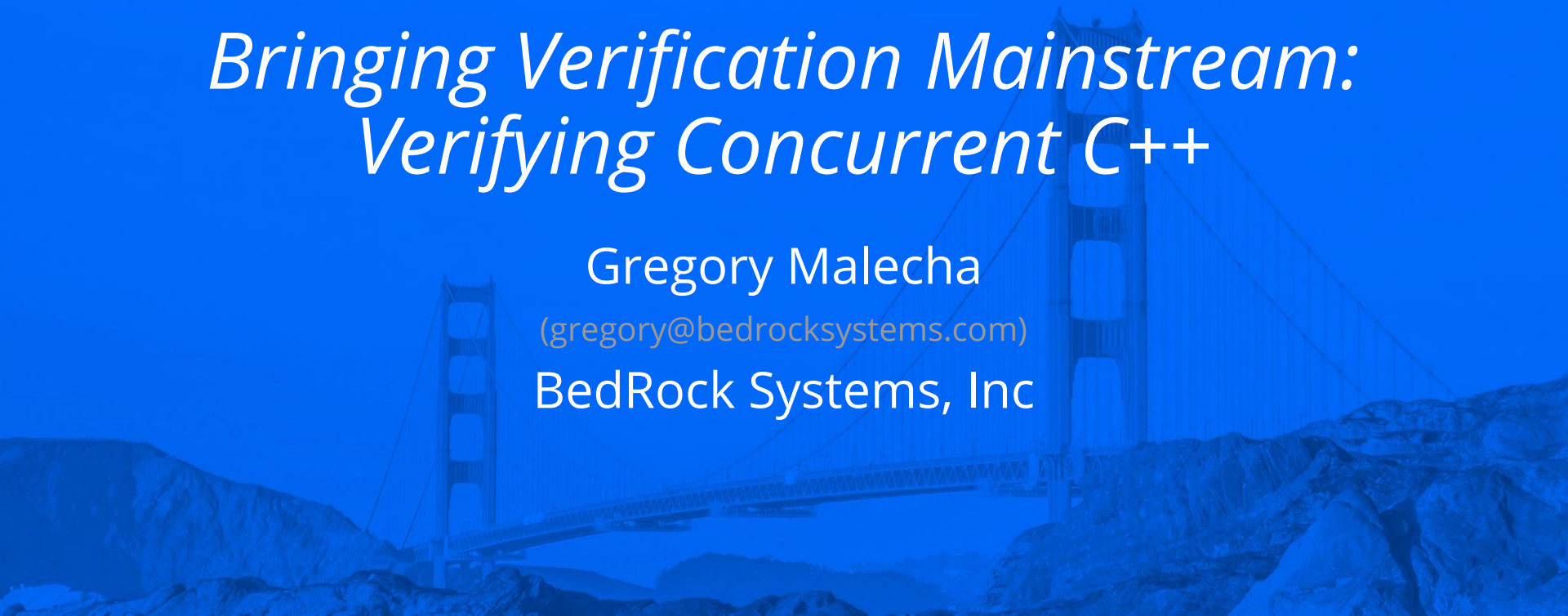
BEDROCK Systems Inc

Bringing Verification Mainstream: Verifying Concurrent C++

Gregory Malecha

(gregory@bedrocksystems.com)

BedRock Systems, Inc



BedRock Systems Inc

What is BedRock Systems?

Our point of view

Modern trustworthy compute base

➔ Virtualization stack

Formal methods from the ground up

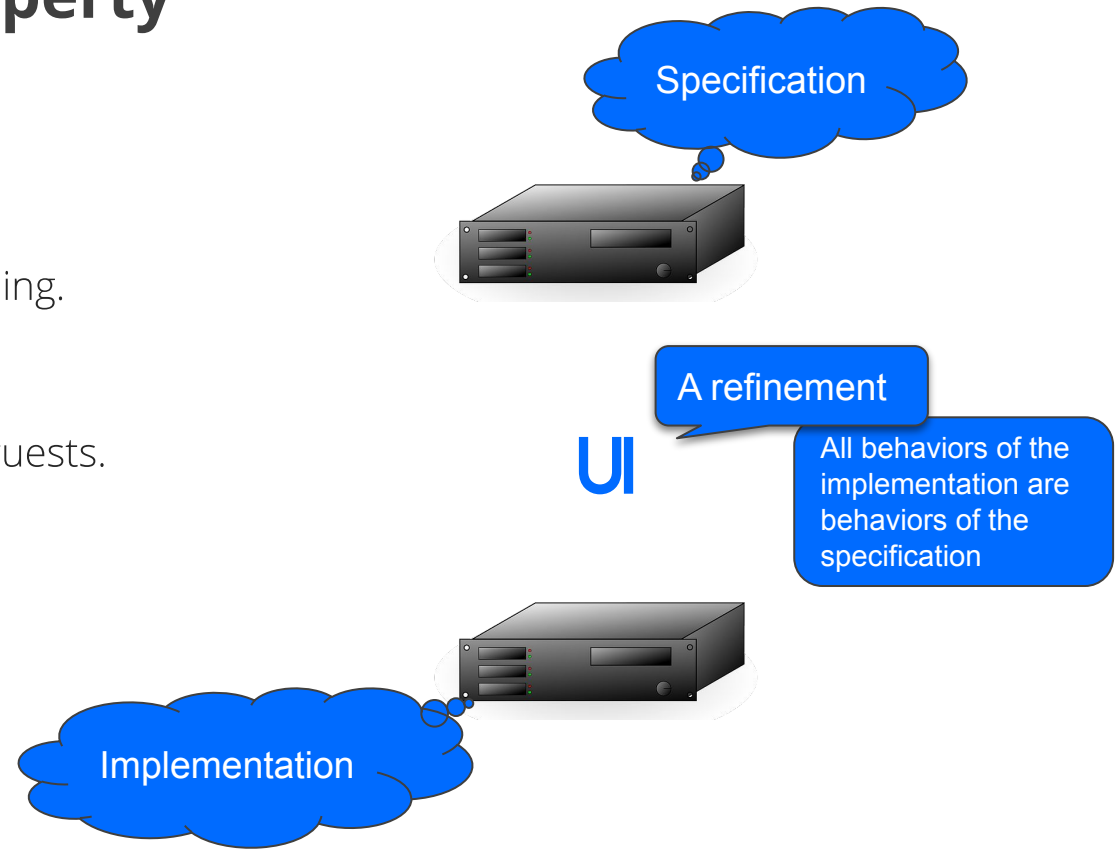
➔ Apply formal methods early and continuously

➔ Collaborate closely with systems engineers

The "Bare Metal" Property

Mathematically "boring", very compelling.

- Tractable because we can run unmodified (potentially buggy) guests.
- Already agreed upon interface.



BedRock Systems Inc

The "Bare Metal" Property (with Introspection)

Provide monitoring, and remediation.

- Omnipotent vantage point.
- Invisible to the guest*.
- Unlimited access to system internals.

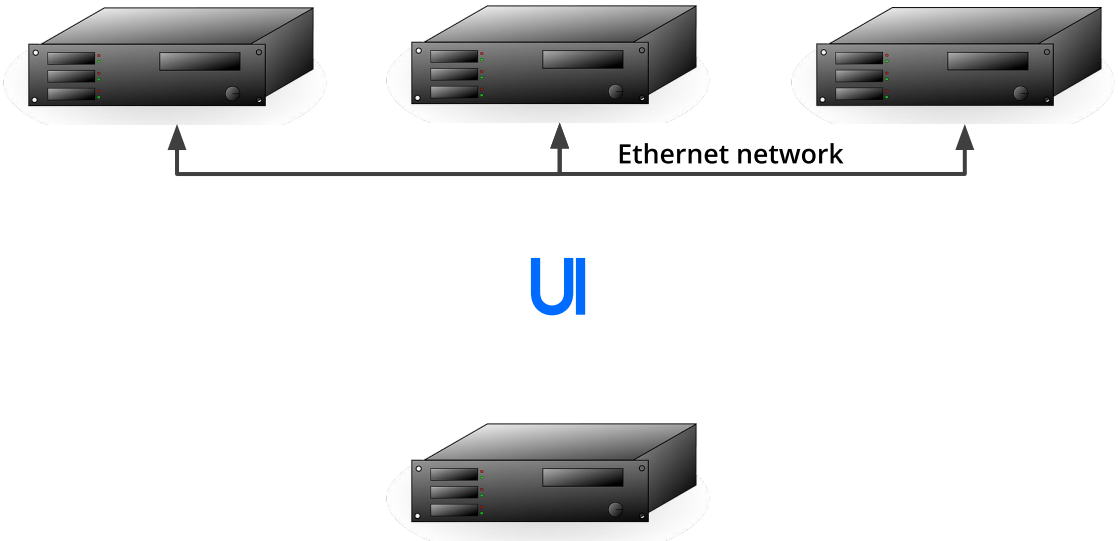
UI

* Under certain conditions (not formally guaranteed).

The "Bare Metal" Property (for Consolidation)

Running multiple guests provides consolidation.

- Related to multi-computer system connected by wires.
- Introspect communication.
- Normal virtualization & consolidation benefits.

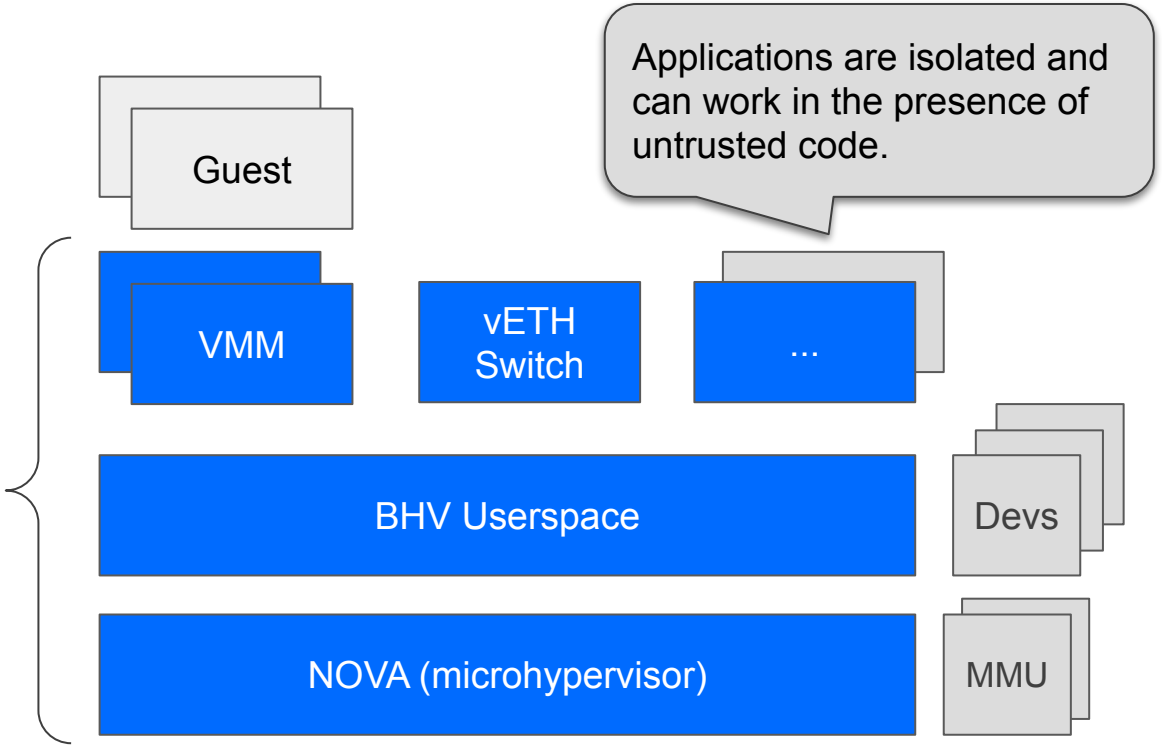


BEDROCK Systems Inc

The BedRock Stack

System built from many components

- Separate verification
- Significant code & proof re-use
- More than just C++
  - Assembly (x86, ARM, ...)
  - Hardware devices
  - Guest code



Lots of interesting & difficult challenges (life is never boring).

BEDROCK Systems Inc

Outline

- BedRock: Who? What? How?
- Verification for C++
- FM Value Proposition
- Separation Logic
- Automation

BedRock
Systems Inc

Verification for C++

Challenges & Benefits

Making Verification Mainstream

A "Better" Language

Convince developers to change their language of choice.

- Ada
- Rust
- ...

An Existing Language

Provide tools to verify software written in existing mainstream languages

- C
- C++

Making Verification Mainstream

A "Better" Language

Convince developers to change their language of choice.

- Ada
- Rust
- ...

- Does it solve our problem completely?
  - Prove the Bare Metal property?
- How easy is it to extend?
- How much do you have to code-around the language?

Making Verification Mainstream

- Some "cruft"
  - Oddities in the language (often historic). Hindsight is 20/20.
- No formal description
- All-at-once complexity
  - "Idiomatic" code and existing libraries use complex language features

Focus on subset of the language and grow over time.

An Existing Language

Provide tools to verify software written in existing mainstream languages

- C
- C++

Making Verification Mainstream

A "Better" Language

Convince mainstream developers to change their language of choice.

- Ada
- Rust
- ...

An Existing Language

Provide tools to verify software written in existing mainstream languages

- C
- C++

On the Shoulders of Giants (From C to C++)

Tempting to try to support everything, but ***prioritize*** *what you really need*.
➔ Build on existing tools / libraries

Surface Complexities

- Parsing
- Type checking
- Overloading
- Syntactic sugar
- constexpr
- templates

Semantic Challenges

- Value categories
- Memory model
- Side-effects
- Modularity

Classes + Objects

- Constructors
- Destructors
- Inheritance
- Virtual dispatch

Some features of C++ are *very* complex.
➔ A sound, incomplete logic.

BEDROCK
Systems Inc

The FM Value Proposition

Beyond bug freedom

More than bug-free?

Implementation-level assurance

- Precise statements
- Guaranteed correctness

"Traditional" value-proposition of formal methods.

More than bug-free
(Confidence & Composition)

Implementation-level assurance

- Precise statements
- Guaranteed correctness

Formal methods as the **science of design**.

Design-level confidence

- Composition
- Abstraction
- Encapsulation

Formal methods provides more than "testing".

Greater potential for longer-lived code.
Easier to evolve and maintain.

Experiences
(clear focus on core problem)

"The specification makes the problem a lot clearer." *~VMM Developer*

- VMM guest memory management

Mediating access to guest memory, races between HW, software instruction emulation, virtual devices, et.al.

- ACLs for network traffic

Generalized the architecture of rules.
Simpler user-facing model and easier to extend.

- Console multiplexer

Clarify the protocol between the control- and data-"planes".
(reusable abstraction)

- Driver architecture

Better understanding of the design space allowed us to evaluate alternatives and find generic specifications.

Formal methods provides a *lens* for evaluating designs, and finding alternatives.

FM and "Best Practice" Tend to Agree

("better" code is easier to specify/understand)

1. **Don't return pointers/references to (certain) internal data**
2. **Mutation is (often) unnecessary**
3. **Avoid duplicate information**
   - Potential for Inconsistent views
   - Atomic update is difficult / expensive
4. **A function should only work at one level of abstraction**
   - Huge layering improvements
5. **Decouple unrelated problems**

Better designed code has smaller, more natural specifications.

Fewer side-conditions, more local.
"Code smells" before the code!

BedRock
Systems Inc

Separation Logic

Concise, modular, & expressive specifications

Separation Logic

Our ***language*** of formal methods

Separation is inherent in *our* understanding,
it should be the cornerstone of our reasoning...

Understandable proofs often lead to better code.

Especially true for complex code.

Core Principles of Separation Logic

Focus on states, not transitions.

Describe the state of your system as the combination of small, ***disjoint*** resources.

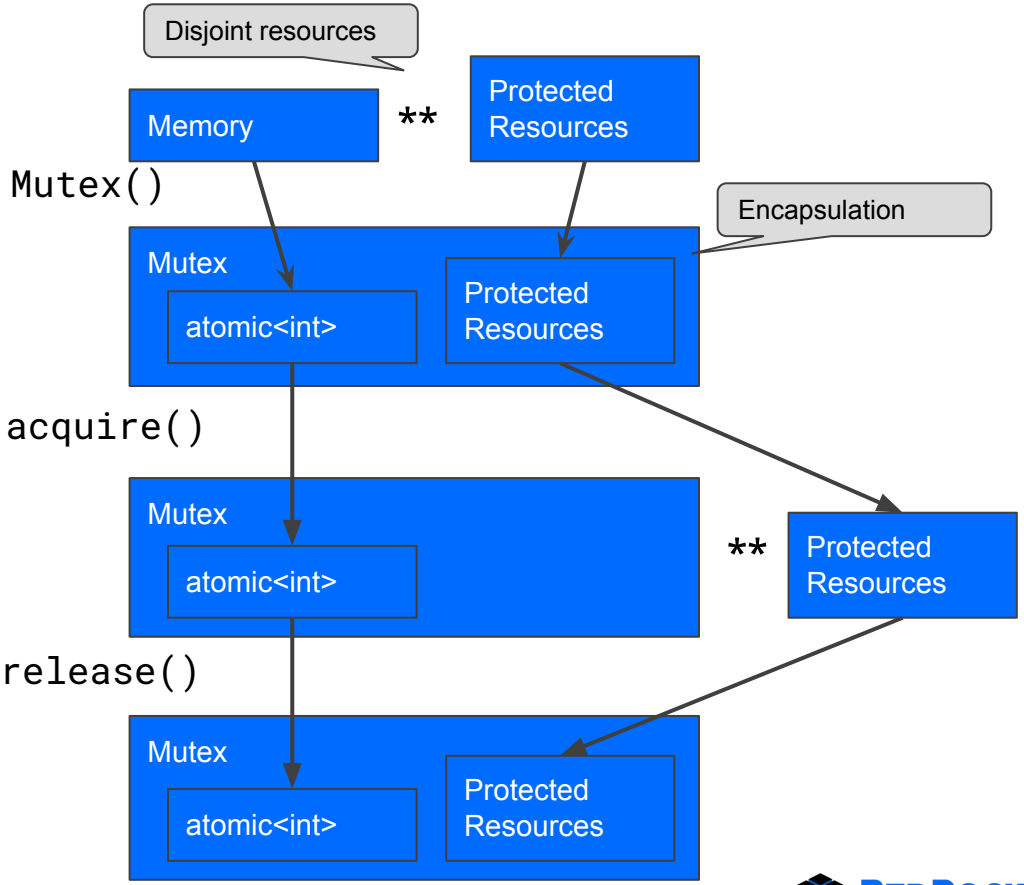
A (formal) logic with standard quantifiers.

- Standard mathematical abstractions.

Separation Logic in Action

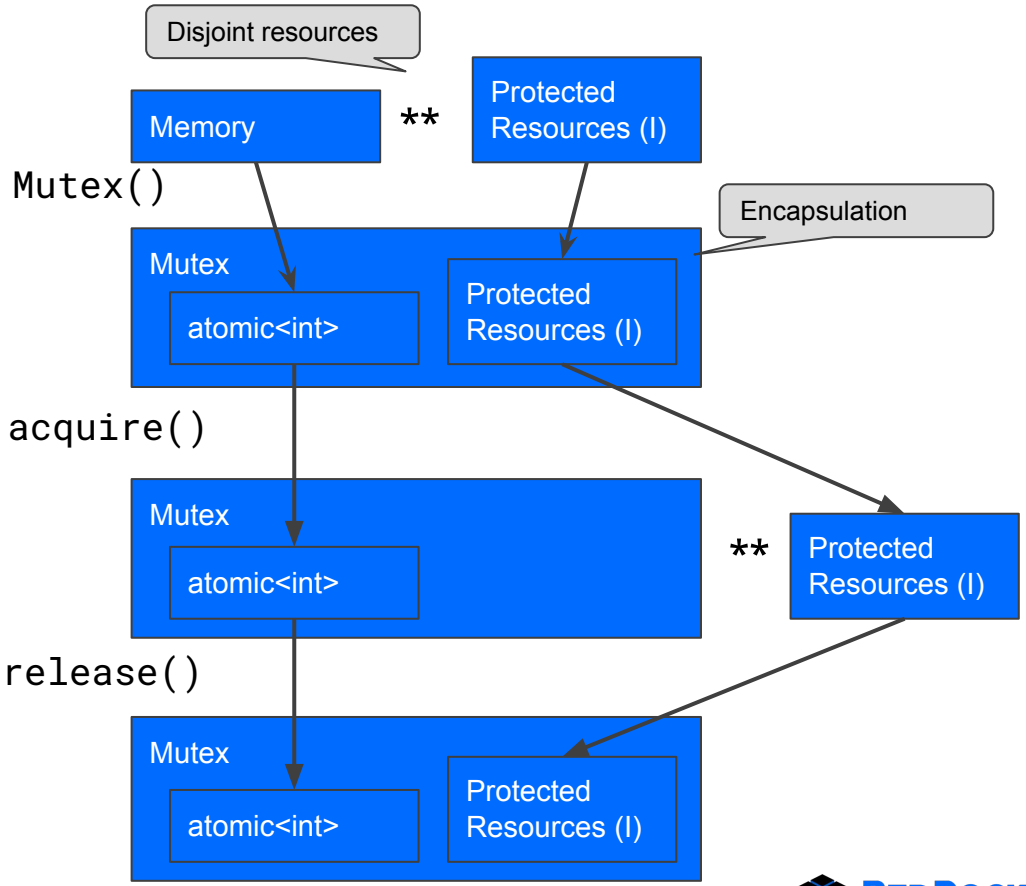
Divide the "world" into disjoint regions.

- Disjointness enables composition
- Implicitly open-world reasoning
  - New functions
  - New threads
  - ***Dynamic*** rather than static



Separation Logic in Action

```
{ I }
Mutex() // initially locked
{ ∃ g, this |-> mutex.R g 1 I }
```

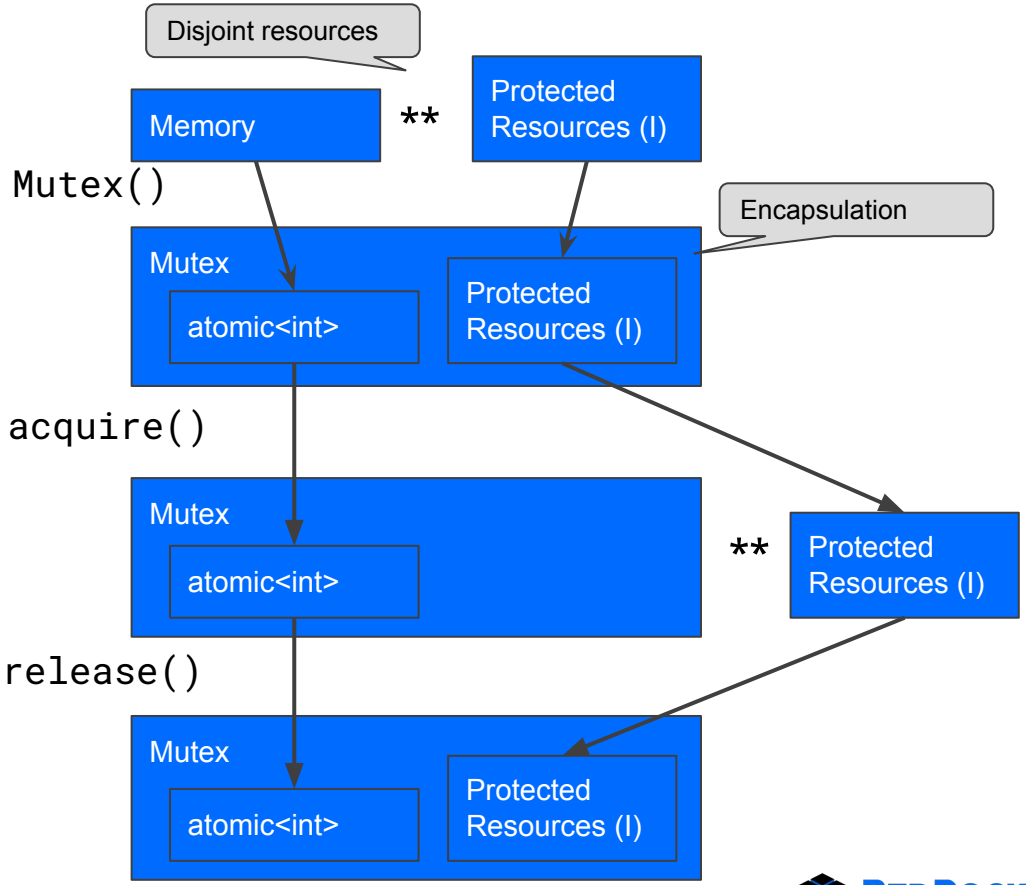


Proofs are *often* quite program directed, "follow your nose".

Separation Logic in Action

```
{ I }
Mutex() // initially locked
{ ∃ g, this |-> mutex.R g 1 I }


{ this |-> mutex.R g q I }
void acquire()
{ this |-> mutex.R g q I **
    I ** Locked g }
```



Proofs are *often* quite program directed, "follow your nose".

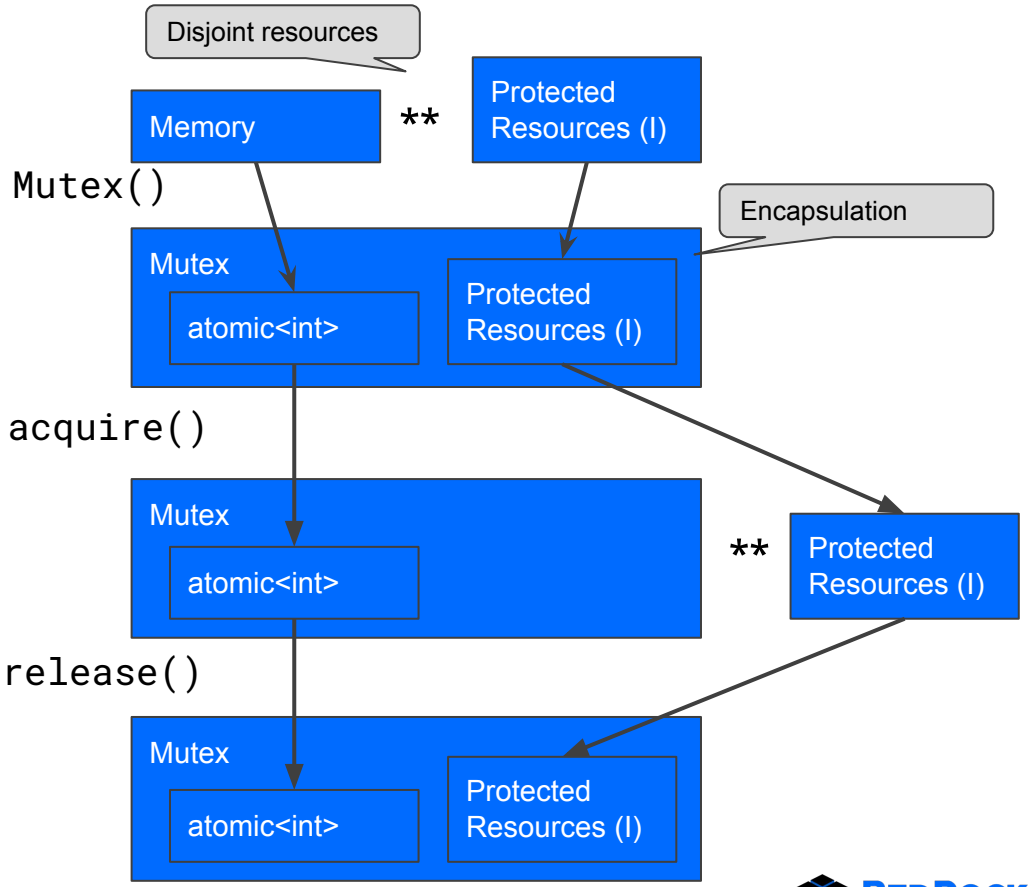
Separation Logic in Action

```
{ I }
Mutex() // initially locked
{ ∃ g, this |-> mutex.R g 1 I }
```

```
{ this |-> mutex.R g q I }
void acquire()
{ this |-> mutex.R g q I **
    I ** Locked g }

{ this |-> mutex.R g q I **
    I ** Locked g }
void release()
{ this |-> mutex.R g q I }
```

Proofs are *often* quite program directed, "follow your nose".



BEDROCK Systems Inc

Lock Specifications

Specifications broken down into:

1. State assertions, e.g. mutex.R
2. Function specifications

Pre- and post-conditions describe the transfer of ownership between parties.

1. Directly connected to the code.
2. Requires only local knowledge about the code
3. Completely open-world assumption

```
\pre{I} I
\post Exists g, this |-> mutex.R g 1 I
Mutex() // initially locked

\pre{g I} this |-> mutex.R g 1 I
\pre Locked g (* require locked *)
\post emp
~Mutex()

\prepost{g q I} this |-> mutex.R g q I
\post I ** Locked g
void acquire()

\prepost{g q I} this |-> mutex.R g q I
\pre I ** Locked g
\post emp
void release()
```

Concurrent Specifications

Build on Iris's theory of atomic updates.

1. First-class representation of commit points in the logic.
2. "Atomic ghost functions"
3. Almost perfectly compositional

A bit more boilerplate to reason about, but *very* (maximally?) expressive.

```
\arg{by} "by" (Vint by)
\pre AU << Ex v, this |-> atomicR 1 v >> @...
      << this |-> atomicR 1 (trim (v + by))
       , COMM Q v >>
\post{v}[Vint v] Q v
int fetch_add(int by)
```

Only caveat to perfect composition is masks, but don't seem to be a problem in practice.

BedRock
Systems Inc

Scaling to the real world

The easy and the complex

Easy Problems, Easy Solutions
(patterns & automation)

Formal methods are pedantic in nature.

- Proofs done in complete detail
- Precise about *everything*

Reduce "boilerplate"

Ensure that the **easy things are easy**

- Leverage the language
- Fall back on the proof assistant

Specification Generators

- ➔ Getters / setters
- ➔ Default operations
- ➔ Simple structures

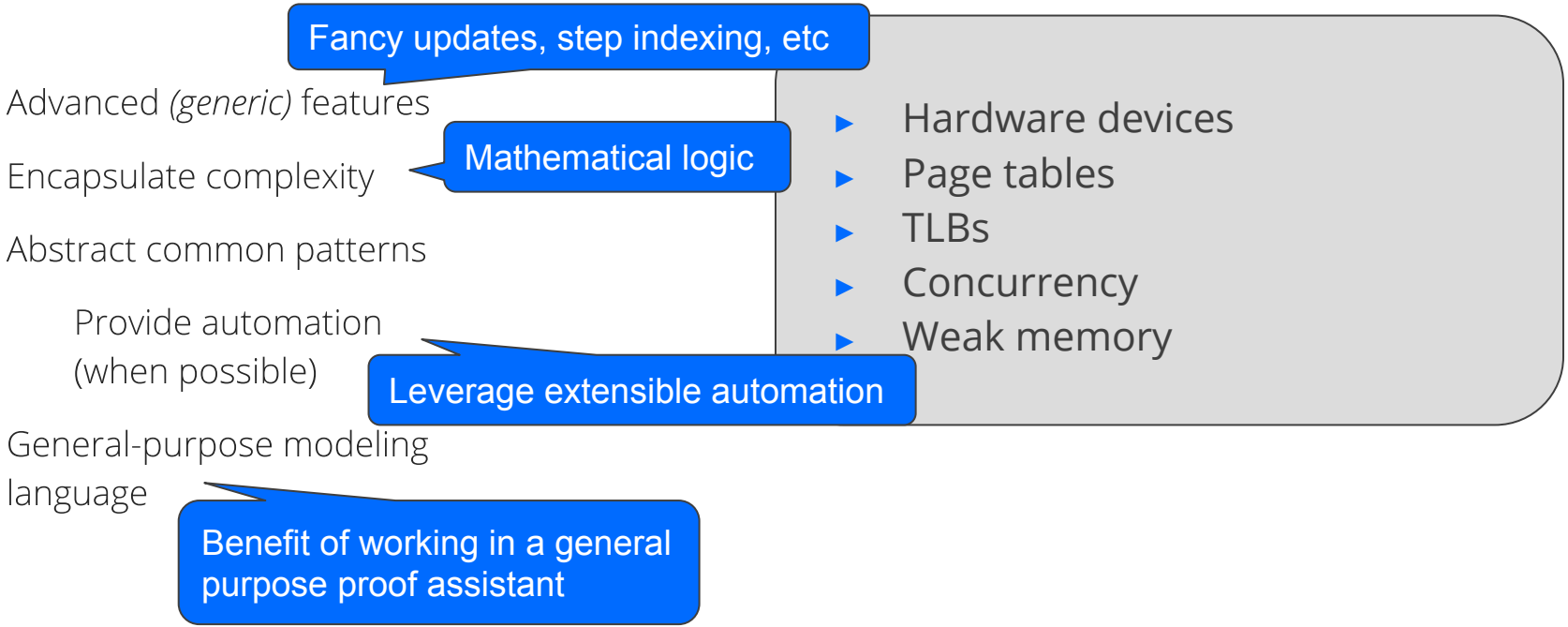
Naive proofs have very low signal-to-noise ratio.

Must avoid excessive bookkeeping.

Customizable Automation

- ➔ Common patterns
- ➔ Domain-specific automation

Hard Problems...
Possible Solutions
(expressivity & patterns)



Thank you!

(Questions?)

- BedRock: Who? What? How?
- Verification for C++
- FM Value Proposition
- Separation Logic
- Automation

Join us? Hiring in FM & Systems

BedRock Systems Inc