

The University of Iowa

Fall 2014

Formal Methods in Software Engineering

Introduction

Copyright 2013, Cesare Tinelli, Pierre-Loïc Garoche, Reiner Hähnle

These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders

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

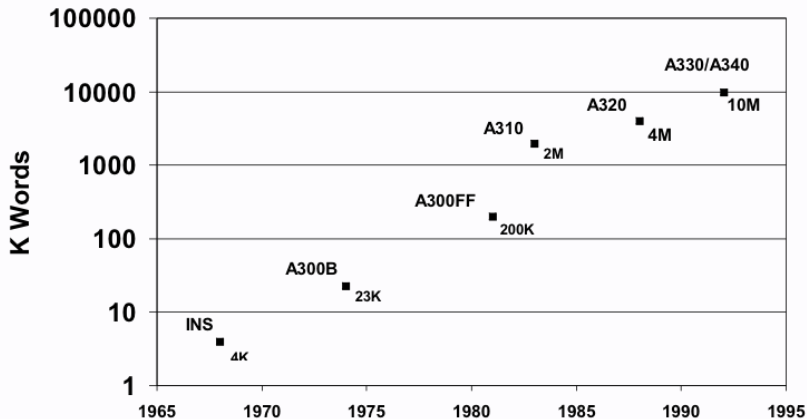
Software systems are embedded everywhere
Some of them are **critical**



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}` ✓
- `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}` ✓

Missed Test Cases!

- `sort({2,1,2}) == {1,2,2}` ☒
- `sort(NULL) == exception` ☒
-

FORMAL VERIFICATION AS THEOREM PROVING

Theorem. The program `sort()` is correct:

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

However, methodology differs from Mathematics:

1. **Formalize** the claim in a logical representation
2. Prove the claim with the help of an **automated reasoner**

FORMAL METHODS

- Rigorous design and development methods for computational (hardware/software) systems

FORMAL METHODS

- Rigorous design and development methods for computational (hardware/software) systems
- Based on mathematics and symbolic logic (**formal**)

FORMAL METHODS

- Rigorous design and development methods for computational (hardware/software) systems
- Based on mathematics and symbolic logic (**formal**)
- Increase confidence in the correctness/robustness/security of a system

FORMAL METHODS

- Rigorous design and development methods for computational (hardware/software) systems
- Based on mathematics and symbolic logic (**formal**)
- Increase confidence in the correctness/robustness/security of a system
- Consider two main artifacts:
 1. system **requirements**
 2. system **implementation**

FORMAL METHODS

- Rigorous design and development methods for computational (hardware/software) systems
- Based on mathematics and symbolic logic (**formal**)
- Increase confidence in the correctness/robustness/security of a system
- Consider two main artifacts:
 1. system **requirements**
 2. system **implementation**
- Are based on
 - a. some **formal specification** of (1)
 - b. some **formal execution model** of (2)

FORMAL METHODS

- Rigorous design and development methods for computational (hardware/software) systems
- Based on mathematics and symbolic logic (**formal**)
- Increase confidence in the correctness/robustness/security of a system
- Consider two main artifacts:
 1. system **requirements**
 2. system **implementation**
- Are based on
 - a. some **formal specification** of (1)
 - b. some **formal execution model** of (2)
- Use tools to verify **mechanically** that (b) satisfies (a)

FORMAL METHODS: THE VISION

- Complement other analysis and design methods
- Are good at finding bugs (in code **and** specification)
- Reduce development (and testing) time
- Can ensure certain **properties** of the formal system **model**
- Should ideally be automatic

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

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 SPECIFICATIONS

*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 SPECIFICATIONS

*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

A WARNING

- The notion of “formality” is often misunderstood (formal vs. rigorous)
- The effectiveness of formal methods is still debated
- There are still persistent myths about their practicality and cost
- Formal methods are not yet widespread in industry

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

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 and earlier detection of inconsistencies and flaws
- Provides a precise reference to check requirements against
- Provides documentation within a team of developers
- Gives direction to latter development phases (leading to coding)
- Provides a basis for reuse via specification matching
- Can replace (infinitely) many test cases
- Facilitates automatic test case generation

SUCCESSFUL FORMAL METHODS ...

- are integrated into the development process, in particular at early design stages
- avoid unreasonable new demands or skills from the user (FMs should be learnable as part of Masters in CS)
- work at large scale
- save time or money in getting a good quality product out
- increase the feasible complexity of products

TYPICAL AREAS

- **Saving time**
Time to market
- **Saving money**
Intel Pentium bug
Smart cards in banking
- **More complex products**
Modern processors, fault tolerant software
- **Saving human lives**
Avionics, X-by-wire

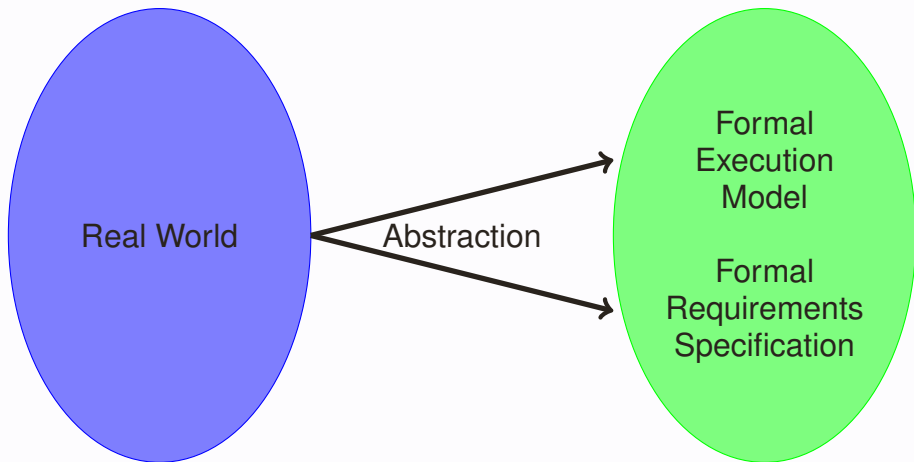
FORMALIZATION HELPS TO FIND BUGS IN SPECS

- 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

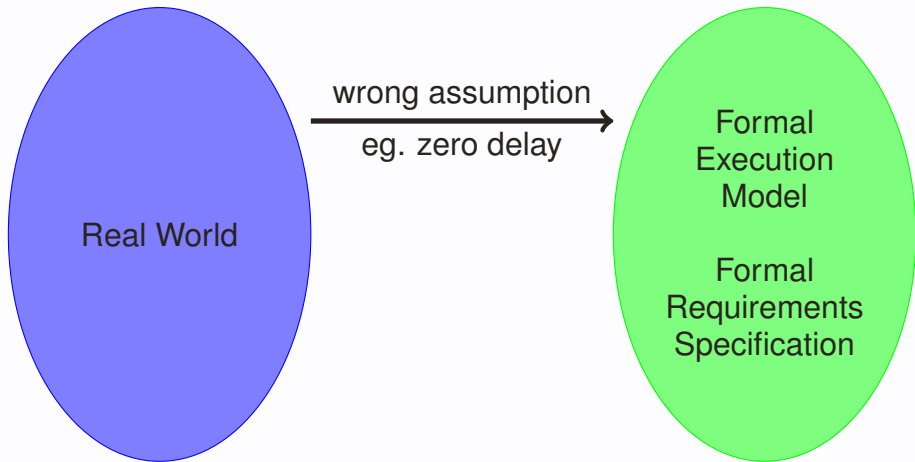
A FUNDAMENTAL FACT

Formalisation of system requirements is hard

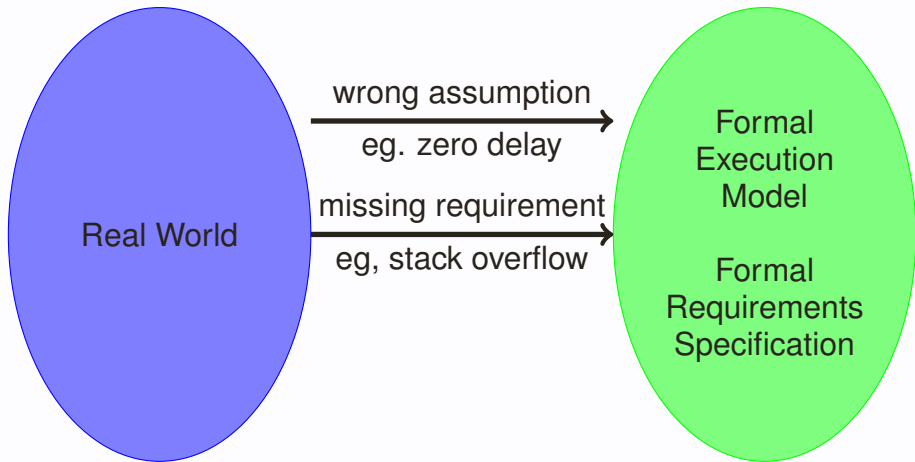
DIFFICULTIES IN CREATING FORMAL MODELS



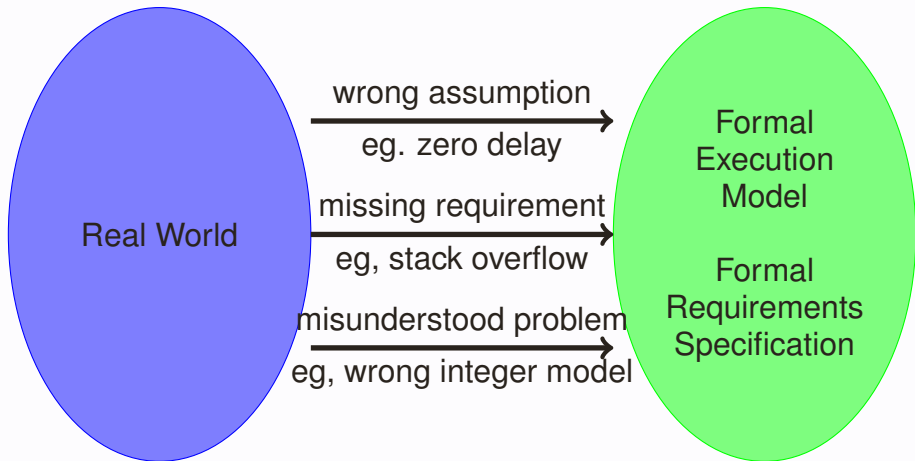
DIFFICULTIES IN CREATING FORMAL MODELS



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 use 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