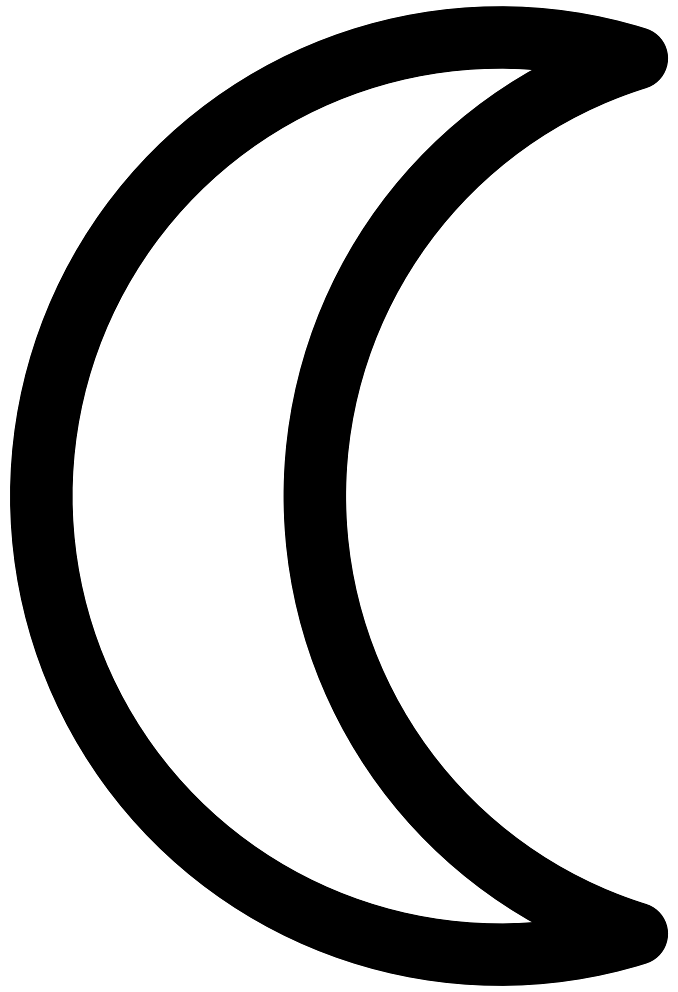
Automated verification of systems software with Serval

Luke Nelson, Emina Torlak, Xi Wang
Paul G. Allen School of Computer Science & Engineering
University of Washington

Outline

- Part 1: Overview
- Part 2: Serval
- Part 3: Rosette

Part 1: overview



$\approx$

specification **proof** **implementation**

Overview outline

- History
- Example: specification & verification
- Project organization

Pioneer efforts of OS kernel verification

- Examples of earlier efforts
  - UCLA Unix security kernel
  - Kit
- seL4 microkernel
  - first functional correctness proof (2009)
  - 10,000 lines of C and assembly

Examples in recent SOSP/OSDI

OSDI'14	SOSP'15	OSDI'16	SOSP'17	OSDI'18	SOSP'19
					AtomFS
					Notary
			DFSCQ	CSPEC	Perennial
Ironclad	FSCQ	CertiKOS	Hyperkernel	Nickel	Serval
Jitk	IronFleet	Yggdrasil	Komodo	SFSCQ	Vigor

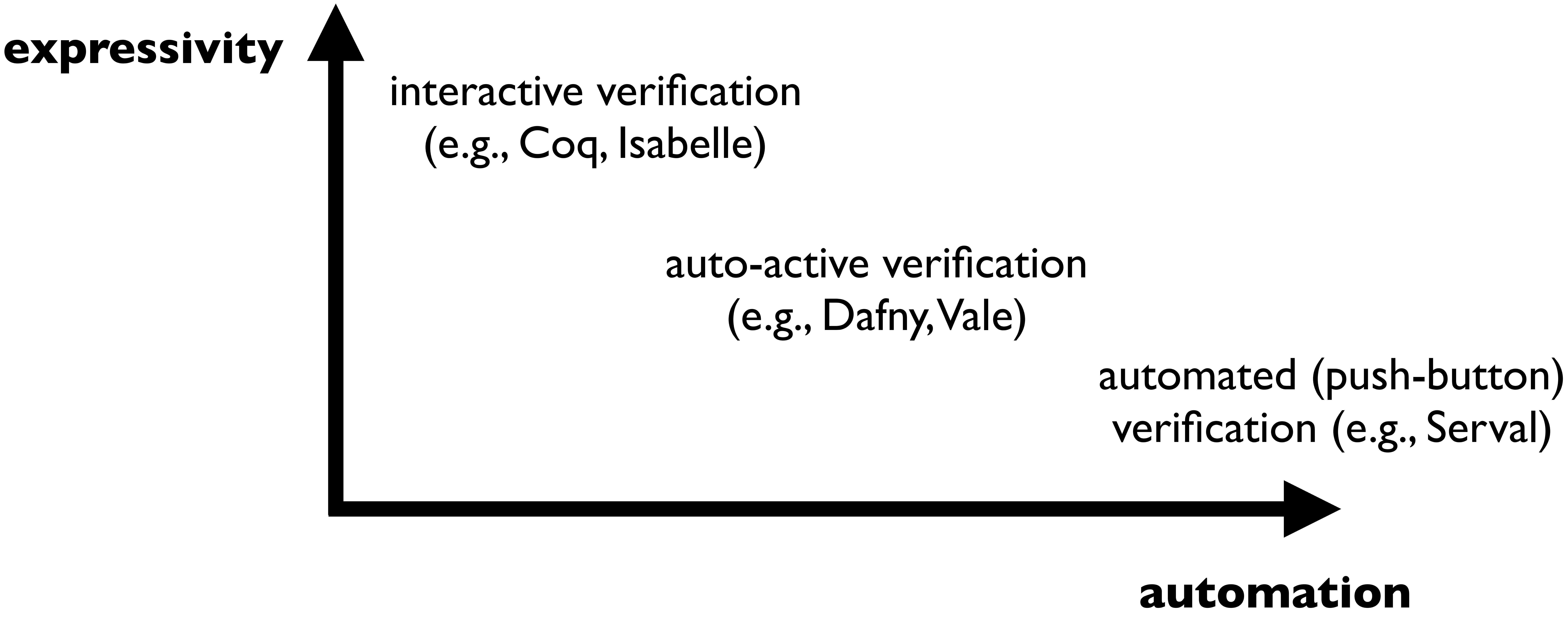
Types of systems verified

- Hardware
- OS kernels & security monitors
- File systems
- Distributed systems
- Networking
- Applications

Properties & specifications

- State-machine refinement
- Noninterference
- Crash safety
- Determinism
- Linearizability
- Liveness

Verification methodologies & tools



Overview outline

- History
- **Example: specification & verification**
- Project organization

Example: verify a 16-bit integer encoder/decoder

```rust
fn encode(x: u16) -> (u8, u8) {
    (x as u8, (x >> 8) as u8)
}

fn decode(b: (u8, u8)) -> u16 {
    (b.0 as u16) | ((b.1 as u16) << 8)
}
```

What's the specification?

Specification example

- Theorem: forall 16-bit x, decode(encode(x)) == x
- Any encoded 16-bit integer can be decoded back
- What bugs <u>cannot</u> be captured by the specification?

Specification example

- Theorem: forall 16-bit x, decode(encode(x)) == x
- Any encoded 16-bit integer can be decoded back
- What bugs cannot be captured by the specification?
  - Correctness of encode (or decode) alone
  - Whether any two bytes can be decoded and encoded back (e.g., parser)

Specification alternatives

- Theorem: forall 16-bit x, decode(encode(x)) == x
- Any encoded 16-bit integer can be decoded back
- Alternative spec: model little endianness for decode/encode
- Alternative spec: memory safety for encode/decode

Summary of specification

- Specification is key to verification
  - Theorem & model of environment
  - Any bugs in specification can invalidate guarantees
- Trade-offs
  - Expressiveness: does the spec prevent the intended bugs?
  - Simplicity: is the spec easy to audit?

Verification

forall 16-bit x, decode(encode(x)) == x

```
fn encode(x: u16) -> (u8, u8) {
    (x as u8, (x >> 8) as u8)
}

fn decode(b: (u8, u8)) -> u16 {
    (b.0 as u16) | ((b.1 as u16) << 8)
}
```

specification ≈ implementation

Verify impl against spec 1/2: enumerating

- Theorem: forall 16-bit x, decode(encode(x)) == x
- Exhaustively enumerate every possible value of x: 0 to 65535
- Easy to automate; hard to scale (if the input space is large)

Verify impl against spec 2/2: rewriting

- Theorem: forall 16-bit x, decode(encode(x)) == x
- Repeatedly apply rewrite rules until true
- Demo: run rustc/llvm

```
fn spec_encode_decode(x: u16) -> bool {
    decode(encode(x)) == x
}
```

- Hard to automate; easy to generalize to larger integer types

Summary of verification

- Two basic proof strategies
  - Search in the space of input data
  - Search in the space of rewriting rules
- Usually use a hybrid approach

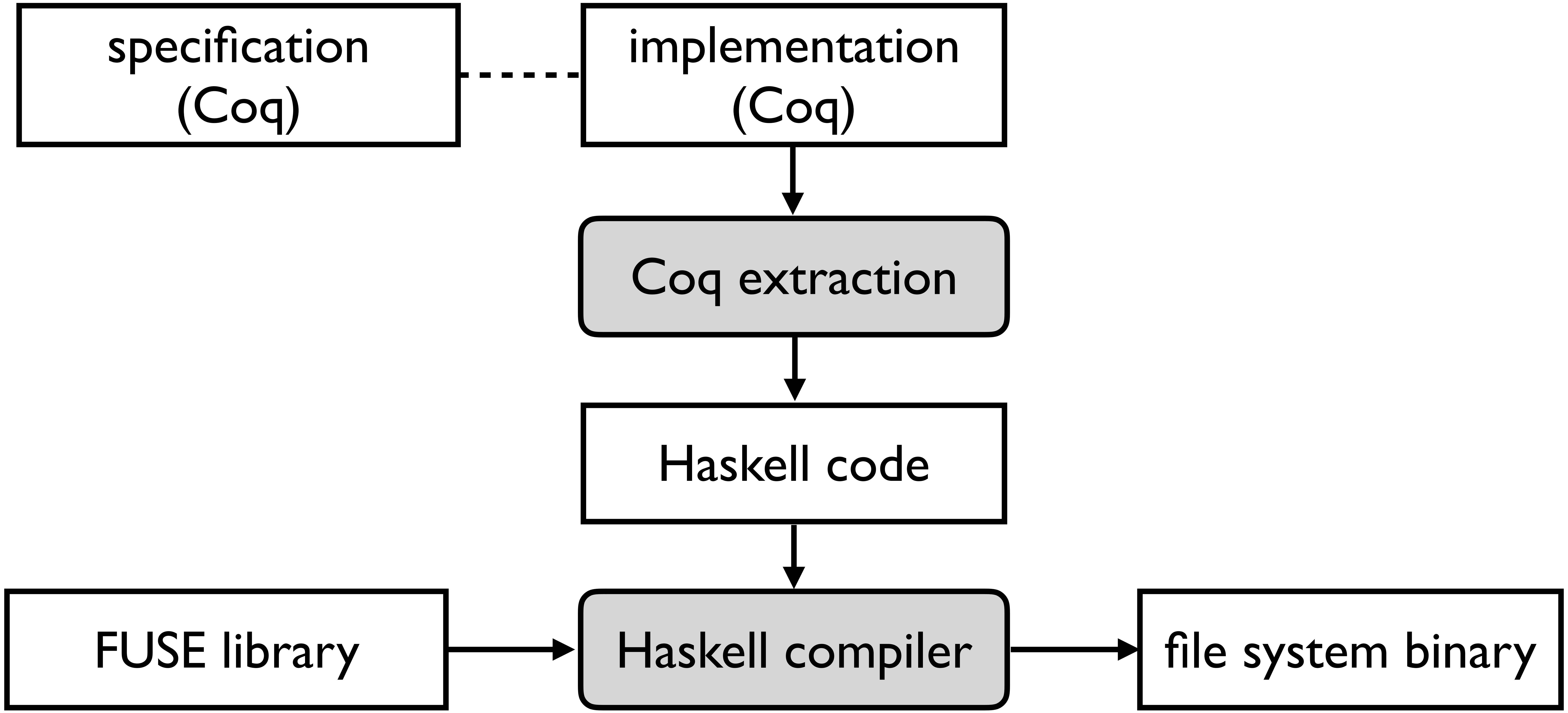
Overview outline

- History
- Example: specification & verification
- **Project organization**

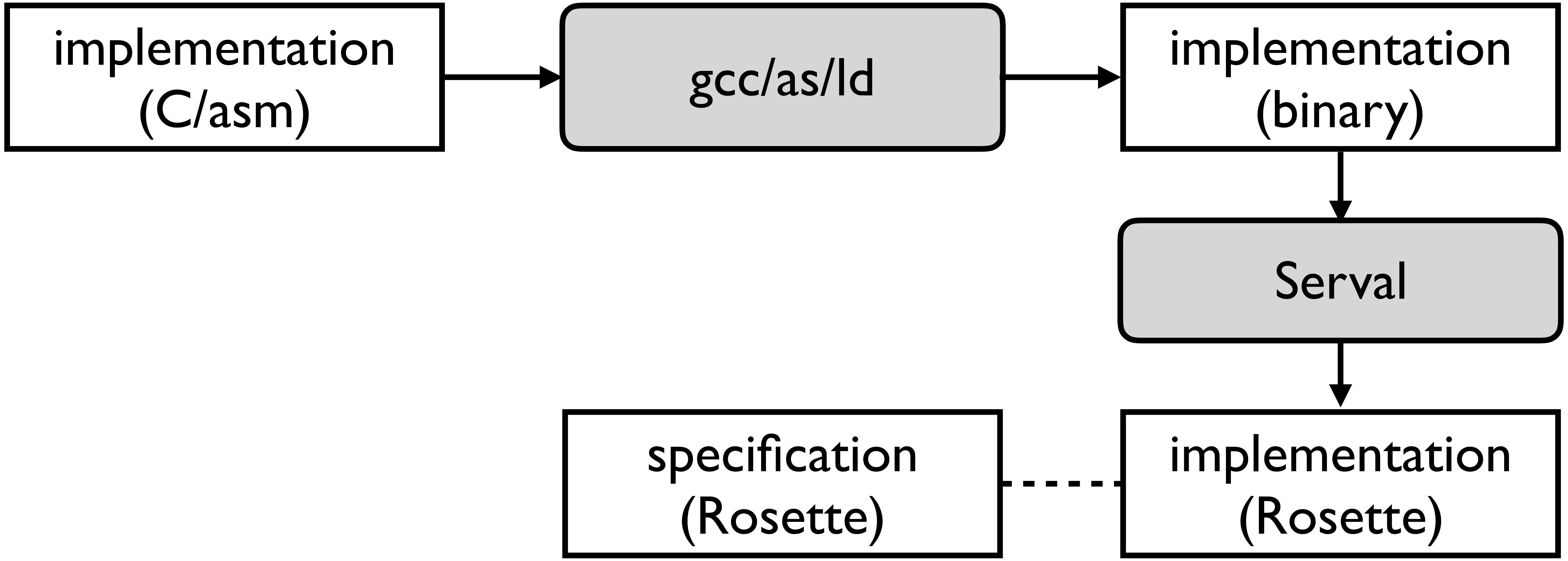
How to organize a verification project

- Extraction
- Parsing
- Verified compilation

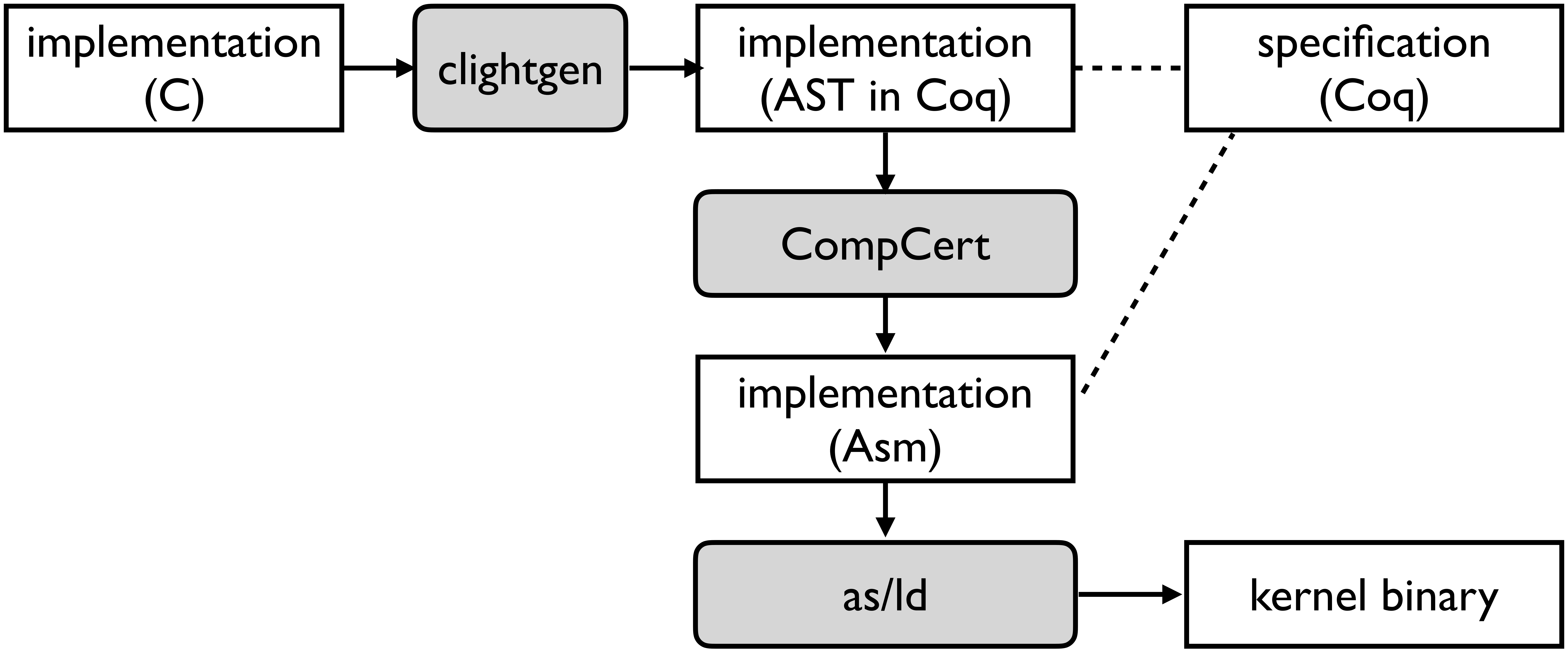
Extraction example: FSCQ



Parsing example: Serval



Verified compilation example: CertiKOS



Summary of Part I

- Verification is effective at eliminating bugs
- Choose the "right" trade-offs for your system
  - Specification & design
  - Verification methodologies
  - Turning implementation to executable code

Part 2: Serval

- Overview of Serval
- A toy monitor
- Example: free of low-level bugs
- Example: functional correctness
- Example: safety properties

Verification stack

System specification

RISC-V instructions

x86-32 instructions

LLVM instructions

BPF instructions

RISC-V verifier

x86-32 verifier

LLVM verifier

BPF verifier

Serval:
Specification library, symbolic optimizations, machine code support

Rosette:
Symbolic evaluation, symbolic profiling, symbolic reflection

Z3

SMT solver:
Constraint solving, counterexample generation

Verification stack

System specification

RISC-V instructions

x86-32 instructions

LLVM instructions

BPF instructions

RISC-V verifier

x86-32 verifier

LLVM verifier

BPF verifier

Serval:
Specification library, symbolic optimizations, machine code support

Rosette:
Symbolic evaluation, symbolic profiling, symbolic reflection

SMT solver:
Constraint solving, counterexample generation

Verification stack

System specification

RISC-V instructions

x86-32 instructions

LLVM instructions

BPF instructions

RISC-V verifier

x86-32 verifier

LLVM verifier

BPF verifier

Serval:
Specification library, symbolic optimizations, machine code support

Rosette:
Symbolic evaluation, symbolic profiling, symbolic reflection

Z3

SMT solver:
Constraint solving, counterexample generation

Verification stack

System specification

RISC-V instructions

x86-32 instructions

LLVM instructions

BPF instructions

RISC-V verifier

x86-32 verifier

LLVM verifier

BPF verifier

Serval:
Specification library, symbolic optimizations, machine code support

Rosette:
Symbolic evaluation, symbolic profiling, symbolic reflection

SMT solver:
Constraint solving, counterexample generation

Verification stack

System specification	RISC-V instructions	x86-32 instructions	LLVM instructions	BPF instructions
	RISC-V verifier	x86-32 verifier	LLVM verifier	BPF verifier

Serval:
Specification library, symbolic optimizations, machine code support

Rosette:
Symbolic evaluation, symbolic profiling, symbolic reflection

SMT solver:
Constraint solving, counterexample generation

Verification stack

System specification

RISC-V instructions

x86-32 instructions

LLVM instructions

BPF instructions

RISC-V verifier

x86-32 verifier

LLVM verifier

BPF verifier

Serval:
Specification library, symbolic optimizations, machine code support

Rosette:
Symbolic evaluation, symbolic profiling, symbolic reflection

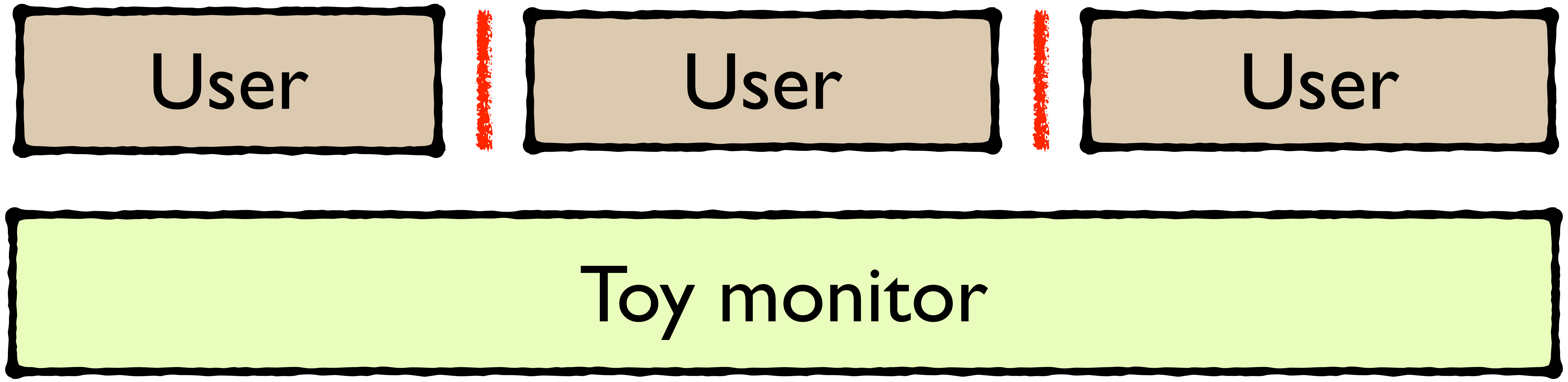
Z3

SMT solver:
Constraint solving, counterexample generation

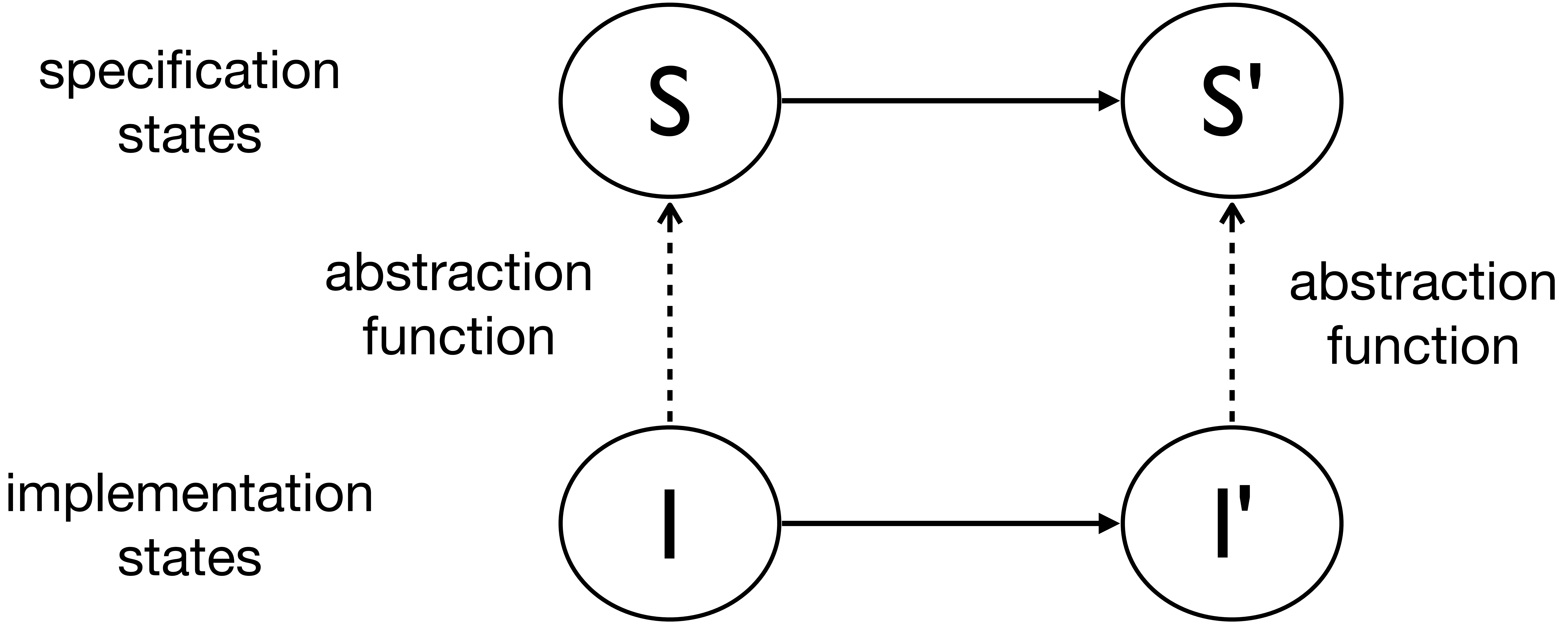
What you can do with Serval

- **Verify functional correctness and safety properties**
- Verify system designs: Keystone
- Verify existing code: BPF JIT in the Linux kernel

A toy monitor

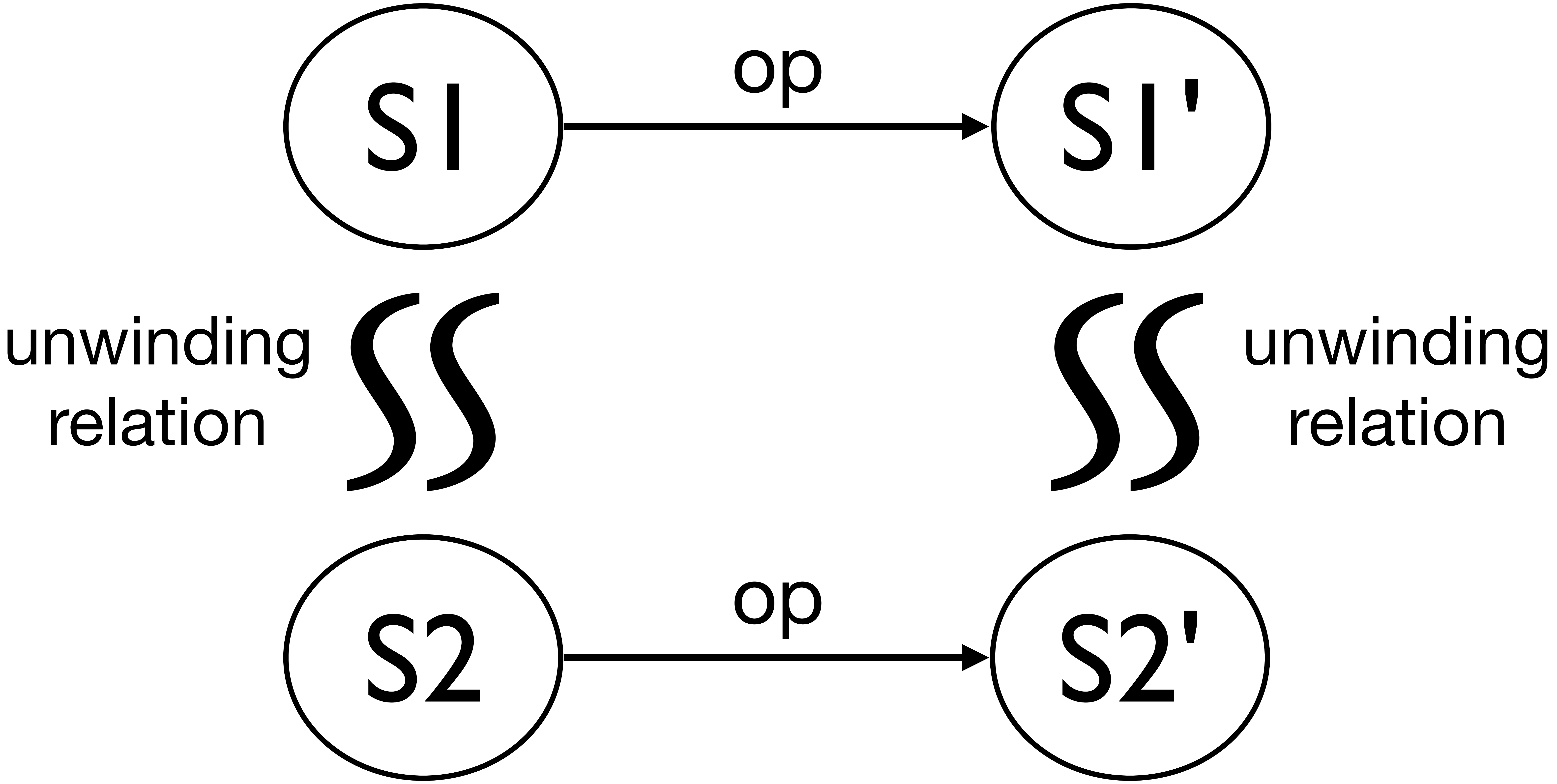


Verifying state-machine refinement



Refinement: both spec and impl state machines move in lock-step

Verifying noninterference (step consistency)



Confidentiality: stepping from two equivalent states results in equivalent states
(e.g., parts of the state that cannot be observed by user P cannot affect its execution)