AI-Assisted Coding

Correct By Construction, Not By Generation

Chapman-Topos Workshop 20240508
Shaowei Lin, Director of Research, Topos Institute
Joint work with Atlas Computing

Challenge

Safety and Security of Sociotechnical Systems

Intentional attacks/
accidental mistakes

Human-led/
AI-enabled

MICROSOFT / TECH / OPENAI

Microsoft and OpenAI say hackers are using ChatGPT to improve cyberattacks

Photo by Amelia Holowaty Krales / The Verge

/ A number of nation-backed groups are starting to use large language models to help with research, scripting, and phishing emails.

By Tom Warren, a senior editor covering Microsoft, PC gaming, console, and tech. He founded WinRumors, a site dedicated to Microsoft news, before joining The Verge in 2012.

Feb 14, 2024, 4:00 AM PST

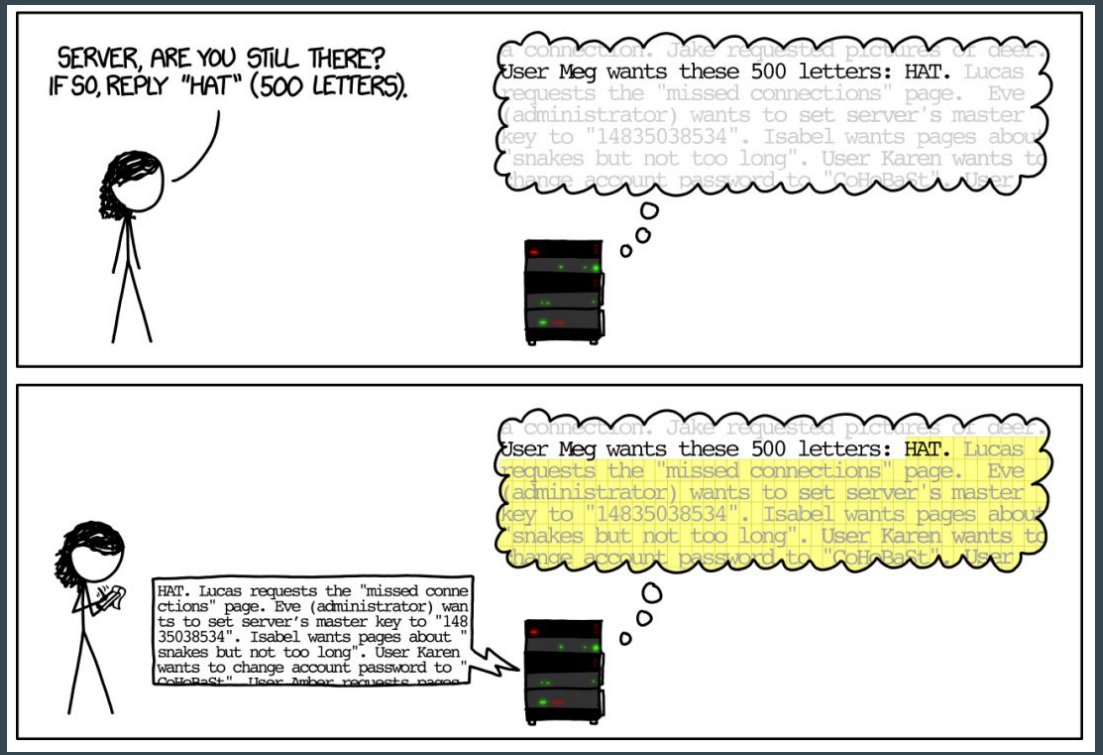
6 Comments (6 New)

If you buy something from a Verge link, Vox Media may earn a commission. ***See our ethics statement.***

https://www.theverge.com/2024/2/14/24072706/microsoft-openai-cyberattack-tools-ai-chatgpt

Hardening Systems with Formal Verification

Do we want to fix our sociotechnical systems one bug at a time?

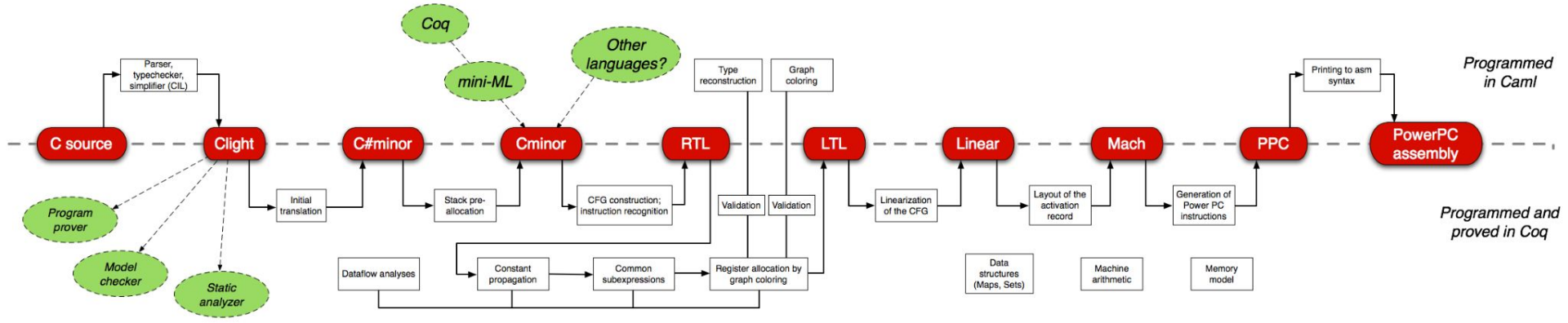


2014 OpenSSL HeartBleed Bug https://xkcd.com/1354/

Hardening Systems with Formal Verification

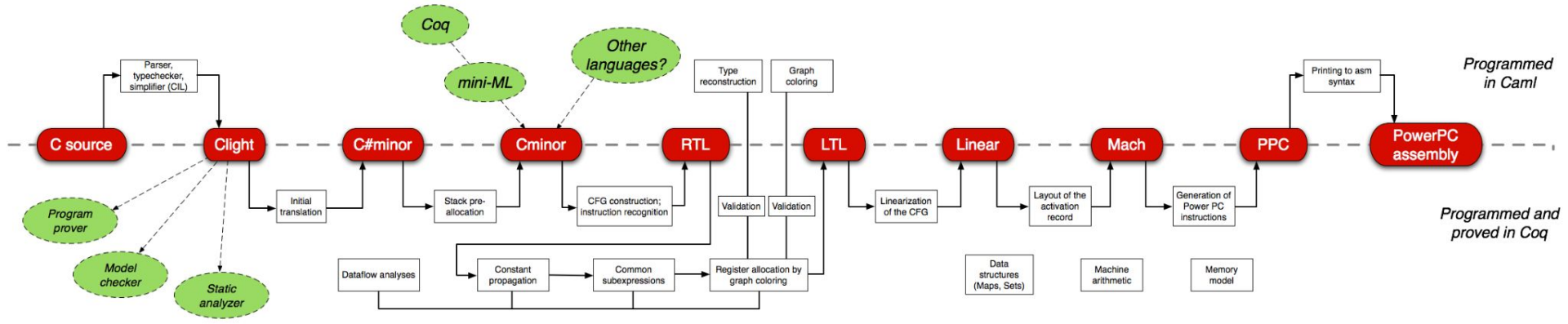
"As of early 2011, the under-development version of **CompCert** is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task."

Yang, X., Chen, Y., Eide, E. and Regehr, J., 2011, June. Finding and understanding bugs in C compilers. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (pp. 283-294).



Challenge of Formal Verification

Writing specifications, implementations and proofs is difficult!



Scaling FV with AI Assistance

Large language models (LLMs) are becoming good at generating code.

How do we ensure they produce code that is correct?

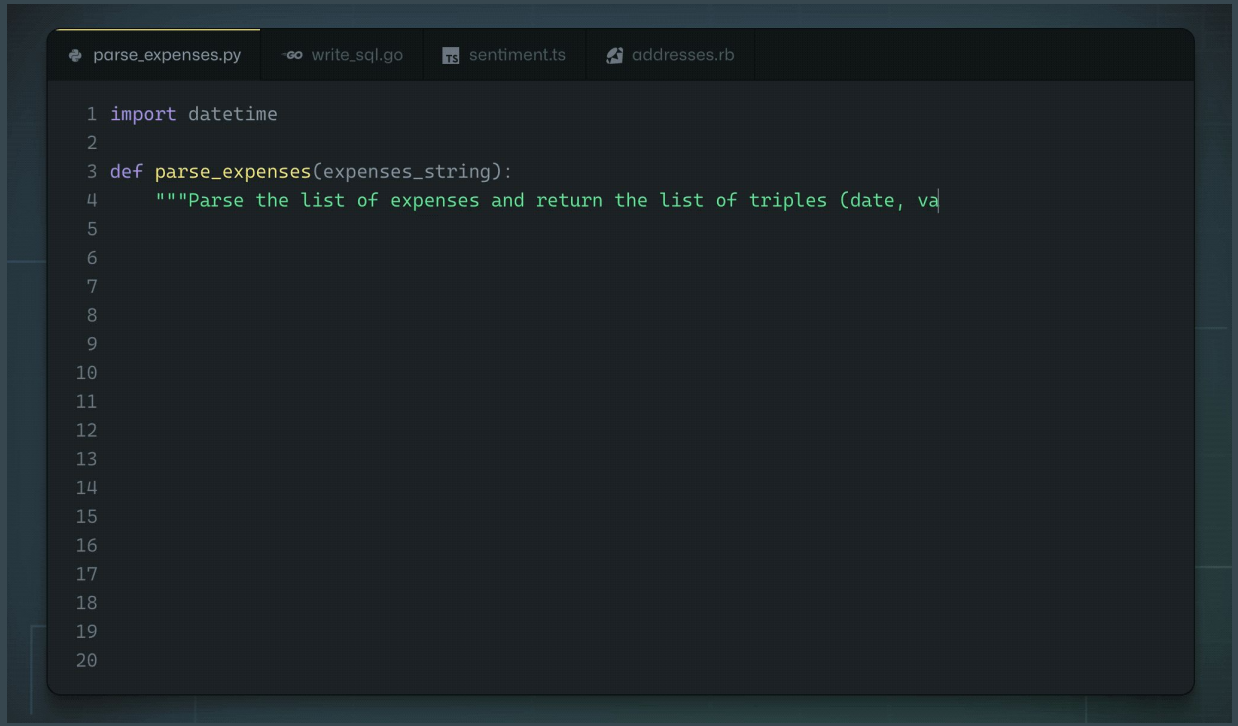
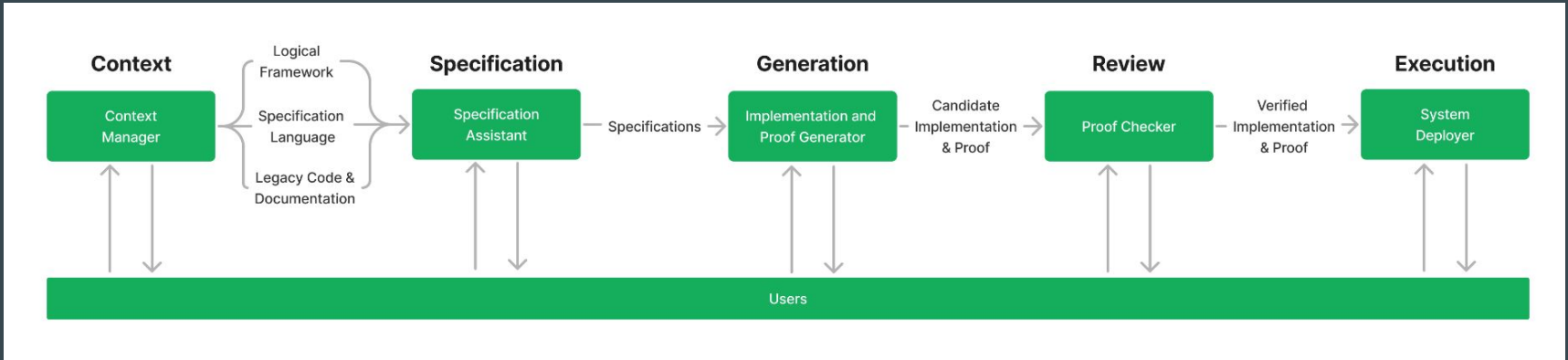


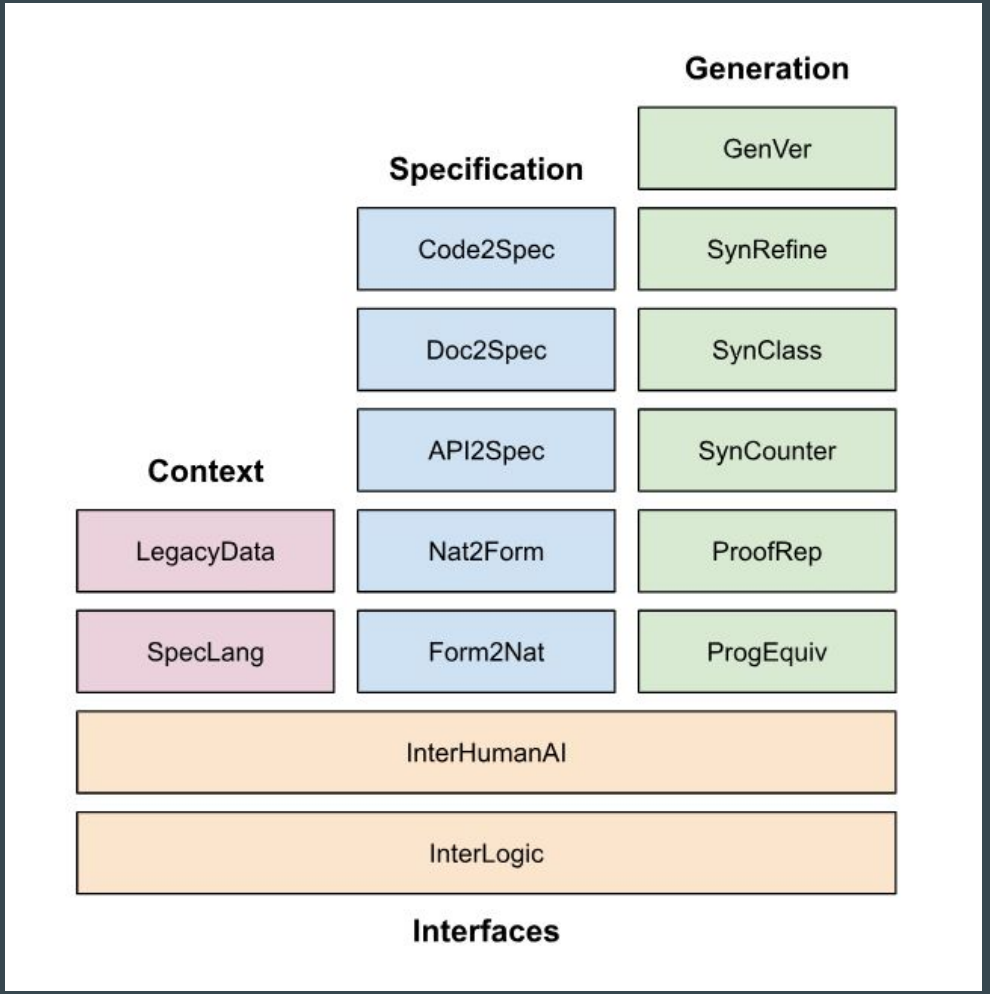
Image Credits: GitHub Copilot

Roadmap

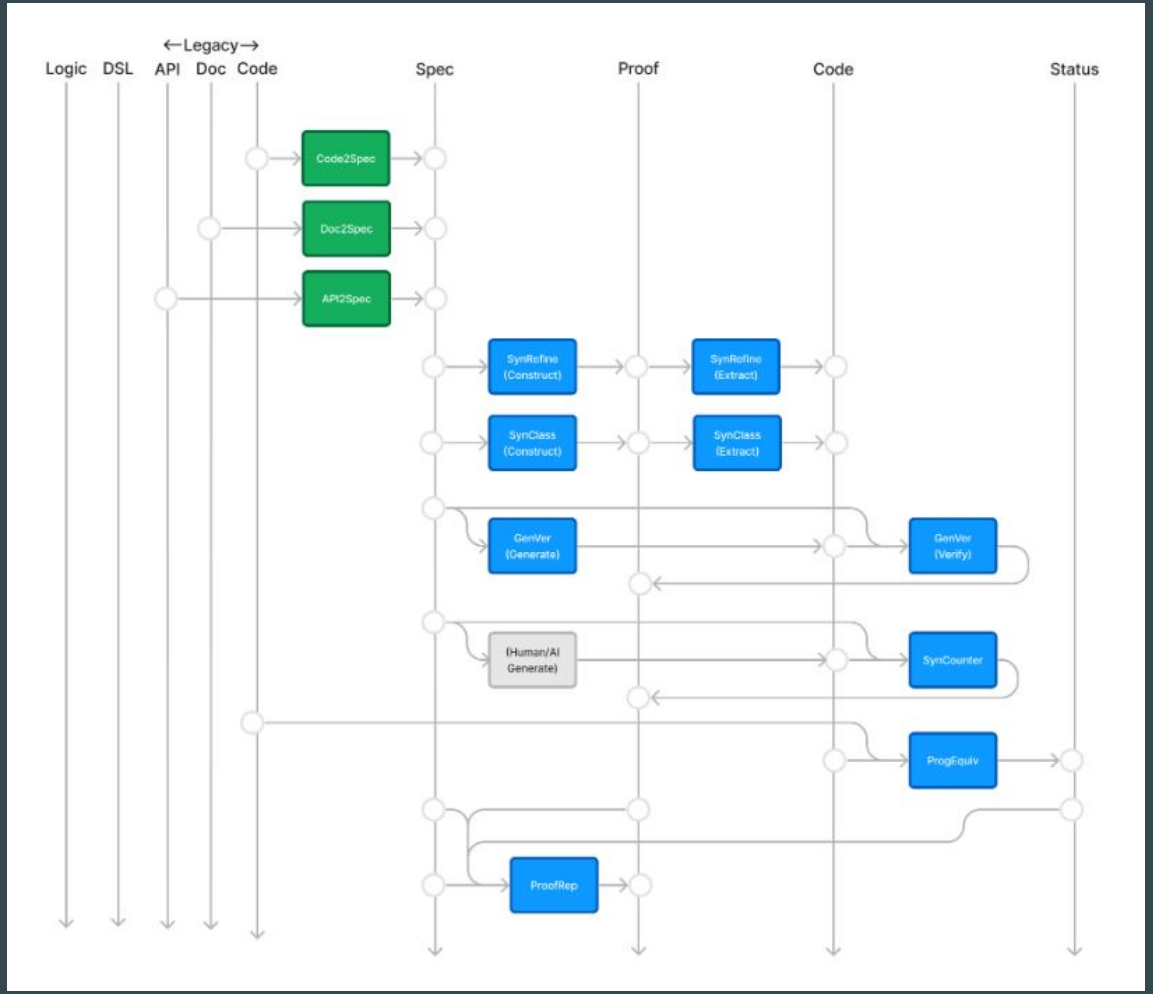
Workflow


Context
Logical Framework
Specification Language
Legacy Code & Documentation
Specification
Generation
Review
Execution
Context Manager
Specification Assistant
Specifications
Implementation and Proof Generator
Candidate Implementation & Proof
Proof Checker
Verified Implementation & Proof
System Deployer
Users

Projects


Generation
GenVer
Specification
Code2Spec
SynRefine
Doc2Spec
SynClass
API2Spec
SynCounter
Context
LegacyData
Nat2Form
ProofRep
SpecLang
Form2Nat
ProgEquiv
InterHumanAI
InterLogic
Interfaces

Dependencies



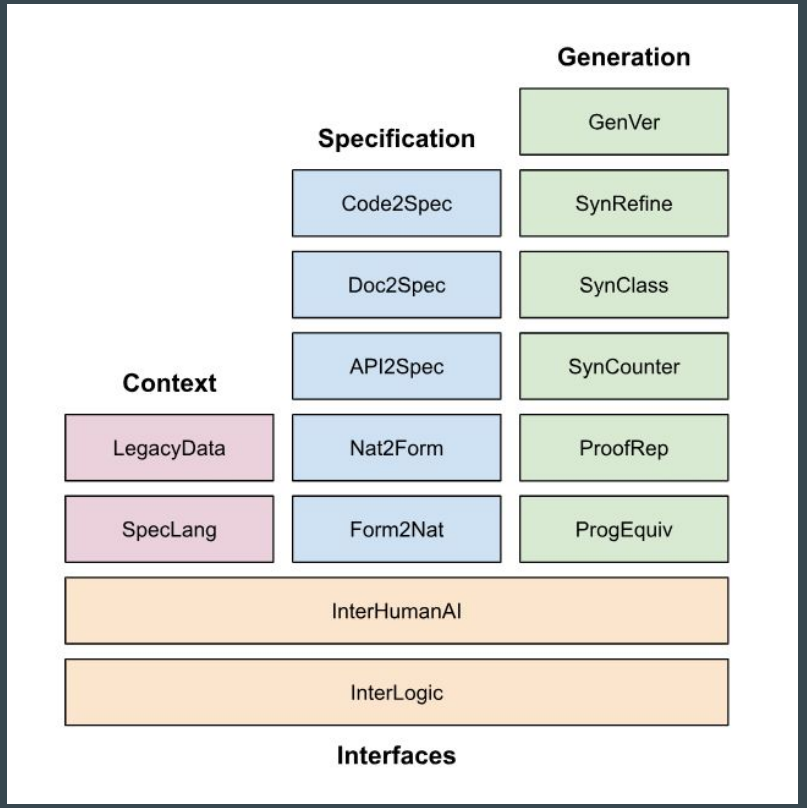
Contexts and Interfaces

Legacy Code and Docs (LegacyData)

- Verify legacy code, fix its bugs?
- Provide data/prompt to AI models for code specification and synthesis!

Specification Languages (SpecLang)

- Domain specific languages



Logic Translations
(InterLogic)

You

Translate the following code from the Coq programming language into the Lean programming language.

```
Definition instrDenote (i : instr) (s : stack) : option stack :=
 match i with
  | iConst n => Some (n :: s)
  | iBinop b =>
   match s with
    | arg1 :: arg2 :: s' => Some ((binopDenote b) arg1 arg2 :: s')
    | _ => None
   end
 end.
```

ChatGPT

To translate the provided Coq code into the Lean programming language, we need to understand the structure of the Coq code and then replicate that logic in Lean. The Coq code defines a function `instrDenote` which takes an instruction `i` and a stack `s`, and then processes the instruction to potentially modify the stack.

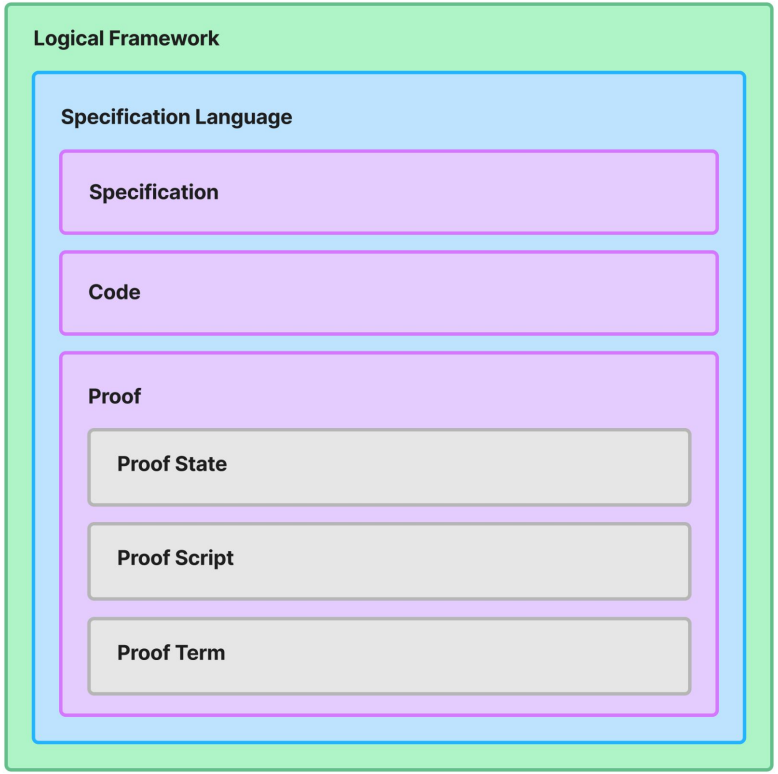
Here's how the same functionality can be implemented in Lean using a similar pattern:

lean — Copy code

```
inductive instr
| iConst : ℕ → instr
| iBinop : (ℕ → ℕ → ℕ) → instr

open instr

def binopDenote (b : ℕ → ℕ → ℕ) (arg1 arg2 : ℕ) : ℕ :=
  b arg1 arg2

def instrDenote : instr → list ℕ → option (list ℕ)
| (iConst n) s := some (n :: s)
| (iBinop b) (arg1 :: arg2 :: s) := some (binopDenote b arg1 arg2 :: s)
| (iBinop _) _ := none
```

Human-AI Interactions
(HumanAI Interface)

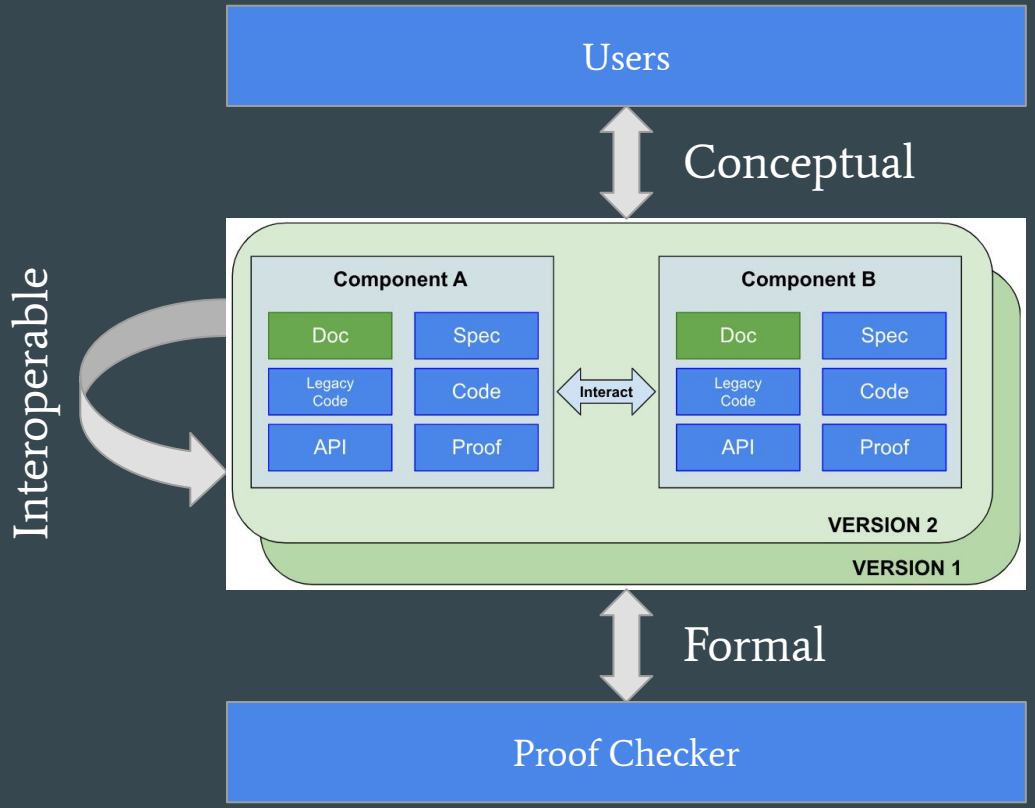
Proof Assistant
+
AI Assistant
+
Human Users



Human-AI Interactions
(InterHumanAI)

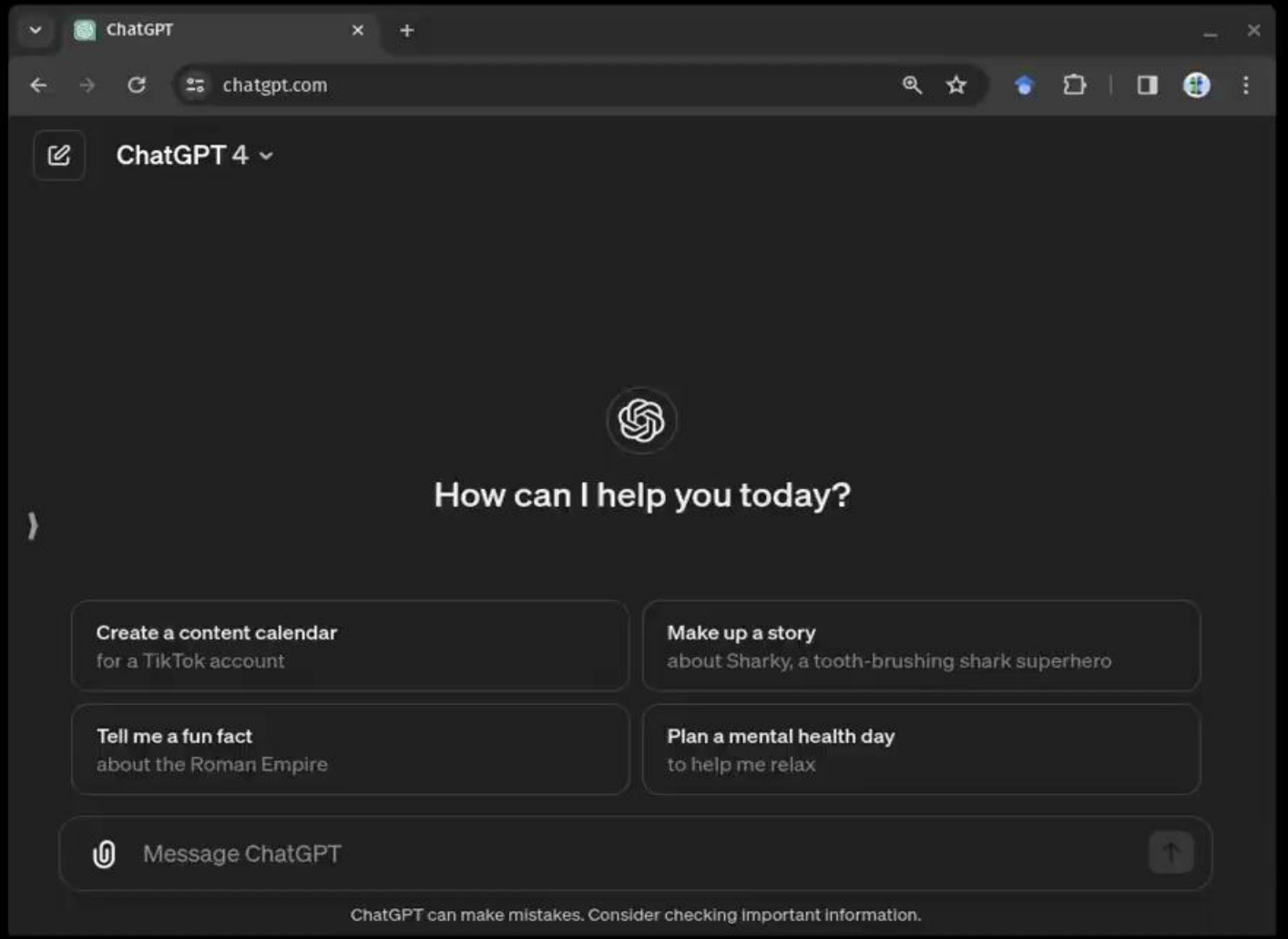
Modelling should be

- Conceptual
- Interoperable
  - Between models
  - Between versions
- Formal



Specifications

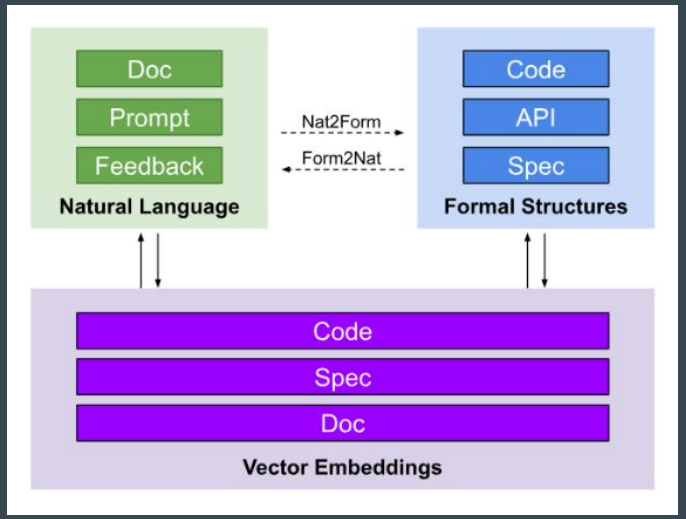
From Natural Language To Specs



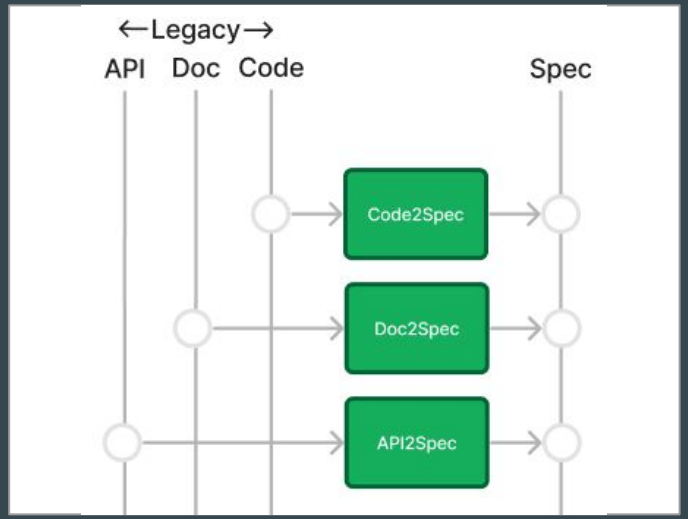
Brandfonbrener, D., Raja, S., Prasad, T., Loughridge, C., Yang, J., Henniger, S., Byrd, W.E., Zinkov, R. and Amin, N., 2024. Verified Multi-Step Synthesis using Large Language Models and Monte Carlo Tree Search. arXiv preprint arXiv:2402.08147

Generating Formal Specifications

How do we ensure that the LLM output is well-formed?



How do we use legacy data to generate accurate specs?



Synthesis

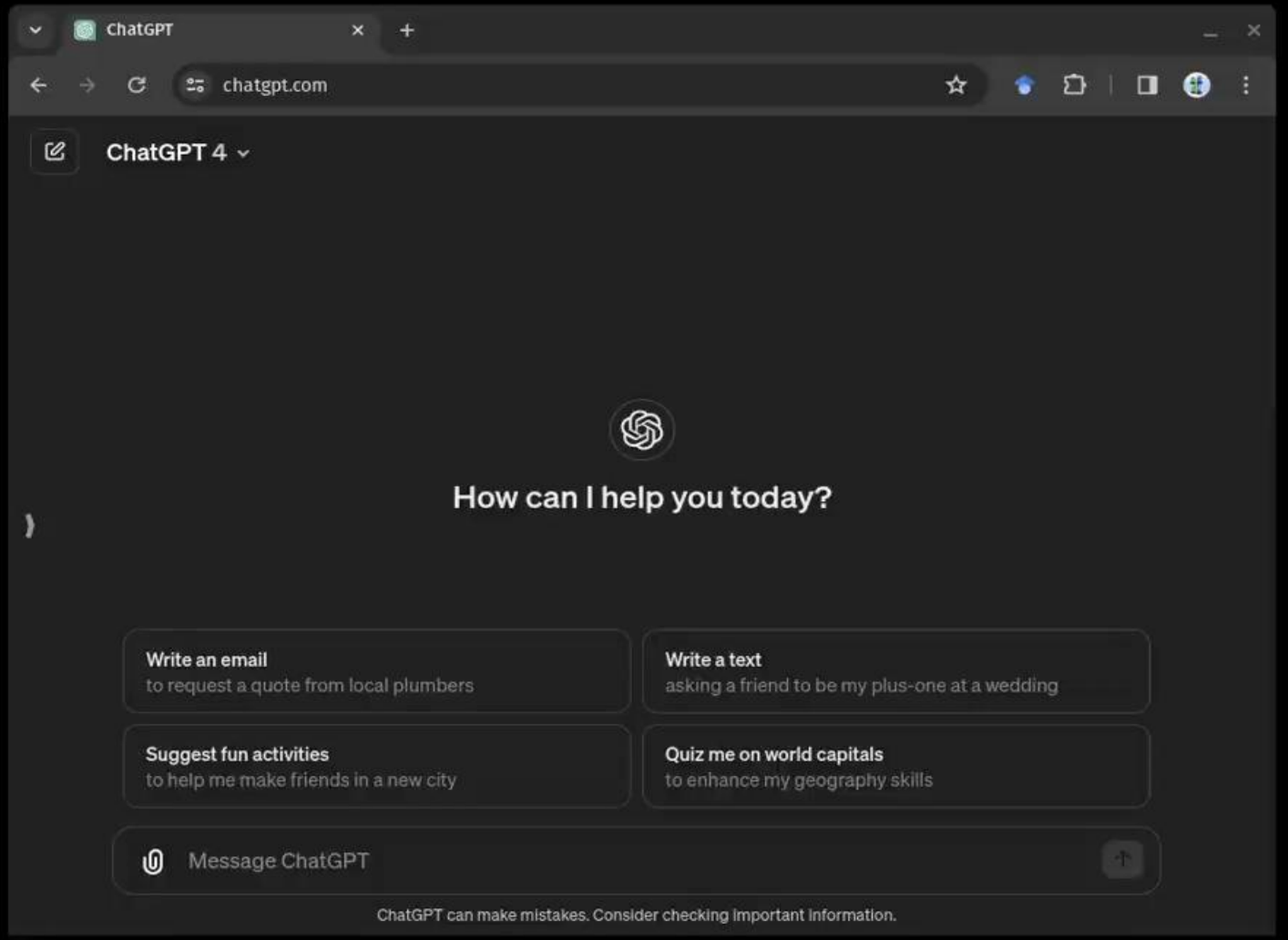
Decidable First-order Theories

Hoare logic

Separation logic

Verification-aware Languages, e.g. Dafny, Frama-C

Find proofs, counterexamples



Sun, C., Sheng, Y., Padon, O. and Barrett, C., 2023. Clover: Closed-Loop Verifiable Code Generation. arXiv preprint arXiv:2310.17807.

Expressive Higher-Order Theories

Dependent type theory

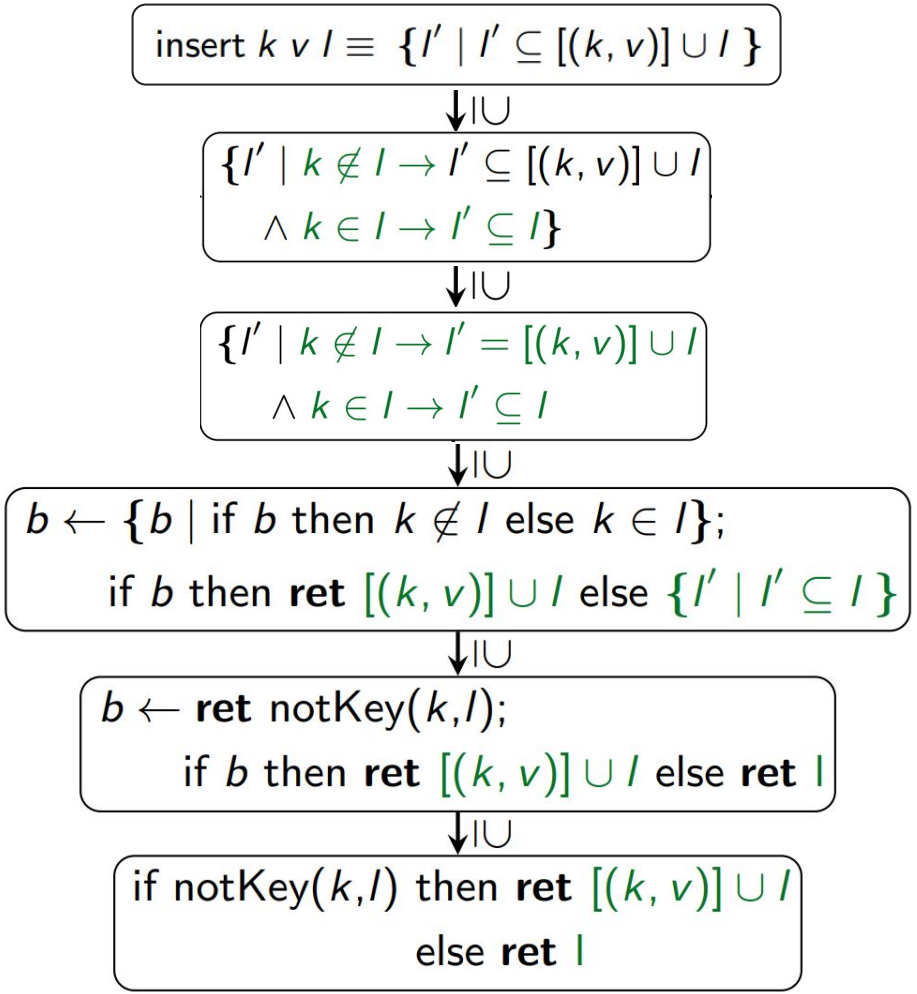
Proof Assistants, e.g. Coq, Lean

```
-- Import necessary libraries for proving
import data.nat.basic
import tactic

-- Using the previously defined factorial function
def factorial : ℕ → ℕ
| 0     := 1
| (n+1) := (n + 1) * factorial n

-- Proof that factorial is always positive
theorem factorial_pos : ∀ n : ℕ, factorial n > 0 :=
begin
  -- Apply mathematical induction on n
  apply nat.rec,
  -- Base case: show that factorial 0 is 1, which is positive
  { show factorial 0 > 0,
    rw factorial,  -- factorial 0 = 1 by definition
    exact nat.zero_lt_one, },
  -- Inductive step: assume factorial n is positive,
  -- prove factorial (n+1) is positive
  { intros n h,
    show factorial (n + 1) > 0,
    rw factorial,  -- expand factorial (n + 1)
    apply nat.mul_pos,  -- product of two positive numbers is positive
    show n + 1 > 0,
    exact nat.succ_pos n,
    exact h, }
end
```

Refinement-based Synthesis

Functionality without sacrificing *performance*

e.g. fiat-crypto library: Estimated that over 95% of HTTPS connections by browsers run the generated algorithm.



Delaware, B., Pit-Claudel, C., Gross, J. and Chlipala, A., 2015. Fiat: Deductive synthesis of abstract data types in a proof assistant. Acm Sigplan Notices, 50(1), pp.689-700.

Correct By Construction

Tactic-driven

Code-carrying proof

Extracting implementations

```
hone method insert. {
      StartMethod.
      setoid_rewrite refine_ReplaceUsedKeyAdd.
      setoid_rewrite refine_SubEnsembleInsert.
      autorewrite with monad laws.
      setoid_rewrite refine_pick_KeyToBeReplaced min.
      setoid_rewrite refine_If_Then_Else_Bind.
      autorewrite with monad laws.
      setoid_rewrite refine_If_Opt_Then_Else_Bind.
      autorewrite with monad laws.
      setoid_rewrite refine_pick_CacheADTwLogIndex AbsR.
      setoid_rewrite refine_pick_KVEnsembleInsertRemove
                           with (1 := EquivKeys H).
      setoid_rewrite refine_pick_KVEnsembleInsert
                           with (1 := EquivKeys H).
      autorewrite with monad laws; simpl.
      finish honing. }
```

Delaware, B., Pit-Claudel, C., Gross, J. and Chlipala, A., 2015. Fiat: Deductive synthesis of abstract data types in a proof assistant. Acm Sigplan Notices, 50(1), pp.689-700.

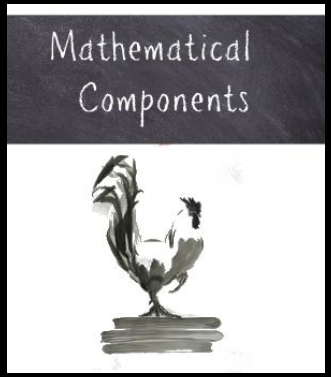
Construction, Not Generation

- Humans do not generate complex programs token by token
- Instead, we construct them – decomposing goals, composing solutions, modifying/refining program states

Construction, Not Generation

To train AI models that construct, we need **generic tactics**

Develop tactics for categories – more precisely, algebraic theories in the form of **typeclasses**




```
Definition eq_axiom T (e : rel T) := forall x y, reflect (x = y) (e x y).

HB.mixin Record hasDecEq T := { eq_op : rel T; eqP : eq_axiom eq_op }.

#[mathcomp(axiom="eq_axiom"), short(type="eqType")]
HB.structure Definition Equality := { T of hasDecEq T }.
```

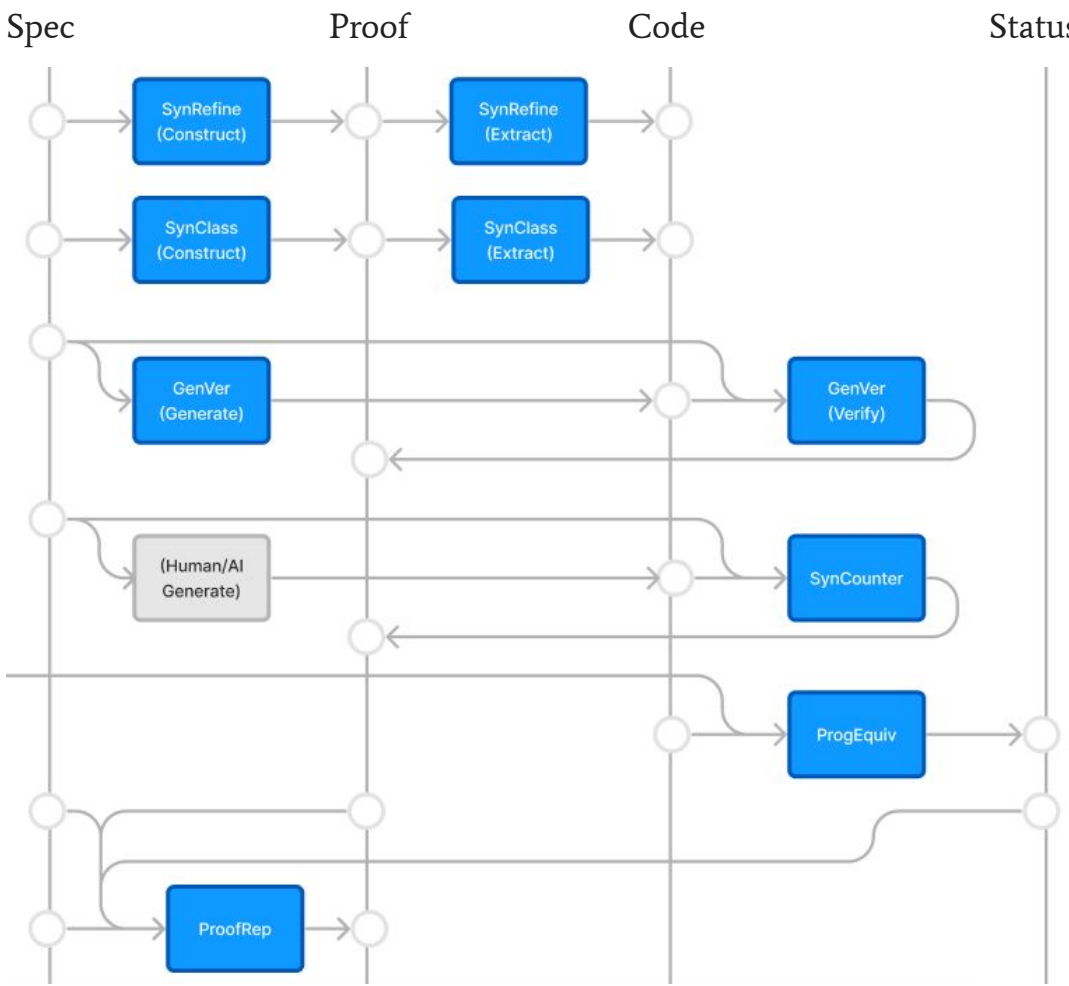
```
Lemma eq_refl (T : eqType) (x : T) : x == x. Proof. exact/eqP. Qed.
Notation eqxx := eq_refl.

Lemma eq_sym (T : eqType) (x y : T) : (x == y) = (y == x).
Proof. exact/eqP/eqP. Qed.

#[global] Hint Resolve eq_refl eq_sym : core.
```

Program Equivalence Checking (ProgEquiv)

- Checking if two implementations have the same behavior

Proof Repair (ProofRep)

- Fixing proofs when libraries, specs, implementations change



Summary

- **Challenge:** Scaling formal verification with AI assistance
- **Context:** Feeding AI models with legacy code and doc
- **Interface:** Formal interoperable conceptual modeling
- **Specification:** From natural language to formal structures
- **Synthesis:** Correct by construction, not by generation

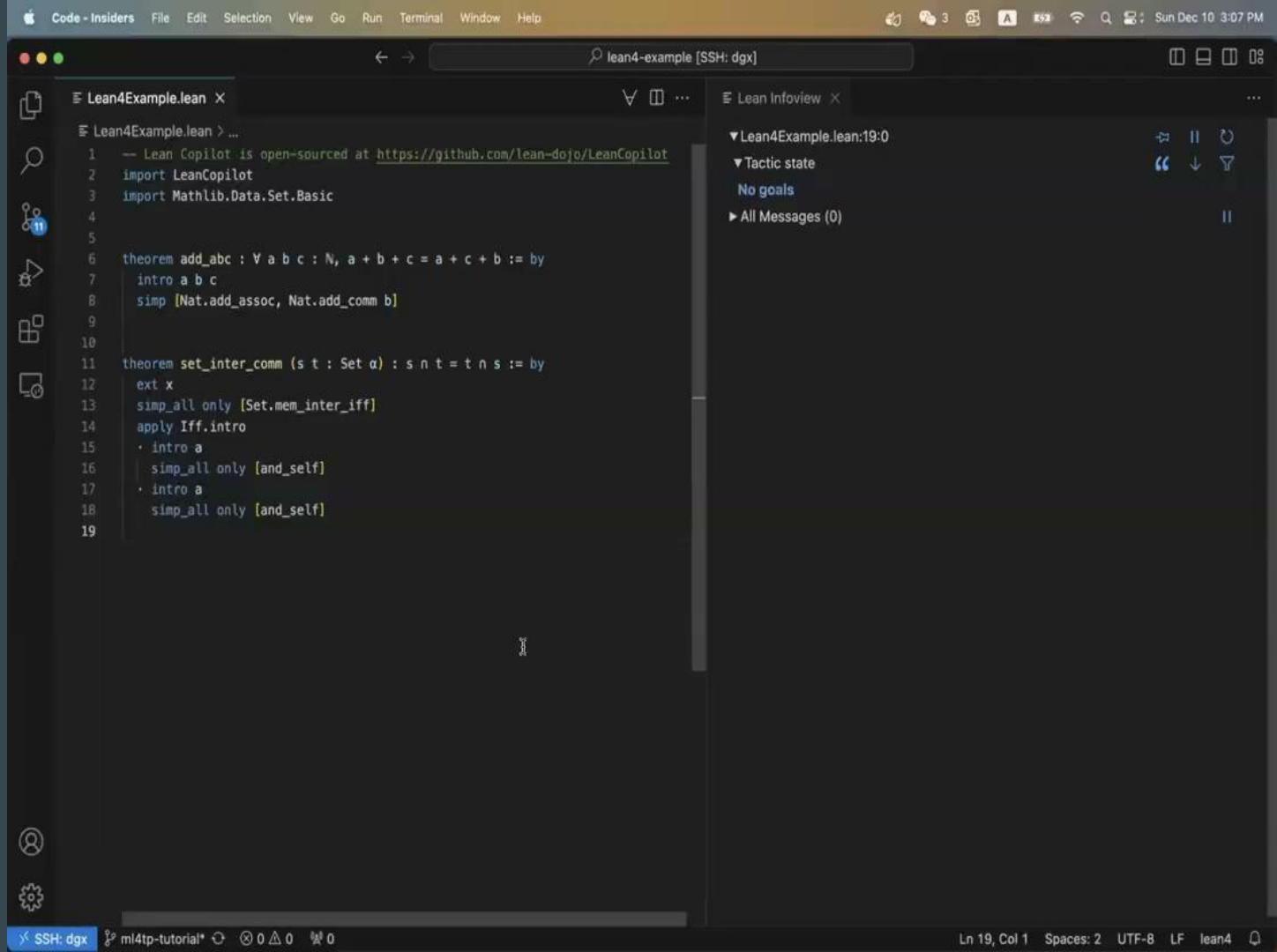
Thank you!

Lean Copilot

suggest_tactics

search_proof

select_premises



Formal AI-Assisted Code Specification And Synthesis

Concrete Steps Towards Safe Sociotechnical Systems

•••

Chapman-Topos Workshop 20240508

Shaowei Lin (Director of Research, Topos Institute)

Joint work with Atlas Computing


TOPOS
INSTITUTE


Atlas Computing