Recent Results in Formal Verification*

The Ugly, The Bad & The Good

John C. Knight
University of Virginia, USA

*Joint work with Wes Weimer, Xiang Yin and Elisabeth Strunk
Funded by NASA & NSF
Why This Topic?

- Other speakers will discuss important, relevant, and interesting topics
- I want to provide a contrast so that you can fully appreciate the other presentations
- I want to discuss a topic that has been around longer than Vic and Marv—difficult…
- I actually work in this field & I think it is important
- Opportunity for an early lunch
- Don’t worry, I have plenty of pictures
Things I Like
Safety-Critical Systems
System Characteristics

- Very serious consequences of failure
- Ultra dependability required
- Dependability *cannot* be demonstrated by testing
- Various forms of analysis demonstrated, e.g.:
  - Static analysis
  - Model checking
  - Property proofs
  - *Low level verification based on code annotations*
- *Formal verification* highly desirable
- Subject of study at UVA
The Ugly
"Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language." [Ref. "An Axiomatic Basis for Computer Programming", C.A.R. Hoare, 1969]
Proofs of Correctness

- Community *was told*: We can prove that an algorithm (program) implies (is correct with respect to) a *specification* (formal verification).

- Community *heard*: “Don’t worry, we will be able to prove that software is correct.”

- Leaving out minor problems such as:
  - The specification
  - The proof
  - The translation to bits

Devastating Impact
A Recent View (NRC)

"Traditional software development methods rely on human inspection and testing for validation and verification. Formal methods also use testing, but they employ notations and languages that are amenable to rigorous analysis, and they exploit mechanical tools for reasoning about the properties of requirements, specifications, designs, and code.

Practitioners have been skeptical about the practicality of formal methods. Increasingly, however, there is evidence that formal methods can yield systems of very high dependability in a cost-effective manner, .....


Most recent date Vic announced his retirement. Do we believe him?
The Bad
Traditional Formal Verification

- Refine specification
- Abstract implementation
- Create proof

Formal Specification
(Z, Statecharts, etc.)

Implementation
(Program in Java, C, etc.)

Compliance Proof

Refinement

Abstraction

Very Difficult
Very Complicated
Very Time Consuming
Traditional Formal Verification

- Series of refinements with proofs at each step
- Progression from abstract to concrete
- Lots of proof elements required
- Exemplified by the B Method

Severe Limitations On Developers
Traditional Formal Verification

- Quite complex
- Lots of detail
- Requires relatively high level of skill
- Lots of effort
- Quite constraining on developers
- Only verifies functionality
The Bad
The Bad

200 grams of Sentex on pressurized 747 (30,000 ft)
The Good

Echo... Echo...

Department of Computer Science

University of Virginia
Echo Research Goals

*Practical formal verification for industrial strength software systems built by real developers*

- Start with safety-critical/security-critical systems
- At least 100,000s of lines of source code
- Real developers using conventional techniques
Fundamental Echo Concept

- Specification
- Annotated Code
- Existing Technology
- Synthesized Specification
- Intermediate Level Representation
- Implementation

Proof

Development

Major Challenge

University of Virginia
19
The Echo Approach

- Original Specification
- Restricted Specification
- Extracted Specification
- Refactored Implementation
- Implementation
- Annotation

```
Restrict specification to be implementable

Initial Refinement

Original Specification

Mechanical proof to show implication

Mechanical translation to specification language

Reverse Synthesis

Speculative-preserving transformations to reduce verification complexity

"Standard" development process

Automatic Floyd/Hoare verification

Conventional Development

Implementation Proof

Implementation

Annexation

Verification Argument

Implication Proof

Verification Argument

Speculative Synthesis
```
Reverse Synthesis (of a Spec)

- Central challenge in Echo

- Refactoring for Verification
  - Change implementation to reduce complexity
  - Facilitate verification
  - Automate to the extent possible
  - Mechanically check human guidance

- Specification Extraction
  - Abstract out irrelevant implementation details
  - Produce *synthetic* specification
Refactoring for Verification

Optimized
Efficient
Hard to verify

Original Implementation

Semantic-preserving transformations

Less Optimized
Less efficient
Easier to verify

Refactored Implementation

Programmers

Mechanical proof

Bend The Program To Make It Verifiable
Specification Extraction

- Eliminate the implementation details:
  - Proved in the implementation proof
  - Irrelevant to remainder of the verification

- Extraction techniques developed for Echo:
  - Architecture Mapping
  - Direct Translation
  - Library Transformation
  - Model Synthesis
The Echo Approach

Now Let’s Look At This Piece Of The Puzzle
Implementation Proof

- Praxis High Integrity Systems:
  - Correctness by Construction process using SPARK Ada

- Microsoft:
  - More than 500,000 annotations in Vista
  - Revealed more than 100,000 software defects

- Current tools very capable:
  - SPARK Examiner & Verifier
  - ESC Java
  - Microsoft SAL
The Echo Approach

- Original Specification
- Restricted Specification
- Implementation
- Refactored Implementation
- Implementation Proof
- Refactoring For Verification
- Annotation
- Conventional Development
- Implication Proof
- Specification Extraction
- Finally The Implication Proof
Implication Proof

- Match the extracted specification to the restricted specification

- Implication, not equivalence

- Implication theorem:

\[
\text{Pre}_{\text{original}} \Rightarrow \text{Pre}_{\text{extracted}} \land \text{Post}_{\text{extracted}} \Rightarrow \text{Post}_{\text{original}}
\]

- Proof between two abstract specification models

- *Almost completely automated, trivial human guidance*
Integrating Complementary Techniques

Original Formal Specification

Restricted Implementable Specification

Primary Refinement

Implementation Property Annotations

Implementation

Code-level Verification Tools

Automatic Property Extractor

Theorem Prover

Extracted Specification

Algebraic Properties

Inferred Temporal Properties

Temporal Properties

Extracted Temporal Model

Model Checker

Other Properties

Other Extracted Results

Other Formal Tools

Implication Proof

Initial Refinement

Developed Temporal Properties

Assurance Argument

Department of Computer Science

University of Virginia
Don’t Forget The Bad
Yet More Bad
First Echo Instantiation

- Specification In PVS
  - Proof Using PVS Theorem Prover
- Synthesized Specification

Development – Any Technique
- Intermediate Level Representation In SPARK Ada
  - Proof Using SPARK Ada Tools
- Implementation In SPARK Ada
Preliminary Assessment

- **Assessment issues:**
  - How practical is Echo?
  - What resources are required?
  - How successful is Echo at allowing freedom during development?

- **Approach:**
  - Apply Echo to a specimen system:
    - That is small but realistic
    - Built by *others*
  - **Target:**
    - *Advanced Encryption Standard* (AES)
    - Key expansion and NIST APIs omitted
Artifacts Employed

- FIPS Specification of AES
  - English + Math
- Reference Implementation
  - C
- Optimized Implementation
  - C

Developers

Verifiers
Evaluation Activities

Scenario:
- Assume specification used “as is”
- Assume Reference Implementation is a preliminary version developed along the way
- Assume Optimized Implementation is the final version to be delivered
- Assume conventional engineering
- Attempt to verify both implementations separately...

Required:
- Translation of these artifacts to other languages
- Annotation of source code
Artifact Transformations

FIPS Specification of AES

English + Math

Reference Implementation

C

Optimized Implementation

C

FIPS Specification of AES

PVS

Reference Implementation

SPARK Ada

Optimized Implementation

SPARK Ada

Possibly Already Available For Other Systems
Echo Instantiation For AES

Natural Language Specification → Conventional Development Artifacts → Reference Implementation → Optimized Implementation

Formal Specification

Implication Proof → Anotated Reference Implementation → Abstract Specification (Reference)

Implication Proof

Abstract Specification (Optimized)

Sequence Of Transformed Implementations

Hypothesized Conventional Development

Echo Verification

Department of Computer Science

University of Virginia
Verification Approach

Abstract Specification

Specification Extraction

Reference Implementation SPARK Ada

Formal Specification

PVS Theorem Prover

SPARK Ada Verification Tools

Abstract Specification

Specification Extraction

Verification Refactoring

Optimized Implementation SPARK Ada
Verification Refactoring

- Reverse each of the following:
  - Loop unrolling
  - Word packing:
    - Four bytes packed into 32-bit words for efficiency
  - Table-look-up implementation:
    - Function calculations were replaced by tables
  - Function in-lining

- Each transformation proved to maintain semantics
- Result: program much more suited to verification
Results—Reference Implementation

- Code compliance with annotations:
  - SPARK Ada toolset
  - Mostly automatic, some trivial human guidance
- Abstract specification:
  - Extracted automatically
  - Shown to imply original specification automatically
- Complete verification argument
Results—Optimized Implementation

- **After refactoring:**
  - Code compliance with annotations:
    - SPARK Ada toolset
    - 136 out of 144 verification conditions discharged automatically
    - Eight required trivial human guidance
  - Abstract specification:
    - Extracted automatically
    - Approx. half implication theorem obligations discharged automatically
    - Remainder required very little human insight to develop proof command sequences
- **Complete verification argument**
Basili Retirement Timeline

Vic announces his retirement
Vic announces his retirement
Vic announces his retirement
Vic announces his retirement
Vic announces his retirement
Vic announces his retirement
Vic announces his retirement
Vic announces his retirement
Vic announces his retirement
Vic announces his retirement


Vic announces his return to Maryland as Chair, Dean and President
Conclusion

- Formal verification is very desirable if it can be done
- Existing techniques have many disadvantages
- Echo tackles the problem in a new way:
  - Makes formal verification easier but *not* easy
  - Permits a high degree of automation
  - Hopeful that it will work on large systems
- Next experiment:
  - NSA’s Tokeneer system
  - ~30,000 lines of Ada
Echo
Verification Challenge
Questions?
Can You Believe These?