Practical Formal Methods

Course Overview and Introduction

Ilya Sergey

November 2024

ilyasergey.net/PFM24

About myself

Undergrad	Saint Petersburg State University, 2008
PhD	KU Leuven, 2012
Currently	Associate Professor at NUS School of Computing (since 2018)
Previously	Assistant Professor at University College London
	Postdoc at IMDEA Software Institute
	Software Engineer at JetBrains (IntelliJ IDEA team: Scala, Groovy)
Research interests	software verification, PL design, concurrent & distributed algorithms

ilyasergey.net

Course Info and Material

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

  https://ilyasergey.net/PFM24/

- 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 Telegram

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

  https://github.com/formal-and-practical

Goals of the Course

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 modelling 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, unsound

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, unsound

Decidable Reasoning: reducing verification to known algorithmic problems

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

Abstract Interpretation: automatic, correct, incomplete, terminating

Practical tools we will learn

Part I: High-Level Design

Language: TLA+

- Lightweight modelling language for system design
- Amenable to a fully automatic analysis
- Aimed at expressing complex behaviour and properties of a software system
- Intuitive structural modelling tool based on Boolean functions
- Automatic analyser 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 modelling language for system design
- Amenable to a fully automatic analysis
- Aimed at expressing complex behaviour and properties of a software system
- Intuitive structural modelling tool based on Boolean functions
- Automatic analyser 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 in 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+: 20%
- Homework 2: SMT: 20%
- Homework 3: Dafny: 20%

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

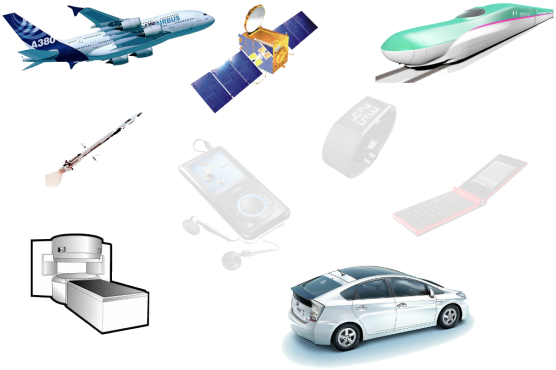
Some of it is critical

Failing software costs money and life!

Embedded Software

Software is now embedded everywhere

Some of it is critical

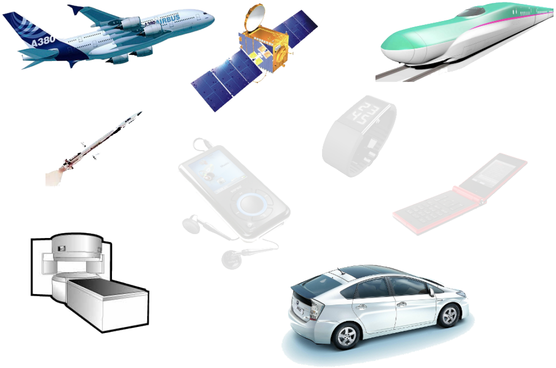


Failing software costs money and life!

Embedded Software

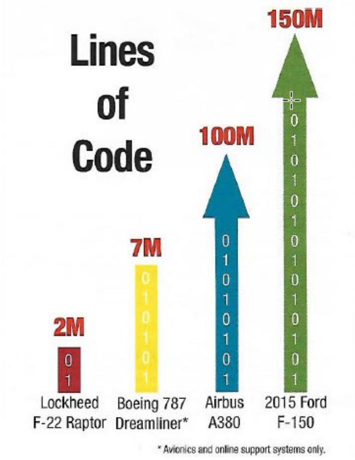
Software is now embedded everywhere

Some of it is critical

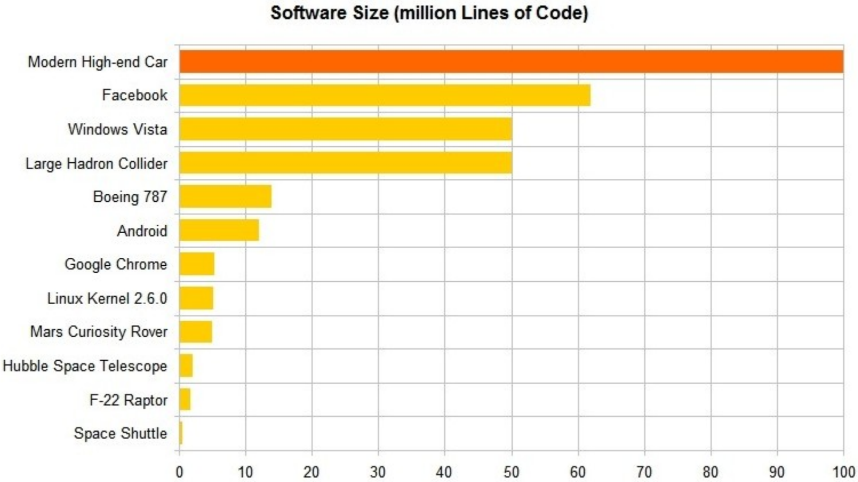


Failing software costs money and life!

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, CrowdStrike outage 2024)

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

- Meltdown, Spectre,
- (too many others 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?

Full-system 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?

Full-system 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. Formalise 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)

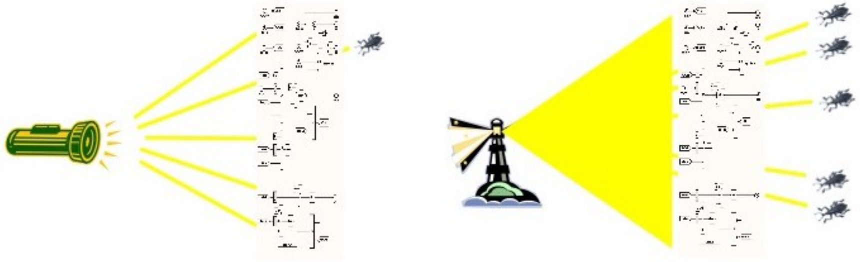
For **any** given non-null int array a, calling the program sort(a) returns an int array that is sorted wrt ≤ *and is a permutation of* a.

However, methodology differs from mathematics:

1. Formalise 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

A suite of methods and techniques
for producing provably correct programs
by employing a mix of
algorithmic and deductive logical reasoning.

- A formal *specification* capturing the *intended behaviour* of the program is assumed to be provided by a human developer.
- The program is then *checked* against the formal specification, and if it is *proved to satisfy the ascribed specification*, it is deemed "*correct*".

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 analyse 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 analyse 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 analyse existing systems
- Based on mathematics and symbolic logic (formal)

Main Artefacts 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 Artefacts 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 Artefacts 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)

Example:

Specifying a Compiler

Specifying a Compiler

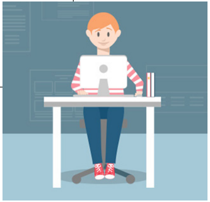
Program in C

```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 Arm 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
```



67

Program P 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 →

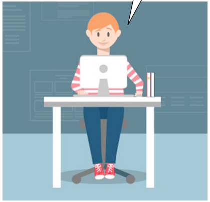
Program *compile*(P) in Arm Assembly

```
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) = R_c = R_{arm} = Result(*compile*(P), input)



68

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 **Arm Assembly**.

or, equivalently

Correctness Theorem:

$$\forall \; P, input, \; \mathit{interpret}_C(P, input) = \mathit{execute}_{arm}(\mathit{compile}(P, input))$$

Correctness Theorem:

$\forall$ P, input, $\textit{interpret}_{C}$(P, input) = $\textit{execute}_{arm}$(*compile*(P, input))

Proof: ???

Assumptions:

- Meaningful definition of *interpret*$_C$ is given and fixed
- Meaningful definition of *execute*$_{arm}$ 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*$_{arm}$(*compile*(P, in))

Proof: ???

once proven,
does not have
to be trusted

71

Why Use Formal Methods

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

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

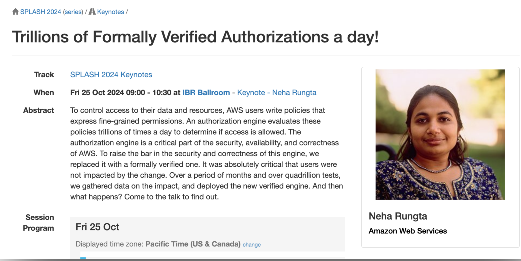
3. Find more flaws and earlier
   (i.e., during specification and design vs. testing and maintenance)

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 (perhaps with AI in the future)

A Warning

- 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


SPLASH 2024 (series) / Keynotes /

Trillions of Formally Verified Authorizations a day!

Track SPLASH 2024 Keynotes

When Fri 25 Oct 2024 09:00 - 10:30 at IBR Ballroom - Keynote - Neha Rungta

Abstract To control access to their data and resources, AWS users write policies that express fine-grained permissions. An authorization engine evaluates these policies trillions of times a day to determine if access is allowed. The authorization engine is a critical part of the security, availability, and correctness of AWS. To raise the bar in the security and correctness of this engine, we replaced it with a formally verified one. It was absolutely critical that users were not impacted by the change. Over a period of months and over quadrillion tests, we gathered data on the impact, and deployed the new verified engine. And then what happens? Come to the talk to find out.

Session Program Fri 25 Oct

Displayed time zone: **Pacific Time (US & Canada)** change

Neha Rungta
Amazon Web Services

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) formalisable
- 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*

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

[Axel 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

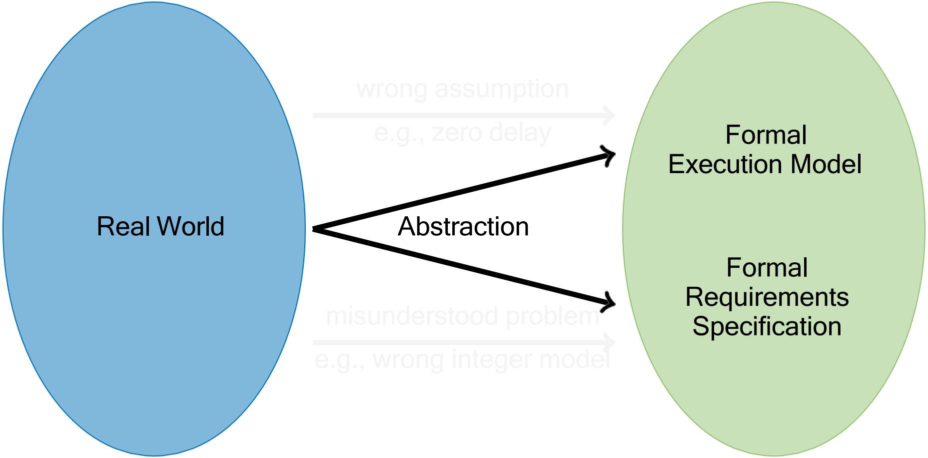
Formalization Helps to Find Bugs in Specs!

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

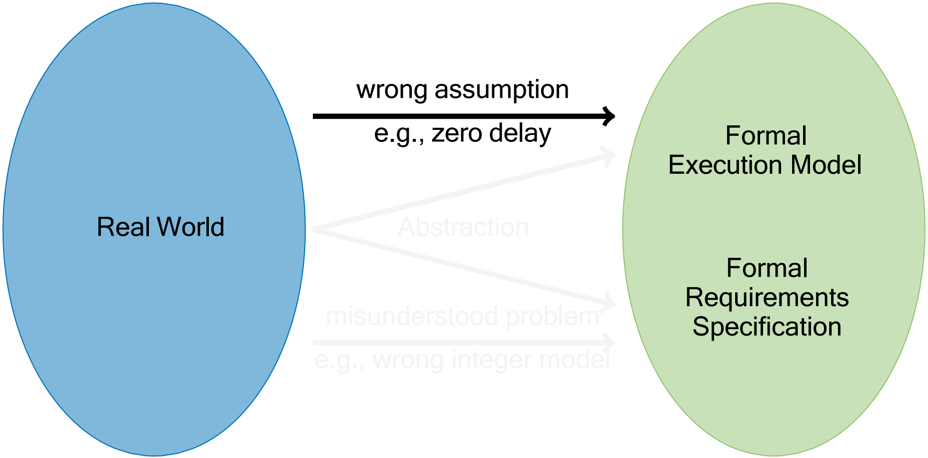
A Fundamental Fact

Formalizing system requirements is hard

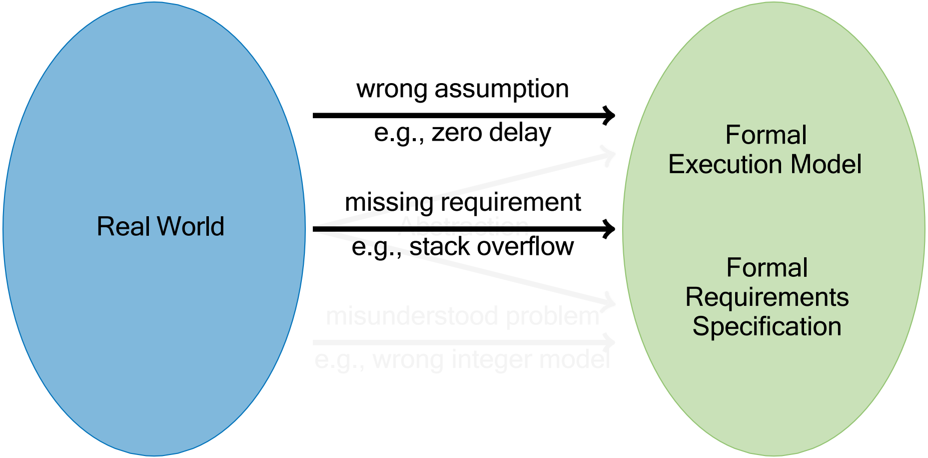
Difficulties in Creating Formal Models



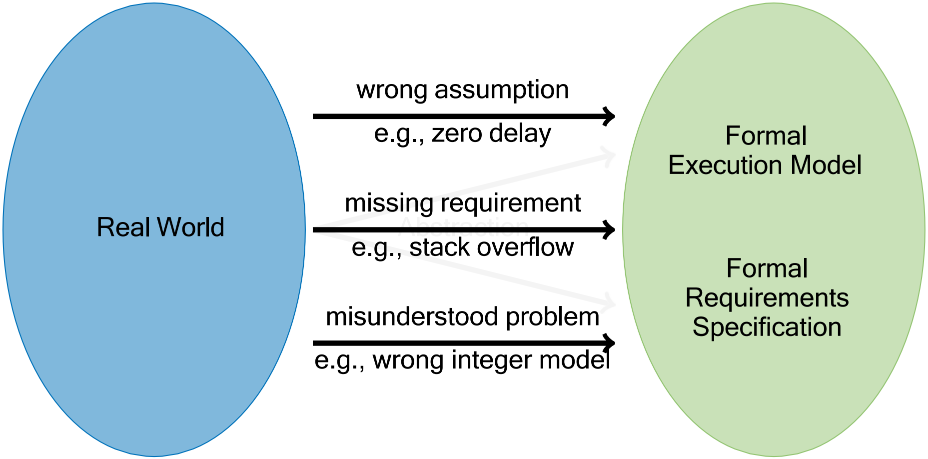
Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Another Fundamental Fact

Proving properties of systems can be hard

Level of System Description

High level (modelling)

- Abstract clean semantics
- Easier to program
- Automatic proofs (sometimes) are possible

⋮

Low level (implementation level)

- Realistic programming language
- Often can be directly executed
- Automatic proofs are (mostly) impossible

Alloy

TLA+

Dafny

Verified Software Toolchain

Summary So Far

- 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: formal methods in action!