CS5232: Formal Specification and Design Techniques

Module Overview and Introduction

Ilya Sergey

Spring 2023

Module Overview

Instructional Staff

A/P Ilya Sergey, instructor

George Pîrlea, TA

Course Info and Material

- All information, including the syllabus, available on **website** at:

  https://ilyasergey.net/CS5232/

- Textbooks:
  - *Specifying Systems* by Leslie Lamport, 2002
  - *Program Proofs* by Rustan Leino, 2020

- Class notes and additional reading material to be posted on the website

- Announcements, submissions and grades on **Canvas**

- Accompanying code on GitHub (send me your GH handle to get access!):

  https://github.com/cs5232

Goals of the Module

1. Learn about formal methods (FMs) in system design and software engineering
2. Understand how FMs help produce high-quality software
3. Learn about formal modeling and specification languages
4. Write and understand formal requirement specifications
5. Learn about main approaches in formal software verification
6. Learn about underpinning for state-of-the-art verification tools
7. Use automated and interactive tools to verify models and code

Course Topics

Software Specification and Validation

- High-level system design
- Foundations of automated reasoning
- Code-level design

Main Software Validation Techniques

Model Checking: often automatic, abstract

Decidable Reasoning: reducing verification to known algorithmic problems

Deductive Verification: typically semi-automatic, precise (source code level)

Course Topics

Software Specification and Validation

- High-level system design
- Foundations of automated reasoning
- Code-level design

Main Software Validation Techniques

Model Checking: often automatic, abstract

Decidable Reasoning: reducing verification to known algorithmic problems

Deductive Verification: typically semi-automatic, precise (source code level)

Abstract Interpretation: automatic, correct, incomplete, terminating

Part I: High-Level Design

Language: TLA+

- Lightweight modeling language for system design
- Amenable to a fully automatic analysis
- Aimed at expressing complex behavior and properties of a software system
- Intuitive structural modeling tool based on Boolean functions
- Automatic analyzer based on bounded model checking

Learning Outcomes

- Design and model software systems in the TLA+ language
- Check models and their properties with the TLC model checker
- Understand the practical limitations of TLA+

Part I: High-Level Design

Language: TLA+

- Lightweight modeling language for system design
- Amenable to a fully automatic analysis
- Aimed at expressing complex behavior and properties of a software system
- Intuitive structural modeling tool based on Boolean functions
- Automatic analyzer based on bounded model checking

Learning Outcomes

- Design and model software systems in the TLA+ language
- Check models and their properties with the TLC model checker
- Understand the practical limitations of TLA+

Part II: Logical Foundations

Language: SAT and SMT formulas

- Basic formalism for encoding systems and their properties
- Foundation of most of existing verification techniques
- Typically, not used explicitly but rather as a compilation target
- Puts strict constraints on expressivity

Learning Outcomes

- Identify problems that can be encoded as SAT or SMT
- Encode decidable verification and synthesis problems
- Using state of the art solvers, such as Z3 and CVC4

Part II: Logical Foundations

Language: SAT and SMT formulas

- Basic formalism for encoding systems and their properties
- Foundation of most of existing verification techniques
- Typically, not used explicitly but rather as a compilation target
- Puts strict constraints on expressivity

Learning Outcomes

- Identify problems that can be encoded as SAT or SMT
- Encode decidable verification and synthesis problems
- Using state of the art solvers, such as Z3 and CVC5

Part III: Code-level Specification

Language: Dafny

- Programming language with specification constructs
- Specifications embedded in source code as formal contracts
- Tool support with sophisticated verification engines
- Automated analysis based on theorem proving techniques

Learning Outcomes:

- Write formal specifications and contracts in Dafny
- Verify functional properties of Dafny programs with automated tools
- Understand what can and cannot be expressed in Dafny

Part III: Code-level Specification

Language: Dafny

- Programming language with specification constructs
- Specifications embedded in source code as formal contracts
- Tool support with sophisticated verification engines
- Automated analysis based on theorem proving techniques

Learning Outcomes:

- Write formal specifications and contracts in Dafny
- Verify functional properties of Dafny programs with automated tools
- Understand what can and cannot be expressed in Dafny

Assessment

Homework Assignments: 30%

- Homework 1: TLA+: 10%
- Homework 2: SAT and SMT: 10%
- Homework 3: Dafny: 10%

Theory Quizzes: 30%

- Quiz 1 (Week 7, 1 hour): Properties of Computations and TLA+: 15%
- Quiz 2 (Week 12, 1 hour): SAT, SMT, and Deductive Verification: 15%

Research Project: 40%

- Done in teams of one or two
- Includes implementation, written report, and presentation
- Part of the score is by means of self- and peer assessment

Introduction

Today’s reality

Software has become critical to modern life

- Communication (internet, voice, video, . . .)
- Transportation (air traffic control, avionics, cars, . . .)
- Health Care (patient monitoring, device control, . . .)
- Finance (automatic trading, banking, . . .)
- Defense (intelligence, weapons control, . . .)
- Manufacturing (precision milling, assembly, . . .)
- Process Control (oil, gas, water, . . .)
- . . .

Embedded Software

Software is now embedded everywhere

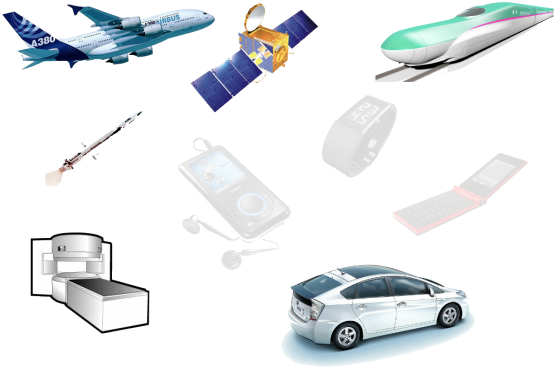
Some of it is critical

Failing software costs money and life!

Embedded Software

Software is now embedded everywhere

Some of it is critical

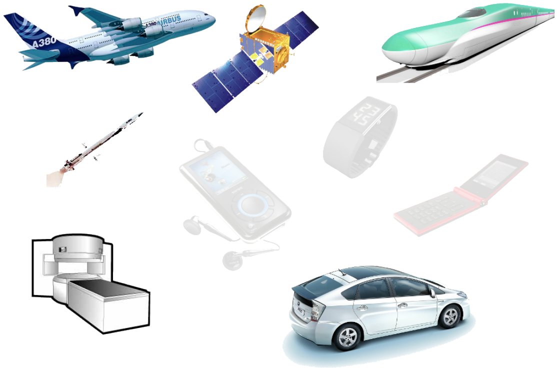


Failing software costs money and life!

Embedded Software

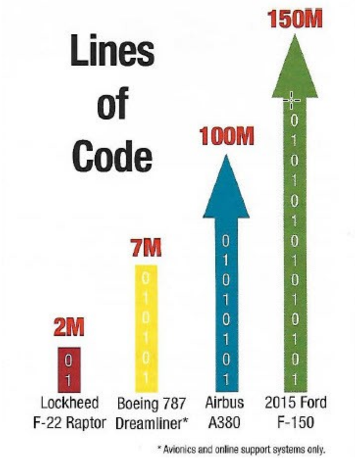
Software is now embedded everywhere

Some of it is critical

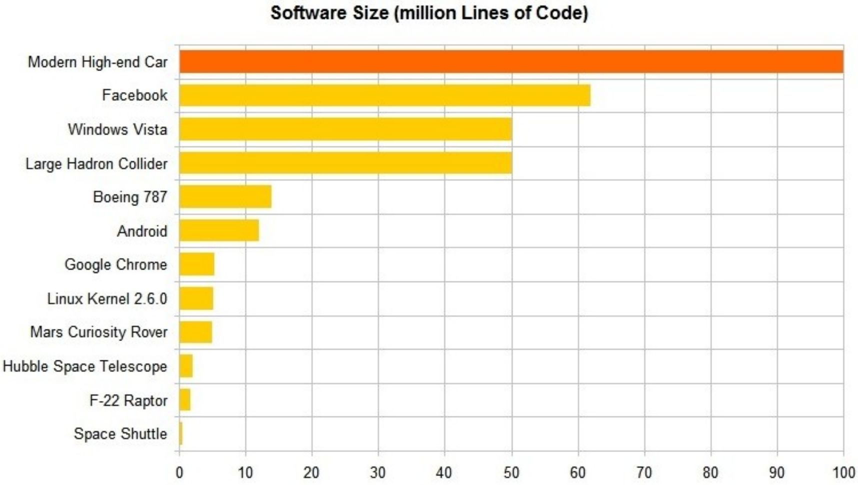


Failing software costs money and life!

Software Systems are Growing Very Large



Software Systems are Growing Very Large



Software Systems are Growing Very Large

Automotive Software

A typical 2022 car model contains >100M lines of code
How do you verify that?

Current cars admit hundreds of onboard functions
How do you cover their combination?

Software Systems are Growing Very Large

Automotive Software

A typical 2022 car model contains >100M lines of code
How do you verify that?

Current cars admit hundreds of onboard functions
How do you cover their combination?

Ex. does braking when changing the radio station and starting the windscreen wiper, affect air conditioning?

Software Systems are Growing Very Large

Automotive Software

A typical 2022 car model contains >100M lines of code

How do you verify that?

Current cars admit hundreds of onboard functions

How do you cover their combination?

Ex. does braking when changing the radio station and starting the windscreen wiper, affect air conditioning?

Failing Software Costs Money

Expensive recalls of products with embedded software

Lawsuits for loss of life or property damage

- Car crashes (e.g., Toyota Camry 2005)

Thousands of dollars for each minute of down-time

- (e.g., Denver Airport Luggage Handling System)

Huge losses of monetary and intellectual investment

- Rocket boost failure (e.g., Ariane 5)

Business failures associated with buggy software

- (e.g., Ashton-Tate dBase, Ethereum DAO)

Failing Software Costs Lives

Potential problems are obvious:

- Software used to control nuclear power plants
- Air-traffic control systems
- Spacecraft launch vehicle control
- Embedded software in cars

A well-known and tragic example: Therac-25 X-ray machine failures

https://en.wikipedia.org/wiki/Therac-25

The Peculiarity of Software Systems

Software seems particularly prone to faults

Tiny faults can have catastrophic consequences

- Ariane 5
- Mars Climate Orbiter, Mars Sojourner
- Pentium-Bug
- . . .

Rare bugs can occur

- avg. lifetime of a passenger plane: 30 years
- avg. lifetime of a car: < 10 years, but > 1.4B cars in 2022

Logic and implementation errors represent security exploits

- (too many to mention)

The Peculiarity of Software Systems

Software seems particularly prone to faults

Tiny faults can have catastrophic consequences

- Ariane 5
- Mars Climate Orbiter, Mars Sojourner
- Pentium FDIV bug. . .

Rare bugs can occur

- avg. lifetime of a passenger plane: 30 years
- avg. lifetime of a car: < 10 years, but > 1.4B cars in 2022

Logic and implementation errors represent security exploits

- (too many to mention)

The Peculiarity of Software Systems

Software seems particularly prone to faults

Tiny faults can have catastrophic consequences

- Ariane 5
- Mars Climate Orbiter, Mars Sojourner
- Pentium-Bug
- ...

Rare bugs can occur

- avg. lifetime of a passenger plane: 30 years
- avg. lifetime of a car: < 10 years, but $> 1.4\mathrm{B}$ cars in 2022

Logic and implementation errors represent security exploits

- (too many to mention)

The Peculiarity of Software Systems

Software seems particularly prone to faults

Tiny faults can have catastrophic consequences

- Ariane 5
- Mars Climate Orbiter, Mars Sojourner
- Pentium-Bug
- ...

Rare bugs can occur

- avg. lifetime of a passenger plane: 30 years
- avg. lifetime of a car: < 10 years, but $> 1.4\mathrm{B}$ cars in 2022

Logic and implementation errors represent security exploits

- (too many to mention)

Observation

Building software is what most of you will do after graduation

- You'll be developing systems in the context above
- Given the increasing importance of software,
  - you may be liable for errors
  - your job may depend on your ability to produce reliable systems

What are the challenges in building reliable and secure software?

Observation

Building software is what most of you will do after graduation

- You’ll be developing systems in the context above
- Given the increasing importance of software,
  - you may be liable for errors
  - your job may depend on your ability to produce reliable systems

What are the challenges in building reliable and secure software?

Achieving Reliability in Engineering

Some well-known strategies from civil/mechanical engineering:

- Precise calculations/estimations of forces, stress, etc.
- Hardware redundancy ("make it a bit stronger than necessary")
- Robust design (single fault not catastrophic)
- Clear separation of subsystems (any airplane flies with dozens of known and minor defects)
- Design follows patterns that are proven to work

Achieving Reliability in Engineering

Some well-known strategies from civil/mechanical engineering:

- Precise calculations/estimations of forces, stress, etc.
- Hardware redundancy ("make it a bit stronger than necessary")
- Robust design (single fault not catastrophic)
- Clear separation of subsystems (any airplane flies with dozens of known and minor defects)
- Design follows patterns that are proven to work

Achieving Reliability in Engineering

Some well-known strategies from civil/mechanical engineering:

- Precise calculations/estimations of forces, stress, etc.
- Hardware redundancy ("make it a bit stronger than necessary")
- Robust design (single fault not catastrophic)
- Clear separation of subsystems (any airplane flies with dozens of known and minor defects)
- Design follows patterns that are proven to work

Achieving Reliability in Engineering

Some well-known strategies from civil/mechanical engineering:

- Precise calculations/estimations of forces, stress, etc.
- Hardware redundancy (“make it a bit stronger than necessary”)
- Robust design (single fault not catastrophic)
- Clear separation of subsystems (any airplane flies with dozens of known and minor defects)
- Design follows patterns that are proven to work

Achieving Reliability in Engineering

Some well-known strategies from civil/mechanical engineering:

- Precise calculations/estimations of forces, stress, etc.
- Hardware redundancy ("make it a bit stronger than necessary")
- Robust design (single fault not catastrophic)
- Clear separation of subsystems (any airplane flies with dozens of known and minor defects)
- Design follows patterns that are proven to work

Achieving Reliability in Engineering

Some well-known strategies from civil/mechanical engineering:

- Precise calculations/estimations of forces, stress, etc.
- Hardware redundancy ("make it a bit stronger than necessary")
- Robust design (single fault not catastrophic)
- Clear separation of subsystems (any airplane flies with dozens of known and minor defects)
- Design follows patterns that are proven to work

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn't help against logical errors
  Redundant SW development only viable in extreme cases
- No physical or modal separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Most SW engineers are untrained in correctness
- Cost efficiency more important than reliability
- Design practice for reliable software is not yet mature

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn't help against logical errors
  Redundant SW development only viable in extreme cases
- No physical or modal separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Most SW engineers are untrained in correctness
- Cost efficiency more important than reliability
- Design practice for reliable software is not yet mature

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn't help against logical errors
  Redundant SW development only viable in extreme cases
- No physical or modal separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Most SW engineers are untrained in correctness
- Cost efficiency more important than reliability
- Design practice for reliable software is not yet mature

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn't help against logical errors
  Redundant SW development only viable in extreme cases
- No physical or modal separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Most SW engineers are untrained in correctness
- Cost efficiency more important than reliability
- Design practice for reliable software is not yet mature

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn't help against logical errors
  Redundant SW development only viable in extreme cases
- No physical or modal separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Most SW engineers are untrained in correctness
- Cost efficiency more important than reliability
- Design practice for reliable software is not yet mature

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn't help against logical errors
  Redundant SW development only viable in extreme cases
- No physical or modal separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Most SW engineers are untrained in correctness
- Cost efficiency more important than reliability
- Design practice for reliable software is not yet mature

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn't help against logical errors
  Redundant SW development only viable in extreme cases
- No physical or modal separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Most SW engineers are untrained in correctness
- Cost efficiency more important than reliability
- Design practice for reliable software is not yet mature

How to Ensure Software Correctness?

A Central Strategy: **Testing**
(others: SW processes, reviews, libraries, . . .)

Testing against inherent SW errors (“bugs”)

1. Design test configurations that hopefully are representative
2. Check that the system behaves as intended on them

Testing against external faults

1. Inject faults (memory, communication) by simulation or radiation
2. Check that the system’s performance degrades gracefully

How to Ensure Software Correctness?

A Central Strategy: **Testing**
(others: SW processes, reviews, libraries, . . .)

Testing against inherent SW errors ("bugs")

1. Design test configurations that hopefully are representative
2. Check that the system behaves as intended on them

Testing against external faults

1. Inject faults (memory, communication) by simulation or radiation
2. Check that the system's performance degrades gracefully

Limitations of Testing

Limitations of Testing

Testing can show the presence of errors, but not their absence

Exhaustive testing viable only for trivial systems

Representativeness of test cases/injected faults is subjective

How to test for the unexpected? Rare cases?

Testing is labor intensive, hence expensive

Limitations of Testing

Testing can show the presence of errors, but not their absence

Exhaustive testing viable only for trivial systems

Representativeness of test cases/injected faults is subjective

How to test for the unexpected? Rare cases?

Testing is labor intensive, hence expensive

Limitations of Testing

Testing can show the presence of errors, but not their absence

Exhaustive testing viable only for trivial systems

Representativeness of test cases/injected faults is subjective

How to test for the unexpected? Rare cases?

Testing is labor intensive, hence expensive

Complementing Testing: Formal Verification

A Sorting Program:

```
int* sort(int* a) {
  ...
}
```

Testing sort:

- sort({3, 2, 5}) == {2, 3, 5} √
- sort({}) == {} √
- sort({17}) == {17} √

Complementing Testing: Formal Verification

A Sorting Program:

```
int* sort(int* a) {
  ...
}
```

Testing `sort`:

- `sort({3,2,5}) == {2,3,5}` √
- `sort({}) == {}` √
- `sort({17}) == {17}` √

Typically missed test cases

- `sort({2,1,2}) == {1,2,2}` ⊠
- `sort(null) == exception` ⊠
- `isPermutation(sort(a),a)` ⊠

Complementing Testing: Formal Verification

A Sorting Program:

```
int* sort(int* a) {
  ...
}
```

Testing `sort`:

- `sort({3,2,5}) == {2,3,5}` √
- `sort({}) == {}` √
- `sort({17}) == {17}` √

Typically missed test cases

- `sort({2,1,2}) == {1,2,2}` ☒
- `sort(null) == exception` ☒
- `isPermutation(sort(a),a)` ☒

Formal Verification as Theorem Proving

Theorem (Correctness of `sort`)
For any given non-null int array `a`, calling the program `sort(a)` returns an int array that is sorted wrt ≤ *and is a permutation of* `a`.

However, methodology differs from mathematics:

1. Formalize the expected property in a logical language
2. Prove the property with the help of an (semi-)automated tool

Formal Verification as Theorem Proving

Theorem (Correctness of `sort`)
For any given non-null int array `a`, calling the program `sort(a)` returns an int array that is sorted wrt ≤ *and is a permutation of* `a`.

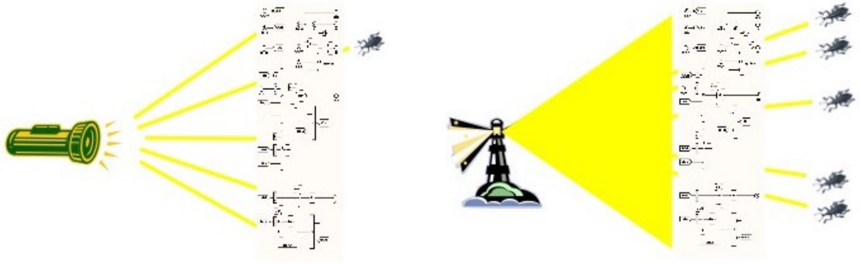
However, methodology differs from mathematics:

1. Formalize the expected property in a logical language
2. Prove the property with the help of an (semi-)automated tool

Contrasting Testing with Formal Verification

Testing Checks Only the Values We Select

Formal Verification Checks Every Possible Value!



Even Small Systems Have Trillions (of Trillions) of Possible Tests!

Finds every exception to the property being checked!

Formal Methods

Rigorous techniques and tools for the development and analysis of computational (hardware/software) systems

- Applied at various stages of the development cycle
- Also used in reverse engineering to model and analyze existing systems
- Based on mathematics and symbolic logic (formal)

Formal Methods

Rigorous techniques and tools for the development and analysis of computational (hardware/software) systems

- Applied at various stages of the development cycle
- Also used in reverse engineering to model and analyze existing systems
- Based on mathematics and symbolic logic (formal)

Formal Methods

Rigorous techniques and tools for the development and analysis of computational (hardware/software) systems

- Applied at various stages of the development cycle
- Also used in reverse engineering to model and analyze existing systems
- Based on mathematics and symbolic logic (formal)

Formal Methods

Rigorous techniques and tools for the development and analysis of computational (hardware/software) systems

- Applied at various stages of the development cycle
- Also used in reverse engineering to model and analyze existing systems
- Based on mathematics and symbolic logic (formal)

Main Artifacts in Formal Methods

1. System requirements
2. System implementation

Formal methods rely on

a. some formal specification of (1)
b. some formal execution model of (2)

They use tools to verify mechanically that implementation satisfies (a) according to (b)

Main Artifacts in Formal Methods

1. System requirements
2. System implementation

Formal methods rely on

a. some formal specification of (1)
b. some formal execution model of (2)

They use tools to verify mechanically
that implementation satisfies (a)
according to (b)

Main Artifacts in Formal Methods

1. System requirements
2. System implementation

Formal methods rely on

a. some formal specification of (1)
b. some formal execution model of (2)

They use tools to verify mechanically that implementation satisfies (a) according to (b)

Why Use Formal Methods

1. Contribute to the overall quality of the final product thanks to mathematical modeling and formal analysis

2. Increase confidence in the correctness/robustness/security of a system

3. Find more flaws and earlier (i.e., during specification and design vs. testing and maintenance)

Formal Methods: The Vision

- Complement other analysis and design methods
- Help find bugs in code and specification
- Reduce development, and testing, cost
- Ensure certain properties of the formal system model
- Should be highly automated

Formal Methods and Testing

- Run the system at chosen inputs and observe its behavior
  - Randomly chosen
  - Intelligently chosen (by hand: expensive!)
  - Automatically chosen (need formalized spec)
- What about other inputs? (test coverage)
- What about the observation? (test oracle)

Challenges can be addressed by/require formal methods

A Warning

- The notion of “formality” is often misunderstood (formal vs. rigorous)
- The effectiveness of FMs is still debated
- There are persistent myths about their practicality and cost
- FMs are not yet as widespread in industry as they could be
- They are mostly used in the development of safety-, business-, or mission-critical software, where the cost of faults is high

The Main Point of Formal Methods is Not

- To show "correctness" of entire systems
  - What is correctness? Go for specific properties!
- To replace testing entirely
  - FMs typically do not go below byte code level
  - Some properties are not (easily) formalizable
- To replace good design practices

There is no silver bullet!

No correct system w/o clear requirements & good design

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Overall Benefits of Using Formal Methods

1. Forces developers to think systematically about issues
2. Improves the quality of specifications, *even without formal verification*
3. Leads to better design
4. Provides a precise reference to check requirements against
5. Provides rigorous documentation within a team of developers
6. Gives direction to later development phases
7. Provides a basis for reuse via specification matching
8. Can replace (infinitely) many test cases
9. Facilitates automatic test case generation

Specifications: What the system should do

- Individual properties
  - Safety properties: something bad will never happen
  - Liveness properties: something good will happen eventually
  - Non-functional properties: runtime, memory, usability, . . .
- “Complete” behaviour specification
  - Equivalence check
  - Refinement
  - Data consistency
  - . . .

Formal Specification

The expression in some formal language and at some level of abstraction of a collection of properties that some system should satisfy [van Lamsweerde]

Formal Specification

The expression in some formal language and at some level of abstraction of a collection of properties that some system should satisfy [van Lamsweerde]

formal language:

- syntax can be mechanically processed and checked
- semantics is defined unambiguously by mathematical means

abstraction:

- above the level of source code
- several levels possible

properties:

- expressed in some formal logic
- have a well-defined semantics

satisfaction:

- ideally (but not always) decided mechanically
- based on automated deduction and/or model checking techniques

Formal Specification

The expression in some formal language and at some level of abstraction of a collection of properties that some system should satisfy [van Lamsweerde]

formal language:

- syntax can be mechanically processed and checked
- semantics is defined unambiguously by mathematical means

abstraction:

- above the level of source code
- several levels possible

properties:

- expressed in some formal logic
- have a well-defined semantics

satisfaction:

- ideally (but not always) decided mechanically
- based on automated deduction and/or model checking techniques

Formal Specification

The expression in some formal language and at some level of abstraction of a collection of properties that some system should satisfy [van Lamsweerde]

formal language:

- syntax can be mechanically processed and checked
- semantics is defined unambiguously by mathematical means

abstraction:

- above the level of source code
- several levels possible

properties:

- expressed in some formal logic
- have a well-defined semantics

satisfaction:

- ideally (but not always) decided mechanically
- based on automated deduction and/or model checking techniques

Formal Specification

The expression in some formal language and at some level of abstraction of a collection of properties that some system should satisfy [van Lamsweerde]

formal language:

- syntax can be mechanically processed and checked
- semantics is defined unambiguously by mathematical means

abstraction:

- above the level of source code
- several levels possible

properties:

- expressed in some formal logic
- have a well-defined semantics

satisfaction:

- ideally (but not always) decided mechanically
- based on automated deduction and/or model checking techniques

Formal Specification

The expression in some formal language and at some level of abstraction of a collection of properties that some system should satisfy [van Lamsweerde]

formal language:

- syntax can be mechanically processed and checked
- semantics is defined unambiguously by mathematical means

abstraction:

- above the level of source code
- several levels possible

properties:

- expressed in some formal logic
- have a well-defined semantics

satisfaction:

- ideally (but not always) decided mechanically
- based on automated deduction and/or model checking techniques

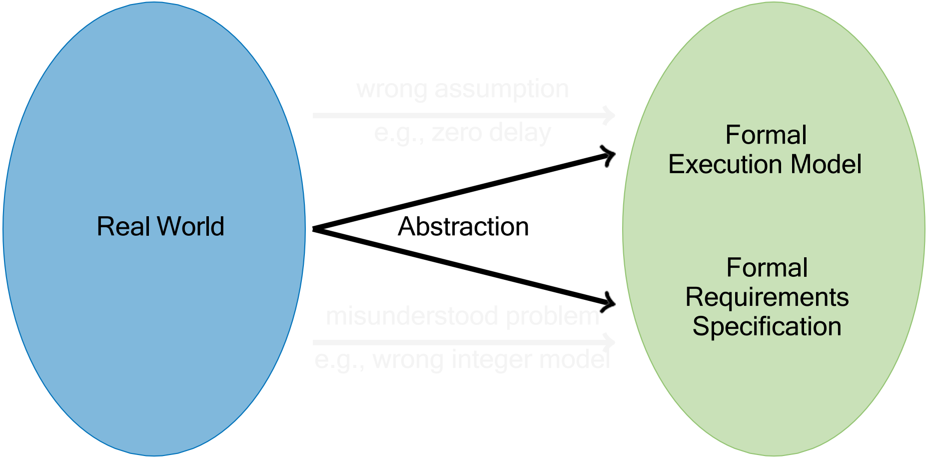
Formalization Helps to Find Bugs in Specs

- Well-formedness and consistency of formal specs are machine-checkable
- Fixed signature (set of symbols) helps spot incomplete specs
- Failed verification of implementation against specs provides feedback on errors
  - in the implementation or
  - in the (formalization of the) spec

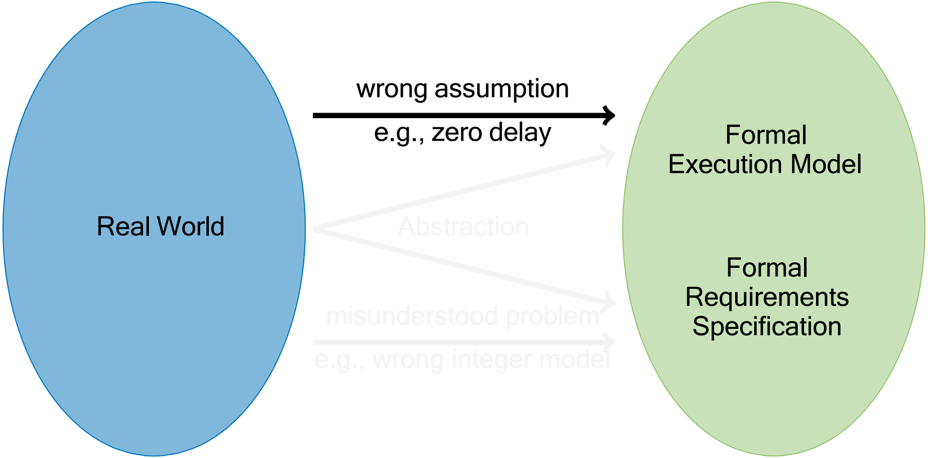
A Fundamental Fact

Formalizing system requirements is hard

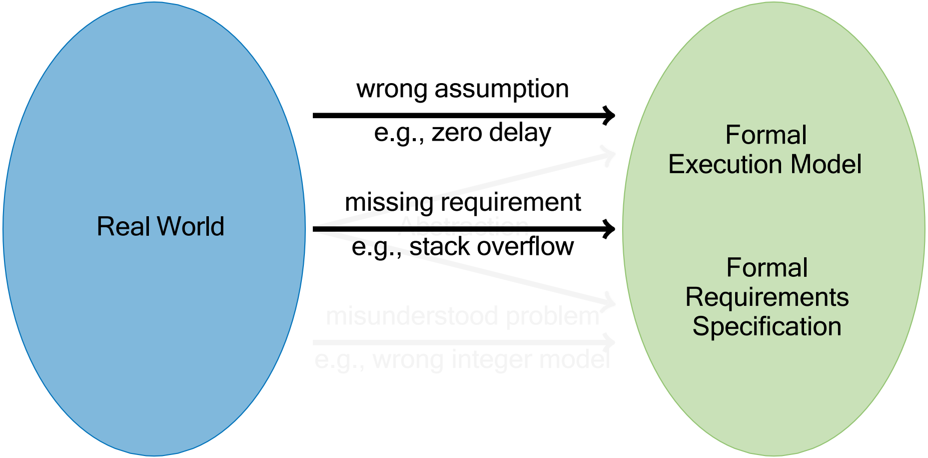
Difficulties in Creating Formal Models



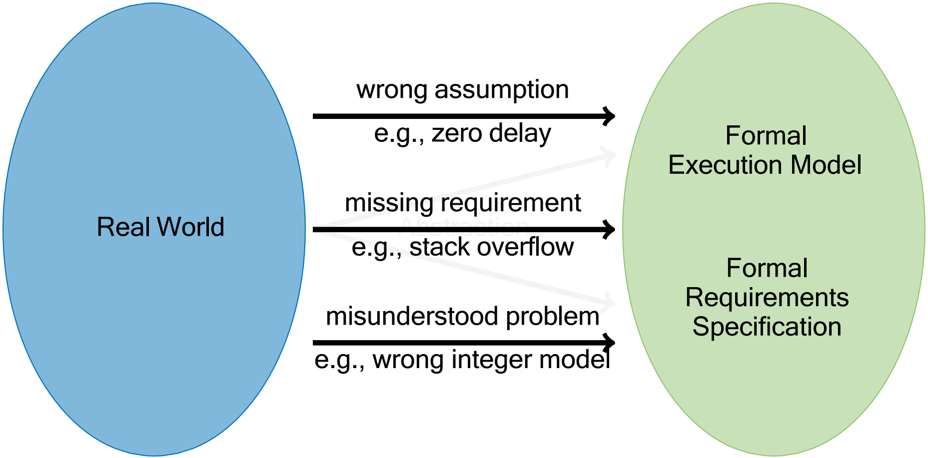
Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Level of System Description

High level (modeling/programming language level)

- Complex datatypes and control structures, general programs
- Easier to program
- Automatic proofs (in general) impossible!

:

Low level (machine level)

- Finitely many states
- Tedious to program, worse to maintain
- Automatic proofs are (in principle) possible

Expressiveness of Specification

High

- General properties
- High precision, tight modeling
- Automatic proofs (in general) impossible!

⋮

Low

- Finitely many cases
- Approximation, low precision
- Automatic proofs are (in principle) possible

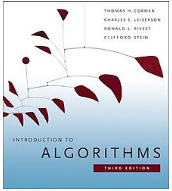
Another Fundamental Fact

Proving properties of systems can be hard

Formal Methods to the Extreme: Formal Verification

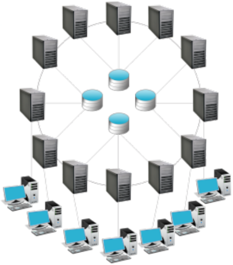
Correctness-critical software

- Implementations of textbook algorithms

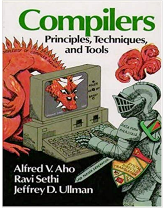


- Operating Systems

- Distributed systems and their applications

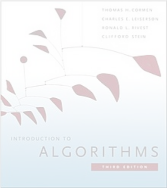


- Compilers



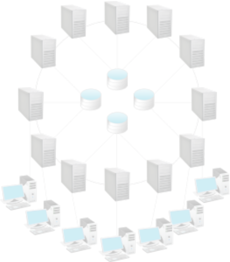
Correctness-critical software

- Implementations of textbook algorithms

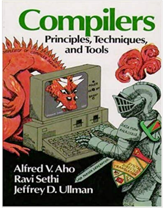


- Operating Systems

- Distributed systems and their applications



- Compilers



Specifying Compilers

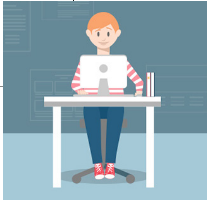
Program in C

```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

```
#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

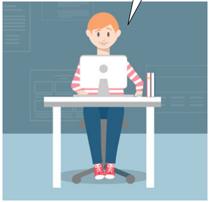
Program *compile*(P) in x86 Assembly

```
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) = $\mathbf{R_c}$ = $\mathbf{R_{x86}}$ = 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, *interpret*$_C$(P, input) = *execute*$_{x86}$(*compile*(P, input))

Proof: ???

Assumptions:

- Meaningful definition of *interpret*$_C$ is given and fixed
- Meaningful definition of *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:

∀ P, in, *interpret*$_C$(P, in) = *execute*$_{x86}$(*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*,

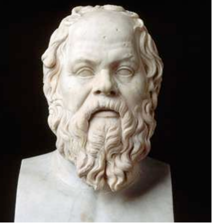
each of which is either *validly derived from those preceding* it
or is an *assumption*,

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.

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



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

reasonable assumptions

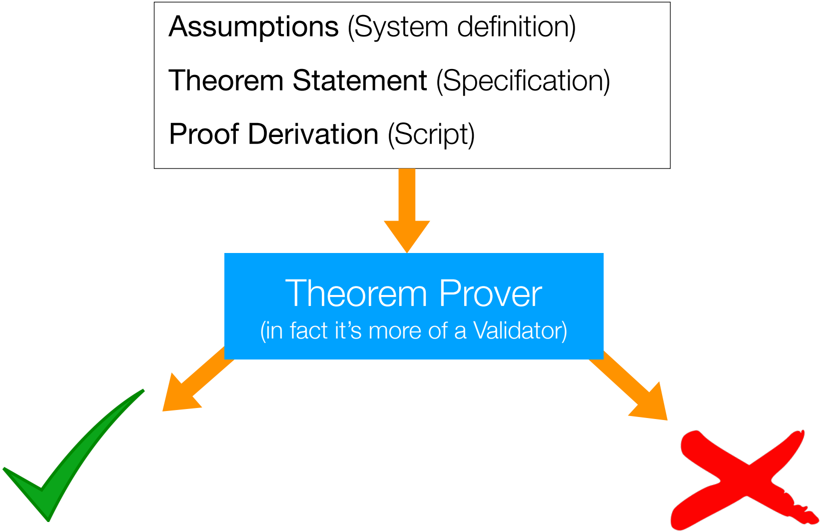
Socrates is a man

is a man ⇒ is mortal

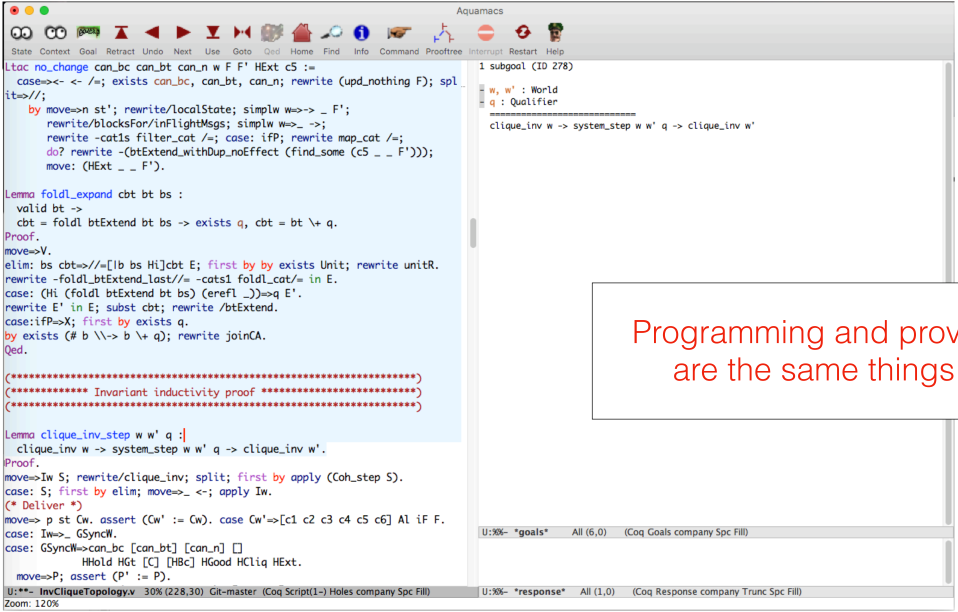
Socrates is mortal

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

Proofs don’t have to be trusted!



Modern Theorem Provers are Awesome



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

Observartions

- *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

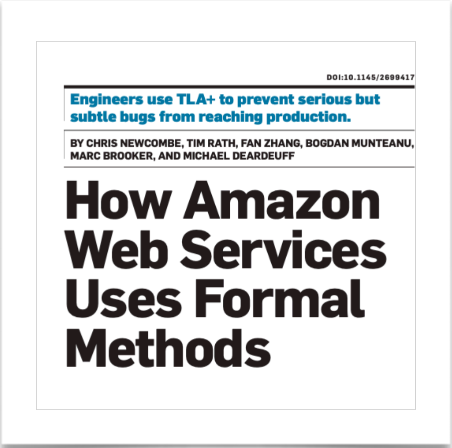
- These assumptions *might be broken* in the real world, thus invalidating the claims of theorems

- *Testing* helps to validate the assumptions.

Current and Future Trends

Slowly but surely formal methods are finding increased used in industry.

- Designing for formal verification
- Combining semi-automatic methods with SAT/SMT solvers, theorem provers
- Combining static analysis of programs with automatic methods and with theorem provers
- Combining testing and formal verification
- Integration of formal methods into development process


DOI:10.1145/2699417

Engineers use TLA+ to prevent serious but subtle bugs from reaching production.

BY CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, BOGDAN MUNTEANU, MARC BROOKER, AND MICHAEL DEARDEUFF

How Amazon Web Services Uses Formal Methods

Current and Future Trends

Need for secure systems is increasing the use of FMs

- Security is intrinsically hard
- Redundant fault-tolerant systems are often used to meet safety requirements
- Fault-tolerance depends on the independence of component failures
- Security attacks are intelligent, coordinated and malicious
- Formal methods provides a systematic way to meet stringent security requirements

Today's Summary

- Software is becoming pervasive and very complex
- Current development techniques are inadequate
- Formal methods …
  - are not a panacea, but will be increasingly necessary
  - are (more and more) used in practice
  - can shorten development time
  - can push the limits of feasible complexity
  - can increase product quality
  - can improve system security
- We will learn to use several different formal methods, for different development stages

Next week: formal methods in action!