Formal Verification of Large Software Systems

Xiang Yin
John Knight
Department of Computer Science
University of Virginia

Apr-19-10

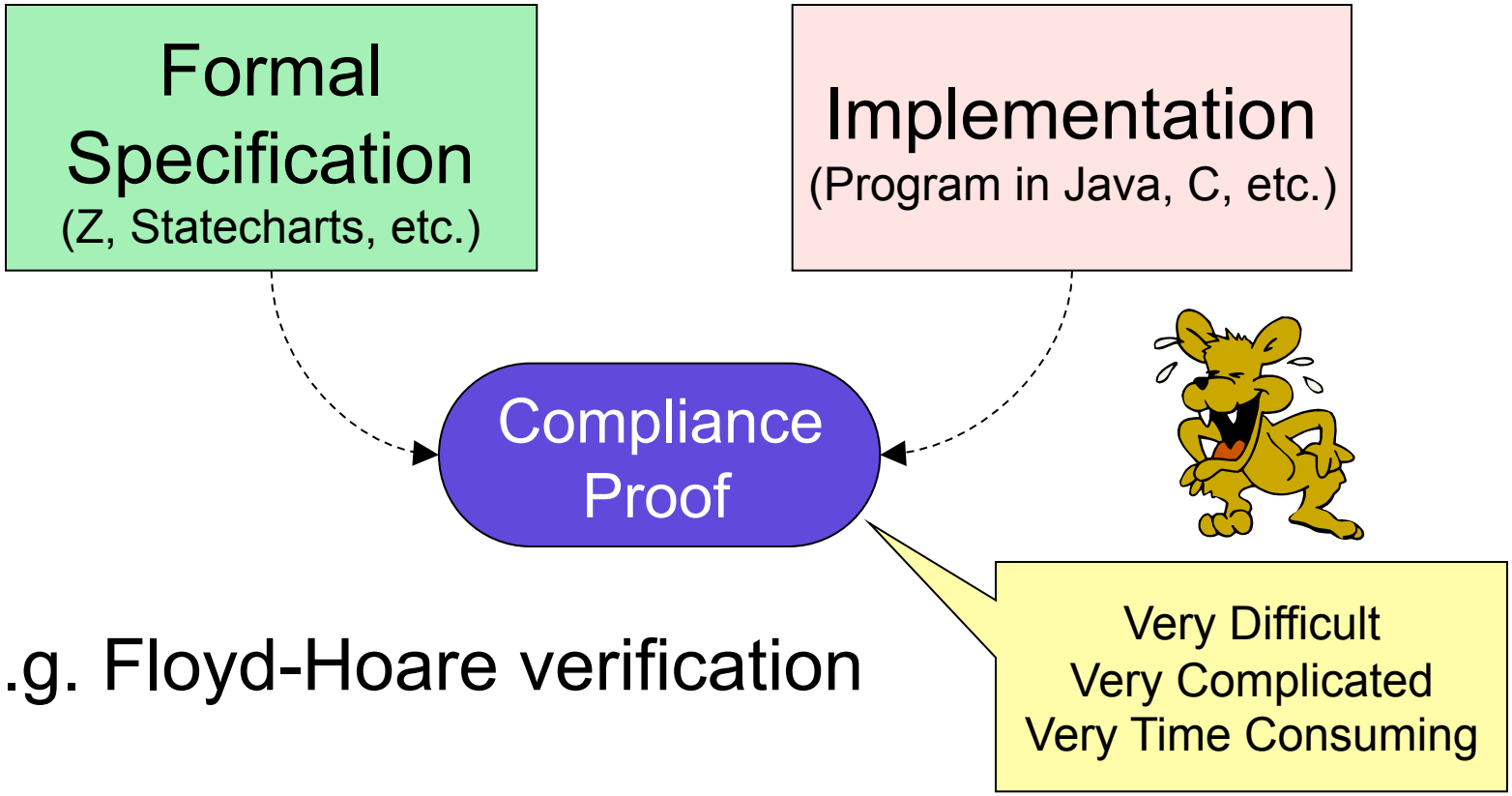
Motivation

- Software Systems
  - Safety and security are critical concerns
  - Formal verification highly desirable

  - Increasing size and complexity
- Current approaches not widely applied
- Formal verification needs to become routine

Proofs in Traditional Verification

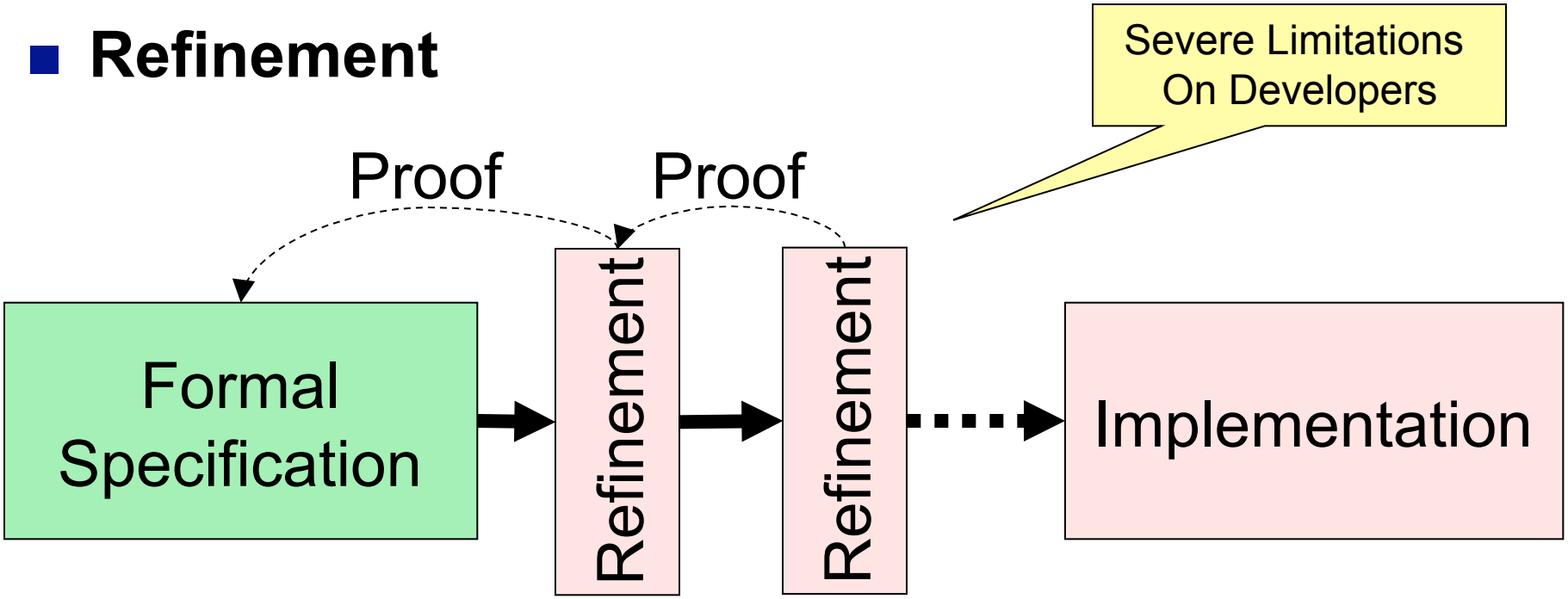
- **Correctness Proof**



- e.g. Floyd-Hoare verification

Proofs in Traditional Verification

- **Refinement**



- e.g. B Method

Our Goal

- Focus on ***functional correctness***
- More **practical** proof structure
- Goals:
  - Relevant
  - Scalable
  - Accessible
  - Efficient

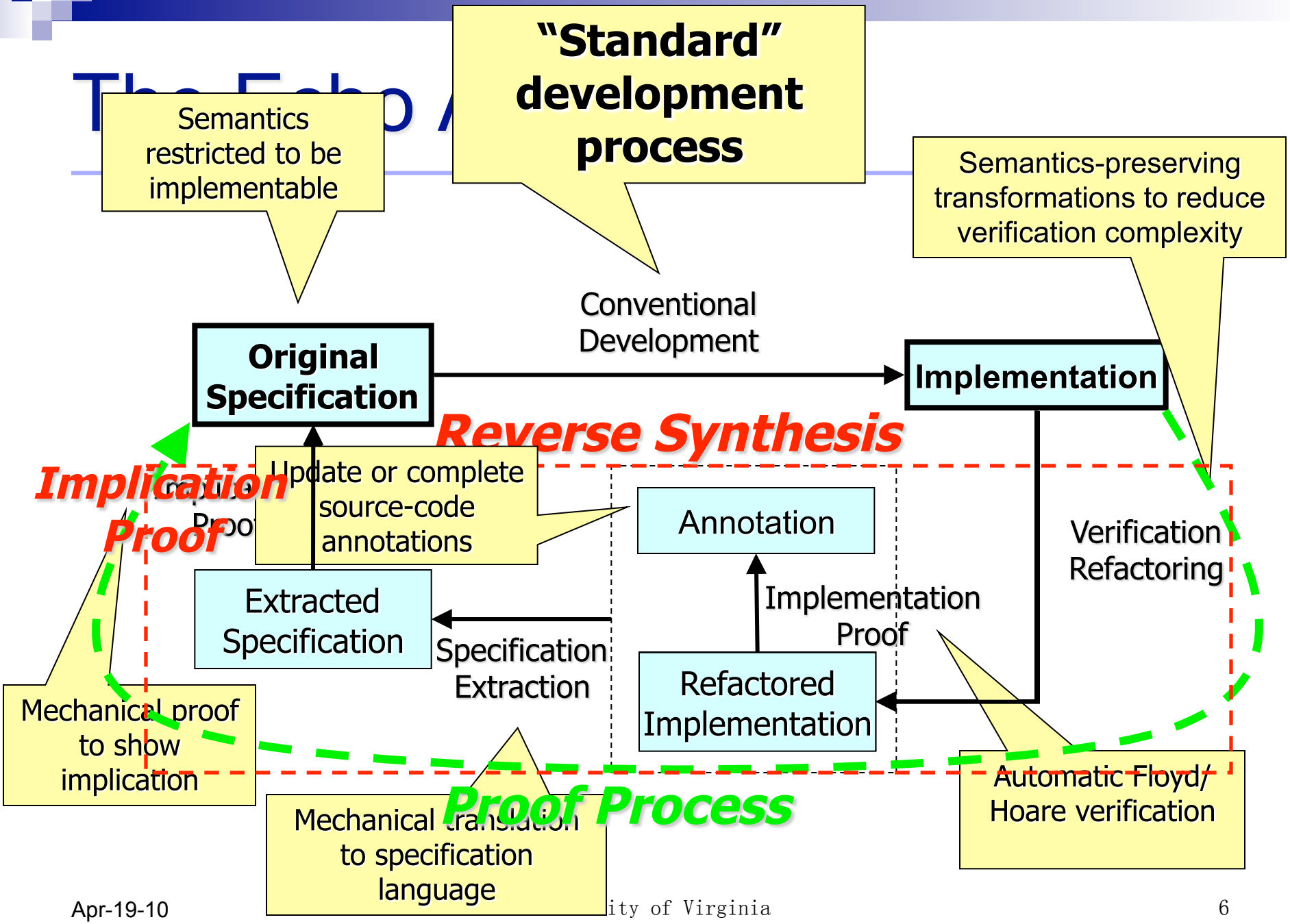
This Is Strictly a Pragmatic Issue

Our Goal

- Focus on ***functional correctness***
- More **practical** proof structure
  - Relevant
    - Benefit from formal verification
  - Scalable
    - Applicable to larger systems
  - Accessible
    - Routine usage
  - Efficient
    - Acceptable time and resource

This Is Strictly a Pragmatic Issue

The Echo A


"Standard" development process
Semantics restricted to be implementable
Semantics-preserving transformations to reduce verification complexity
Conventional Development
Original Specification
Implementation
Reverse Synthesis
Implication Proof
Update or complete source-code annotations
Annotation
Verification Refactoring
Extracted Specification
Implementation Proof
Specification Extraction
Refactored Implementation
Mechanical proof to show implication
Automatic Floyd/ Hoare verification
Proof Process
Mechanical translation to specification language

Structural Matching Hypothesis

- High-level structure of a specification retained in the implementation
  - Specification: contain design information
  - Implementation: often similar in structure, at least in partial
    - Save design effort
    - More maintainable

- e.g. Z schema ⇒ System operation
- e.g. model-based specifications: states & operations

```
state: TYPE = [# a: int, b: int #]
foo(st: state) : state
```

⇒

```
type state is
  record
    a: Integer;
    b: Integer;
  end record;

procedure foo(st: in out state);
--# derives st from st;
```

Structural Matching Hypothesis

High-level structure of specification tends to be retained in the implementation

- Example: Model-based specifications states & operations:

Z schema → System operation

```
state: TYPE = [# a: int, b: int #]
foo(st: state) : state
```

→

```
type state is
  record
    a: Integer;
    b: Integer;
  end record;

procedure foo(st: in out state);
--# derives st from st;
```

- Advantages to implementer:
  - Save design effort, more maintainable

Proof by Parts

- Implementation I, Specification S: `I => S`
  - `pre(S) => pre(I) ∧ post(I) => post(S)`
  - Weakens the pre-condition
  - Decreases non-determinism
- Rely on reverse synthesis:
  - Break into two proofs
  - Make implication proof between two abstract specifications
- Rely on structural matching hypothesis:
  - Pairs of matching elements: types, states, operations
  - Implication lemma for each *distinct* element

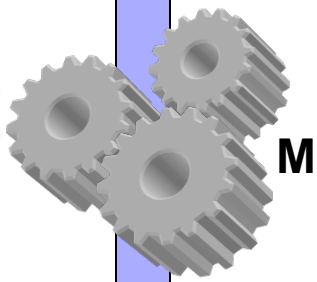
Verification Refactoring

Hard to verify

Original Implementation

Programmers

Semantics-preserving transformations



Mechanical proof

Easier to verify

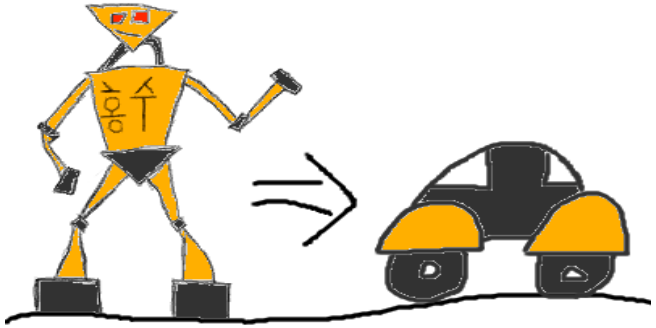
Refactored Implementation

***Bend* The Program To Make It Verifiable**

Verification Refactoring

- Transform implementation to facilitate verification

- <u>Simplify verification conditions</u>
- Reduce complexity introduced in design
  - Careful treatment of special cases
  - Compact data structures
  - Efficient algorithms
  - Complexity in the control- and data-flow



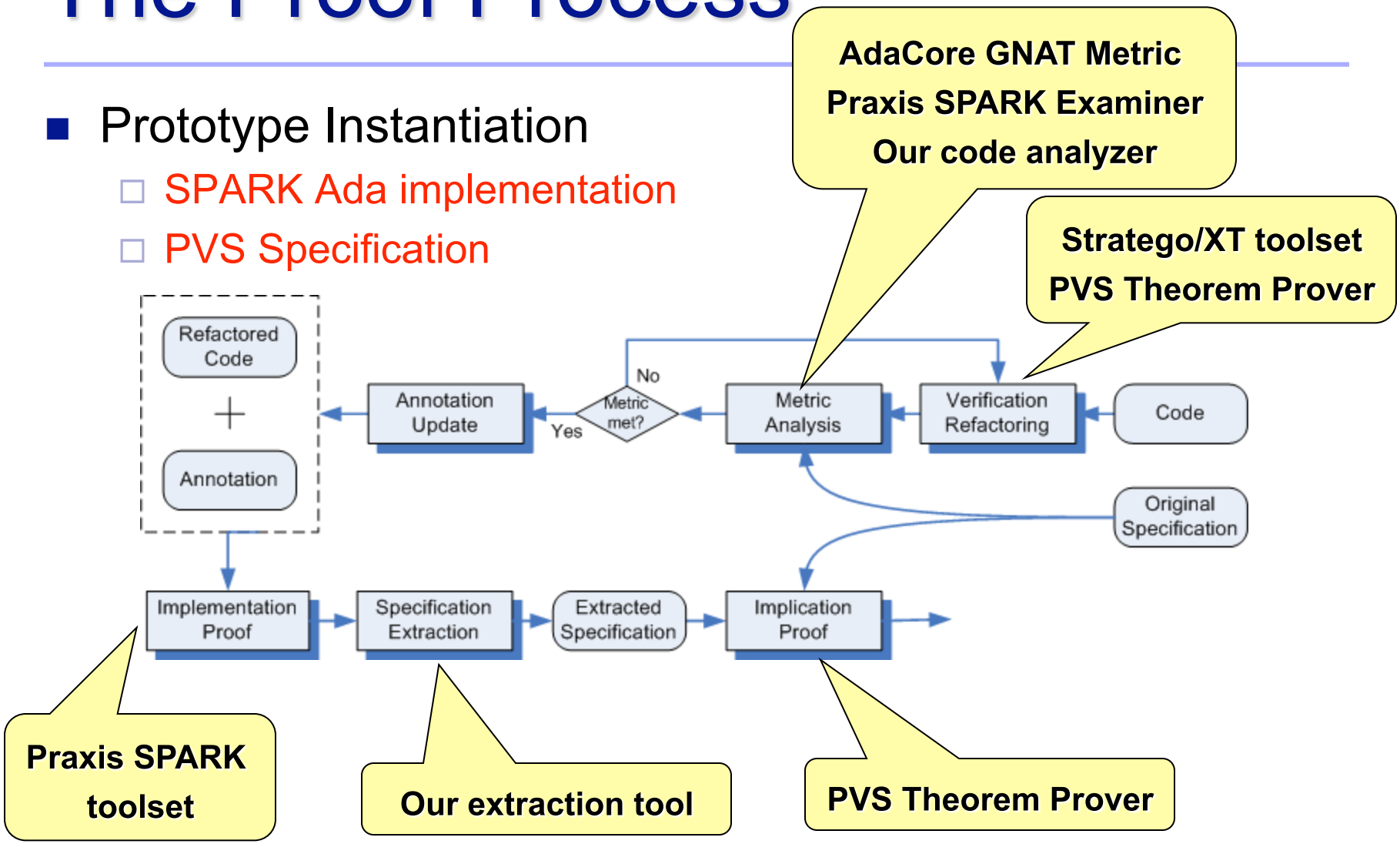
- **<u>Support proof by parts</u>**
- Align the structure
  - Matches extracted specification & original specification
  - Allows an efficient overall proof structure

Metric Analysis

- A hybrid of metrics for review:
  - *Element metrics*
    - Lines of code, number of statements, construct nesting level, etc.
  - *Complexity metrics*
    - McCabe cyclomatic complexity, loop nesting level, etc.
  - *Verification condition metrics*
    - Number and size of VCs, machine time to analyze the VCs, etc.
  - ***Specification matching metric***
    - Support proof by parts
    - Summary of the structures of the original and the extracted specifications
    - Visually inspected and evaluated match-ratio
- Indicate likely difficulty of proof

The Proof Process

- Prototype Instantiation
  - SPARK Ada implementation
  - PVS Specification



Specification Extraction

- Extraction from annotation
  - Proved pre- and post-condition annotation
  - Introduced as a proved lemma
  - Leave out the unrelated implementation details
    - Correctness of the output but not actual algorithm
  - e.g.

```
type state is
  record
    a: Integer;
    b: Integer;
  end record;

procedure foo(st: in out state)
--# derives st from st;
--# pre st.a = 0;
--# post st = st~[a => 1];
is
begin
  -- procedure body
  …
end foo;
```

```
state: TYPE = [# a: int, b: int #]

foo_pre(st: state): bool = (st`a = 0)

foo_post(st_, st: state): bool =
  (st = st_ WITH [`a := 1])

foo(st: state): state

foo: LEMMA FORALL (st: state):
  foo_pre(st) => foo_post(st,
foo(st))
```

Specification Extraction

- Direct extraction from code
  - No proper annotation
  - Not helpful in abstracting out details
  - e.g.

```
procedure foo(st: in out state)
is
begin
  foo1(st);
  st.a = 1;
  foo2(st);
end foo;
```

→

```
foo(st: state): state =
  LET st1 = foo1(st) IN
  LET st2 = st1 WITH [`a := 1] IN
  LET st3 = foo2(st2) IN
  st3
```

- Skeleton extraction
  - Lightweight version, structure only
  - Facilitate metric analysis

- Component Reuse & Model Synthesis

Implication Proof

- Implication ***lemma*** for each pair of matching elements
- Implication ***theorem*** as conjunction of all lemmas

- *Type Lemma*
  - Type refinement

- *State Lemma*
  - State match
  - State initialization

- *Operation Lemma*
  - Applicability
  - Correctness

Implication Proof

- Operation Lemma
  - Set up according to behavior subtyping
  - Applicability
    - The extracted operation has a weaker pre-condition
    - Applicable whenever the original operation is

```
FORALL st:
   Pre_org(R(st)) => Pre_ext(st)
```

  - Correctness
    - The extracted operation has a stronger post-condition if applicable
    - When applicable, generate allowed output of the original operation

```
FORALL st1, st2 | st2 = f(st1):
      Post_ext(st2) AND pre_org(R(st1)) => post_org(R(st2))
```

Evaluation

- Target: The ***Tokeneer ID Station***
  - Hypothetical secure enclave protection software
    - Defined by NSA as security challenge problem
    - Developed by Praxis High Integrity Systems

Developers

Z Specification (117 pages)

PVS Specification (2336 lines)

SPARK Ada Implementation (9939 lines)

Verifiers

- Scenario:
  - Public available artifacts (developed by others)
  - Non-trivial application
  - Several thousand lines long
  - In a domain requiring high assurance
  - Focus on **functional proof**

Tokeneer Proof

- Proof: correctness of functionality:
  - Different from Praxis' correctness by construction proof
- Structural matching hypothesis:
  - Upon review:
    - Source code structure resembled specification closely
  - Skeleton extraction:
    - Structure match ratio 74.7%
- Verification refactoring:
  - Sufficiently similar to proceed without major refactoring
- Specification extraction:
  - 5622 lines of PVS extracted automatically

Tokeneer Proof

- Implementation Proof
  - Pre- / post-condition annotations, freedom from run-time exceptions
  - SPARK toolset: Over 2600 VCs generated, 95% VCs discharged automatically
- Implication Proof
  - Matching elements identified straightforwardly
    - Can be partly automatically suggested by names
  - Over 300 implication lemmas
    - Most TCCs discharged automatically
  - 10% of the lemmas discharged automatically
  - 90% required straightforward human intervention
    - expansion of function definitions
    - introduction of type predicates
    - application of extensionality
    - etc.
- Complete Proof
  - Identified mismatches that were documented design decisions

Conclusion

- **Proof by parts**
  - Focus on proof of functional correctness
  - Designed to scale for large software systems
    - Demonstrated on a program several thousand lines long
  - Does not impose restrictions on software development
  - Also eases the location of implementation defects
  - Infeasible if structural matching hypothesis does not hold
    - Verification refactoring can help align the structures

Questions?