# Model Checking and Systems Correctness

Mike He, Yihong Zhang

# Agenda

- More backgrounds on model checking
  - Transition systems and temporal logics
  - Explicit-state model checking
  - Symbolic model checking
  - Abstract model checking
- Comparison with other systems verification tools
  - Interactive Verification
  - Automatic Verification (SMT)
  - Static Analysis
  - Symbolic Execution / Concolic Execution

# Model Checking

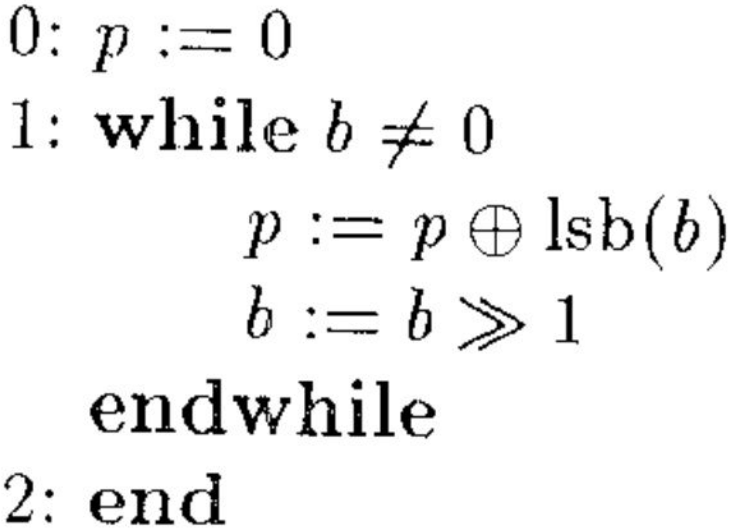
- An automated technique for verifying if a finite-state program satisfies the specification.
- Goal: proving M, s ⊧ P, where M is a finite state transition system, s is a state in M and P is a temporal logic formula.
- Why and when model checking.
    - Used for finite-state programs.
    - Control-intensive and concurrent systems (e.g., distributed systems).
    - When the specification can be expressed in fragments of temporal logic like CTL, LTL that can be solved efficiently.
    - Completely automatic, no manual proof burden.

# Transition Systems

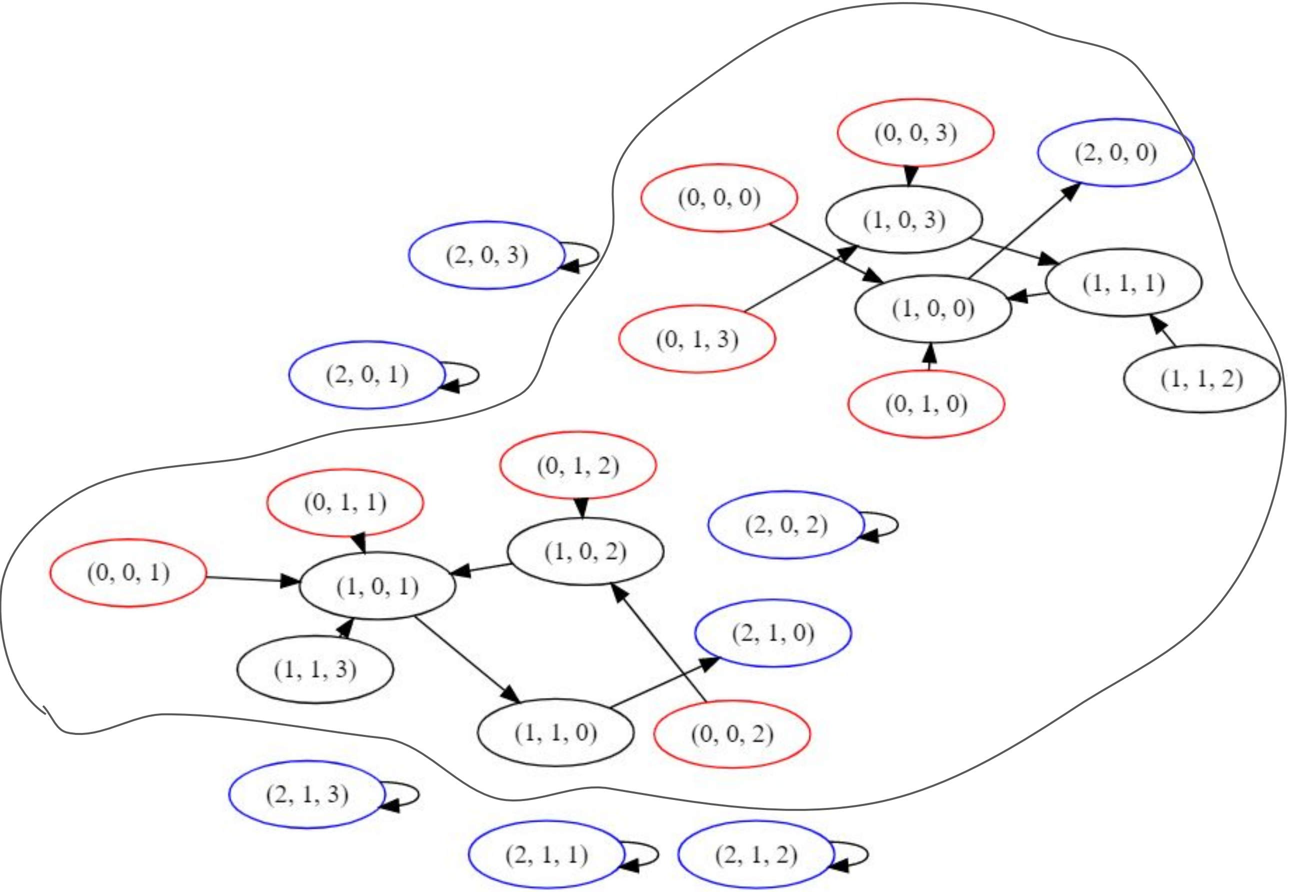
- For a program with state $d \in D$. We define a transition system M over D to be a triple (S, I, R) where
    - $S = D$ is the set of states,
    - $I \subseteq S$ is a set of initial states, and
    - $R \subseteq S \times S$ is a transition relation.
- If D is finite, then M is called a finite-state transition system.

# Modeling programs as transition systems

- M = (S, I, R) **where**:
- S = { (PC, p, b) **for** PC **in** [0, 2], p, b **in** [0, 2^k) }
- I = { (0, p, b) **for all** p, b }
- R = {
- (0, p, b) => (1, 0, b) **for all** p, b;
- (1, p, 0) => (2, p, 0) **for all** p;
- (1, p, b) => (1, p⊕lsb(b), b>>1) **for all** p, b s.t. b ≠ 0
- }



```
0: $p := 0$
1: while $b \neq 0$
           $p := p \oplus \mathrm{lsb}(b)$
           $b := b \gg 1$
   endwhile
2: end
```



Red = initial states
Blue = terminal states
Reachable terminal states = {(2,0,0), (2,1,0)}

# Temporal Logic

- We are interested in the transition behavior over time.
- Five basic temporal operators:
    - **X** p: p holds at the next point in time.
    - p **U** q: p holds until q holds.
    - p **V** q: p releases q; q holds until p holds (if ever).
    - **F** p: p holds at some future point (= true **U** q).
    - **G** p: p holds globally on the path (= false **V** p).
- These operators describe properties of a path π.
    - A path π is defined as an infinite sequence s0,s1,s2,... such R(si, si+1).

# Examples

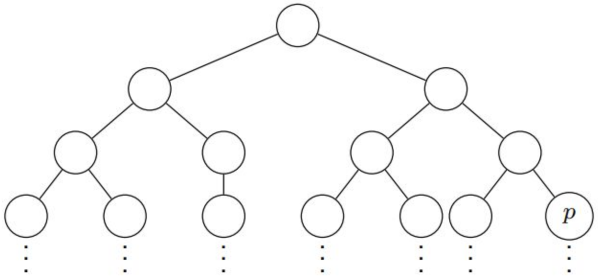
- **F** *lunch*
  - Lunch will come eventually
- *terminate* **U** *serve*
  - Requests are served until the connection terminates (if ever)
- **G** (*request* -> **F** *response*)
  - Each request is always followed by a response
- **F G** (not *p*)
  - *p* holds only finitely often
- **G** (*start* -> **F** *heat*)
  - Whenever the start button is pressed, the oven heats eventually

# Linear Temporal Logic

- State formula (corresponding to facts that hold in a particular state)
  - f ::= true | false | p | not f | f1 or f2 | f1 and f2
- Path formula (only makes sense when evaluated along a particular path)
  - g ::= f | not g | g1 or g2 | g1 and g2 | **X** g | **F** g | **G** g | g1 **U** g2 | g1 **V** g2
- LTL universally quantifies over paths, describes linear-time properties (i.e., no branches).
- M, π ⊧ f ⇔ “path formula f holds along path π in M”.
- M, s ⊧ f ⇔ “path formula f holds along all path starting at s in M”.
- We can extend LTL with quantifications over path.

# Computational Tree Logic * (CTL*)

- We extend **state** formulas with two new forms
- **E** g is a state formula, if g is a path formula
    - M, s ⊧ **E** g ⇔ these exists a path π starting at s where M, π ⊧ g.
- **A** g is a state formula, if g is a path formula
    - M, s ⊧ **A** g ⇔ for all paths π starting at s, M, π ⊧ g.
- CTL* formulas describe branching-time properties.
    - Path quantifiers can talk about multiple possible futures
    - E.g., **EF** p

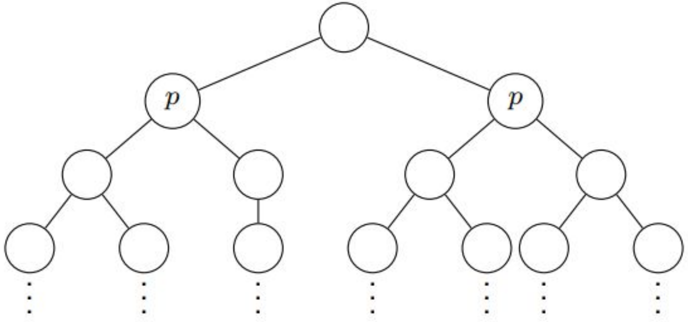


# Computation Tree Logic (CTL)

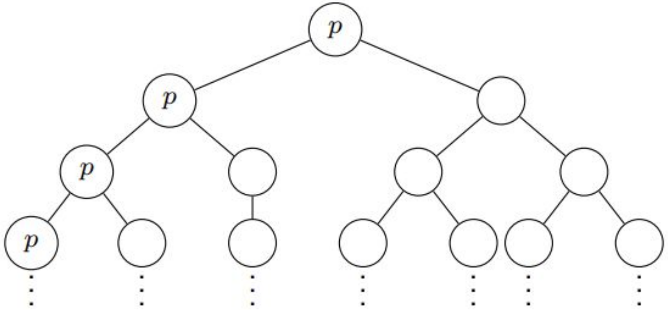
- CTL is a fragment of CTL* where each temporal operator is preceded by a path quantifier.
- Equivalently, in CTL, if p and q are state formulas,
  - **AX** p, **AF** p, **AG** p, **A** (p **U** q), **A** (p **V** q) are state formulas, and
  - **EX** p, **EF** p, **EG** p, **E** (p **U** q), **E** (p **V** q) are state formulas.

# Examples: CTL

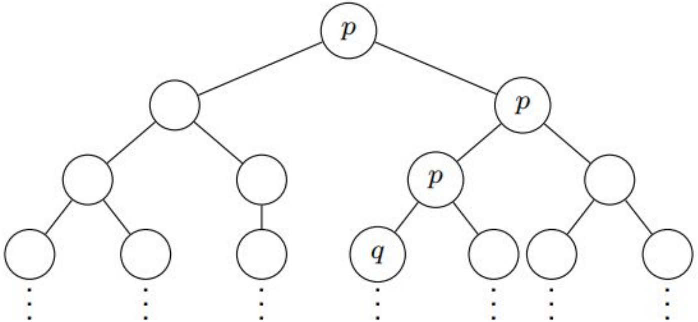
- what does **AX** p look like?



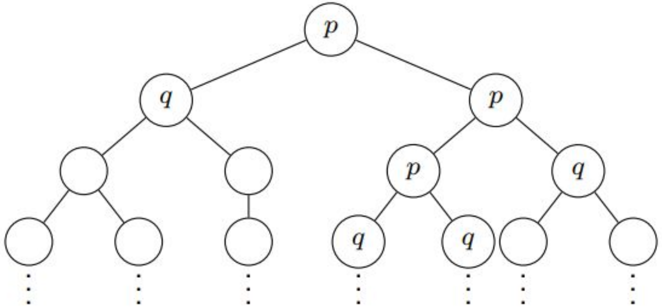
- what does **EG** p look like?



- what does **E** (p **U** q) look like?



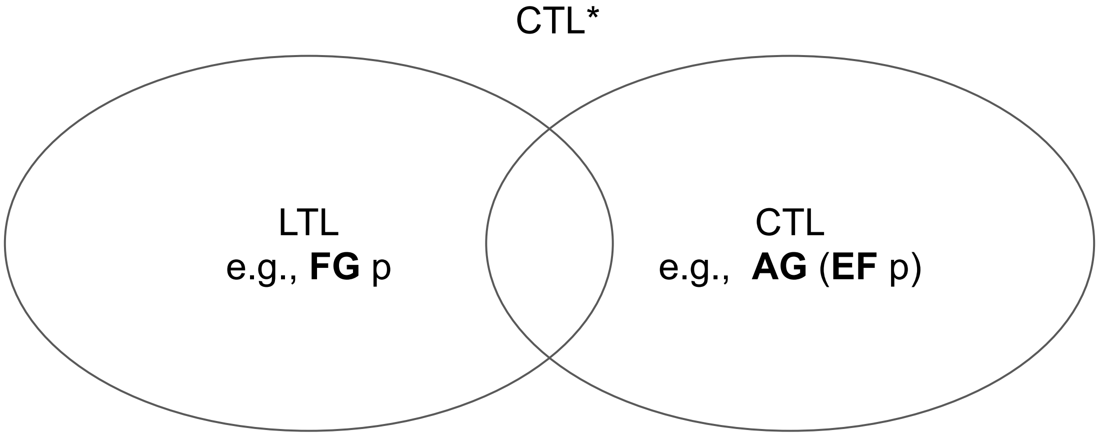
- what does **A** (p **U** q) look like?



# Examples

- **EF** (start and not ready)
  - Can get to a state where start holds by ready doesn’t
- **AG** (**EF** restart)
  - It is always possible (from any state) to enter restart
- **AF** (**AG** p)
  - No matter what, p eventually always holds on all paths
- **AG** (**AF** crit)
  - On all paths at all times, can always enter crit later

# LTL vs CTL



CTL*

LTL
e.g., **FG** p

CTL
e.g., **AG** (**EF** p)

# Discussion

https://tinyurl.com/550mc

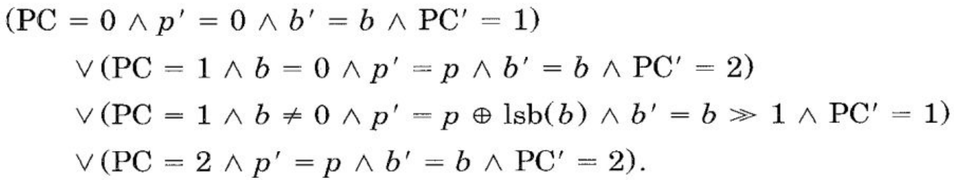
1. What kind of system properties people would like to verify?
2. Are they expressible in temporal logic?

# Explicit-state Model Checking

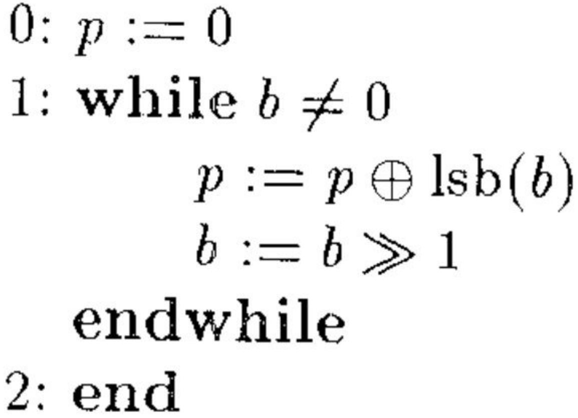
- To check whether M, s ⊧ P for all s in initial states I
- Compute sat(P), the set of all states that satisfy P
    - … by reasoning about the temporal operators
- Check if I is a subset of sat(P).
- The state explosion problem.

# Symbolic Model Checking

- Idea: represent the initial states and the transition relation as predicates.



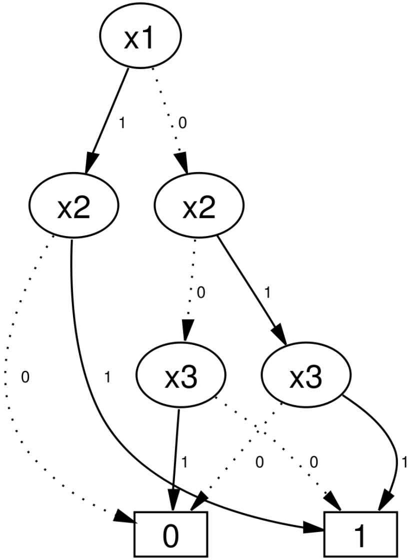
$$
\begin{aligned}
&(\mathrm{PC} = 0 \land p' = 0 \land b' = b \land \mathrm{PC}' = 1)\\
&\quad \lor (\mathrm{PC} = 1 \land b = 0 \land p' = p \land b' = b \land \mathrm{PC}' = 2)\\
&\quad \lor (\mathrm{PC} = 1 \land b \neq 0 \land p' = p \oplus \mathrm{lsb}(b) \land b' = b \gg 1 \land \mathrm{PC}' = 1)\\
&\quad \lor (\mathrm{PC} = 2 \land p' = p \land b' = b \land \mathrm{PC}' = 2).
\end{aligned}
$$



```
0: $p := 0$
1: while $b \neq 0$
       $p := p \oplus \mathrm{lsb}(b)$
       $b := b \gg 1$
   endwhile
2: end
```

# Symbolic Model Checking

- Doing it naively will still instantiate all the states.
    - E.g., **E** (p1 **U** p2).
- Use Binary Decision Diagram.



# Abstractions in Model Checking

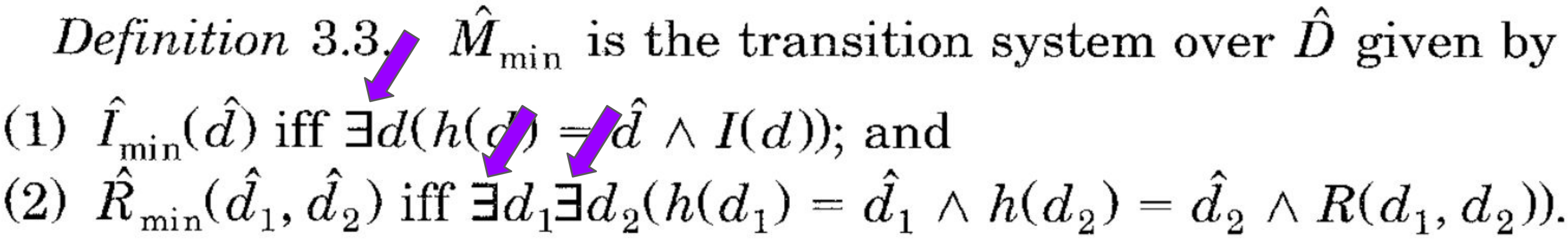
- Still too many states for even moderate programs.
- Consider a ~1000 LoC program with a few dozen 32-bit int variables
    - 1000 × 36 × 2^32 ≈ 1.5×10^14 states.
- Idea: build an alternative TS that is an approximation of the original TS.
    - The approximate TS should be more conservative.
    - If M’ is an conservative approximation of M, then properties hold for M’ also hold for M.
    - Therefore, if M’ verifies, so will M.

# Abstractions in Model Checking

- Define the abstract state to be D’.
- Define h to be the abstraction function from D to D’.
- M’ is an abstract model of M if
  - For all d’, if exists some d s.t. d’=h(d) and d is some initial state in M, then d’ should also be an initial state in M’; and
  - For all d’1, d’2, if exists some d1, d2, s.t., d’1=h(d1), d’2=h(d2) where (d1, d2) is a transition in M, then (d’1, d’2) should also be a transition in M’.
- M’ is a minimal model of M if changing replace the “then” part with “if and only if”.
  - Any other abstract model of M is necessarily a superset of the minimal model.

# Problems with the minimal model

- To check minimal model, one still need to find *global* concrete witnesses from the transition relation of the original model (i.e., d1, d2 where R(d1, d2) holds).
- Intuitively, no information is lost in the minimal model, so checking the minimal model is as hard as the checking the original model.



*Definition 3.3.* $\hat{M}_{\min}$ is the transition system over $\hat{D}$ given by

(1) $\hat{I}_{\min}(\hat{d})$ iff $\exists d(h(d) = \hat{d} \wedge I(d))$; and

(2) $\hat{R}_{\min}(\hat{d}_1, \hat{d}_2)$ iff $\exists d_1 \exists d_2(h(d_1) = \hat{d}_1 \wedge h(d_2) = \hat{d}_2 \wedge R(d_1, d_2))$.

# Approximate Models

- Instead, the approximate model loses information and is more practical.
- Existential quantifiers are pushed inside the formula
- Now, only need to find local witnesses for each transition.

# Example: Approximate Models

- Program
    - 0: x = x > 0 **and** x < 0 **?** 1 **:** 0
    - 1: **print** b
- Minimal Model
    - R'(x'1, x'2) **iff** exists x1, x2 such that
        - h(x1) = x'1,
        - h(x2) = x'2
        - ((x1 > 0 **and** x1 < 0) **implies** x2 = 1)
        - (**not** ((x1 > 0 **and** x1 < 0)) **implies** x2 = 0)
- Approximate Model
    - R'(x'1, x'2) iff
        - ((exists x1: h(x1)=x'1 **and** x1 > 0) **and**
        - (exists x1: h(x1)=x'1 **and** x1 < 0) **and**
        - (exists x2: h(x2)=x'2 **and** x2 = 1)) **OR**
        - **…**

# Common abstractions described in the paper

- Congruence modulo an integer
- Representation by Logarithm
- Single-Bit and Product Abstractions
- Symbolic abstractions

# Predicate abstractions

- Idea: only track predicates we are interested in on program states
- Abstraction function: $h(s) = (\phi 1(s), \ldots, \phi n(s))$
- Each state in the transition relation maps to a vector of predicate values
- Use hoare logic to reason the transition of abstract states (i.e., weakest preconditions).
- Often times, the tracked predicate may be too coarse to prove the desired properties
  - Idea: Refine tracked predicates with new, spurious counterexamples, rerun model checking again
  - Counterexample-Guided Abstraction Refinement (CEGAR)

# Case study: TLA+ at Amazon Web Services

- TLA+ & PlusCal
    - TLA+: A high-level specification language
    - PlusCal: Imperative language that compiles to TLA+
    - Checked with TLC model checker
- Motivation
    - Complexity of the AWS services become high
    - Interaction between different part of the system (e.g, load balancing, consistency, concurrency control, etc.) require modifications on the algorithms themselves
    - Verification techniques (unit test, fault-injection testing, etc.) are not sufficient: hard to catch “rare” cases in concurrent programs
- Have helped identify bugs in S3, DynamoDB, etc.
    - Was able to find a bug that will only be triggered after ~30 steps

# Other correctness tools

- Formal Verification (Interactive Verification / Automatic Verification)
- Static Analysis
- Symbolic Execution / Concolic Testing

# Interactive Verification

- Use formal logic to reason about programs.
- Requires developers to write machine-checkable proof.
  - Some famous proof checkers (assistants): Coq, Isabelle, Lean, Agda, Arend, etc.
- Mechanically checked by theorem provers.
- High confidence!

# Case study: seL4 [SOSP ‘09]

- First formally verified Operating System Kernel
- Prototyped in Haskell and C; Formalized and proved in Isabelle/HOL
- 10k lines of code (in Haskell / C / assembly), ~160k lines of mechanized proof, takes 25–30 person years
- Compiled using CompCert, a formally verified C compiler

# Case study: Verdi [PLDI ‘15]

- Formally verified distributed KV store based on Raft.
- Formalize network as operational semantics and build semantics for fault models (e.g., drop, failure).
- Proved in Coq and extracted in OCaml.
- 1k lines of implementation, 5k lines for proving serializability, 30k lines for proving state machine safety.
- Once verified, 10% performance overhead.

# Engineering Overhead - Proof Engineering

- seL4
    - Implementation: **10k** lines of code (in Haskell / C / assembly)
    - Proves: **~160k** lines of mechanized proof, takes 25–30 person years

- Verdi
    - Implementation: **1k** lines of Coq
    - Proves: **5k** lines for proving serializability, **30k** lines for proving state machine safety in Coq

**Implementation <<< Proof engineering**

Survey of Proof Engineering: *QED at Large* by Talia Ringer et al.

# Automated Verification

- Use automated decision procedure like SMT solvers to prove the correctness of programs.
- Needs handling of infinitary paths.
  - Finitize programs
  - User-annotated loop invariant
- “Push-button” verification.

# Case study: Hyperkernel [SOSP ‘17]

- Verified OS kernels using the Z3 SMT solver.
- Verification is done at the LLVM level to avoid complicated bugs.
- “Verification [of the implementation] finishes within about 15 minutes on an 8-core machine.”
- Kernel Implementation: **~7600 LoC** in C and Assembly
- Specifications: **~1000 LoC** in Python
- Verifier: **~2800 LoC** in C++ and Python

# Case study: Dafny & F🌟 (F-star)

- Dafny and F* are languages for **auto-active verification**
  - **Auto-active = Automatic + Interactive**
- Dafny: Imperative, Java-like; Hoare triple inference
  - Weakest precondition
  - Need to manually provide loop invariants
- SMT solving could be very slow in practice
- Dafny is used in Amazon AWS for verifying security-critical libraries (e.g. encryption services)
- F*: Functional, Refinement Type; the aim is similar

# Static Analysis

- Directly done on source code / bytecode
- Examples: Abstract Interpretation (AI), Extended Static Checking (ESC)
- Pros: mostly automated
- Cons: limited capabilities; more false positives (AI overestimates errors like abstract model checking)

# Case study: INFER (Facebook, aka Meta)

- Infer: a static analysis tool for C#, Java, Obj-C to catch potential bugs (for memory safety)
- Can check errors like dereferencing null pointers, resource leak, deadlock, etc
- Uses Separation Logic, which analyzes programs with pointers (references).
- Fairly efficient: seconds to minutes
- Used for lots of industrial applications
  - Integrated in CI and **code review** pipeline of Spotify, Facebook, etc.
- See more at Volume 6 of Software Foundations

# Symbolic Execution / Concolic Testing

- Symbolic Execution: instead of running the program with an actual input, treat inputs as symbolic values and encode the state of the program at a specific point into machine-checkable witnesses (e.g. SMT formulae)
- Concolic Testing: Concolic = Concrete + Symbolic
  - Keep a record of symbolic path conditions while running concretely
  - To improve test coverage, it will generate new concrete inputs to test the program by, for instance, negating the expression on a condition check in an If statement.

# Case study: Klee

- **a Symbolic Execution Engine** ([https://klee.github.io/](https://klee.github.io/))
- Runs on LLVM
- Supports Concolic testing
- Influential in both academia and industry
  - KleeNet: detect bugs in wireless sensors
  - ConcFuzzer at Baidu X-Lab (KLEE Workshop, 2018)
  - Embedded Software Testing at Fujitsu (KLEE Workshop, 2018)

# Discussion

https://tinyurl.com/550mc

Questions:

1. In which scenario(s) does the following approaches work better
    a. Model Checking
    b. Interactive theorem proving
    c. Static Analysis (INFER, extended static checking)
    d. Automated / Auto-active verification (Dafny, F🌟)
    e. Symbolic Execution / Concolic Testing
2. What are the challenges blocking industrial companies from adopting these correctness tools
3. Will you use any (or none) of them if you are a founder of a tech startup and why?