22c181:
Formal Methods in Software Engineering

The University of Iowa

Spring 2008

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, . . .)**

Failing software costs money and life!

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., Arianne 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**
- **. . .**

Observation

Building software is what the majority of you will do after graduation

- **You'll be developing systems in the context we just mentioned**
- **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 in immature state

How to Ensure Software Correctness/Compliance?

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 intentionally 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 subjective How to test for the unexpected? Rare cases?
- Testing is labor intensive, hence expensive

A Complement to Testing: Formal Verification

A Sorting Program:

```
public static Integer[] sort(Integer[] a) {
  ...
}
```

A Complement to Testing: Formal Verification

A Sorting Program:

```
public static Integer[] sort(Integer[] a) {
  ...}
```

Testing `sort()`:

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

A Complement to Testing: Formal Verification

A Sorting Program:

```
public static Integer[] sort(Integer[] a) {
  ...}
```

Testing `sort()`:

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

Missed Test Cases!

- $\mathtt{sort}(\{2, 1, 2\}) == \{1, 2, 2\}$ ☒
- $\mathtt{sort}(\mathrm{NULL}) == \{1, 2, 2\}$ ☒

Formal Verification as Theorem Proving

Theorem. The program $\mathtt{sort}()$ is correct.

For any given array of integers $\mathtt{a}$, calling the program $\mathtt{sort}(\mathtt{a})$ returns an array of integers that is sorted and is a permutation of $\mathtt{a}$.

Proof.

Methodology differs from Mathematics!

1. Formalize the claim in a logical representation
2. Prove the claim with the help of a theorem prover

Formal Methods: The Scenario

- Rigorous methods used in system design and development
- Mathematics and symbolic logic $\Rightarrow$ formal
- Increase confidence in a system
- Two aspects:
  - System specification
  - System implementation
- Make formal model of both and use tools to prove mechanically that formal execution model of the implementation satisfies formal requirements of the specification

Formal Methods: The Vision

- **Complement other analysis and design methods**
- **Are good at finding bugs**

  (in code and specification)
- **Reduce development (and test) 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 behaviour
  - 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
- 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 usually) decided mechanically**

The Main Point of Formal Methods is Not

- **To show "correctness" of entire systems**

  What IS correctness? Always go for specific properties!

- **To replace testing entirely**

  Formal methods work on source code or, at most, bytecode level

  Non-formalizable properties

- **To replace good design practices**

There is no silver bullet!

No correct system w/o clear requirements & good design

This holds as well for Formal Methods

But . . .

- Formal proof can replace (infinitely) many test cases
- Formal methods can be used in automatic test case generation
- Formal methods improve the quality of specs

  (even without formal verification)

Successful Formal Methods

- . . . are integrated into the development process,

  in particular at early design stages

- . . . avoid unreasonable new demands or skills from the user

  FM 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

Typical Areas

- **Saving time**

  Time to market

- **Saving money**

  Intel Pentium bug

  Smart cards in banking

Typical Areas

- **Saving time**

  Time to market

- **Saving money**

  Intel Pentium bug

  Smart cards in banking

- **More complex products**

  Modern processors, fault tolerant software

Typical Areas

- Saving time

  Time to market

- Saving money

  Intel Pentium bug

  Smart cards in banking

- More complex products

  Modern processors, fault tolerant software

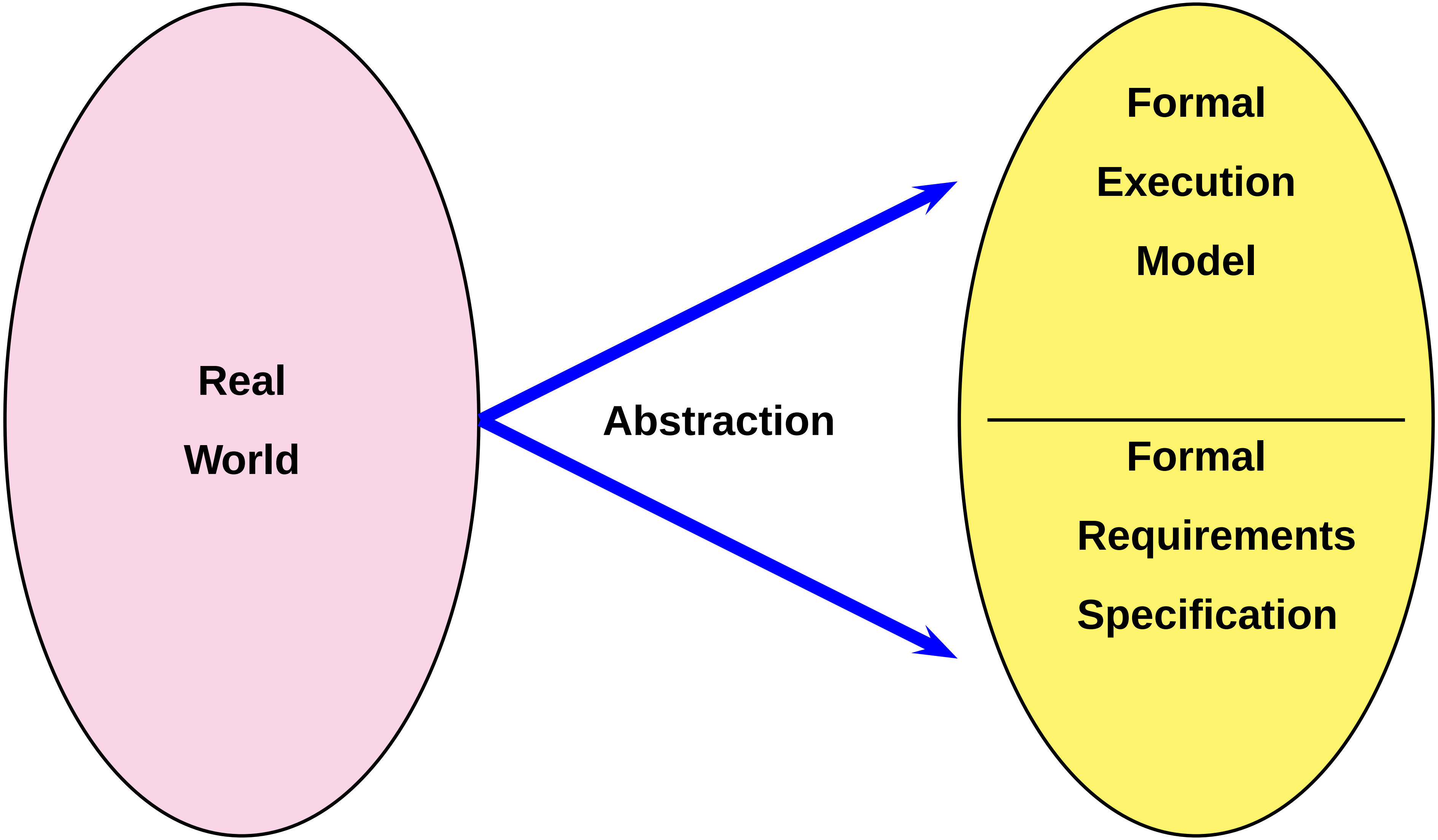
- Saving human lives

  Avionics, X-by-wire

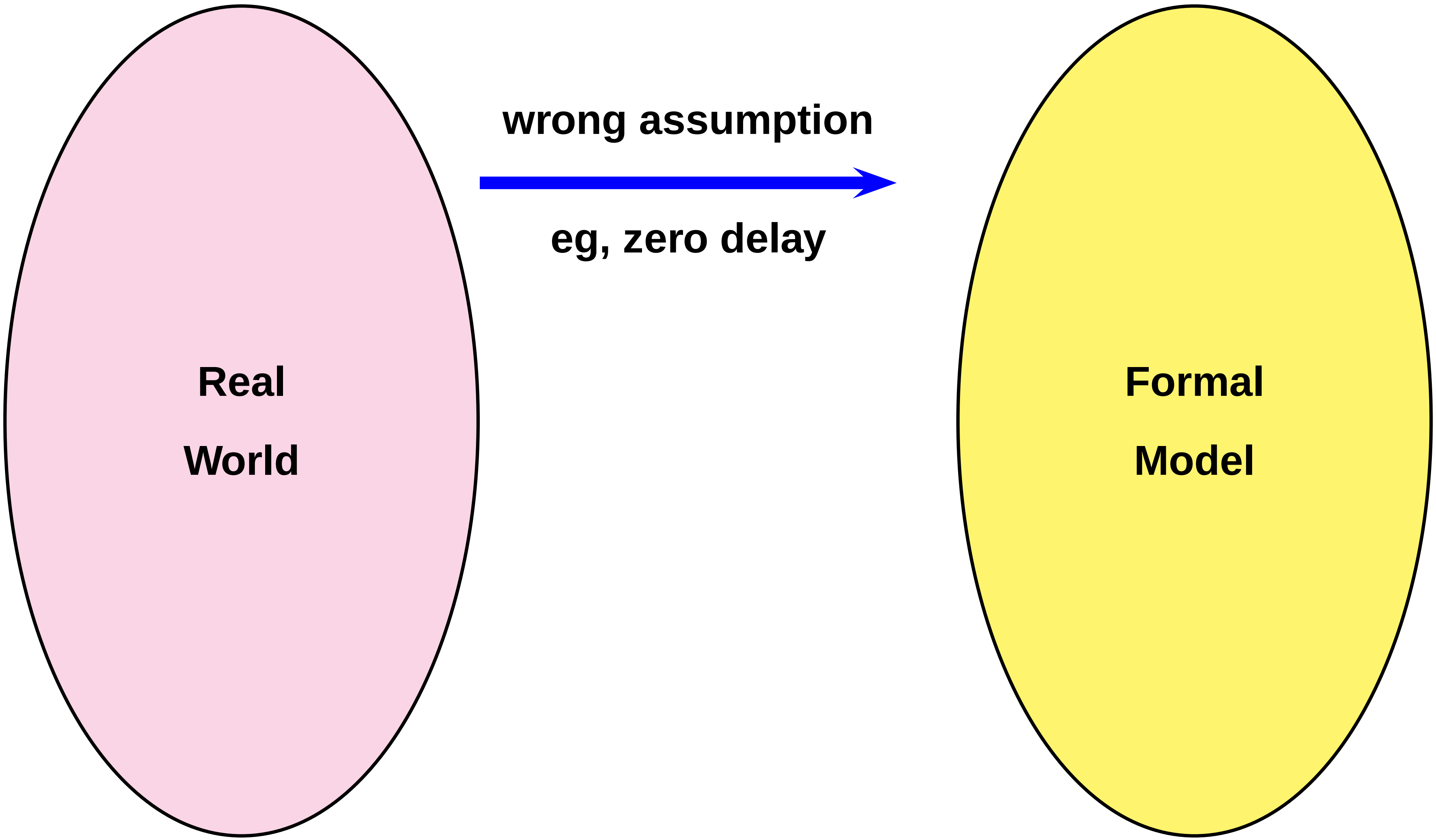
A Fundamental Fact

Formalisation of system requirements is hard

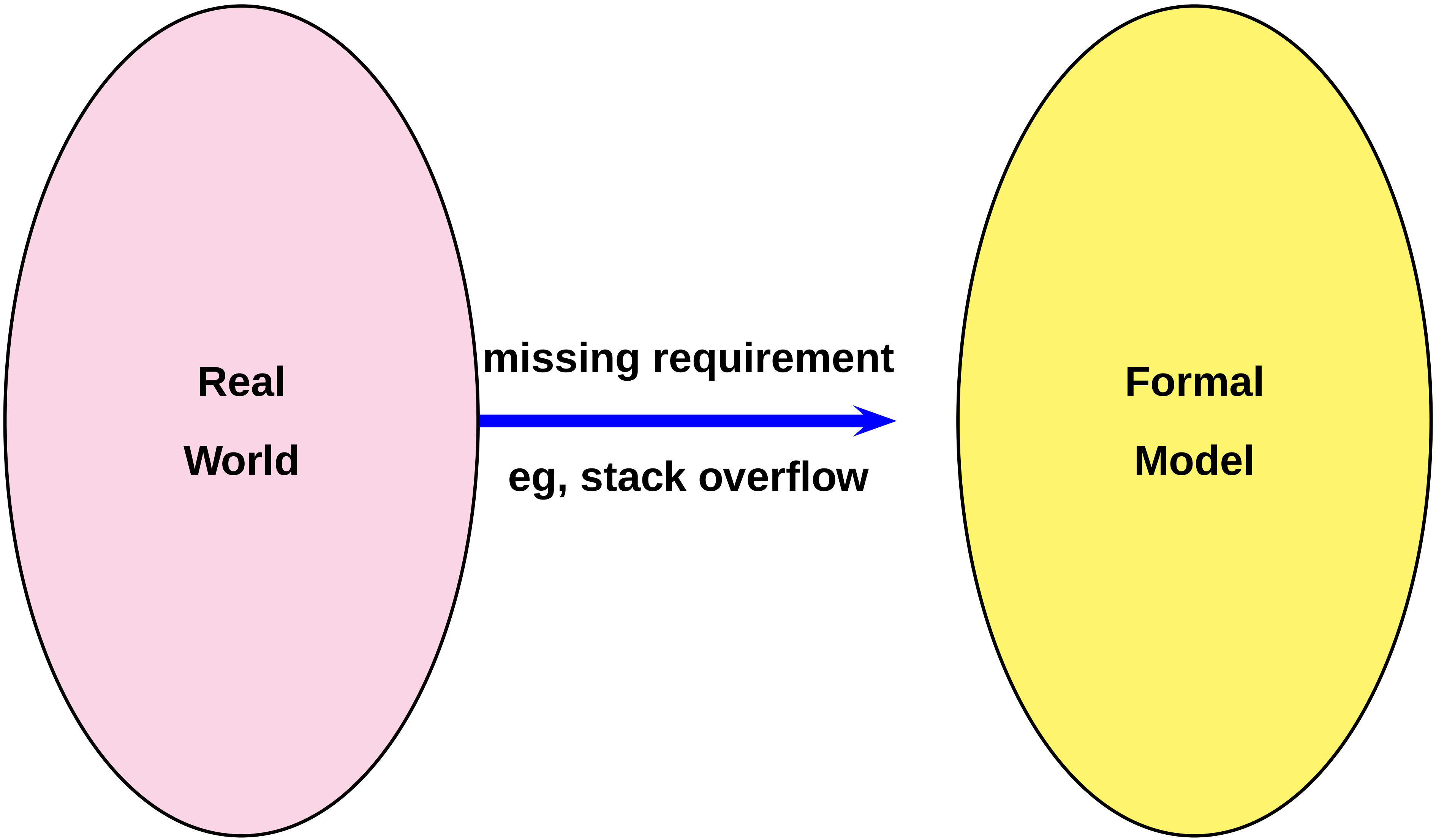
Difficulties in Creating Formal Models



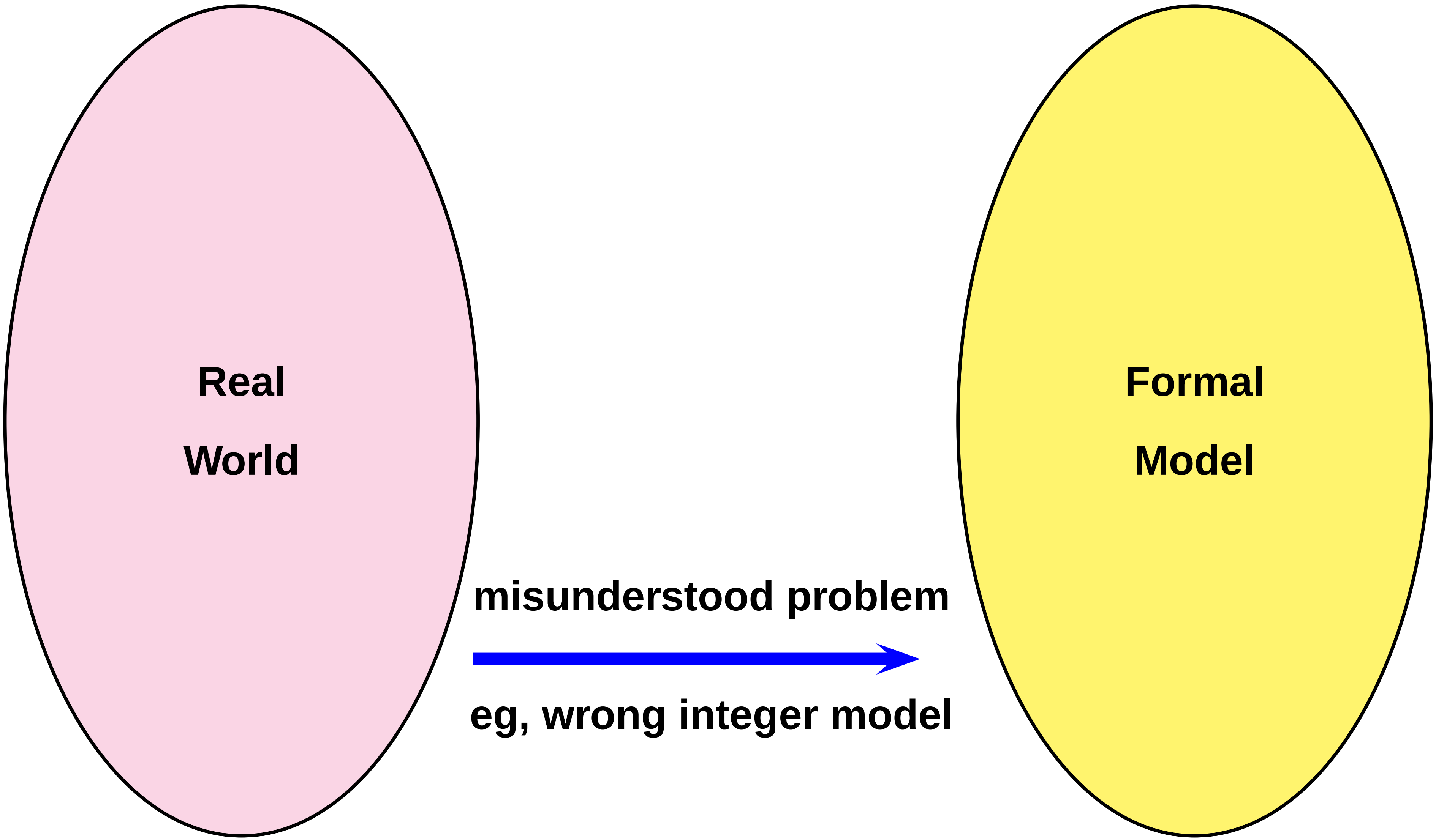
Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Formalization Helps to Find Bugs in Specs

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

Another Fundamental Fact

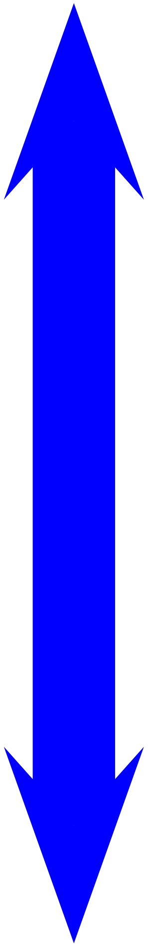
Proving properties of systems can be hard

Level of System (Implementation) Description

- **Low level**
  - **Finitely many states**
  - **Tedious to program, worse to maintain**
  - **Automatic proofs are (in principle) possible**
- **High 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!**



Main Approaches

High-level programs, Complex properties	High-level programs, Simple properties
Low-level programs, Complex properties	Low-level programs, Simple properties

Main Approaches

High-level programs, Complex properties	High-level programs, Simple properties
Low-level programs, Complex properties	Low-level programs, Simple properties

Lustre

1^{st} part of course

Main Approaches

KeY

2^{nd} **part of course**

High-level programs, **Complex properties**	**High-level programs,** **Simple properties**
Low-level programs, **Complex properties**	**Low-level programs,** **Simple properties**

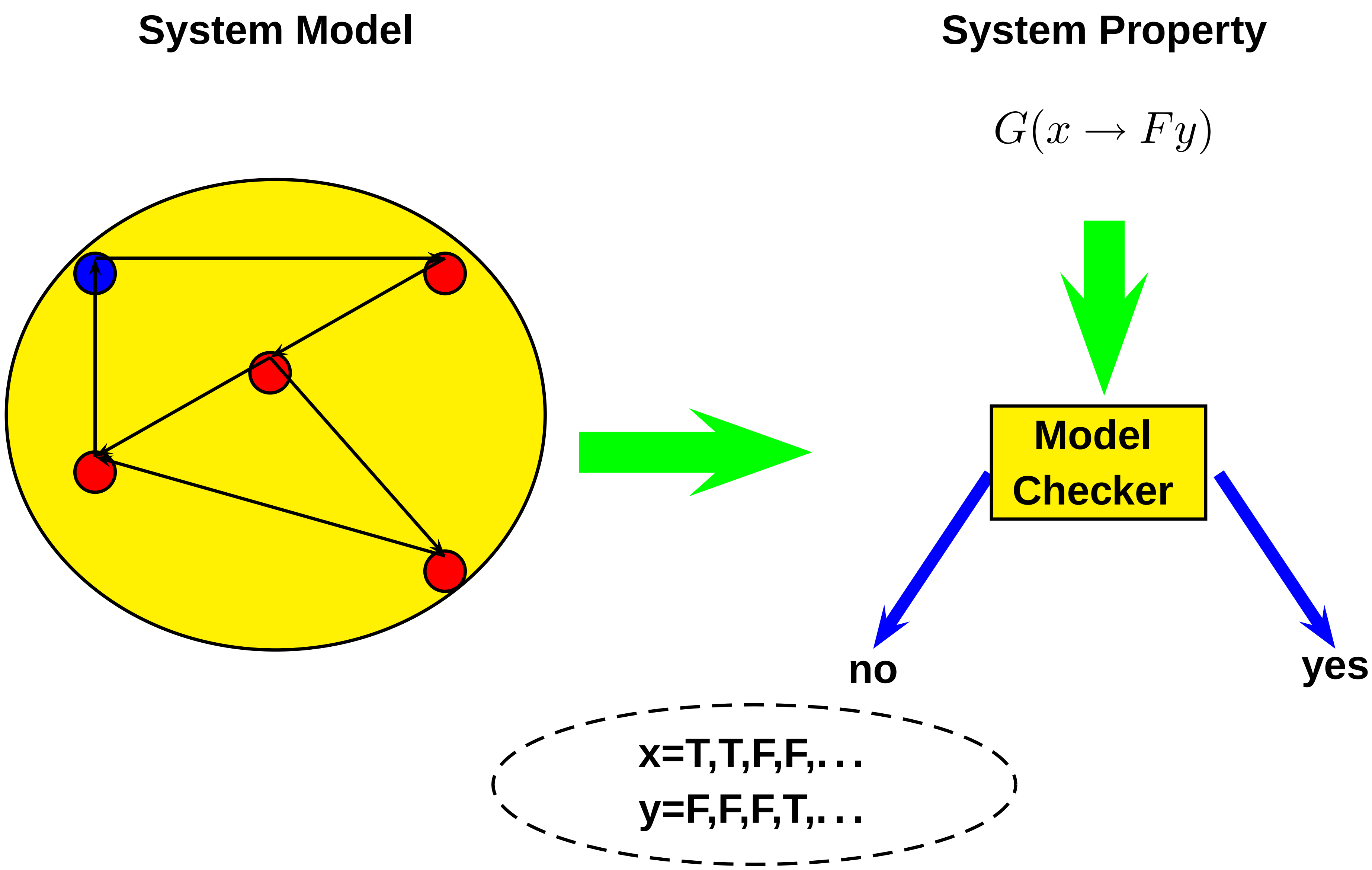
Lustre

1^{st} **part of course**

Proof Automation

- “Automatic” Proof
  - No interaction
  - Sometimes help is required anyway
  - Formal specification still “by hand”
- “Semi-Automatic” Proof
  - Interaction may be required
  - Very often proof tool suggests proof rules
  - Proof is checked by tool

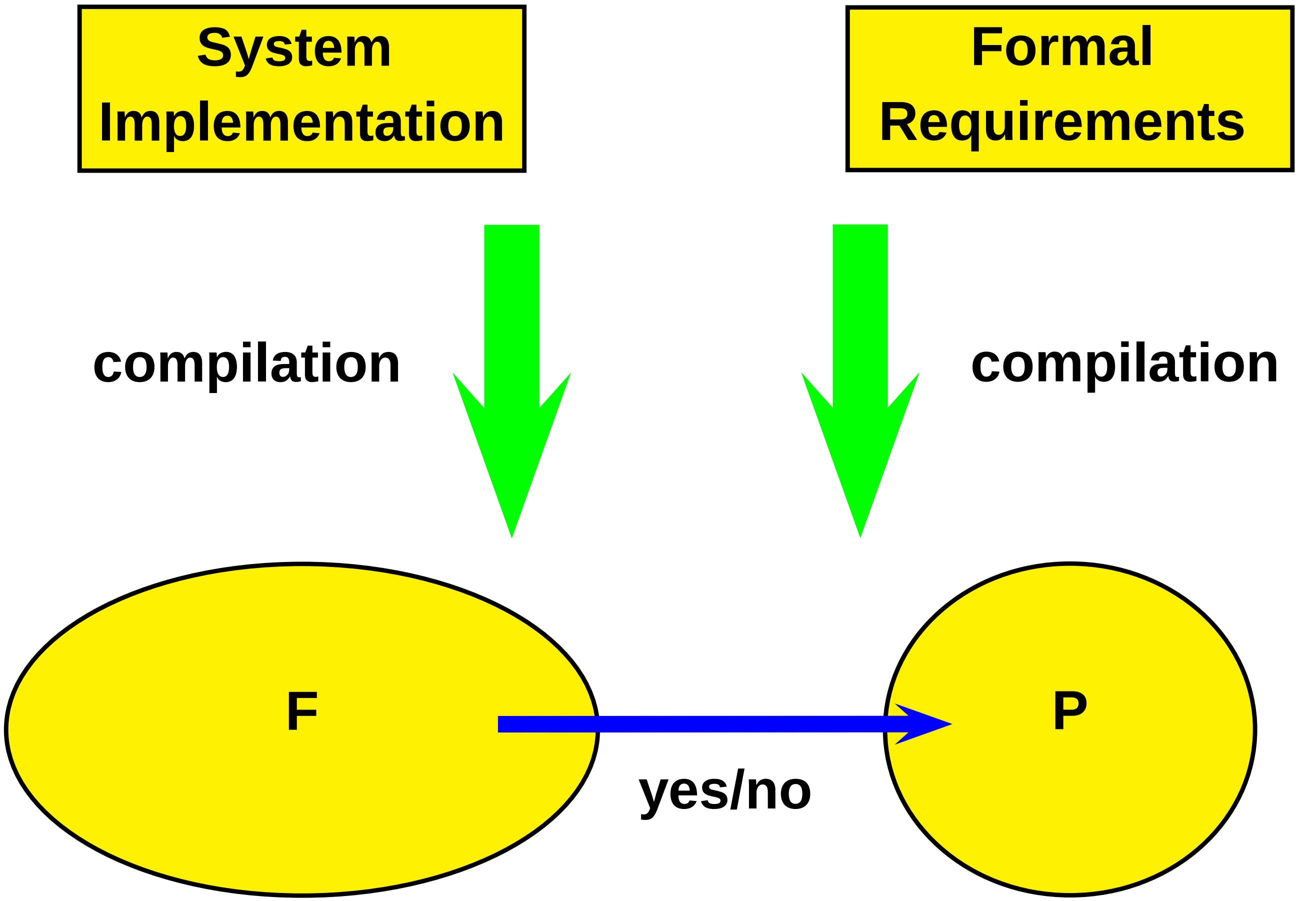
Model Checking



Model Checking in Industry

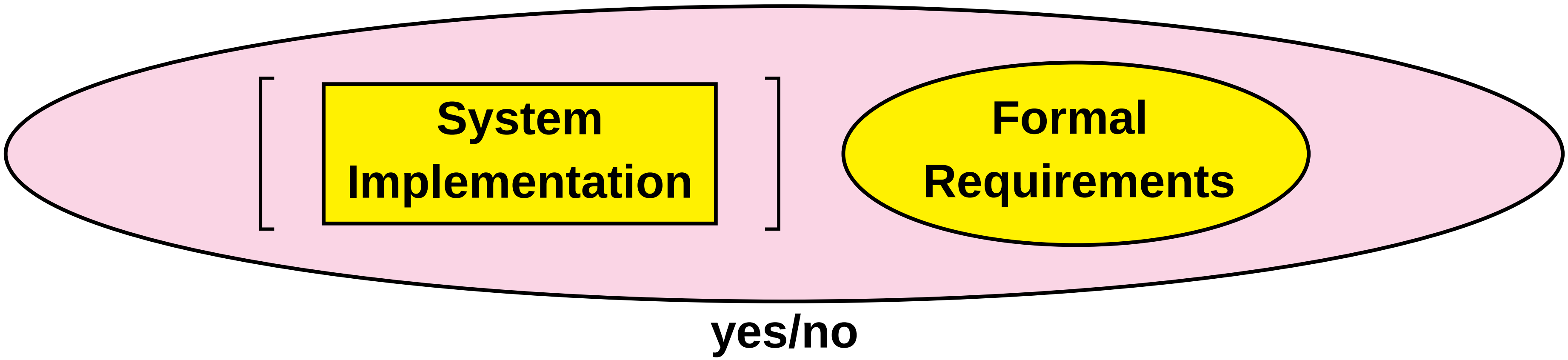
- **Hardware verification**
  - **Good match between limitations of technology and application**
  - **Intel, Motorola, IBM, . . .**
- **Software verification**
  - **Specialized software: control systems, protocols**
  - **Typically no checking of executable source code, but of abstraction thereof**
  - **Ericsson, Microsoft, Rockwell-Collins**

Proof Based Methods (I)



Proof rules establish relation "implementation conforms to specs"

Proof Based Methods (II)



Apply proof rules to establish validity of formula that encodes relation "implementation conforms to specs"

Proof Methods in Industry

- **Hardware verification**
  - **For large systems**
  - **Intel, Motorola, AMD, . . .**
- **Software verification**
  - **Safety critical systems, libraries**
  - **Paris driverless metro (Meteor), Emergency closing system**
  - **Rockwell-Collins, Avionics software**

SPIN at Bell Labs

- Feature interaction for telephone call processing software
- Tool works directly on C source code, automatic abstraction
- Web interface to track properties
- Work farmed out to large numbers of computers
- Finds shortest possible error trace
- 18 months, 300 versions, 75 bugs found
- Main burden: Defining meaningful properties

Static Driver Verifier/SLAM at Microsoft

- **Device drivers running in “kernel mode” should respect API**
- **Third-party device drivers do not respect APIs responsible for 90% of Windows crashes**
- **SLAM inspects C code, builds a finite state machine, checks requirements**
- **Static Driver Verifier β-released as part of the Windows Driver Foundation**

Static Driver Verifier/SLAM at Microsoft

- **Device drivers running in "kernel mode" should respect API**
- **Third-party device drivers do not respect APIs responsible for 90% of Windows crashes**
- **SLAM inspects C code, builds a finite state machine, checks requirements**
- **Static Driver Verifier β-released as part of the Windows Driver Foundation**

Future Trends

- 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
- Integration of formal method tools into CASE tools
- Applying formal methods to dependable systems design

Summary

Formal Methods . . .

- **Are (more and more) used in practice**
- **Can shorten development time**
- **Can push the limits of feasible complexity**
- **Can increase product quality**

Summary

Formal Methods . . .

- **Are (more and more) used in practice**
- **Can shorten development time**
- **Can push the limits of feasible complexity**
- **Can increase product quality**

Those responsible for software management should consider formal methods, especially within the realm of safety-critical, security-critical, and cost-intensive software