The University of Iowa

Fall 2015

CS:5810

Formal Methods in Software Engineering

Introduction

A TRUISM

Software has become critical to modern life

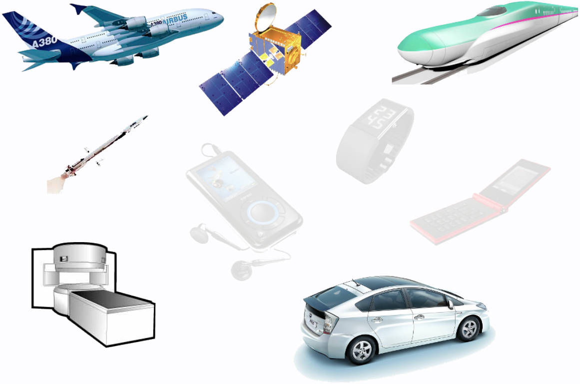
- Process Control (oil, gas, water, . . .)
- Transportation (air traffic control, . . .)
- Health Care (patient monitoring, device control . . .)
- Finance (automatic trading, bank security . . .)
- Defense (intelligence, weapons control, . . .)
- Manufacturing (precision milling, assembly, . . .)

EMBEDDED SOFTWARE

Software systems are embedded everywhere

Embedded Software

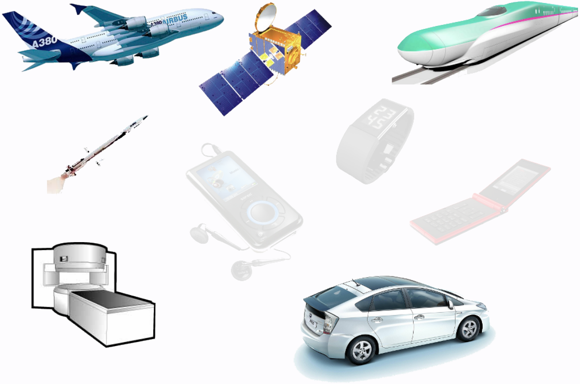
Software systems are embedded everywhere
Some of them are critical



Embedded Software

Software systems are embedded everywhere

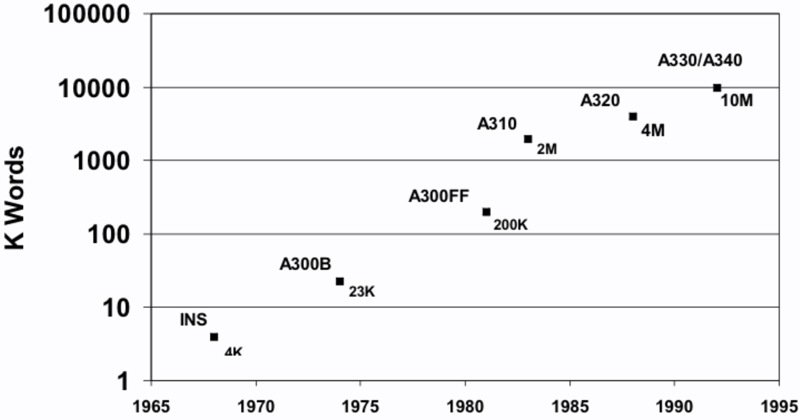
Some of them are critical



Failing software costs money and life!

Software Systems are Growing Very Large

Millions of LOCs in aircraft software



Software Systems are Growing Very Large

Car software:

- The GM Volt contains +10M lines of code: how do you verify that?
- Current cars admit hundreds of onboard functions: how do you cover their combination?

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

FAILING SOFTWARE COSTS MONEY

- Thousands of dollars for each minute of factory down-time
- 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)

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 radiation machine failures

The Peculiarity of Software Systems

Tiny faults can have catastrophic consequences

Software seems particularly prone to faults:

- Ariane 5
- Mars Climate Orbiter, Mars Sojourner
- London Ambulance Dispatch System
- Denver Airport Luggage Handling System
- Pentium-Bug
- (more at `http://www5.in.tum.de/~huckle/bugse.html`)

Rare bugs can happen

- Lifetime of a civil aircraft $\equiv$ 30 years
- Lifetime of a car $<$ 10 years but . . . 1 billions cars in 2010

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 software?

Achieving Reliability in Engineering

Some well-known strategies from civil 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 bugs
  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 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”)

- Design test configurations that hopefully are representative and
- ensure that the system behaves as intended on them

Testing against external faults

- Inject faults (memory, communication) by simulation or radiation

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) {
  ...
}
```

Complementing Testing: Formal Verification

A Sorting Program:

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

Testing `sort`:

- sort({3, 2, 5}) == {2, 3, 5} $\surd$
- sort({}) == {} $\surd$
- sort({17}) == {17} $\surd$

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 $\leq$ *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

What are Formal Methods?

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

What are Formal Methods?

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

- Applied at various stages of the development cycle

What are 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

WHAT ARE 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

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)

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

Why Use Formal Methods

- Mathematical modeling and analysis contribute to the overall quality of the final product
- Increase confidence in the correctness/robustness/security of a system
- 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 formal methods is still debated
- There as still persistent myths about their practicality and cost
- Formal methods are not yet widespread in industry
- 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
  - Formal methods do not go below byte code level
  - Some properties are not 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

- Forces developers to think systematically about issues
- Improves the quality of specifications, even without formal verification
- Leads to better design
- Provides a precise reference to check requirements against
- Provides documentation within a team of developers
- Gives direction to latter development phases
- Provides a basis for reuse via specification matching
- Can replace (infinitely) many test cases
- Facilitates automatic test case generation

SPECIFICATIONS: WHAT THE SYSTEM SHOULD DO

- Simple 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 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

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]

- 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

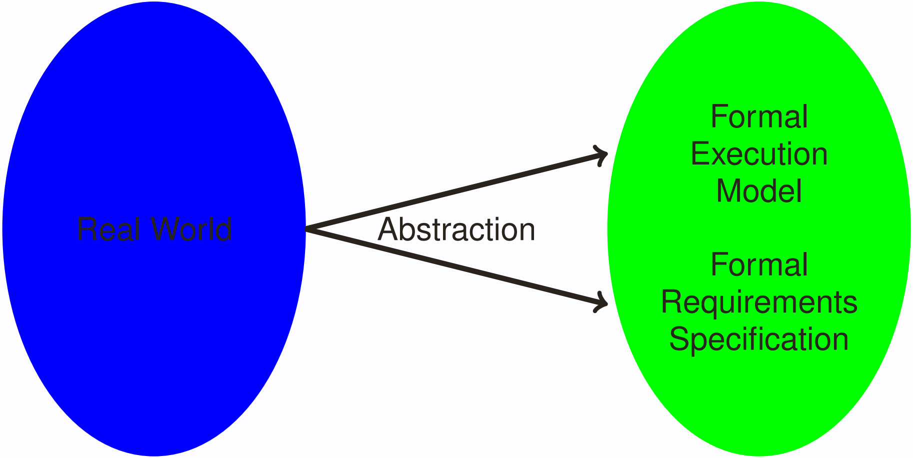
Formalization Helps to Find Bugs in Specs

- Wellformedness and consistency of formal specs checkable with tools
- Fixed signature (symbols) helps spot incomplete specs
- Failed verification of implementation against spec gives feedback on erroneous formalization

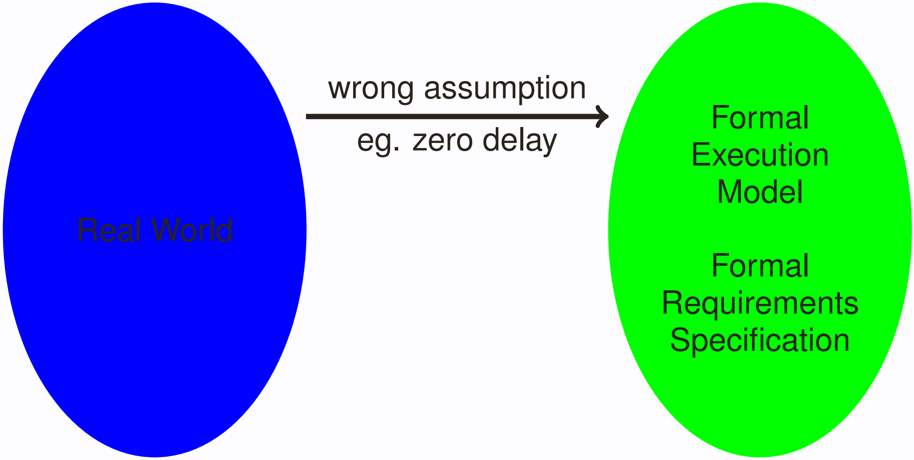
A FUNDAMENTAL FACT

Formalisation of system requirements is hard

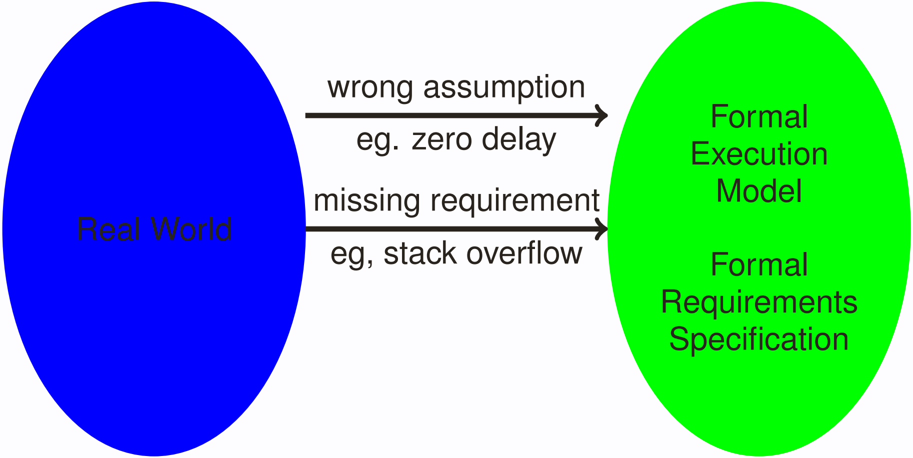
Difficulties in Creating Formal Models



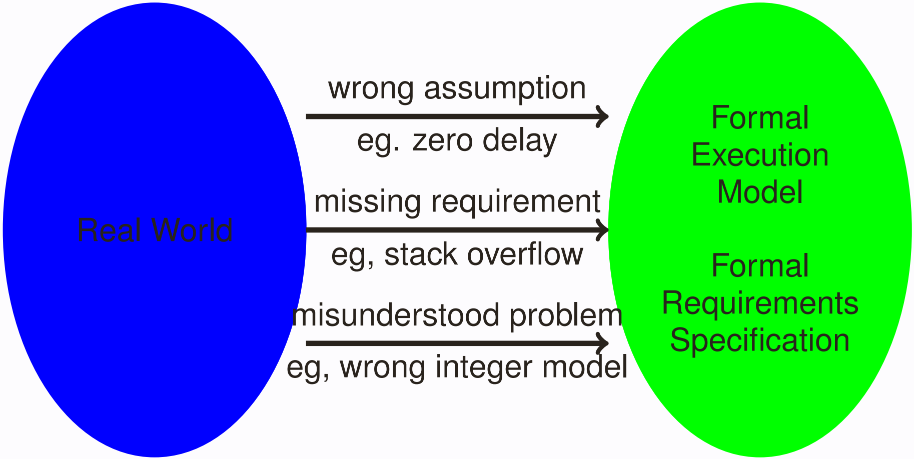
Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



ANOTHER FUNDAMENTAL FACT

Proving properties of systems can be hard

Level of System Description

Low level (machine level)

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

⋮

High level (programming language level)

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

Expressiveness of Specification

Simple

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

⋮

Complex

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

Current and Future Trends

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

- Design for formal verification
- Combining semi-automatic methods with SAT, theorem provers
- Combining static analysis of programs
  with automatic methods and with theorem provers
- Combining test and formal verification
- Integration of formal methods into SW development process

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
- We will learn to use several different formal methods, for different development stages