What We Talk about When We Talk about Formally Verified Systems

Ilya Sergey

Associate Professor, Yale-NUS College
Lead Language Designer, Zilliqa

http://ilyasergey.net

Formal Verification

Proving Correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

Formal Verification

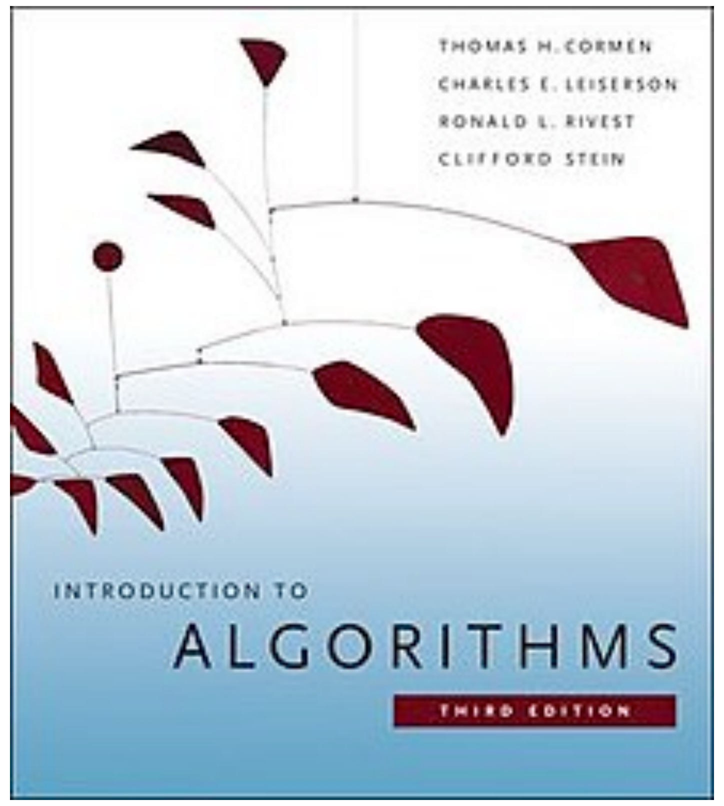
Proving Correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

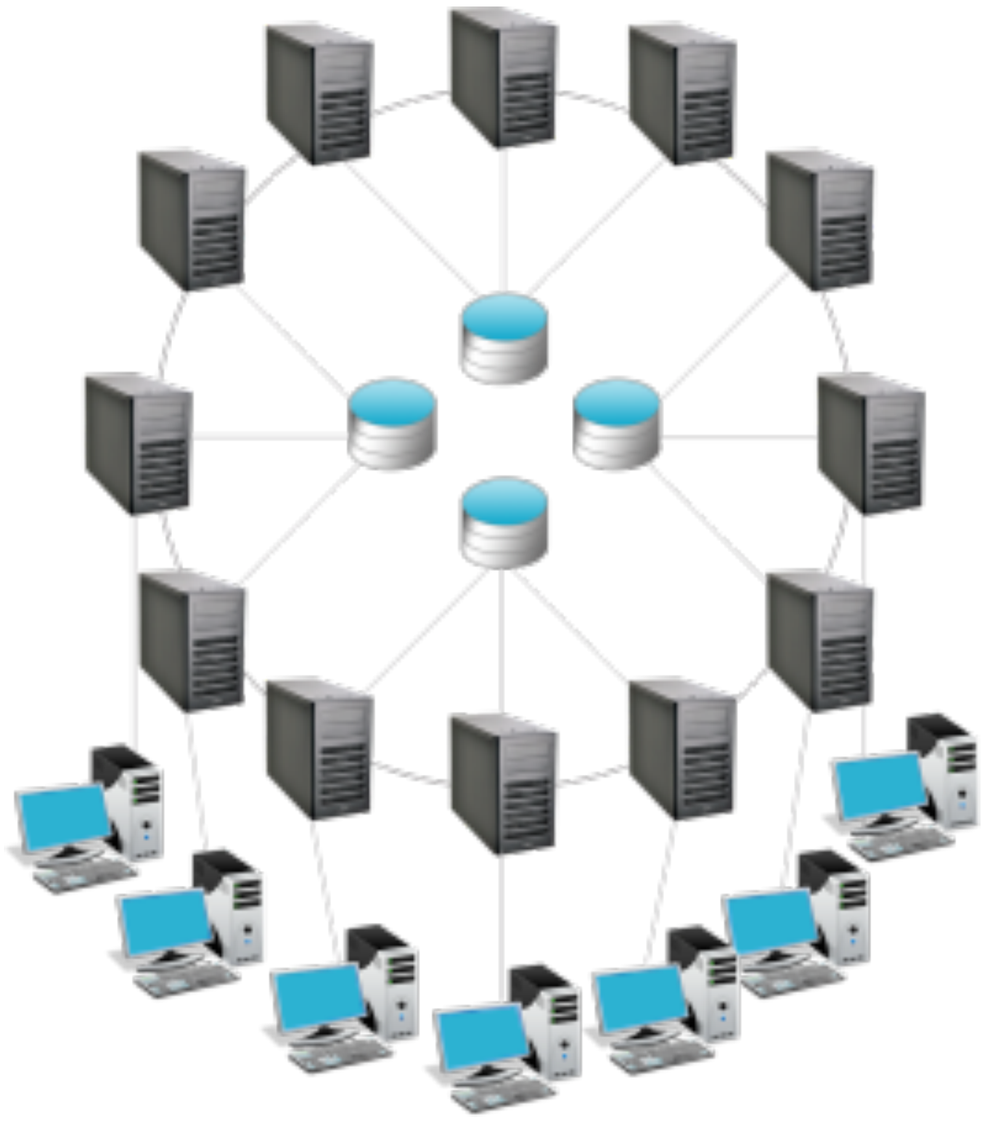
Correctness - critical software

- Implementations of textbook algorithms

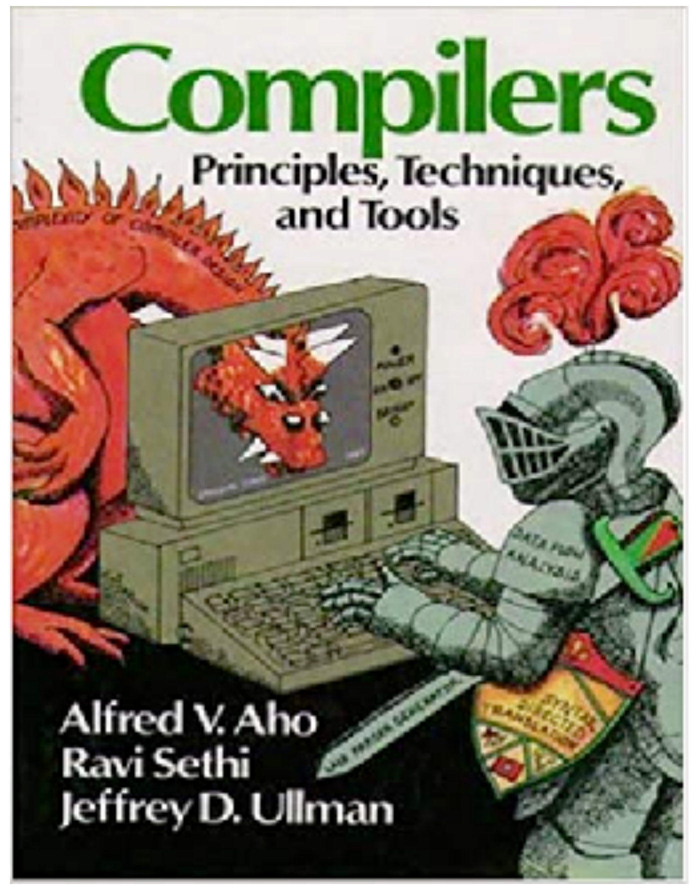


- Operational Systems

- Distributed Systems and their Applications



- Compilers



Formal Verification

Proving Correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

Formal Verification

Proving Correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

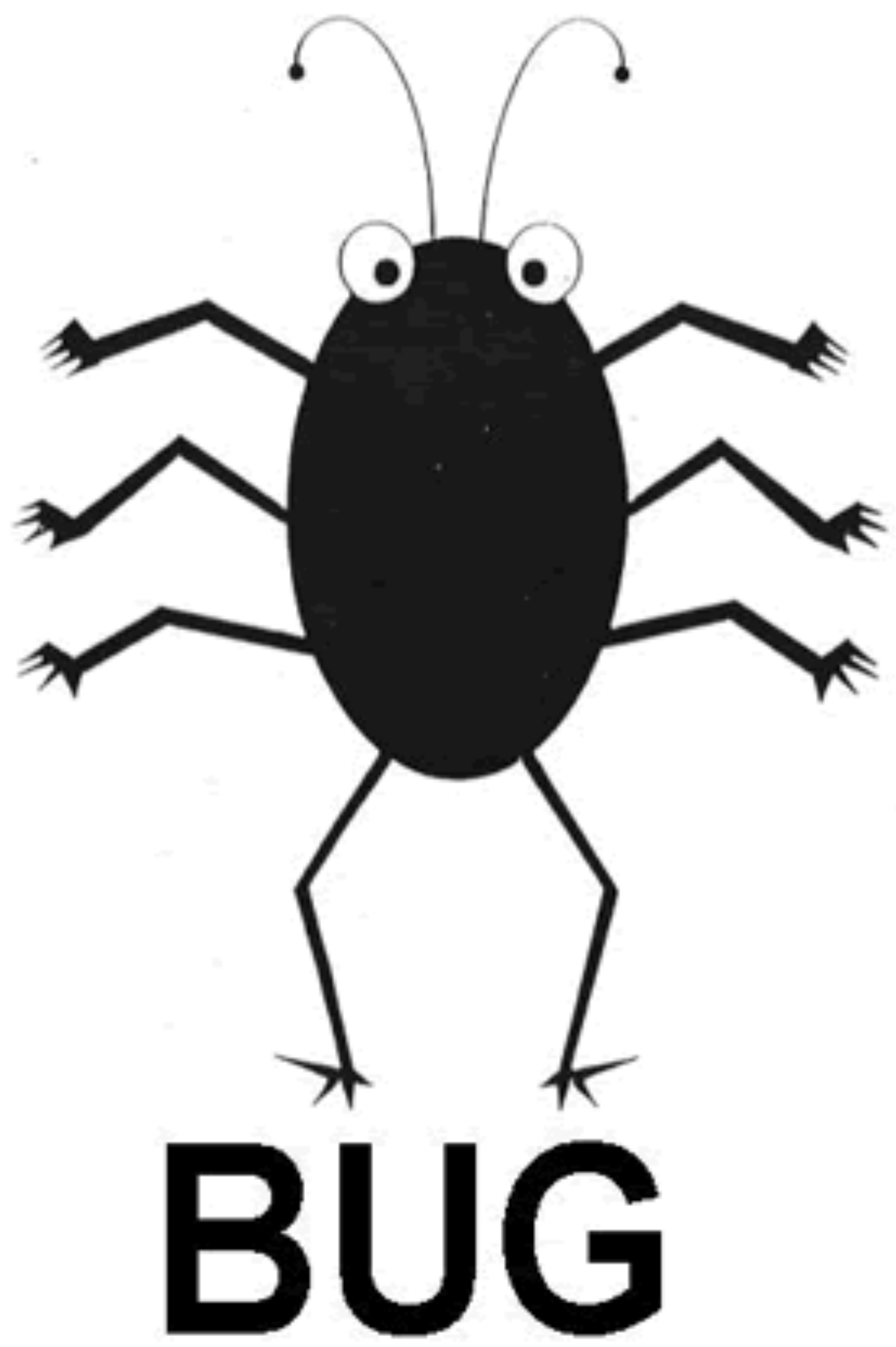
Formal Verification $\neq$ Testing

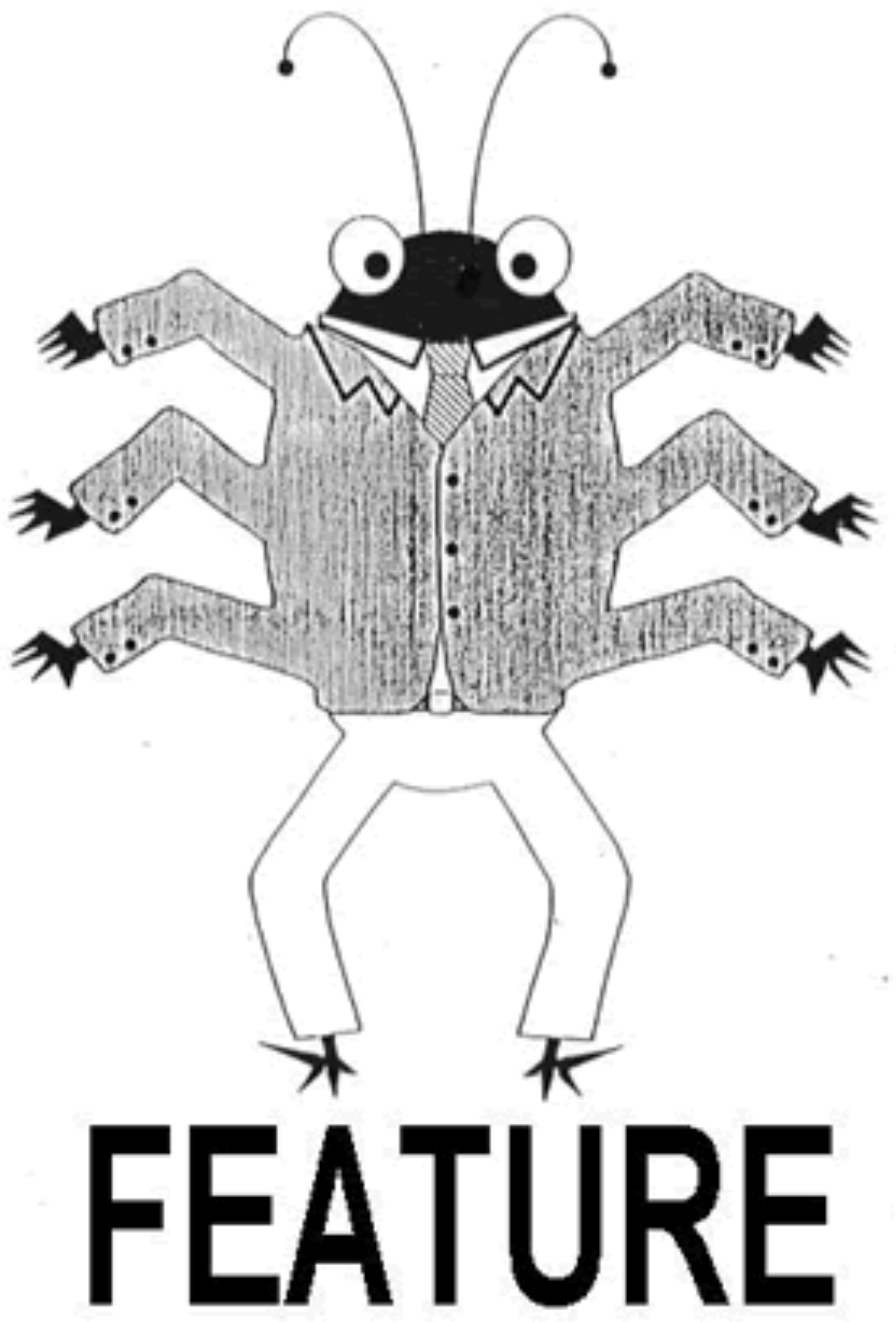
"Program testing can be used to show the ***presence*** *of bugs, but never to show their* ***absence****!"*

Edsger W. Dijkstra

But the bugs are in the eye of the beholder!

But the bugs are in the eye of the beholder!





specification

Formal Verification

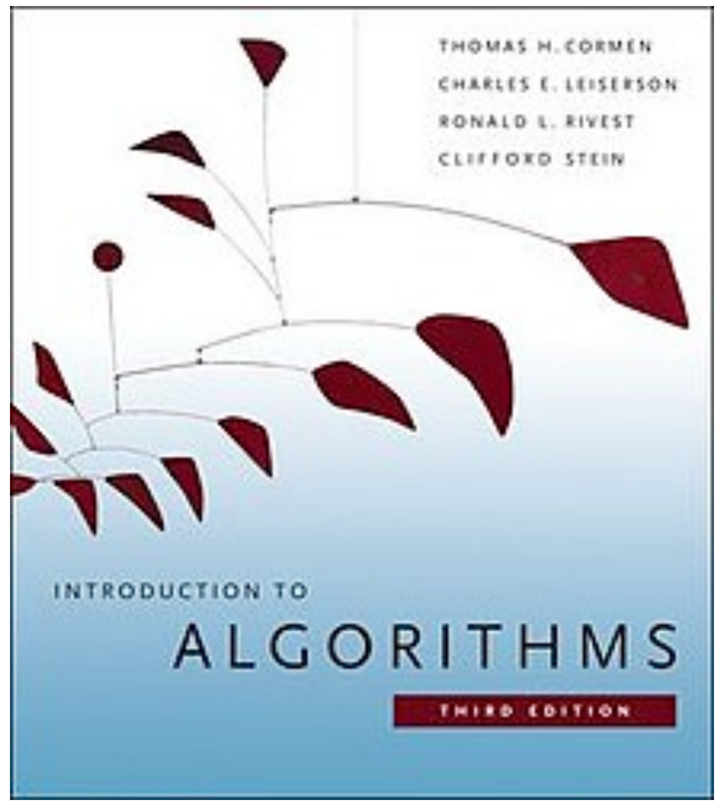
Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

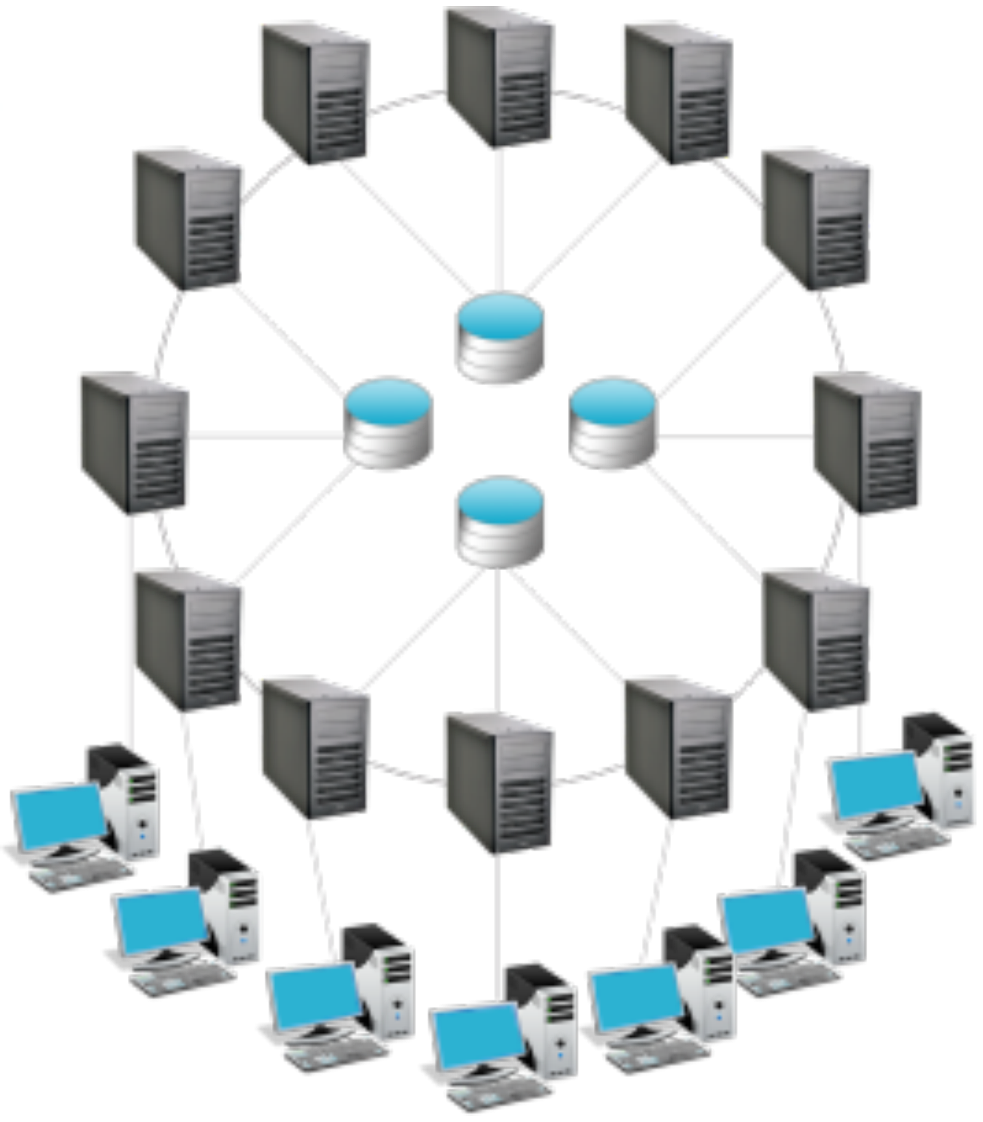
using mathematical reasoning.

Correctness-critical software

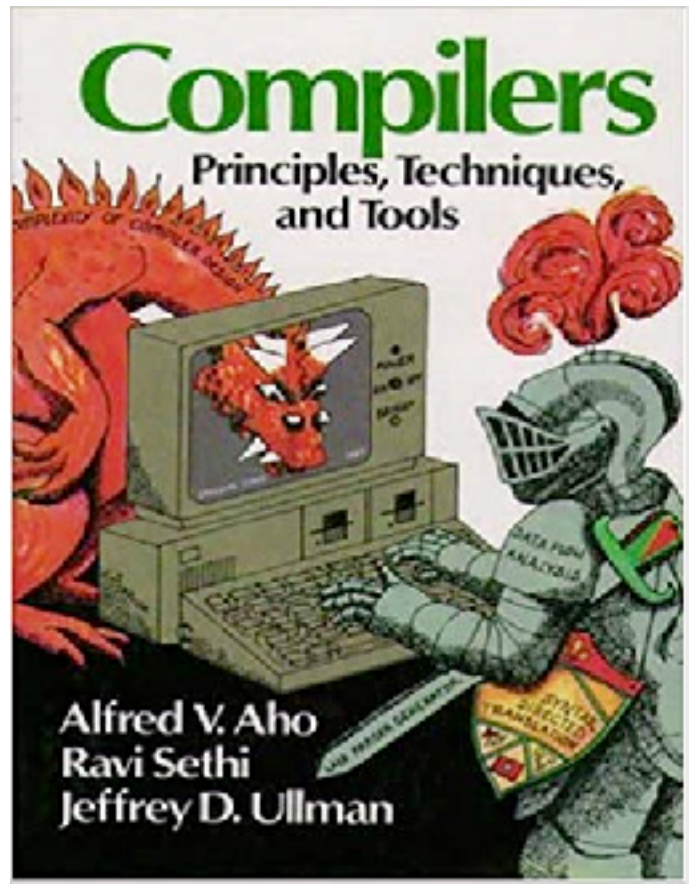
- Implementations of textbook algorithms



- Operational Systems

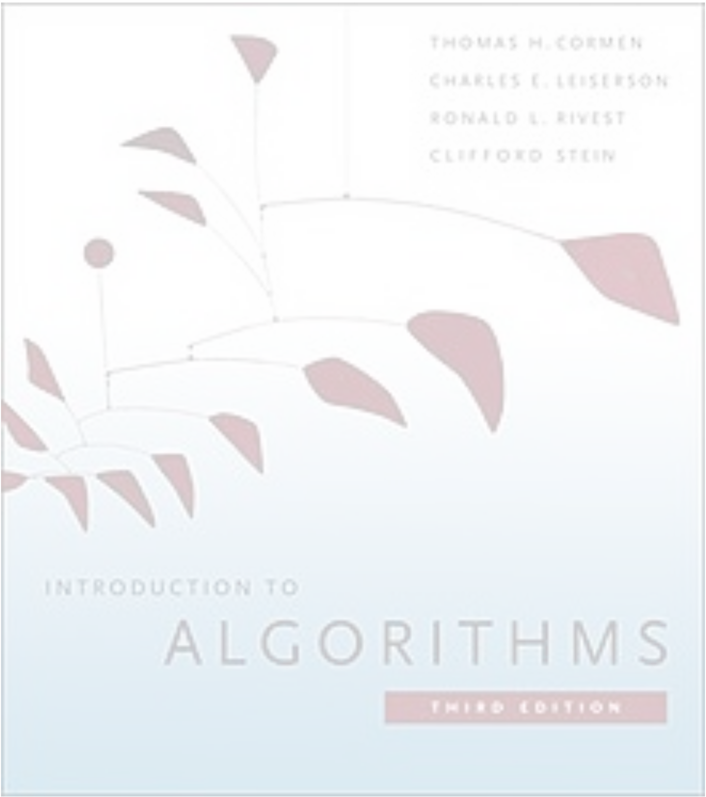


- Distributed systems and their applications
- Compilers

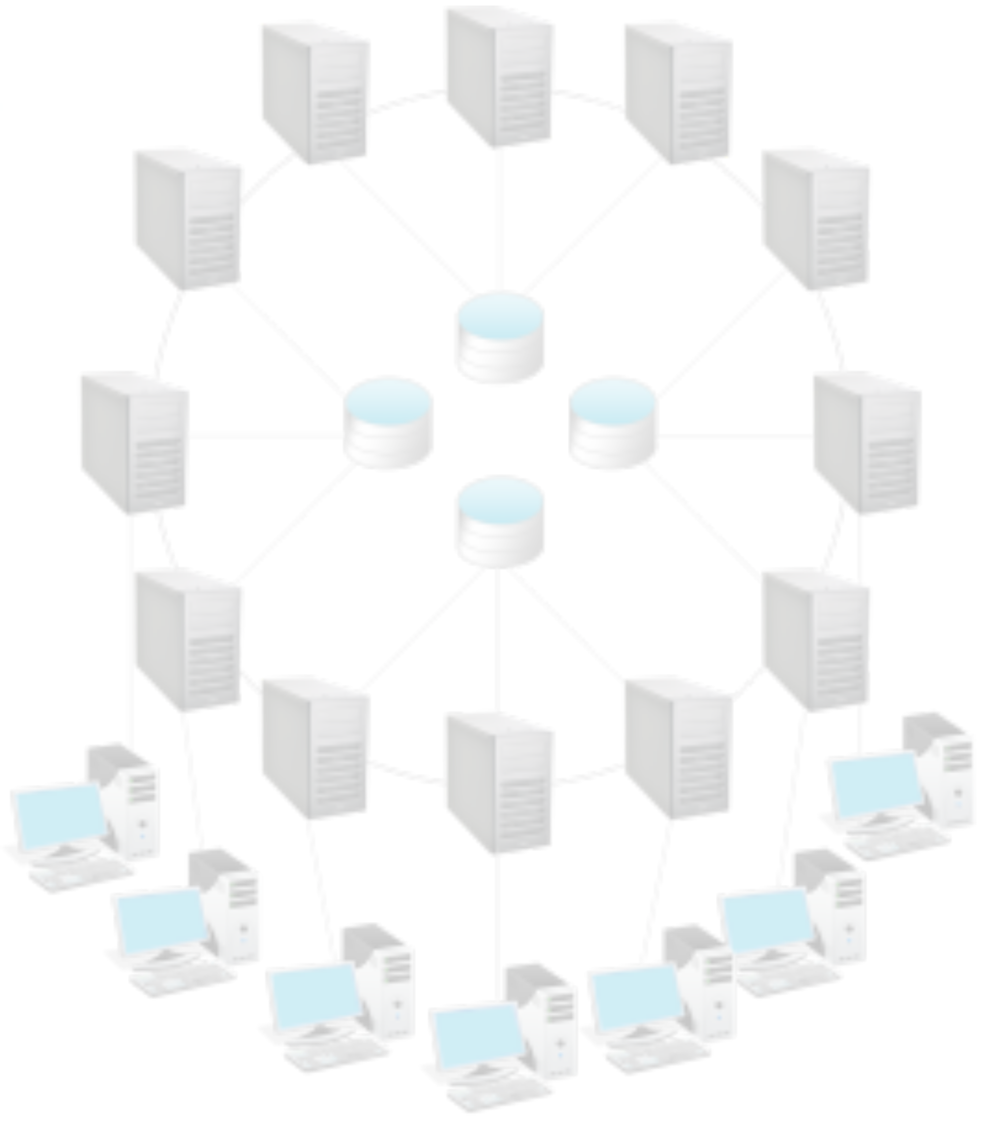


Correctness-critical software

- Implementations of textbook algorithms

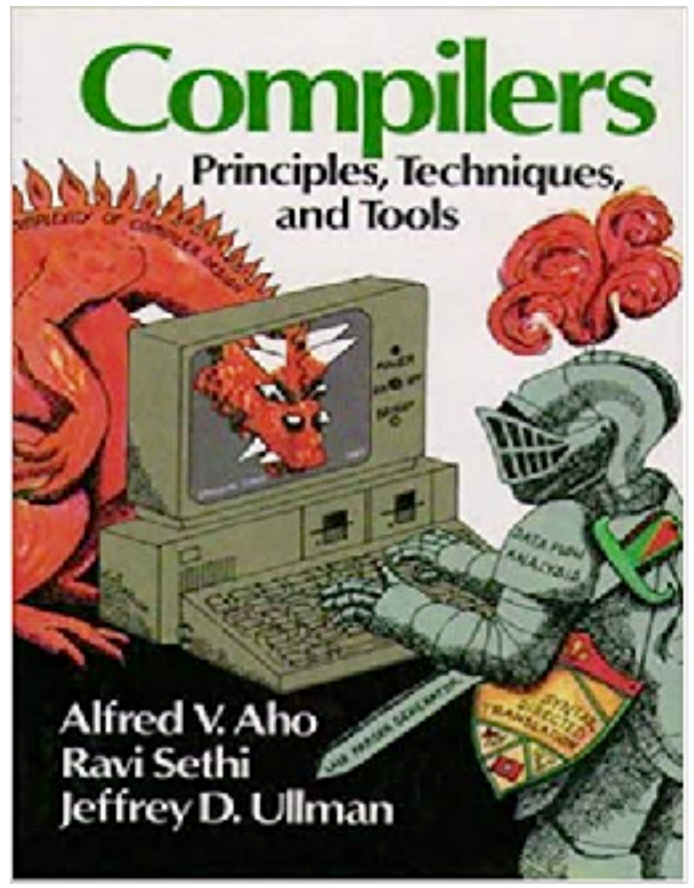


- Operational Systems



- Distributed systems and their applications

- Compilers



Specifying Compilers

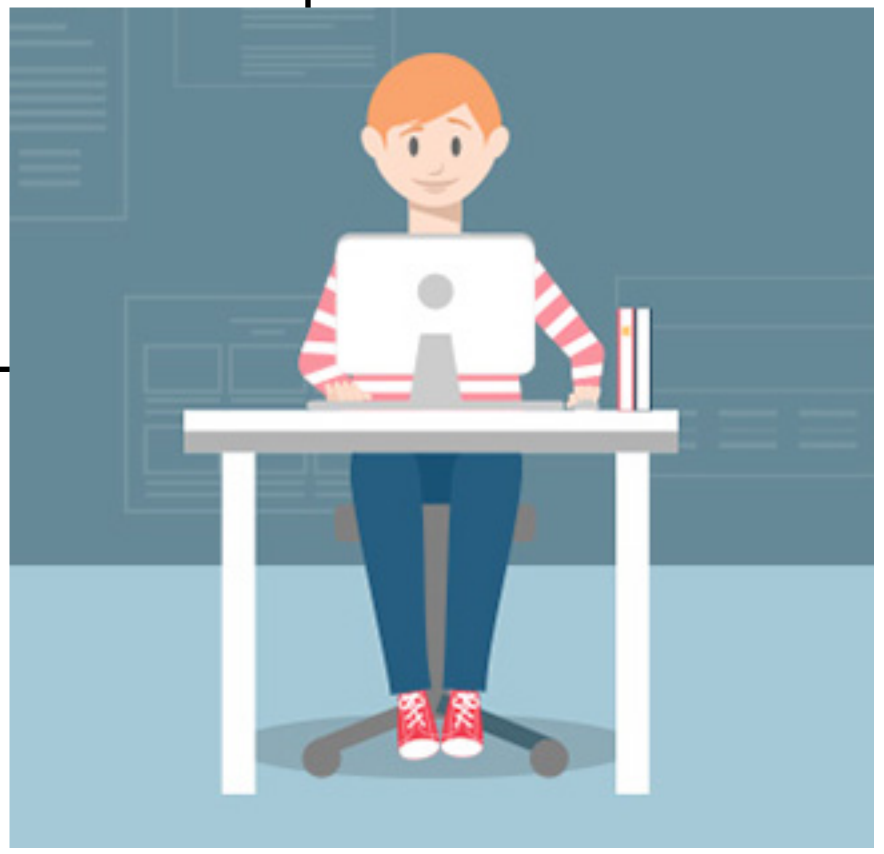
Program in C

```c
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```



compile

Program in x86 Assembly

```
792415C0    55       push ebp
792415C1    89E5     mov ebp, esp
792415C3    8B45 08  mov eax, [ebp+0x08]
792415C6    DB28     fld tword [eax]
792415C8    8B4D 0C  mov ecx, [ebp+0x0C]
792415CB    DB29     fld tword [ecx]
792415CD    DEC1     faddp
792415CF    8B55 10  mov edx, [ebp+0x10]
792415D2    DB3A     fstp tword [edx]
792415D4    DB68 0A  fld tword [eax+0x0A]
792415D7    DB69 0A  fld tword [ecx+0x0A]
792415DA    DEC1     faddp
792415DC    DB7A 0A  fstp tword [edx+0x0A]
                     pop ebp
                     ret 0x000C
```

Program P in C

```c
#include <stdio.h>

#define IN  1    /* inside a word */
#define OUT 0    /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

compile →

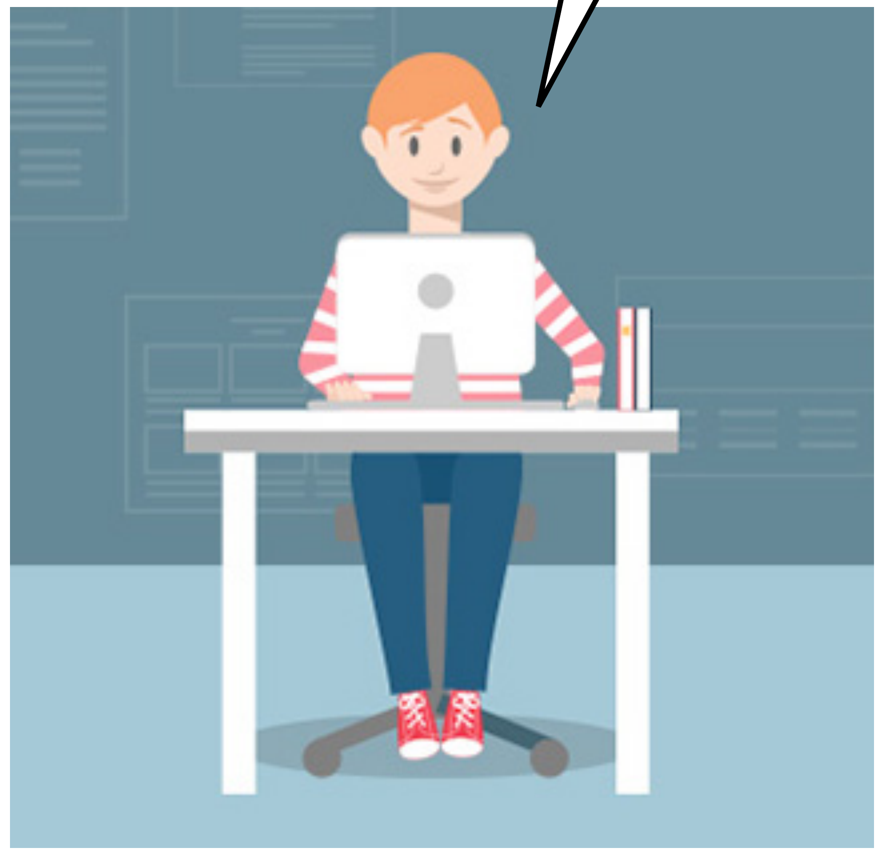
Program *compile*(P) in x86 Assembly

```
792415C0    55          push ebp
792415C1    89E5        mov ebp, esp
792415C3    8B45 08     mov eax, [ebp+0x08]
792415C6    DB28        fld tword [eax]
792415C8    8B4D 0C     mov ecx, [ebp+0x0C]
792415CB    DB29        fld tword [ecx]
792415CD    DEC1        faddp
792415CF    8B55 10     mov edx, [ebp+0x10]
792415D2    DB3A        fstp tword [edx]
792415D4    DB68 0A     fld tword [eax+0x0A]
792415D7    DB69 0A     fld tword [ecx+0x0A]
792415DA    DEC1        faddp
792415DC    DB7A 0A     fstp tword [edx+0x0A]
792415DF    5D          pop ebp
792415E0    C2 0C00     ret 0x000C
```

interpret-as-C ↓

interpret-as-x86 ↓

Result(P, input) = R_c **=** R_{x86} = Result(*compile*(P), input)



Compiler **Specification:**

For *any* program P, and *any* input,
the result of *interpreting* P with input in **C** is the same as
the result of *executing* *compilation* of P with input in **x86 Assembly**.

or, equivalently

Correctness Theorem:

$$\forall\ P, \text{input}, \textit{interpret}_{C}(P, \text{input}) = \textit{execute}_{x86}(\textit{compile}(P, \text{input}))$$

Correctness Theorem:

$$\forall\ P, input,\ \textit{interpret}_C(P, input) = \textit{execute}_{x86}(\textit{compile}(P, input))$$

Proof: ???

Assumptions:

- Meaningful definition of $\textit{interpret}_C$ is given and fixed
- Meaningful definition of $\textit{execute}_{x86}$ is given and fixed
- Specific implementation of *compile* is given and fixed
- Considered programs P is are valid and written in C

must be trusted
(*i.e.*, better be "sane")

Correctness Theorem:

$$\forall\ P, in,\ \textit{interpret}_C(P, in) = \textit{execute}_{x86}(\textit{compile}(P, in))$$

Proof: ???

once proven,
does not have
to be trusted

Formal Verification

Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

Formal Verification

Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

What is a Proof?

A proof is sufficient evidence
or an argument for the truth of a proposition.

Better Definition

A proof is a *sequence of logical statements*,

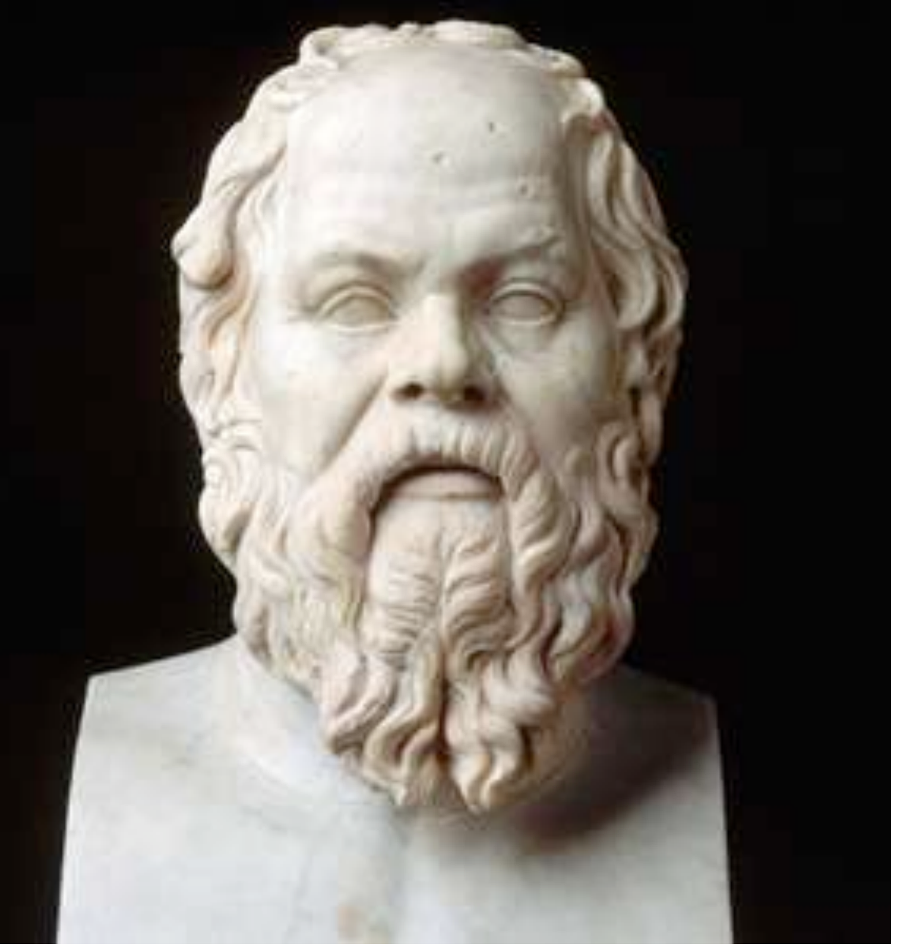
each of which is either *validly derived from those preceding* it
or is an *assumption*,

and the final member of which,
the conclusion, is the statement
of which the truth is thereby established.

Deriving Valid Proofs

The proposition A is true, and, moreover, A being true implies that B is true; then we can derive that B is true.

$$\frac{\vdash A \qquad \vdash A \Rightarrow B}{\vdash B}$$



$$\frac{\vdash A \quad \vdash A \Rightarrow B}{\vdash B}$$

reasonable assumptions

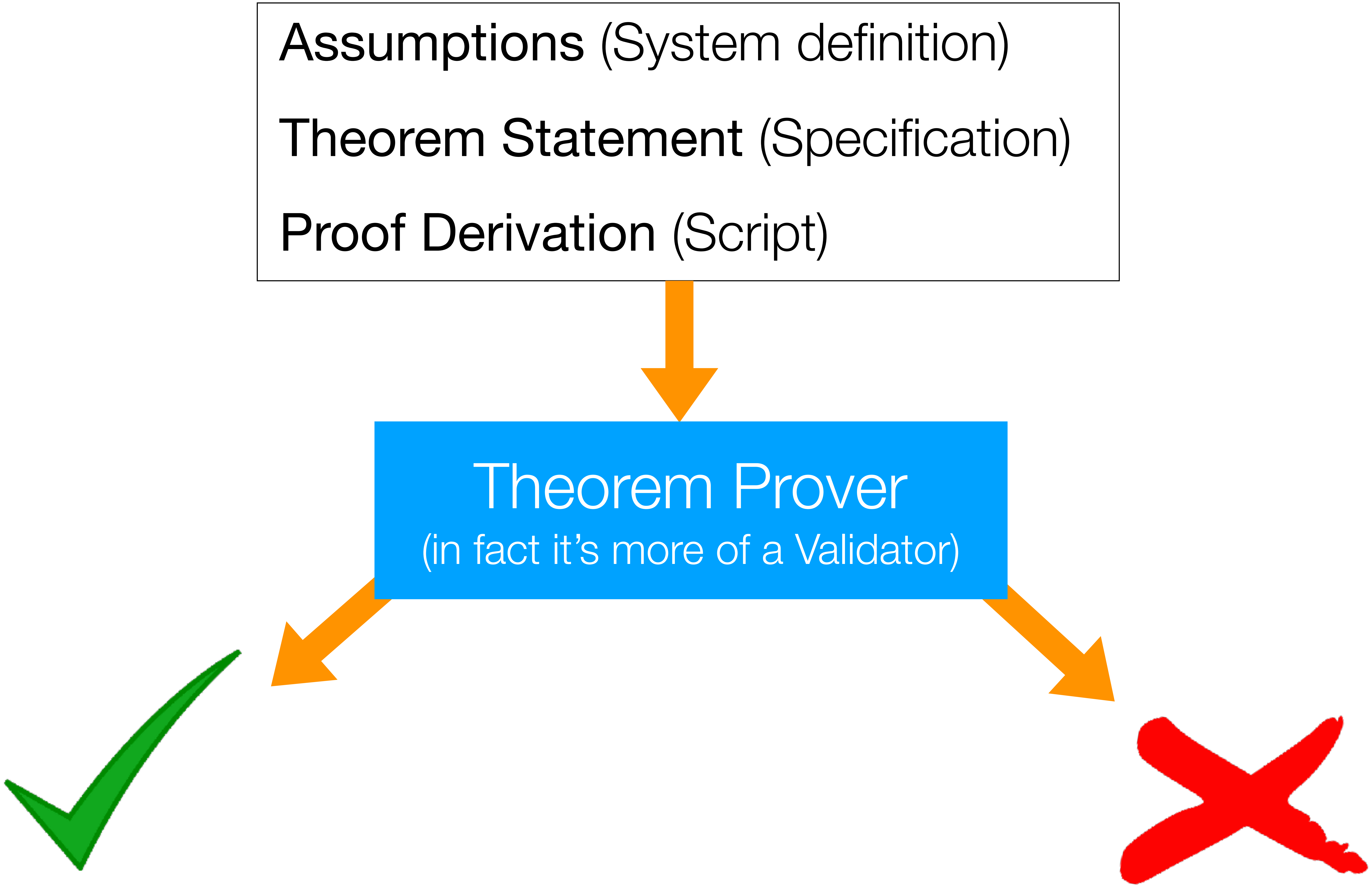
Socrates is a man

is a man ⇒ is mortal

Socrates is mortal

Overall, this is a valid proof, hence the conclusion it true

Proofs don’t have to be trusted!



Modern Theorem Provers are Awesome

Aquamacs

State Context Goal Retract Undo Next Use Goto Qed Home Find Info Command Prooftree Interrupt Restart Help

```
Ltac no_change can_bc can_bt can_n w F F' HExt c5 :=
  case=><- <- /=; exists can_bc, can_bt, can_n; rewrite (upd_nothing F); spl
it=>//;
    by move=>n st'; rewrite/localState; simplw w=>-> _ F';
       rewrite/blocksFor/inFlightMsgs; simplw w=>_ ->;
       rewrite -cat1s filter_cat /=; case: ifP; rewrite map_cat /=;
       do? rewrite -(btExtend_withDup_noEffect (find_some (c5 _ _ F')));
       move: (HExt _ _ F').

Lemma foldl_expand cbt bt bs :
  valid bt ->
  cbt = foldl btExtend bt bs -> exists q, cbt = bt \+ q.
Proof.
move=>V.
elim: bs cbt=>//=[|b bs Hi]cbt E; first by by exists Unit; rewrite unitR.
rewrite -foldl_btExtend_last//= -cats1 foldl_cat/= in E.
case: (Hi (foldl btExtend bt bs) (erefl _))=>q E'.
rewrite E' in E; subst cbt; rewrite /btExtend.
case:ifP=>X; first by exists q.
by exists (# b \\-> b \+ q); rewrite joinCA.
Qed.

(*******************************************************************)
(************* Invariant inductivity proof **************************)
(*******************************************************************)

Lemma clique_inv_step w w' q :
  clique_inv w -> system_step w w' q -> clique_inv w'.
Proof.
move=>Iw S; rewrite/clique_inv; split; first by apply (Coh_step S).
case: S; first by elim; move=>_ <-; apply Iw.
(* Deliver *)
move=> p st Cw. assert (Cw' := Cw). case Cw'=>[c1 c2 c3 c4 c5 c6] Al iF F.
case: Iw=>_ GSyncW.
case: GSyncW=>can_bc [can_bt] [can_n] []
           HHold HGt [C] [HBc] HGood HCliq HExt.
  move=>P; assert (P' := P).
```

U:**- InvCliqueTopology.v 30% (228,30) Git-master (Coq Script(1-) Holes company Spc Fill)

```
1 subgoal (ID 278)

  w, w' : World
  q : Qualifier
  ============================
  clique_inv w -> system_step w w' q -> clique_inv w'
```

U:%%- *goals* All (6,0) (Coq Goals company Spc Fill)

U:%%- *response* All (1,0) (Coq Response company Trunc Spc Fill)

Zoom: 120%

Programming and proving are the same things!

Formal Verification

Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

Mechanised Formal Verification

Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning,

whose validity is machine-checked.

(assuming that you trust the checker)

Checkpoint

- For a fully specified system, correctness is a *mathematical theorem*
- It can be proven using rules of *mathematical logic*
- Typically, the proofs rest on some unprovable assumptions, which must be *trusted*
- *Mechanised proof checking* ensures validity of the proof, but requires to *trust the checker implementation.*

State of the Art
in Formally Verified Systems

CompCert (2006-now)

a mechanically verified C compiler

Formal Certification of a Compiler Back-end

***or*: Programming a Compiler with a Proof Assistant**

Xavier Leroy
INRIA Rocquencourt
Xavier.Leroy@inria.fr

- Specification: source and target programs are equivalent
- Assumptions: underlying hardware semantics, unverified parser
- Proof effort: 146 kLOC of specifications and proofs

Verdi (2015)

a formally verified Raft consensus implementation

Verdi: A Framework for Implementing and Formally Verifying Distributed Systems

James R. Wilcox Doug Woos Pavel Panchekha
Zachary Tatlock Xi Wang Michael D. Ernst Thomas Anderson
University of Washington, USA
{jrw12, dwoos, pavpan, ztatlock, xi, mernst, tom}@cs.washington.edu

- **Specification:** Raft provides *transparent replication*
- **Assumptions:** unlimited memory, TCP works atomically, …
- **Proof effort:** 50 kLOC of specifications and proofs

FSCQ (2015)

a crash-tolerant file system

Using Crash Hoare Logic for Certifying the FSCQ File System

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

- Specification: asynchronous disk writes are not affected by crashes
- Assumptions about semantics of extraction and linking with other drivers
- Proof effort: 81 kLOC of specifications and proofs

Does it really work?

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{ jxyang, chenyang, eeide, regehr }@cs.utah.edu

(in PLDI 2011)

Compilers should be correct.

To improve the quality of C compilers, we created Csmith, a **randomized test-case generation tool**, and spent **three years** using it to find compiler bugs.

During this period we reported **more than 325 previously unknown bugs** to compiler developers.

The striking thing about our **CompCert** results is that the middle-end bugs we found in all other compilers are **absent**.

As of early 2011, the under-development version of **CompCert** is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.

The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

So, bye-bye testing?

Formal Verification is Expensive

- CompCert
  146 kLOC
- Verdi
  50 kLOC
- FSCQ
  81 kLOC

Formal Verification is Expensive

- CompCert
  146 kLOC, 10+ person-years
- Verdi
  50 kLOC, 3+ person-years
- FSCQ
  81 kLOC, 5+ person-years

Formal Verification is Expensive

- CompCert
  146 kLOC, 10+ person-**years**
- Verdi
  50 kLOC, 3+ person-**years**
- FSCQ
  81 kLOC, 5+ person-**years**

Assumptions Matter

Story 1: CompCert

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{ jxyang, chenyang, eeide, regehr }@cs.utah.edu

The second CompCert problem we found was illustrated by two bugs that resulted in generation of code like this:

```
stwu r1, -44432(r1)
```

Here, a large PowerPC stack frame is being allocated. The problem is that the 16-bit displacement field is overflowed. CompCert's PPC semantics failed to specify a constraint on the width of this immediate value, on the assumption that the assembler would catch out-of-range values. In fact, this is what happened. We also found a

Wrong assumption about compiled assembly execution!

Story 2: Verdi

An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca Kaiyuan Zhang Xi Wang Arvind Krishnamurthy
University of Washington

Overall, 7 bugs are found

4.3 Resource Limits

This section describes three bugs that involve exceeding resource limits.

Bug V6: *Large packets cause server crashes.*

The server code that handled incoming packets had a bug that could cause the server to crash under certain conditions. The bug, due to an insufficiently small buffer in the OCaml code, caused incoming packets to truncate large packets and subsequently prevented the server from correctly unmarshaling the message.

Wrong assumption about the crash model!

Story 3: FSCQ

We found a bug in a verified file system! We ran Crashmonkey's suite of tests on MIT's FSCQ and found that it does not persist data on fdatasync properly. We emailed the authors, they have acked and fixed the bug.

Come see our paper at #osdi18!

Details: github.com/utsaslab/crash…

> **Vijay Chidambaram** @vj_chidambaram
> Excited to share our #osdi18 paper on finding crash-consistency bugs in Linux file systems! I will explain the intuition behind our system in this thread....
> Show this thread

Story 3: FSCQ

We found a bug in a verified file system! We ran Crashmonkey's suite of tests on MIT's FSCQ and found that it does not persist data on fdatasync properly. We emailed the authors, they have acked and fixed the bug.

Come see our paper at #osdi18!

Details: github.com/utsaslab/crash

Vijay Chidambaram @vj_chidambaram
Excited to share our #osdi18 paper on finding crash-consistency b
Linux file systems! I will explain the intuition behind our system in t
thread....
Show this thread

John Regehr @johnregehr · Oct 3
Replying to @vj_chidambaram
what was the root cause of their failure to find this bug during verification?

1 3

Vijay Chidambaram @vj_chidambaram · Oct 3
Even verified file systems have unverified parts :) it was due to a buggy optimization in the Haskell-c bindings.

1 5 4

Checkpoint

- *Costs* of formal verification *are high*,
  but so are the provided *correctness guarantees*
- *Realistic systems* are always verified in the presence of *non-trivial assumptions* about their usage
- These assumptions *might be broken* in the real world, thus invalidating the claims of theorems
- *Testing* helps to validate the assumptions.

What about Blockchains and their Applications?

What about Blockchains and their Applications?

(application layer)

- We're at the stage of proving specifications of *smart contracts*

- We can also verify properties of *executable protocols*

(system layer)

Verifying Protocol Implementations

Mechanising Blockchain Consensus

George Pîrlea
University College London, UK
george.pirlea.15@ucl.ac.uk

Ilya Sergey
University College London, UK
i.sergey@ucl.ac.uk

Abstract

We present the first formalisation of a blockchain-based distributed consensus protocol with a proof of its consistency mechanised in an interactive proof assistant.

Our development includes a reference mechanisation of the *block forest* data structure, necessary for implementing provably correct per-node protocol logic. We also define a

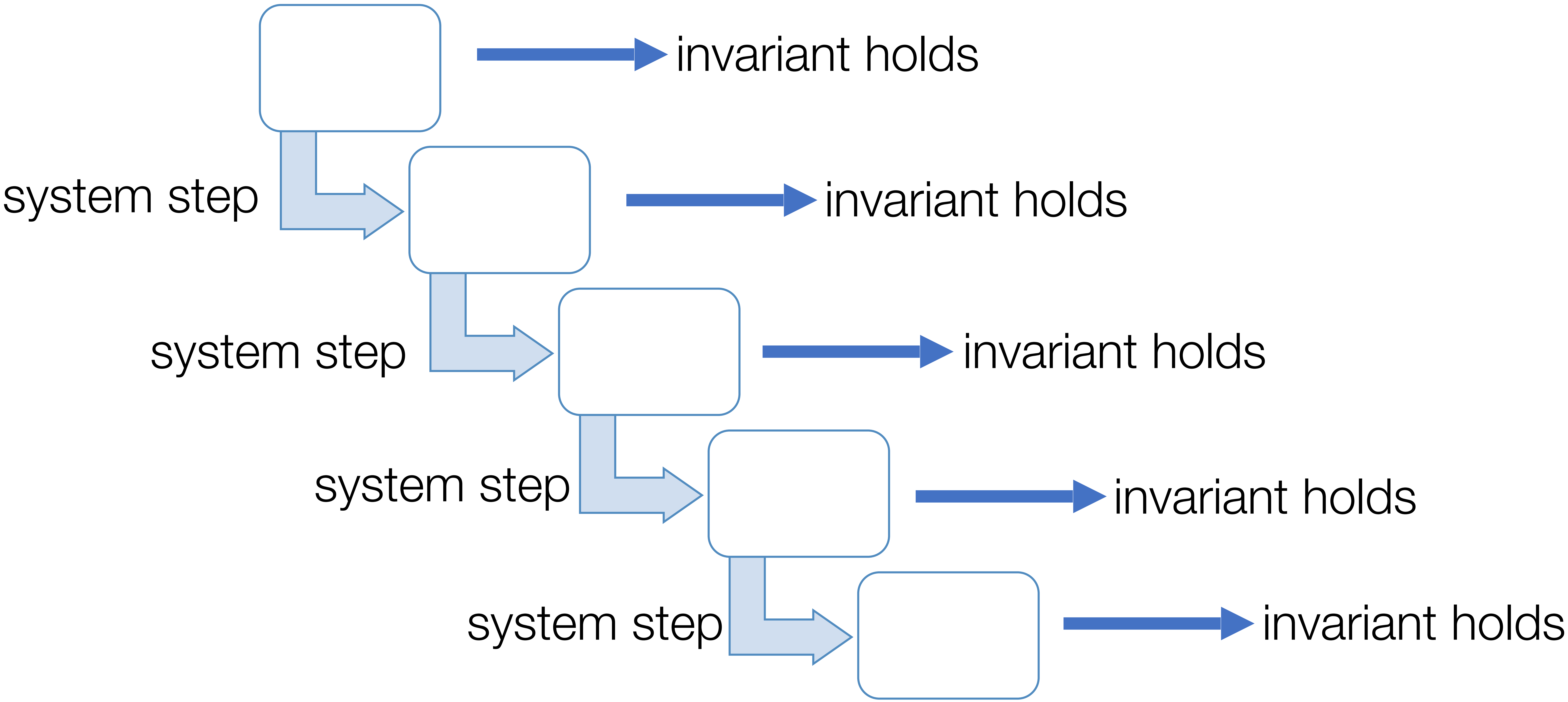
1 Introduction

The notion of decentralised blockchain-based consensus is a tremendous success of the modern science of distributed computing, made possible by the use of basic cryptography, and enabling many applications, including but not limited to cryptocurrencies, smart contracts, application-specific arbitration, voting, *etc.*

- **Specification:** nodes, asynchronously exchanging blocks, reach *agreement*
- **Assumptions** *clique* topology, *fork-chain rule* properties,
  no restrictions wrt. PoW hardness of minting a block.
- **Proof effort:** 3 kLOC of specifications and proofs

Definitions	• blocks, ledgers, block forests
Assumptions	• *hashes* are collision-free • *FCR* imposes strict total order
Theorem	• when all block messages are delivered, everyone agrees
Invariant	• local state + messages “in flight” global

Invariant is inductive



Invariant implies *Quiescent Consistency* (QC)

- QC: when all blocks *delivered*, everyone *agrees*

How:

- local state + ~~“in-flight”~~ = global
- use FCR to extract “heaviest” chain out of local state

- since everyone has *same state & same FCR*
  - ➢**consensus**

(more interesting properties are yet to be proven…)

Verifying Smart Contract Properties

SCILLA: a Smart Contract Intermediate-Level LAnguage

Automata for Smart Contract Implementation and Verification

Ilya Sergey
University College London
i.sergey@ucl.ac.uk

Amrit Kumar
National University of Singapore
amrit@comp.nus.edu.sg

Aquinas Hobor
Yale-NUS College
National University of Singapore
hobor@comp.nus.edu.sg

Principled model for computations — **System F** with small extensions

Not Turing-complete — Only *primitive recursion*/iteration

Explicit Effects — *State-transformer* semantics

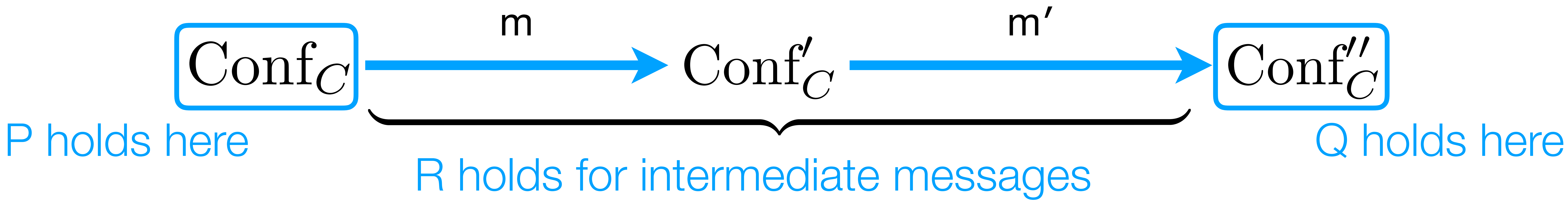
Communication — Contracts are *communicating automata*

Reasoning about Scilla Contracts

- What can be specified and proven
  - Local properties (e.g., *"transition does not throw an exception"*)
  - Invariants (e.g., *"balance is always strictly positive"*)
  - Temporal Properties (*something good eventually happens*)

Temporal Properties

$$Q \textit{ since } P \textit{ as long } R \stackrel{\text{def}}{=} \forall\, conf\ conf',\ \ conf \rightarrow_R^{*} conf',\ P(conf) \Rightarrow Q(conf, conf')$$



- "Token price only goes up"
- "No payments accepted after the quorum is reached"
- "No changes can be made after locking"
- "Consensus results are irrevocable"

Assumptions for Scilla-enabled Formal Verification

- *Translation* from Scilla to Coq correct (in the compiler sense)
  - **future work**: verified Scilla interpreter *implemented* in Coq

- Formalised in Coq *model of message-passing* corresponds precisely to the *blockchain back-end*.

Looking Ahead

- What are the right properties of Blockchain systems to prove?
  - Most of the interesting properties require *probabilistic reasoning*
  - *Chain-growth*, *common-prefix*, etc. — ***none*** are proven for *real code*!
- What are the right specifications for smart contracts?
  - Can we reason about *incentives for interaction* with smart contracts?
  - Can we *teach non-experts* in FM to state them?
- What should be the *reusable libraries* to make mechanised formal reasoning about blockchains *tractable* and *scalable*?

To Take Away

What We Talk about When We Talk about Formally Verified Systems

- *Formal verification* requires *precise specification* and cannot be conducted without *reasonable assumptions*

- *Mechanically-checked proofs* provide the best correctness guarantees

- Yet, *testing* shouldn't be dismissed: it helps *check the assumptions*

- Mechanised formal reasoning is *expensive* but might well worth it for *correctness-critical* systems—especially blockchains and smart contracts

Thanks!