Finally: Practical Formal Verification of Large Software Systems

John C. Knight
University of Virginia

and a cast of thousands

SEHC'13
5th International Workshop on Software Engineering in Health Care

Submission Deadline – February 7

Motivation

- System characteristics
  - Very serious consequences of failure
  - Safety and security are critical concerns
  - ***Formal verification*** highly desirable
- Subject software
  - Compact
  - Efficient
  - Highly functional
  - ***Complexity*** limits formal verification

Traditional Formal Verification

- **Correctness Proof**

Formal Specification
(Z, Statecharts, etc.)

Implementation
(Program in Java, C, etc.)

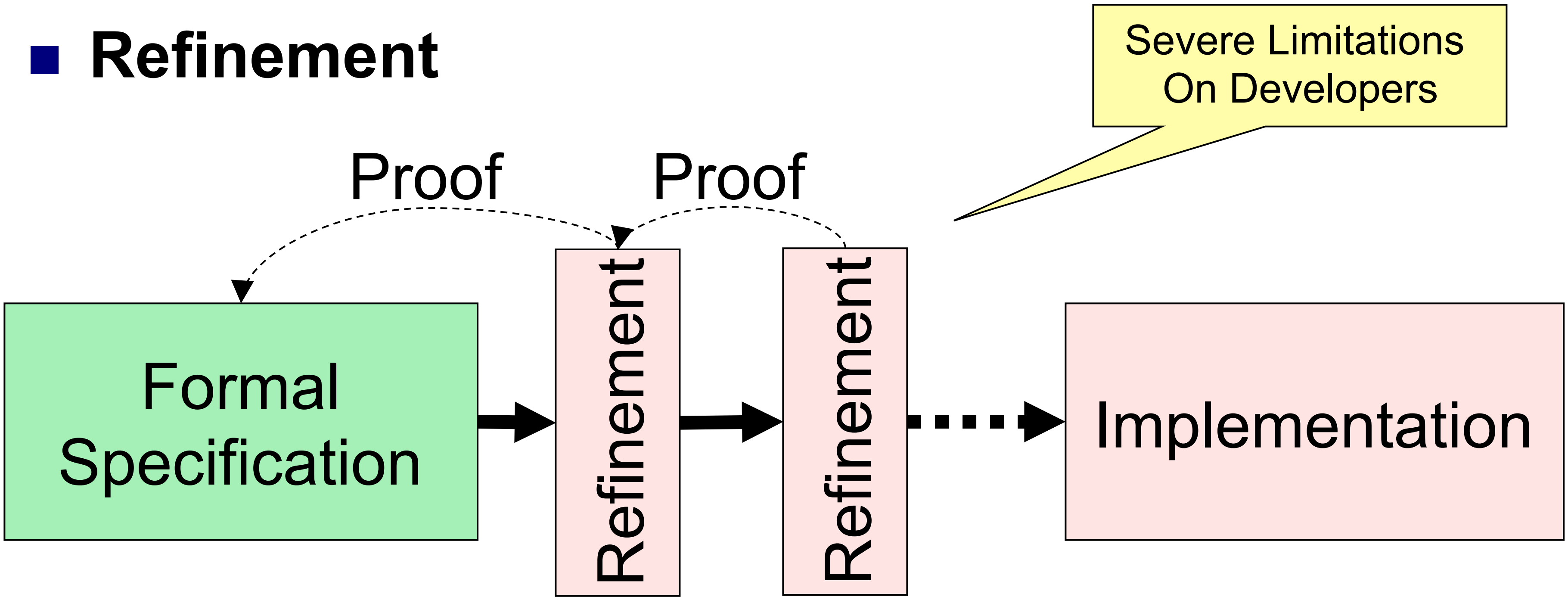
Compliance Proof

Very Difficult
Very Complicated
Very Time Consuming

- e.g. Floyd-Hoare verification

Refinement

- **Refinement**



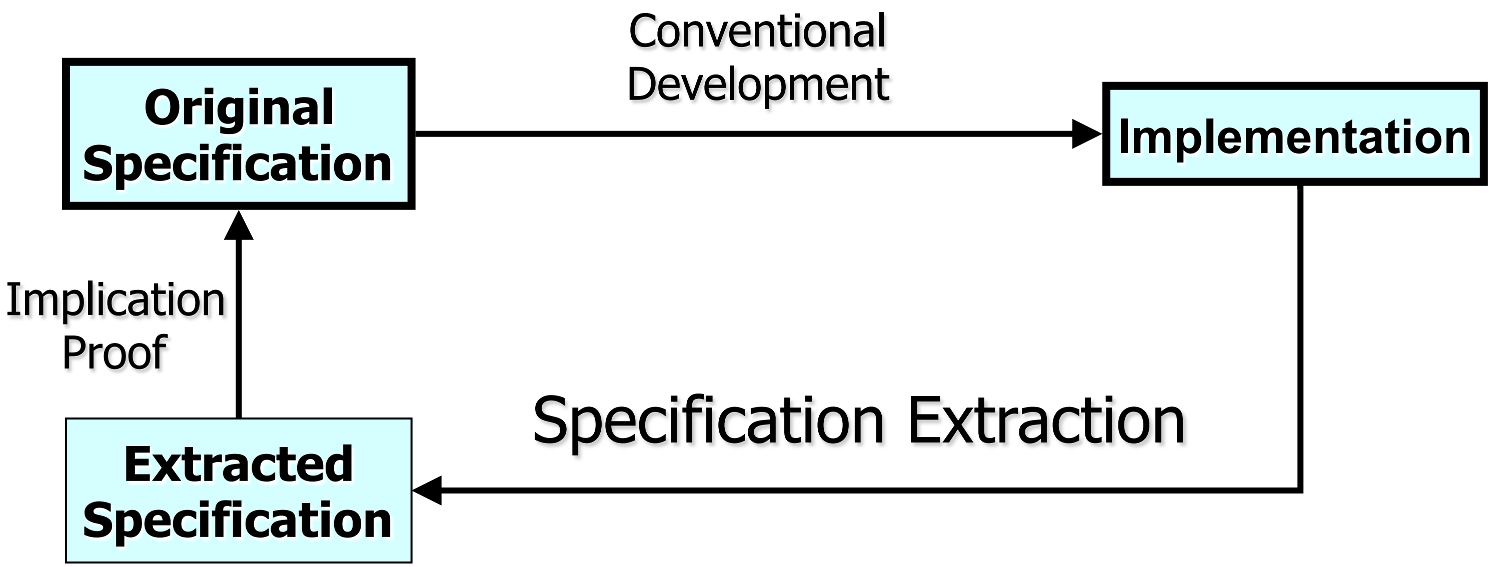
- e.g. B Method

Goals Of *Echo* Verification

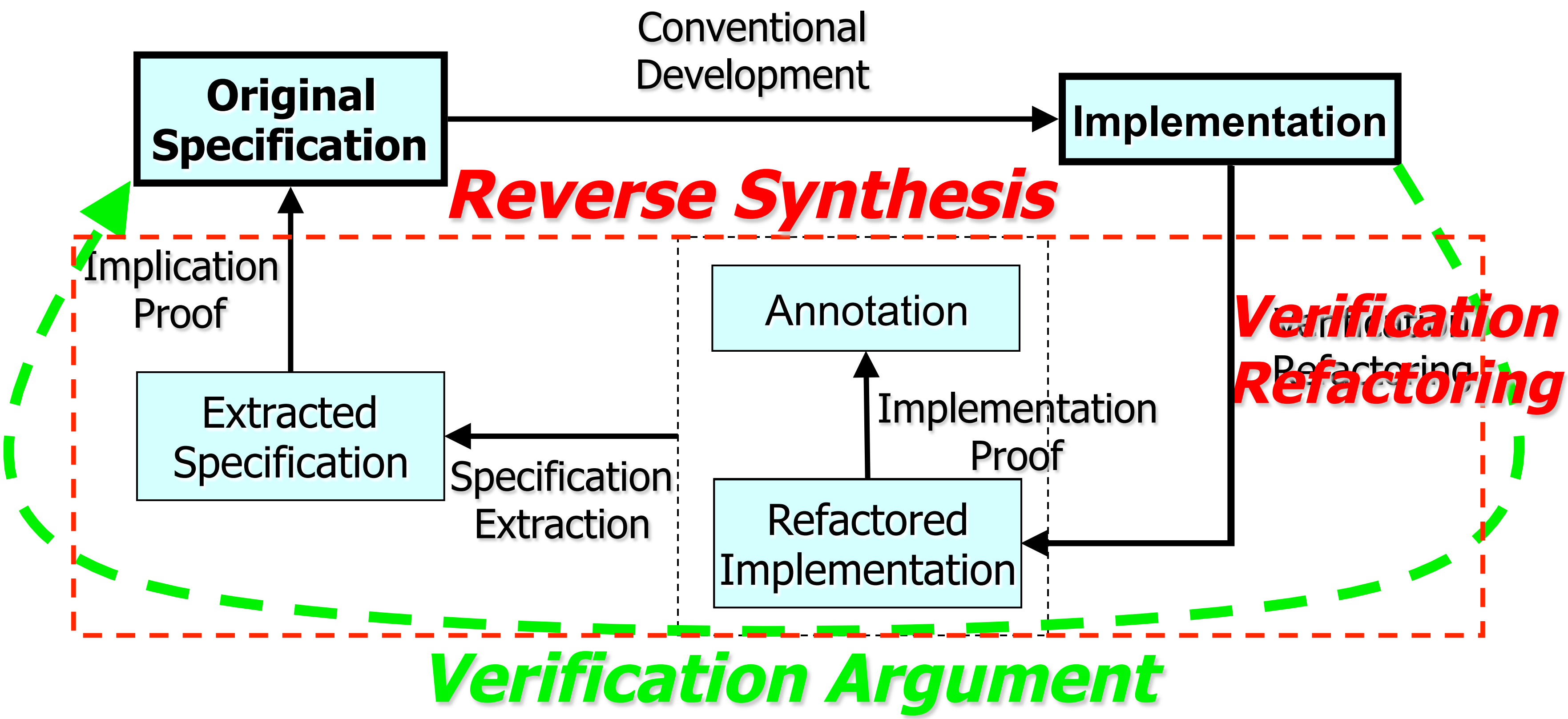
- Focus on ***functional correctness***:
  - No verification of timing
- More **practical** proof structure
  - Relevant
    - Benefit from formal verification
  - Scalable
    - Applicable to larger systems
  - Accessible
    - Routine usage
  - Efficient
    - Acceptable time and resource

This Is Strictly a Pragmatic Issue

Echo Concept

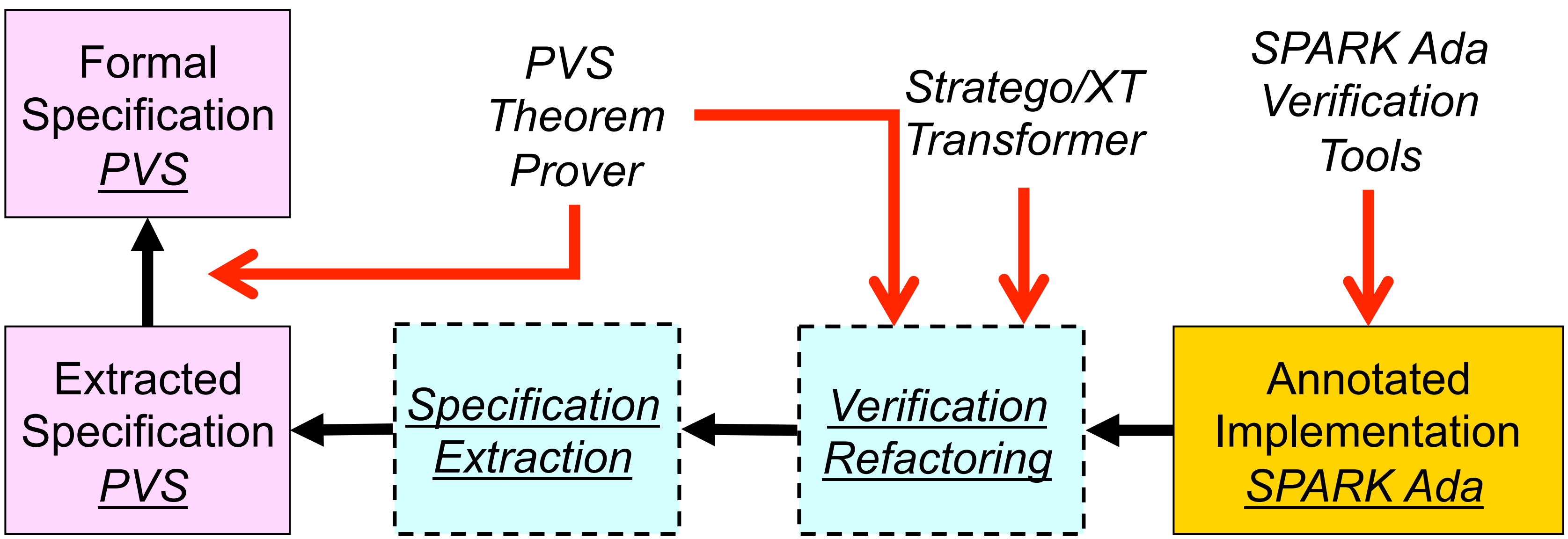


Echo Approach



Prototype Instantiation

- SPARK Ada implementation
- PVS specification
- Stratego transformer



Practicality

- Combines existing powerful techniques
  - Intermediate level representation
  - Partitioned proof

Annotated Code

- Introduces reverse synthesis
  - Fills the gap between proofs
  - Links proof and provides end-to-end argument

Refactoring for Verification
+
Specification Extraction

- Permits use of existing implem[illegible]
  - Development decisions minimally restricted

Reverse Synthesis

Refactoring for Verification

- Change implementation to reduce complexity
- Facilitate verification and proofs
- Transform the code instead of transforming the proof obligations
- Human guided, mechanically checked

- **Specification Extraction**
  - Abstract out irrelevant implementation details
  - Automatically produce ***synthetic*** specification

Implication Proof

- Match the extracted specification to the original specification
- ***Implication, not equivalence***
- Implication theorem:

$$Pre_{original} \Rightarrow Pre_{extracted} \wedge Post_{extracted} \Rightarrow Post_{original}$$

- Proof between two abstract specification models

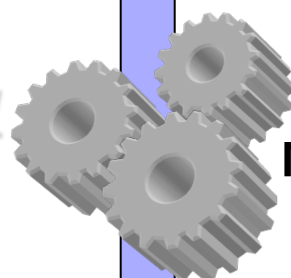
Verification Refactoring

Optimized
Efficient
Hard to verify

Original Implementation

Programmers

<u>Semantics-preserving</u> *transformations*



Mechanical proof

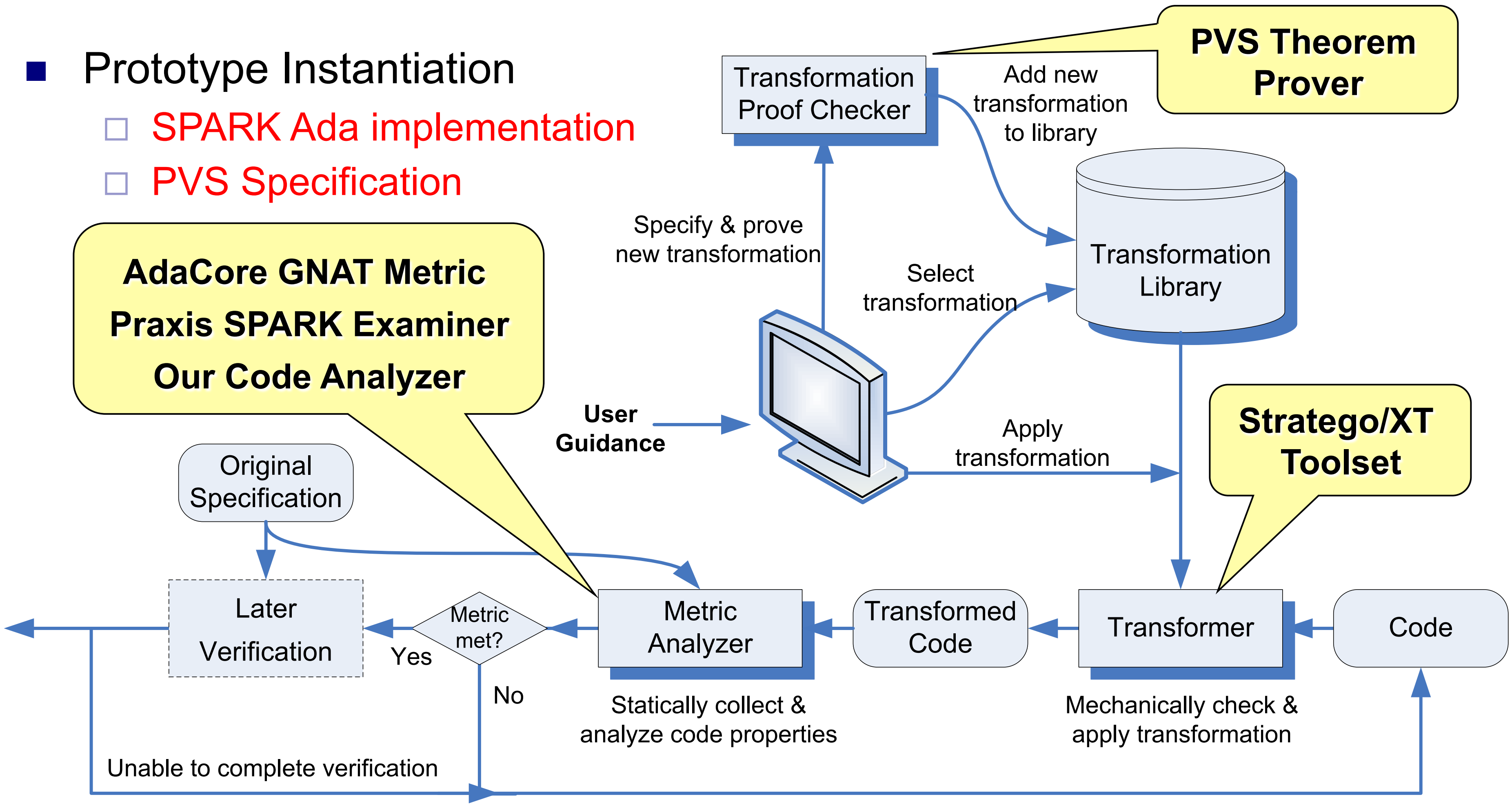
Less Optimized
Less efficient
Easier to verify

Refactored Implementation

Bend **The Program To Make It Verifiable**

Verification Refactoring Process

- Prototype Instantiation
  - SPARK Ada implementation
  - PVS Specification



Complexity Metrics

- A hybrid of metrics for review:
  - Element metrics
    - Lines of code, number of declarations and statements, size of subprograms, construct nesting level, etc.
  - Complexity metrics
    - McCabe cyclomatic complexity, essential complexity, loop nesting level, etc.
  - Verification condition metrics
    - Number and size of VCs, machine time to analyze the VCs, etc.
  - Specification structure metrics
    - Summary of the architectures of the original and the extracted specifications, visually inspected and evaluated match-ratio
- Indicate likely difficulty of proofs
- Interpretation of the metrics is subjective

Example – AES

- Original AES implementation
  - Various optimization
    - Unrolled loops, 32-bit word packing, pre-computed tables, inlined functions
  - SPARK tools ran out of resources
    - Generated VCs too complex

- Verification refactoring
  - Human guided process
  - 50 transformations in 8 categories
    - AES specific transformations
      - Adjusting data structures
      - Reversing table lookups

- Refactored code annotated and verified

AES Verification Results

- Implementation Proof
  - Annotation: pre- / post-conditions, loop invariants
  - SPARK toolset: 306 VCs, 87% VCs discharged automatically in minutes
  - Trivial human guidance on remaining VCs
    - Length of the VCs remained completely manageable
- Specification Extraction
  - Automatically extracted using architectural and direct mapping
  - Showed great similarity in structure to the original specification
- Implication Proof
  - Easily constructed due to structure similarity
  - 201 TCCs, all discharged automatically or subsumed in seconds
  - Implication theorem required straightforward human intervention
    - 32 major lemmas, each proved interactively in a few minutes
- Complete Verification Argument

Refactoring and Defect Detection

- Software defects revealed by failure of proof

- Stages to expose defects:
  - **Application of refactoring**
    - Inconsistency with transformation template
  - ***Implementation* proof**
    - Inconsistency between code and annotations
    - Detected by the SPARK tools
    - Defect in either or both
  - ***Implication* Proof**
    - Unprovable lemma in PVS theorem prover
    - Defective code with corresponding defective annotation
    - Annotation not complete or strong enough

Evaluation of Defect Detection

- Evaluation using seeded defects
  - 15 seeded defects into AES
    - Simple but reflect common errors
    - Randomly change numeric value, array index, operator, variable, statement, function call

- Annotation for defective code
  - Describe actual functional behavior
    - e.g. misunderstanding of the specification
  - Describe desired functional behavior
    - e.g. implementation error

Defect Detection Results

- **Setup 1**: Annotation according to code
- **Setup 2**: Annotation according to specification

Verification Stage	Setup 1 Defects Caught	Setup 1 Defects Left	Setup 2 Defects Caught	Setup 2 Defects Left
Initial state		**15**		**15**
Verification refactoring	**4**	**11**	**4**	**11**
Implementation proof in SPARK	**2**	**9**	**10**	**1**
Implication proof in PVS	**8**	**1**	**0**	**1**

benign

Structural Matching Hypothesis

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

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

PVS

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

⇒

Ada

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

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

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 <u>*distinct*</u> element

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)

SPARK Ada Implementation (9939 lines)

PVS Specification (2336 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

- Formal verification that works:
  - Large programs
  - Realistic development environments
- Verification refactoring to deal with:
  - Unworkably large verification conditions
  - Rigid development process
- Complexity metrics to guide refactoring:
  - Select transformations
  - Determine when the program was likely to be amenable to proof
- Defect detection:
  - Fairly straightforward
  - Demonstrated by seeded errors
- Makes formal verification easier but ***not*** easy

Questions?