Formale Methoden im Software Entwurf

Einführung / Introduction

Reiner Hähnle

Department of Computer Science
Technische Universität Darmstadt

15. Oktober 2018

Kontakt und Kurswebseite

Dozenten

- Reiner Hähnle (RH), S2|02/A204, Vorlesung & Kursverantwortlicher
  E-Mail: haehnle@cs.tu-darmstadt.de
- Richard Bubel (RB), S2|02/A225, Übungen
  E-Mail: bubel@cs.tu-darmstadt.de

Sprechstunden: nach Vereinbarung
Kontakt: E-Mail (nicht Moodle Nachrichtensystem)

Moodle Kurswebseite

https://moodle.informatik.tu-darmstadt.de/course/view.php?id=466

- Bekanntgabe aller wichtigen Informationen (Termine, etc.)
- Folien der Vorlesungen, Übungsaufgaben & weiterführende Literatur
- Diskussionsforen

Klausur

Die Klausur findet statt am . . .

Donnerstag, 28. März 2019, 8:00

Allgemeines

1. Schriftliche Klausur
2. Dauer: 90 Minuten
3. Bonus von einem Notenschritt (0,3 bzw. 0,4):
   Erfolgreiches Bearbeiten von beiden Labs
   - Voraussetzung zum Einlösen: Bestehen der Klausur ohne Bonus
   - Details: siehe spätere Folien

Achtung: Bei der Prüfungsordnung (PO) 2009 sind die Labs eine für die Zulassung zur Klausur verpflichtend zu erbringende Studienleistung!

Übungen

Übungen

- Übungsblätter (Montag nachmittags via Moodle ab 22.10.2018)
- Lösungen werden in den Tutorien diskutiert
  - Tutorien starten in der Woche ab dem 29. Oktober 2018
  - Anmeldung über Moodle Webreg (TUCAN Anmeldung ist ungültig)

Anmeldezeitraum
Donnerstag, 18.10.2019 bis Montag, 22.10.2018 um 10:00 (**strikt**)

- Es soll ernsthaft versucht werden, die Übungen vor Besuch der Tutorien zu lösen (starke Korrelation mit Prüfungserfolg)
- Die Bearbeitung der Übungsaufgaben ist nicht verpflichtend, aber sehr empfohlen
- Es bietet sich an, Laptops mit den installierten Tools zu den Tutorien mitzubringen

Labs

2 Labs (1 Model Checking + 1 Deduktive Verifikation)

- Verfügbar über die Kurswebseite (ca. 2–3 Wochen vor Abgabe; Bekanntgabe via Moodle) — Deadlines:

  Model Checking Montag, 17. Dezember 2018, 8:00
  Deduktive Verifikation Montag, 11. Februar 2019, 8:00

- Abgabe der Lösungen erfolgt über Moodle. Genauere Informationen werden rechtzeitig über Moodle bekanntgegeben.

Lab-Gruppen

Gruppenanmeldung und Bonus

- Bonus ab $\geq$80 % der Gesamtpunktzahl beider Labs
  (beide Labs sind gleich gewichtet, d.h. dieselbe Punktzahl)
  - Nur PO 2009: Bestehen der Studienleistung ab $\geq$ 50 % der Punkte
- Arbeit in Gruppen von vier Personen

  Gruppenanmeldung für die Labs via Moodle:

  Montag 22.10.2018 bis Montag, 29.10.2018, 9:00 (**strikt**)
  - Forum zur Gruppenfindung in Moodle

Vorläufiger Zeitplan

Wird in Moodle laufend aktualisiert und ist einsehbar unter

https://moodle.informatik.tu-darmstadt.de/mod/page/view.php?id=15999

Vorlesungen sind i.d.R. Montags 10 Uhr 45
Achtung: Die rot markierten Termine finden Montags ab 9 Uhr 50 statt.

Empfohlene Literatur

The course books

Ben-Ari Mordechai Ben-Ari: Principles of the Spin Model Checker, Springer, 2008.
Authored by receiver of ACM award for outstanding Contributions to CS Education. Excellent text book.
Read online (from TU Darmstadt domain) at
https://www.springerlink.com/content/978-1-84628-769-5

KeY book Ahrendt et al., editors. Deductive Software Verification – The KeY Book, vol 10001 of *LNCS*. Springer, 2016.
Chapters 2, 3, 7, 15, and 16 Written by course developers.
Read online (from TU Darmstadt domain) at
https://link.springer.com/book/10.1007/978-3-319-49812-6

Further reading

Huth M. Huth, M. Ryan: Logic in Computer Science: Modeling and Reasoning about Systems, Cambridge, 2004.

Voraussetzungen und hilfreiche Vorkenntnisse

Voraussetzung

Gute Kenntnisse im objekt-orientierten Programmieren (z.B. Java)

Grundkenntnisse in den Gebieten

- Nebenläufiges Programmieren
- Logik (Aussagenlogik, Prädikatenlogik, Sequenzenkalkül)
- Endliche Automaten

sind hilfreich (verringern den Aufwand), aber falls nicht vorhanden:

Keine Sorge, wir werden alles Notwendige einführen

Motivation: Software Defects

Software Defects Can Endanger Human Lives

- Breast screening error: "a computer algorithm failure was to blame, which meant, in some cases, women approaching their 71st birthday were not sent an invitation for a final breast scan"
- Autonomes Uber-Auto in tödlichen Unfall verwickelt, bei dem ein Software-Fehler zum fatalen Ausgang beitrug

Software Defects Can Be Very Expensive

- Cryptocurrency Ether (2017):
  $300m accidentally lost forever due to software bug

Software Defects Can Be Hard To Patch ⇐ KeY

- Uncaught exception of Java Android's `sort()` for certain inputs:
  On all Android devices until version 5.1.1, still ca. 25 %

Better not create software defects in the first place!

Traditional Reliability Measures 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
- Design follows patterns that are proven to work

Why Does This 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
- Often no complete separation of subsystems possible
  - Many SW properties (e.g., security) are not compositional
  - Local failures can propagate through whole system
- Software design patterns don't address system complexity
- Many SW engineers untrained to address safety & security
- Cost efficiency/short time-to-market favoured over reliability
- Design practise for reliable software in partially immature state for complex, particularly distributed, systems

How to Ensure Software Correctness/Compliance?

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

Testing against internal SW errors ("bugs")

- Design (hopefully) representative test configurations
- Validate intended system behaviour on those

Testing against external faults

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

Limitations of Testing

- Testing shows presence of errors, not their absence (E. W. Dijkstra)
  (exhaustive testing viable only for very small systems)
- Representativeness of test cases/injected faults subjective
  How to test for the unexpected? Rare cases? Coverage?
  How to test against usage errors?
- Testing is labour intensive, hence expensive, hence limited

Formal methods can improve on and partially replace testing

But they can't replace testing! (See next slide why)

What Are Formal Methods?

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

Formal methods can't exclude errors that are not modelled
(while testing potentially can exhibit them)

What Are Formal Methods Good For?

- Complement (not replace!) other analysis and design methods
- Good at finding bugs
  (in code and specification)
- Reduce overall development time (testing/maintenance included)
  - can automate test case generation
  - can support debugging
- Ensure certain properties of the system model
- Should ideally be as automatic as possible

and

- Training in Formal Methods helps to write high quality code:
  - Practice in writing precise specifications
  - Raising awareness for missing/implicit assumptions
  - "What you can't verify you don't really understand"

Formal Specification and Testing

Testing:

- Run the system on chosen inputs and observe its behaviour
  - Randomly chosen (no guarantees, but can find bugs)
  - Intelligently chosen (by hand: expensive!)
  - Automatically chosen (need formalised spec or model)
- Are inputs representative? (test coverage)
- What about the observation? (test oracle = spec)
- (Formal) specifications are desirable even for testing!

Specification — What a System Should Do

- Simple properties
  - Safety properties
    Something bad will never happen (eg, mutual exclusion)
  - Liveness properties
    Something good will happen eventually
- Schematic properties, mainly of concurrent/distributed systems
  - Deadlock-freedom, no starvation, fairness
- Non-functional properties
  - Runtime, memory, usability, ...
- Full behavioural specification
  - Implementation satisfies a contract that describes its functionality
  - Data consistency, system invariants
  - Modularity, encapsulation
  - Program equivalence, e.g., compiler correctness
  - Relational properties: refinement, information flow, fault propagation

The Main Point of Formal Methods is Not

- To show "correctness" of entire systems
- To replace testing entirely
  - Formal methods work on models of source code or bytecode: hard to formally specify at machine level
  - Many non-formalizable properties (usability, ...)
- To replace good design practises or processes

There is no silver bullet!

- No correct system w/o clear requirements & good design
- One can't formally verify messy code with unclear specs

But . . .

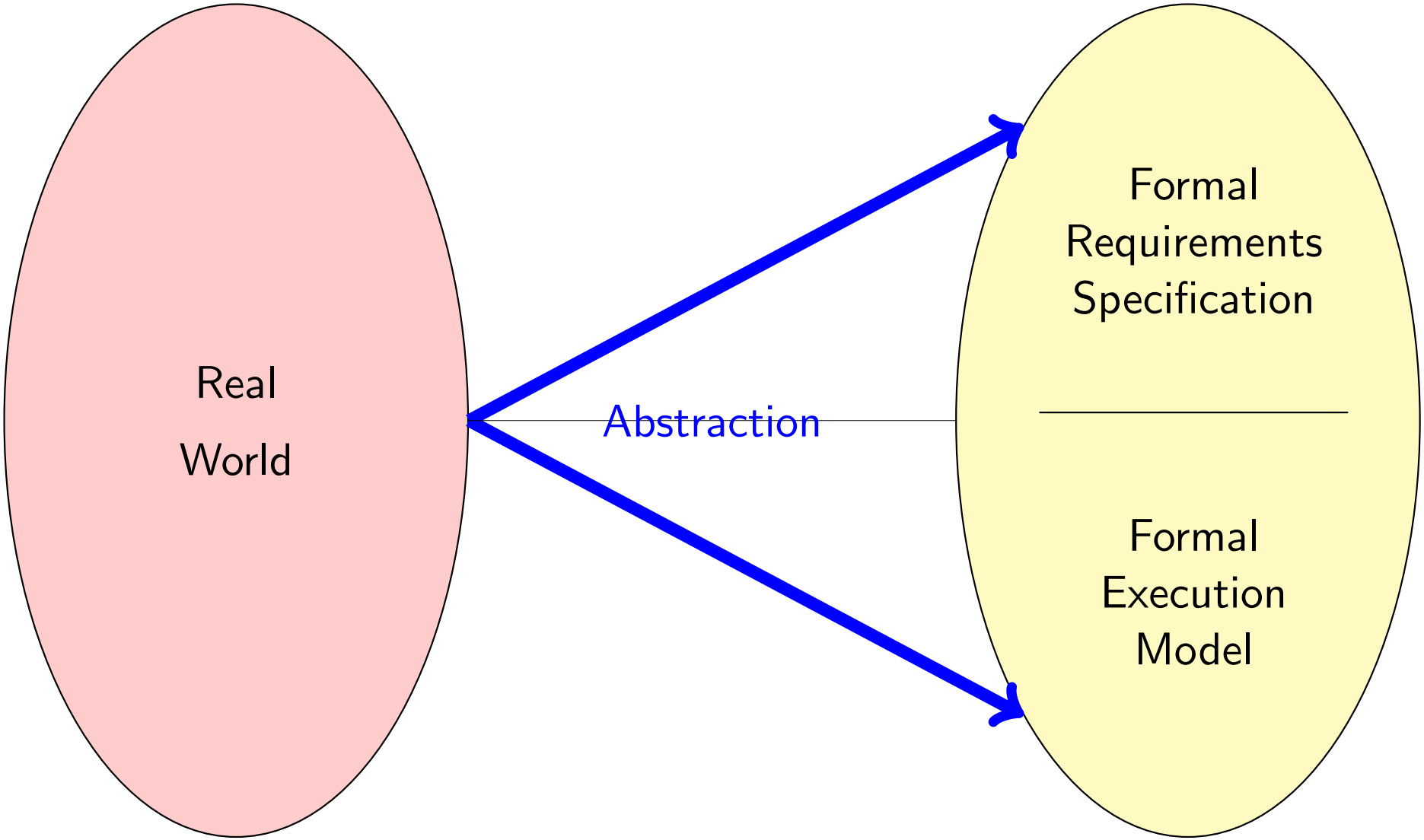
- Formal proof can replace a myriad of test cases
- Formal methods improve the quality of specs
  (even without formal verification)
- Formal methods guarantee specific properties of system model

A Fundamental Fact

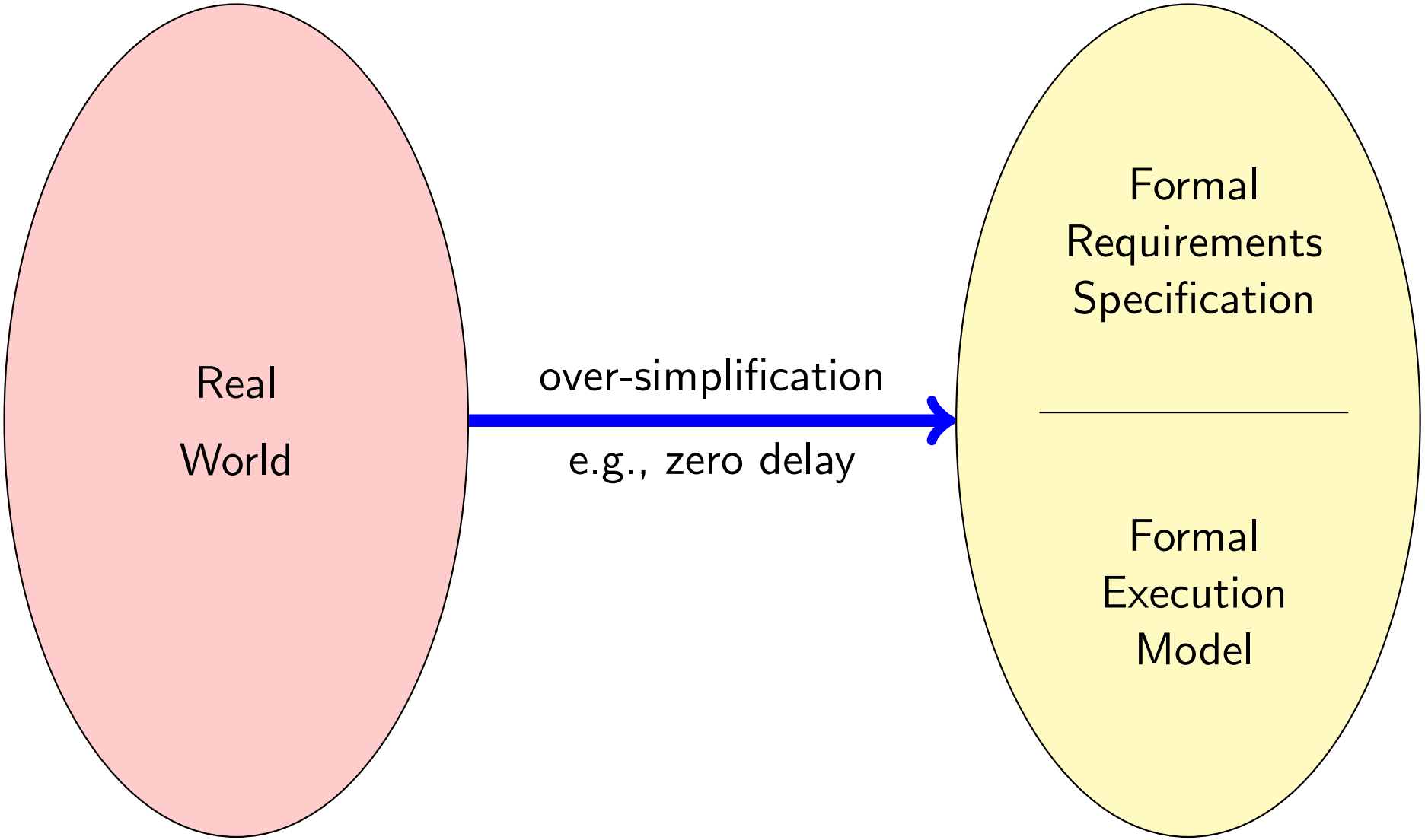
Formalization of system requirements is hard

Let's see why ...

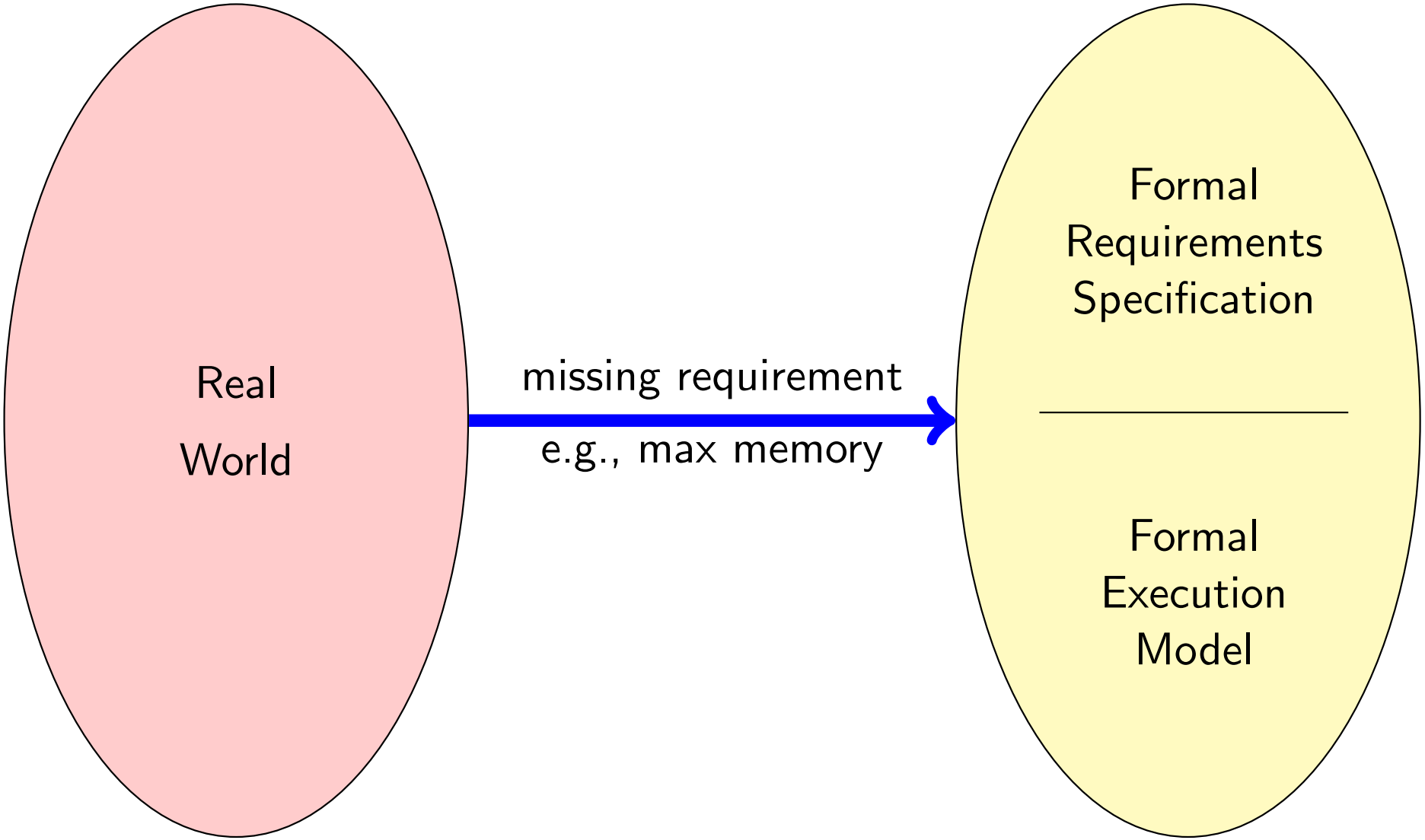
Difficulties in Creating Formal Models



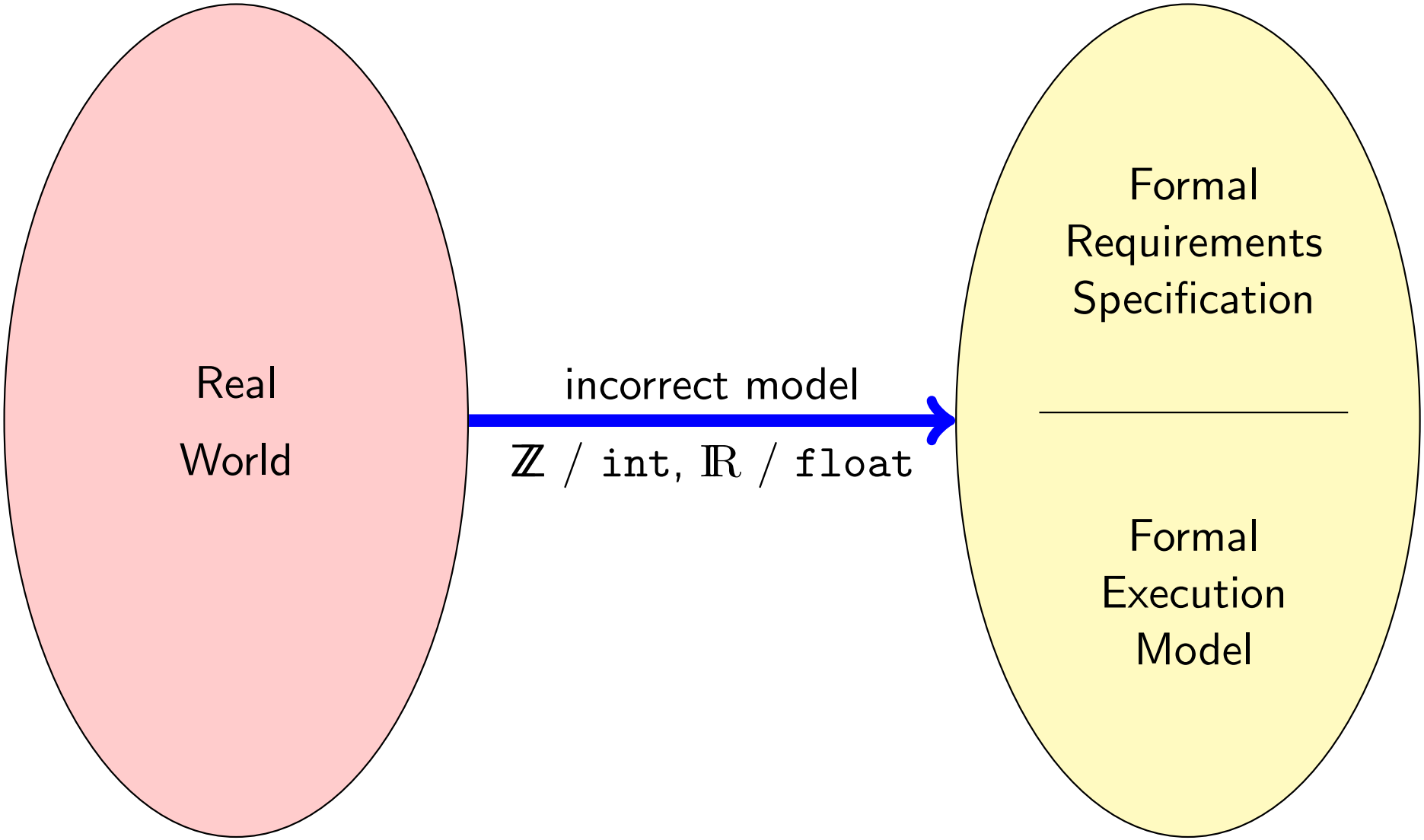
Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



But: Formalization Can Improve Specs

- Well-formedness and consistency of formal specs partly machine-checkable
- Need to declare signature (symbols) helps to spot incomplete specs
- Failed verification of implementation against spec gives feedback on erroneous formalization

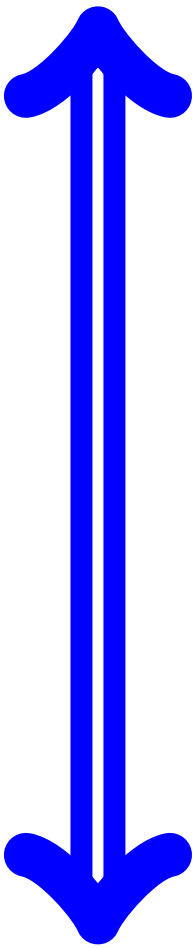
Errors in specifications are at least as common as errors in code, but their discovery gives deep insights in (mis)conceptions of the system

Another Fundamental Fact

Proving systems requirements is hard

Let's have a closer look ...

Level of Abstraction of System Model

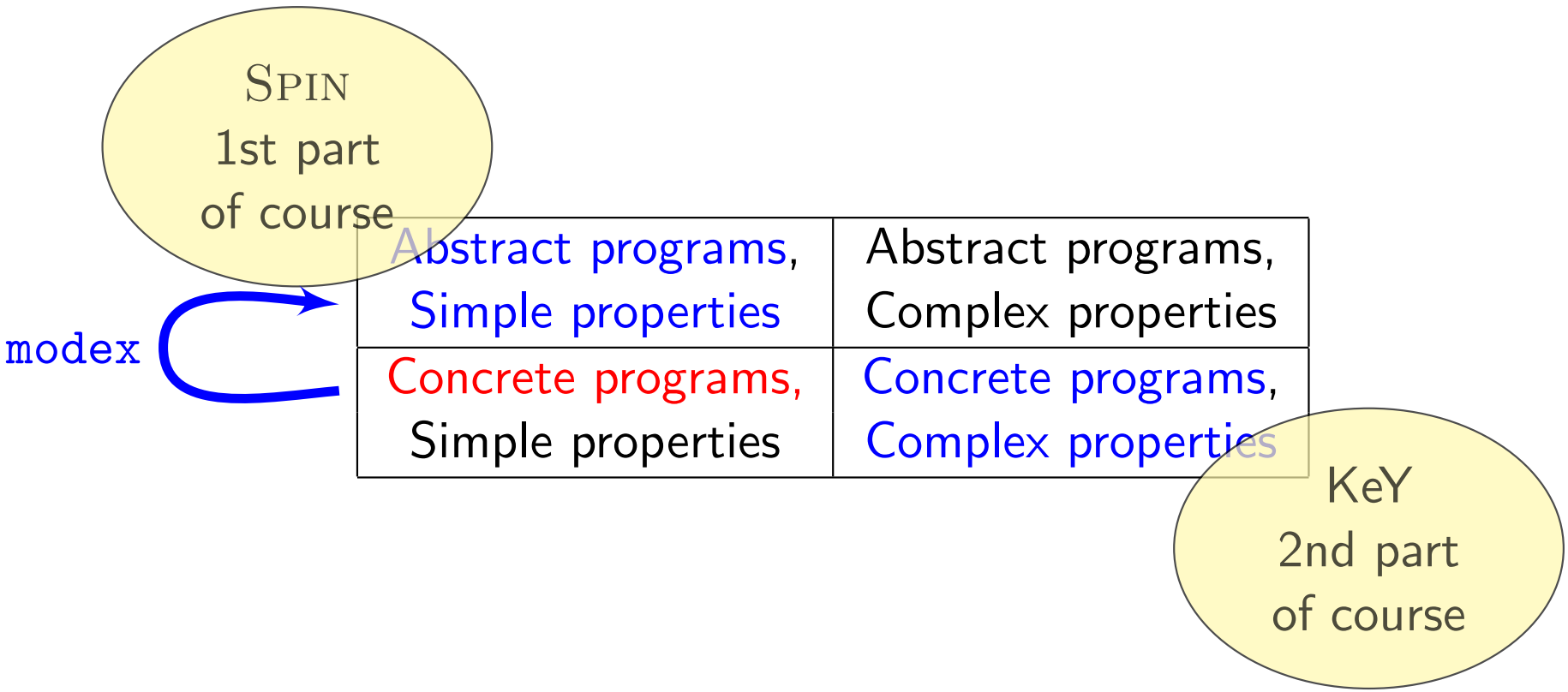


- Abstract, e.g., behavior given as finite automaton
  - Finitely many execution states (finite datatypes)
  - Automatic proofs are (in principle) possible
  - Simplification, unfaithful modeling inevitable
  - Still may capture essential characteristics
    (e.g., protocol among distributed processes)
- Concrete level, e.g., JAVA program
  - Unbounded datatypes
    (lists, arrays, streams)
  - Complex datatypes and control structures
    (exceptions, static initialization, objects, ...)
  - Automatic proofs (in general) impossible
  - Precise characterization of semantics

Expressiveness of Specification

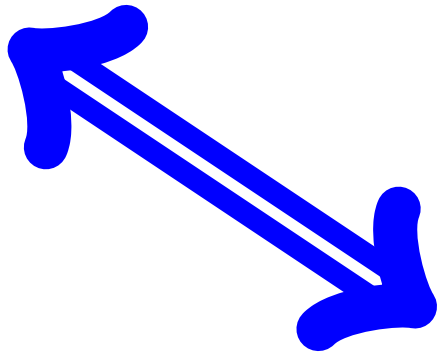
- Simple
  - Simple or schematic properties
  - Finitely many case distinctions
  - Approximation with propositional logic
  - Automatic proofs are (in principle) possible
- Complex
  - Full behavioural specification
  - Quantification over infinite domains
  - Precise modeling with first-order logic
  - Automatic proofs (in general) impossible

Different Combinations



Automation of Proof Search

- "Automatic" Proof
  ("batch" mode, "auto-active" mode)
  - No interaction during actual verification necessary
  - Proof may fail or result can be inconclusive
  - Tuning of tool parameters, patching of program, specs necessary
  - Formal specification of non-trivial properties "by hand"



- "Semi-Automatic" Proof
  ("interactive")
  - Interaction may be required during proof
  - Need certain knowledge of tool internals Intermediate inspection can help
  - Proof is checked by tool

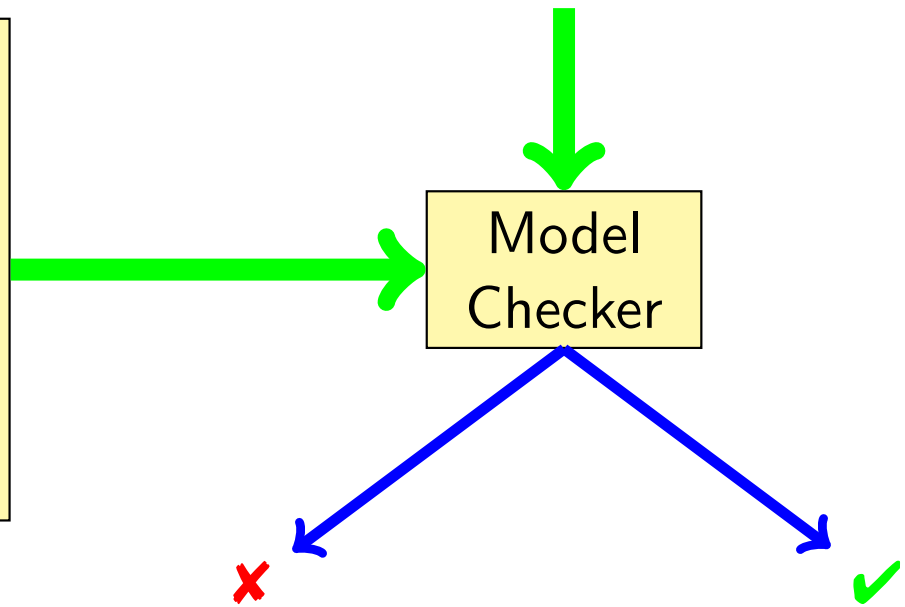
Model Checking Principle

System Model

System Property

```
[] ! (criticalSectP && criticalSectQ)
```

```
byte n = 0;
active proctype P() {
  n = 1;
}
active proctype Q() {
  n = 2;
}
```




```
criticalSectP = 0  1  1
criticalSectQ = 1  0  1
```

Model Checking in Industry

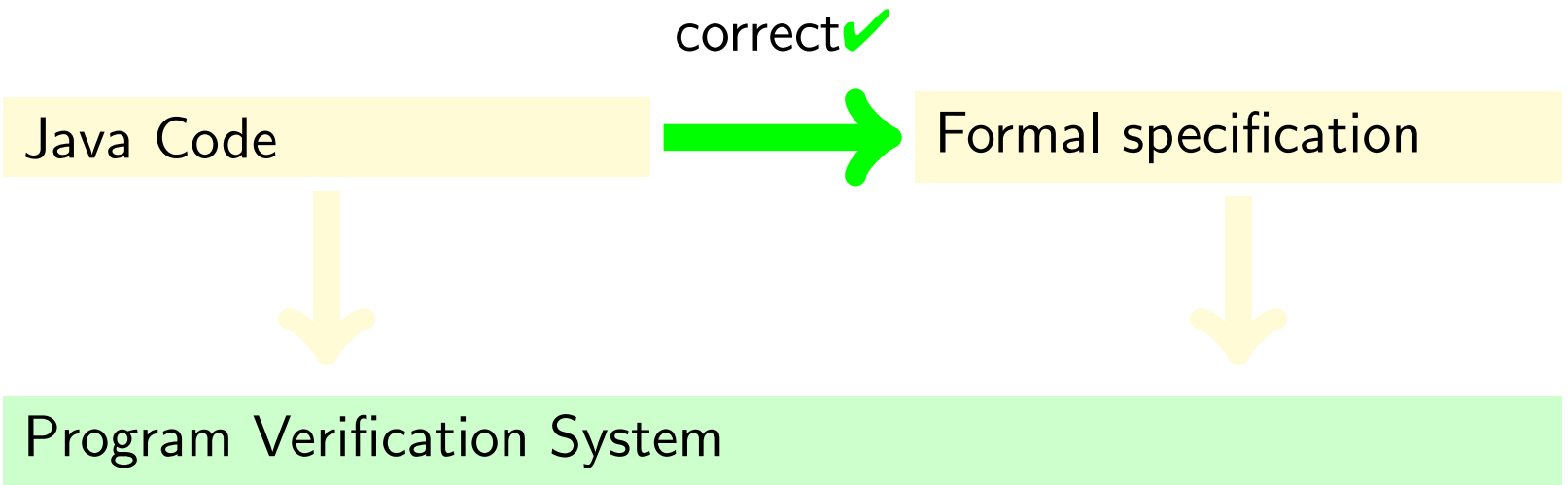
- Hardware verification
  - Good match between limitations of technology and application
  - Intel, Motorola, AMD, QualComm, Nvidia, Xilinx, ...
- Software verification
  - Specialized software: control systems, protocols
  - Typically no checking of executable source code, but of abstraction
  - Bosch, Ericsson, Huawei, Microsoft, ...

A Major Case Study with SPIN

Checking feature interaction for telephone call processing software

- Software for PathStar^{TM} server from Lucent Technologies
- Automated abstraction of unchanged C code into PROMELA
- Web interface, with SPIN as back-end, to:
  - track properties (ca. 20 temporal formulas)
  - invoke verification runs
  - report error traces
- Finds shortest possible error trace, reported as C execution trace
- Work farmed out to 16 computers, daily, overnight runs (AD 2000)
- 18 months, 300 versions of system model, 75 bugs found
- Strength: detection of undesired feature interactions
  (difficult with traditional testing)
- Main challenge: defining meaningful properties

Deductive Verification



Proof rules establish relation "implementation conforms to specs"

Computer support essential for verification of real programs

`boolean ArrayList:contains(Object o)`

- Typical Java library method implementation
- ca. 6,400 proof steps, ca. 8 secs with KeY
- ca. 40 case distinctions, fully automatic

Deductive Verification in Industry

- Hardware verification
  - For complex systems, most of all floating-point processors
  - Intel, Motorola, AMD, ...
- Software verification
  - Safety critical systems:
    - Paris driver-less metro (Meteor)
    - Emergency closing system in North Sea
  - Libraries
    - Java Card 2.2.1 API verified reference implementation
  - Implementations of Protocols

A Major Case Study with KeY

Android & OpenJDK's `static void sort(int[])`

- Extremely complex hybrid sorting algorithm, derived from Python
- Discovered bug: throws uncaught exception for certain input arrays
- Fixed bug in Android (5.1.1), Apache, Python, Haskell
- Verified absence of bug for unaltered library implementation
  - proof of `sort` + seven auxiliary methods
  - Ca. 500 LoC and 500 LoS, ca. 3,000,000 nodes
- Total effort several person months
- $>$ 99 % automation, but still $>$ 20,000 interactions
- Main challenge: proof size and complexity, finding loop invariant

Tool Support is Essential

Some Reasons for Using Tools

- Automate repetitive tasks
- Avoid clerical errors, etc.
- Cope with large/complex programs
- Make verification certifiable

Tools (with GUIs) are Used in this Course in Both Parts:

SPIN to verify PROMELA programs against Temporal Logic specs

KeY to verify Java programs against contracts in JML

Both are free, open source and run on Windows/Unixes/Mac

Install first SPIN on your computer

Follow installation instructions for SPIN on course home page

Literature for this Lecture

FM in SE B. Beckert, R. Hähnle, T. Hoare, D. Smith, C. Green, S. Ranise, C. Tinelli, T. Ball, and S. K. Rajamani: Intelligent Systems and Formal Methods in Software Engineering. *IEEE Intelligent Systems*, 21(6):71–81, 2006. ieeexplore.ieee.org/document/4042539/

KeY R. Hähnle: Quo Vadis Formal Verification? In: W. Ahrendt et al., editors. *Deductive Software Verification: The KeY Book*, pp 1–19, vol. 10001 of *LNCS*. Springer, 2017. link.springer.com/chapter/10.1007/978-3-319-49812-6_1

SPIN Gerard J. Holzmann: A Verification Model of a Telephone Switch. In: *The Spin Model Checker*, pp 299–324, Chapter 14, Addison Wesley, 2004.

TimSort S. de Gouw, J. Rot, F. de Boer, R. Bubel, R. Hähnle: OpenJDK's `Java.utils.Collection.sort()` Is Broken: The Good, the Bad and the Worst Case. CAV 2015: pp. 273–289, vol. 9206 of *LNCS*. Springer, 2015. link.springer.com/chapter/10.1007/978-3-319-21690-4_16

During This Course You Will Gain Experience in . . .

- Software modeling, and modeling languages
- Formal specification, and formal specification languages
- In-depth analysis of possible system behaviour
- Typical types of software and specification errors
- Reasoning about system (mis-)behaviour
- Modeling the behavior of software with logic

To Do-List after Today's Lecture

1. Form a lab group with three other students
2. Install SPIN and `ispin` on your computer
3. Read Intelligent Systems and Formal Methods in Software Engineering