

# Formal Methods for Software Development

## Introduction

Wolfgang Ahrendt

Department of Computer Science and Engineering  
Chalmers University of Technology  
and  
University of Gothenburg

29 August 2017

## Teachers

- ▶ Wolfgang Ahrendt (WA)    examiner, lecturer
- ▶ Mauricio Chimento (MC)    teaching assistant
- ▶ Andreas Löow (AL)    teaching assistant
- ▶ Raúl Pardo (RP)    teaching assistant

## Teachers

- ▶ Wolfgang Ahrendt (WA) **examiner, lecturer**
- ▶ Mauricio Chimento (MC) **teaching assistant**
- ▶ Andreas Löow (AL) **teaching assistant**
- ▶ Raúl Pardo (RP) **teaching assistant**

course assistant activities include:

- ▶ giving exercise classes
- ▶ correcting lab hand-ins
- ▶ student support via:
  - ▶ e-mail
  - ▶ meetings on e-mail request
    - ▶ Mauricio, room 5446
    - ▶ Andreas, room 5461
    - ▶ Raúl, room 5447

# Organisational Stuff

## Course Home Page

Linked from Chalmers and GU course portals

## Google News Group

- ▶ Sign up via course home page (see [News Group](#))
- ▶ Changes, updates, questions, discussions (don't post solutions)

## Passing Criteria

- ▶ **Individual, oral examination** in exam week (week 43)
- ▶ Two lab hand-ins
- ▶ Oral exam and labs can be passed separately

# Course Structure

## Course Structure

Topic	# Lectures	# Exercises	Lab
Intro	1	✗	✗
Modeling & Model Checking with <i>PROMELA &amp; SPIN</i>	6	3	✓
Specification & Verification with <i>JML &amp; KeY</i>	7	3	✓

*PROMELA & SPIN* abstract programs, model checking, automated

*JML & KeY* concrete Java, deductive verification, semi-automated

... more on this later!

## Lectures

- ▶ Please ask questions during lectures
- ▶ Please respond to my questions

## Lectures

- ▶ Please ask questions during lectures
- ▶ Please respond to my questions; 'wrong' answers highly welcome

## Lectures

- ▶ Please ask questions during lectures
- ▶ Please respond to my questions
- ▶ Slides appear online shortly *after* each lecture



## Exercises

- ▶ One exercise web page (almost) each week (6 in total)
- ▶ Discussed in next exercise class
- ▶ Play around with the exercises before coming to the class
- ▶ Exercises **highly** recommended
- ▶ Bring laptops if you have  
(ideally w. installed tools or browser interfaces working)

## Labs

- ▶ 2 Lab handins: PROMELA/SPIN 29 Sep, JML/KeY 23 Oct
- ▶ 2 Lab Questions Sessions
- ▶ Submission via **Fire**, linked from course home page
- ▶ If submission is returned, roughly one week for correction

## Labs

- ▶ 2 Lab handins: PROMELA/SPIN 29 Sep, JML/KeY 23 Oct
- ▶ 2 Lab Questions Sessions
- ▶ Submission via **Fire**, linked from course home page
- ▶ If submission is returned, roughly one week for correction
- ▶ You work in groups of **two**. No exception!<sup>a</sup>  
You pair up by either:
  1. talk to people
  2. post to the Google group
  3. participate in pairing at first exercise session

In case all that is **not** sufficient, contact Mauricio by e-mail.

---

<sup>a</sup>Only PhD students have to work alone.

# Schedule

see course homepage

# Course Evaluation

1. course evaluation group:
  - ▶ student representatives
    - ▶ randomly selected (Chalmers)
    - ▶ volunteers (GU)
  - ▶ one meeting during the course, one after
2. web questionnaire after the course

# Course Evaluation

1. course evaluation group:
  - ▶ student representatives
    - ▶ randomly selected (Chalmers)
    - ▶ volunteers (GU)
  - ▶ one meeting during the course, one after
2. web questionnaire after the course

Randomly selected Chalmers students:

- ▶ Elmar Aliyev
- ▶ Daniel Eineving
- ▶ Mohamad Mortada
- ▶ André Samuelsson
- ▶ Yan Wang

# Course Evaluation

1. course evaluation group:
  - ▶ student representatives
    - ▶ randomly selected (Chalmers)
    - ▶ volunteers (GU)
  - ▶ one meeting during the course, one after
2. web questionnaire after the course

Randomly selected Chalmers students:

- ▶ Elmar Aliyev
- ▶ Daniel Eineving
- ▶ Mohamad Mortada
- ▶ André Samuelsson
- ▶ Yan Wang

GU students: please consider volunteering

- ▶ In part I, we partly use:

**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. Recommended by G. Holzmann. Excellent student text book.*

(E-book at [link.springer.com](http://link.springer.com))

- ▶ Relevant for part II:

**KeYbook** W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich, editors.

**Deductive Software Verification - The KeY Book**

Vol 10001 of LNCS, Springer, 2016

(E-book at [link.springer.com](http://link.springer.com))



# Additional Literature

**Holzmann** Gerard J. Holzmann  
**The Spin Model Checker**  
Addison Wesley, 2004

**BayerKatoen** Christel Baier, Joost-Pieter Katoen  
**Principles of Model Checking**  
MIT Press, 2008

# Connection to other Courses

## Prerequisites

- ▶ Skills in first-order logic and temporal logic, e.g., from
  - ▶ Logic in Computer Science, or
  - ▶ Discrete Event Systems
- ▶ Skills in object-oriented programming (like Java).

## Related courses (not assumed!)

- ▶ Concurrent Programming
- ▶ Finite Automata
- ▶ Testing, Debugging, and Verification

# Connection to other Courses

## Prerequisites

- ▶ Skills in first-order logic and temporal logic, e.g., from
  - ▶ Logic in Computer Science, or
  - ▶ Discrete Event Systems
- ▶ Skills in object-oriented programming (like Java).

## Related courses (not assumed!)

- ▶ Concurrent Programming
- ▶ Finite Automata
- ▶ Testing, Debugging, and Verification

if you took any of those: nice

# Connection to other Courses

## Prerequisites

- ▶ Skills in first-order logic and temporal logic, e.g., from
  - ▶ Logic in Computer Science, or
  - ▶ Discrete Event Systems
- ▶ Skills in object-oriented programming (like Java).

## Related courses (not assumed!)

- ▶ Concurrent Programming
- ▶ Finite Automata
- ▶ Testing, Debugging, and Verification

if you took any of those: nice

if not: don't worry, we introduce everything we use here

# Motivation:

## Software Defects cause BIG Failures

Tiny faults in technical systems can have catastrophic consequences

### In particular, this goes for software systems

- ▶ Ariane 5
- ▶ Mars Climate Orbiter
- ▶ London Ambulance Dispatch System
- ▶ NEDAP Voting Computer Attack
- ▶ ...

# Motivation:

## Software Defects cause OMNIPRESENT Failures

Ubiquitous Computing results in Ubiquitous Failures

### Software is almost everywhere:

- ▶ Mobiles
- ▶ Smart devices
- ▶ Smart cards
- ▶ Cars
- ▶ ...

# Motivation:

## Software Defects cause OMNIPRESENT Failures

Ubiquitous Computing results in Ubiquitous Failures

### Software is almost everywhere:

- ▶ Mobiles
- ▶ Smart devices
- ▶ Smart cards
- ▶ Cars
- ▶ ...

software/specification quality is a growing commercial and legal issue

# Achieving Reliability in Engineering

## Well-known strategies from mechanical and civil engineering

- ▶ Precise calculations/estimations of forces, stress, etc.



# Achieving Reliability in Engineering

## Well-known strategies from mechanical and civil engineering

- ▶ Precise calculations/estimations of forces, stress, etc.
- ▶ Hardware redundancy (“make it a bit stronger than necessary”)

# Achieving Reliability in Engineering

## Well-known strategies from mechanical and civil engineering

- ▶ Precise calculations/estimations of forces, stress, etc.
- ▶ Hardware redundancy (“make it a bit stronger than necessary”)
- ▶ Robust design (single fault not catastrophic)

# Achieving Reliability in Engineering

## Well-known strategies from mechanical and 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

# Achieving Reliability in Engineering

## Well-known strategies from mechanical and 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 This Does Not (Quite) Work For Software?

- ▶ Software systems compute **non-continuous** functions.  
Single bit-flip may change behaviour completely.

# Why This Does Not (Quite) 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 special cases.

# Why This Does Not (Quite) 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 special cases.
- ▶ Insufficient **separation** of subsystems.  
Seemingly correct sub-systems may together behave incorrectly.

# Why This Does Not (Quite) 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 special cases.
- ▶ Insufficient **separation** of subsystems.  
Seemingly correct sub-systems may together behave incorrectly.
- ▶ Software designs have very high logical **complexity**.



# Why This Does Not (Quite) 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 special cases.
- ▶ Insufficient **separation** of subsystems.  
Seemingly correct sub-systems may together behave incorrectly.
- ▶ Software designs have very high logical **complexity**.
- ▶ Most SW engineers **untrained** to address correctness.

# Why This Does Not (Quite) 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 special cases.
- ▶ Insufficient **separation** of subsystems.  
Seemingly correct sub-systems may together behave incorrectly.
- ▶ Software designs have very high logical **complexity**.
- ▶ Most SW engineers **untrained** to address correctness.
- ▶ Cost efficiency favoured over reliability.

# Why This Does Not (Quite) 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 special cases.
- ▶ Insufficient **separation** of subsystems.  
Seemingly correct sub-systems may together behave incorrectly.
- ▶ Software designs have very high logical **complexity**.
- ▶ Most SW engineers **untrained** to address correctness.
- ▶ Cost efficiency favoured over reliability.
- ▶ Design practise for reliable software in **immature** state  
for complex (e.g., distributed) systems.

# How to Ensure Software Correctness/Compliance?

A central strategy: [testing](#)

(others: SW processes, reviews, libraries, ...)

# How to Ensure Software Correctness/Compliance?

A central strategy: **testing**

(others: SW processes, reviews, libraries, ...)

**Testing against internal SW errors (“bugs”)**

- ▶ find (hopefully) representative test configurations
- ▶ check intentional system behaviour on those

# How to Ensure Software Correctness/Compliance?

A central strategy: [testing](#)

(others: SW processes, reviews, libraries, ...)

## Testing against internal SW errors (“bugs”)

- ▶ find (hopefully) representative test configurations
- ▶ check intentional 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 (exhaustive testing viable only for trivial systems)

# Limitations of Testing

- ▶ Testing shows presence of errors, 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?



# Limitations of Testing

- ▶ Testing shows presence of errors, 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 labour intensive, hence expensive

# What **are** Formal Methods

- ▶ Rigorous methods for system design/development/analysis
- ▶ Mathematics and symbolic logic  $\Rightarrow$  **formal**
- ▶ Increase confidence in a system
- ▶ Two aspects:
  - ▶ System **requirements**
  - ▶ System **implementation**
- ▶ Make formal model of both
- ▶ Use **tools** for
  - ▶ **exhaustive** search for failing scenario, or
  - ▶ mechanical **proof** that implementation satisfies requirements

# What are Formal Methods **for**

- ▶ Complement other analysis and design methods
- ▶ Increase confidence in system correctness
- ▶ Good at finding bugs  
(in code **and** specification)
- ▶ *Ensure* certain **properties** of the system (model)
- ▶ Should ideally be as automated as possible

# What are Formal Methods **for**

- ▶ Complement other analysis and design methods
- ▶ Increase confidence in system correctness
- ▶ Good at finding bugs  
(in code **and** specification)
- ▶ *Ensure* certain **properties** of the system (model)
- ▶ Should ideally be as automated as possible

and

- ▶ **Training in Formal Methods increases high quality development skills**

# 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

# 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
- ▶ General properties of concurrent/distributed systems
  - ▶ deadlock-free, no starvation, fairness

# 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
- ▶ General properties of concurrent/distributed systems
  - ▶ deadlock-free, no starvation, fairness
- ▶ Non-functional properties
  - ▶ Execution time, memory, usability, ...

# 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
- ▶ General properties of concurrent/distributed systems
  - ▶ deadlock-free, no starvation, fairness
- ▶ Non-functional properties
  - ▶ Execution time, memory, usability, ...
- ▶ Full behavioural specification
  - ▶ Code functionality described by **contracts**



# 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
- ▶ General properties of concurrent/distributed systems
  - ▶ deadlock-free, no starvation, fairness
- ▶ Non-functional properties
  - ▶ Execution time, memory, usability, ...
- ▶ Full behavioural specification
  - ▶ Code functionality described by **contracts**
  - ▶ Data consistency, system **invariants**  
(in particular for efficient, i.e., redundant, data representations)

# 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
- ▶ General properties of concurrent/distributed systems
  - ▶ deadlock-free, no starvation, fairness
- ▶ Non-functional properties
  - ▶ Execution time, memory, usability, ...
- ▶ Full behavioural specification
  - ▶ Code functionality described by **contracts**
  - ▶ Data consistency, system **invariants**  
(in particular for efficient, i.e., redundant, data representations)
  - ▶ Modularity, encapsulation

# 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
- ▶ General properties of concurrent/distributed systems
  - ▶ deadlock-free, no starvation, fairness
- ▶ Non-functional properties
  - ▶ Execution time, memory, usability, ...
- ▶ Full behavioural specification
  - ▶ Code functionality described by **contracts**
  - ▶ Data consistency, system **invariants**  
(in particular for efficient, i.e., redundant, data representations)
  - ▶ Modularity, encapsulation
  - ▶ Refinement relation

# The Main Point of Formal Methods is **Not**

- ▶ to show correctness of entire systems
- ▶ to replace testing
- ▶ to replace good design practises

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 ...

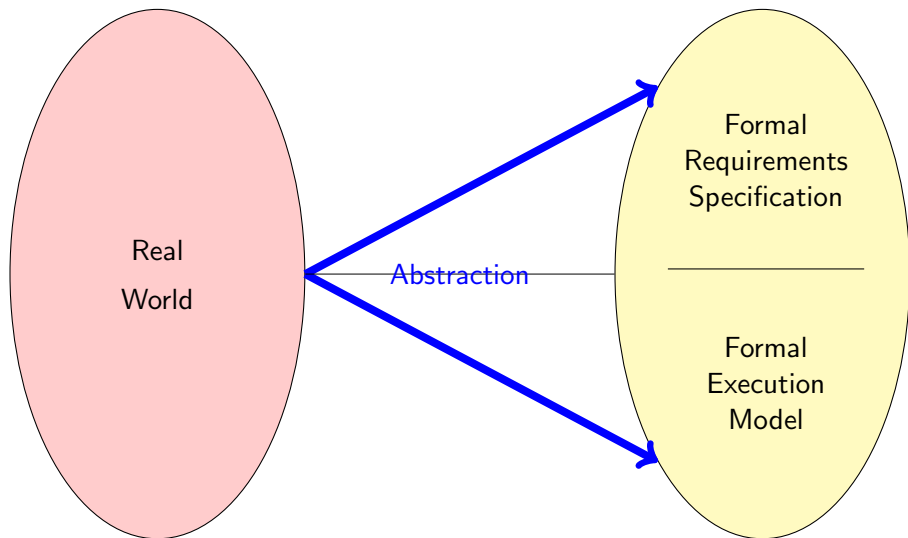
- ▶ Formal proof can replace (infinitely) many test cases
- ▶ Formal methods improve the quality of specs (even without formal verification)
- ▶ Formal methods guarantee specific properties of system model

# A Fundamental Fact

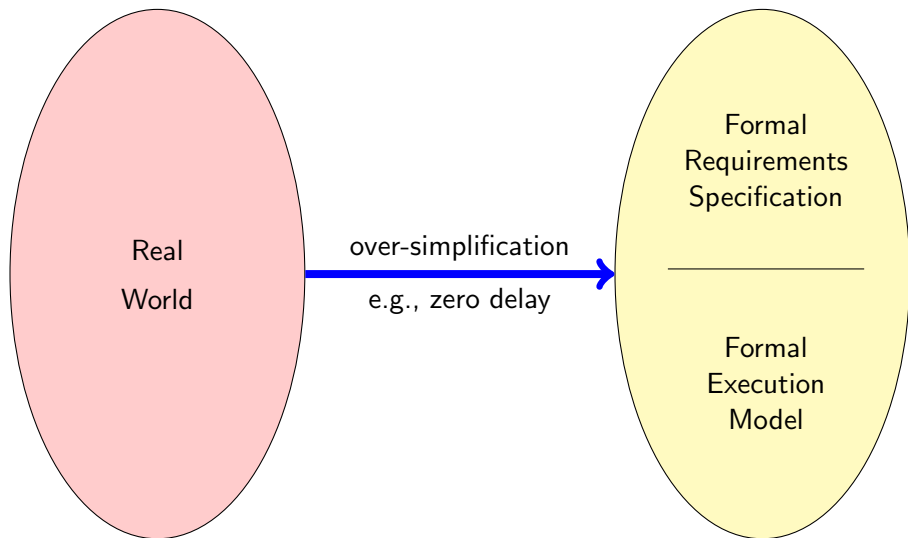
Formalisation of system requirements is hard

Let's see why ...

# Difficulties in Creating Formal Models

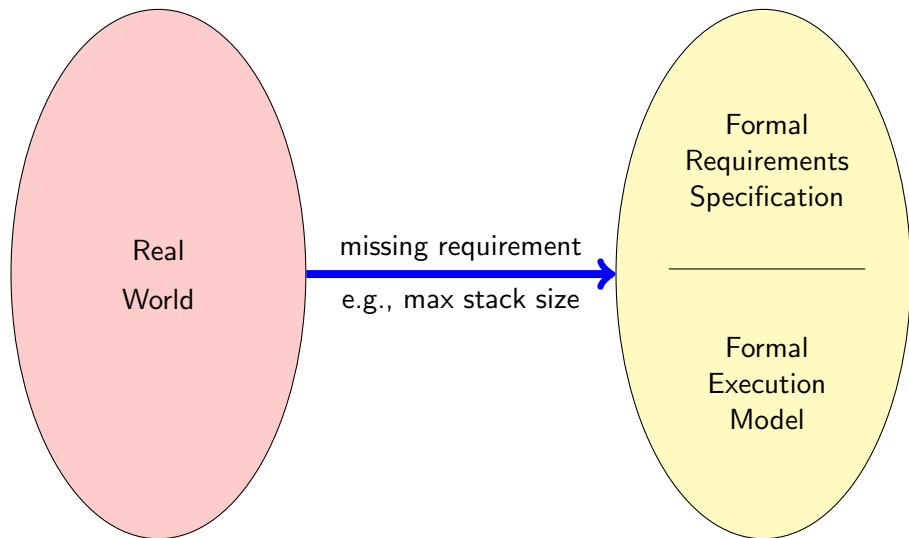


# Difficulties in Creating Formal Models

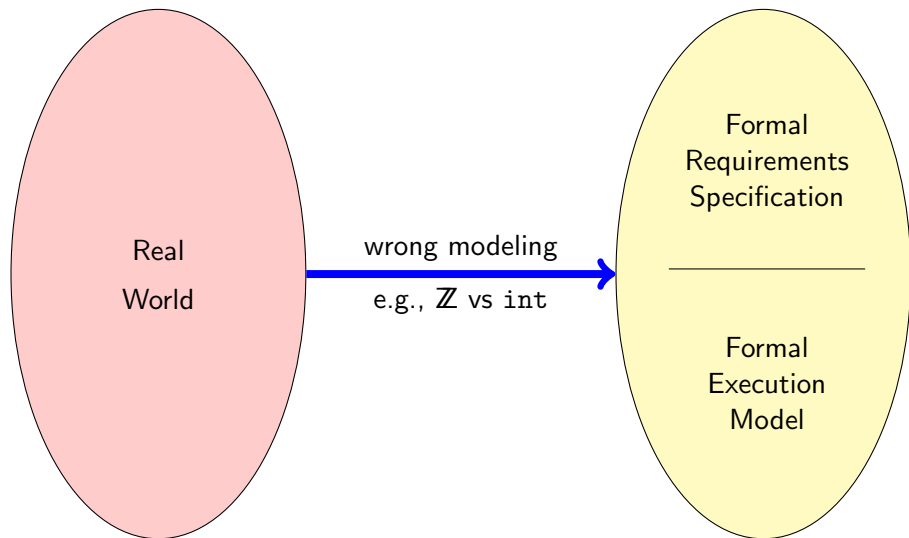




# Difficulties in Creating Formal Models



# Difficulties in Creating Formal Models



# Formalization Helps to Find Bugs in Specs

Errors in specifications are as common as errors in code

# Formalization Helps to Find Bugs in Specs

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

# Formalization Helps to Find Bugs in Specs

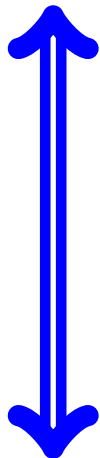
Errors in specifications are as common as errors in code, but their discovery gives deep insights in (mis)conceptions of the system.

- ▶ Wellformedness and consistency of formal specs partly machine-checkable
- ▶ Declared 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



- ▶ **Abstract level**

- ▶ Finitely many states (bounded size datatypes)
- ▶ Automated proofs are (in principle) possible
- ▶ Simplification, unfaithful modeling inevitable

- ▶ **Concrete level**

- ▶ Unbounded size datatypes  
(pointer chains, dynamic containers, streams)
- ▶ Complex datatypes and control structures
- ▶ Realistic programming model (e.g., Java)
- ▶ Automated proofs hard or impossible!

# Expressiveness of Specification



## ▶ Simple

- ▶ Simple or general properties
- ▶ Finitely many case distinctions
- ▶ Approximation, low precision
- ▶ Automated proofs are (in principle) possible

## ▶ Complex

- ▶ Full behavioural specification
- ▶ Quantification over infinite or large domains
- ▶ High precision, tight modeling
- ▶ Automated proofs hard or impossible!



# Main Approaches

Abstract programs, Simple properties	Abstract programs, Complex properties
Concrete programs, Simple properties	Concrete programs, Complex properties

# Main Approaches

SPIN  
1st part  
of course

Abstract programs, Simple properties	Abstract programs, Complex properties
Concrete programs, Simple properties	Concrete programs, Complex properties

# Main Approaches

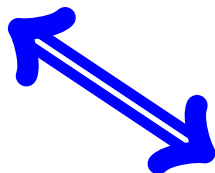
SPIN  
1st part  
of course

Abstract programs, Simple properties	Abstract programs, Complex properties
Concrete programs, Simple properties	Concrete programs, Complex properties

KeY  
2nd part  
of course

# Proof Automation

- ▶ “Automated” Proof  
(“batch-mode”)
  - ▶ No interaction (or lemmas) necessary
  - ▶ Proof may fail or result inconclusive  
Tuning of tool parameters necessary
  - ▶ Formal specification still “by hand”
  
- ▶ “Semi-Automated” Proof  
(“interactive”)
  - ▶ Interaction (or lemmas) may be required
  - ▶ Need certain knowledge of tool internals  
Intermediate inspection can help
  - ▶ User steps are checked by tool



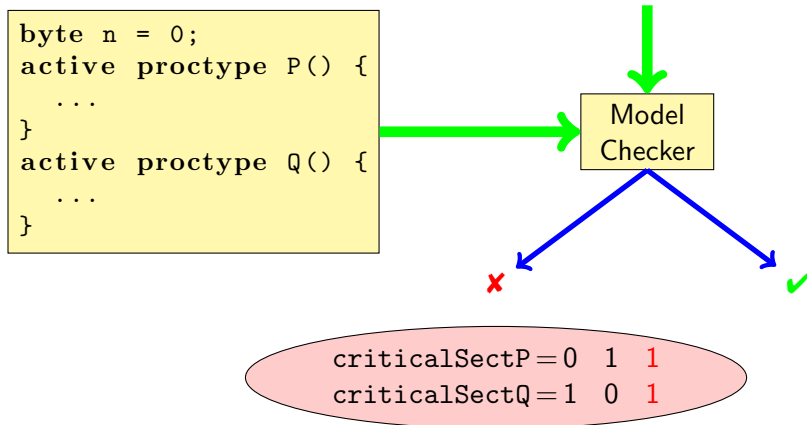
# Model Checking with SPIN

## System Model

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

## System Property

`[]!(criticalSectP && criticalSectQ)`



# Model Checking in Industry—Examples

- ▶ Hardware verification
  - ▶ Good match between limitations of technology and application
  - ▶ Intel, Motorola, AMD, . . .
- ▶ Software verification
  - ▶ Specialized software: control systems, protocols
  - ▶ Typically no direct checking of executable system, but of abstractions
  - ▶ Bell Labs, Microsoft

# A Major Case Study with SPIN

## Checking feature interaction for telephone call processing software

- ▶ Software for PathStar<sup>©</sup> server from Lucent Technologies
- ▶ Automated abstraction of unchanged C code into PROMELA
- ▶ Web interface, with SPIN as back-end, to:
  - ▶ determine properties (ca. 20 temporal formulas)
  - ▶ invoke verification runs
  - ▶ report error traces
- ▶ Finds error trace, reported as C execution trace
- ▶ Work farmed out to 16 computers, daily, overnight runs
- ▶ 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 with KeY

Java Code

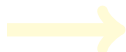
Formal specification



# Deductive Verification with KeY

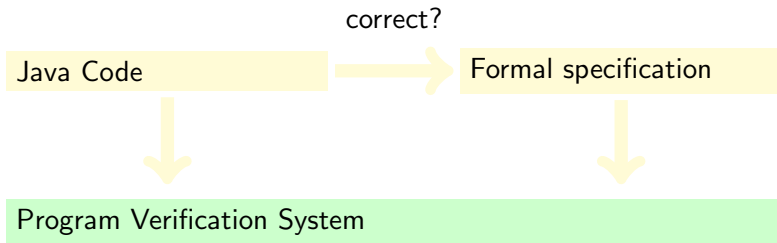
correct?

Java Code



Formal specification

# Deductive Verification with KeY



# Deductive Verification with KeY

correct ✓

Java Code

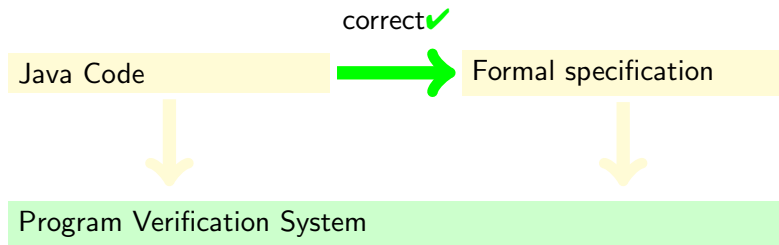


Formal specification



Program Verification System

# Deductive Verification with KeY



Proof rules establish relation “implementation conforms to specs”

**Computer support essential for verification of real programs**

`synchronized StringBuffer append(char c)`

- ▶ ca. 15.000 proof steps
- ▶ ca. 200 case distinctions
- ▶ Two human interactions, ca. 1 minute computing time

# Deductive Verification in Industry—Examples

- ▶ Hardware verification
  - ▶ For complex systems, mostly floating-point processors
  - ▶ Intel, Motorola, AMD, ...
- ▶ Software verification
  - ▶ Safety critical systems:
    - ▶ Paris driver-less metro (Meteor)
    - ▶ Emergency closing system in North Sea
  - ▶ Libraries
  - ▶ Implementations of Protocols

## Java Card 2.2.1 API Reference Implementation

- ▶ Reference implementation and full functional specification
- ▶ All Java Card 2.2.1 API classes and methods
  - ▶ 60 classes; ca. 5,000 LoC (250kB) source code
  - ▶ specification ca. 10,000 LoC
- ▶ Conformant to implementation on actual smart cards
- ▶ All methods fully verified against their spec
  - ▶ 293 proofs; 5–85,000 nodes
- ▶ Total effort several person months
- ▶ Most proofs fully automatic
- ▶ Main challenge: [getting specs right](#)

# Major Case Studies with KeY: Timsort

## Timsort

Hybrid sorting algorithm (insertion sort + merge sort) optimized for partially sorted arrays (typical for real-world data).

# Major Case Studies with KeY: Timsort

## Timsort

Hybrid sorting algorithm (insertion sort + merge sort) optimized for partially sorted arrays (typical for real-world data).

## Facts

- ▶ Designed by Tim Peters (for Python)
- ▶ Since Java 1.7 default algorithm for non-primitive arrays/collections



# Major Case Studies with KeY: Timsort

## Timsort

Hybrid sorting algorithm (insertion sort + merge sort) optimized for partially sorted arrays (typical for real-world data).

## Facts

- ▶ Designed by Tim Peters (for Python)
- ▶ Since Java 1.7 default algorithm for non-primitive arrays/collections

## Timsort is used in

- ▶ Java (standard library), used by Oracle
- ▶ Python (standard library), used by Google
- ▶ Android (standard library), used by Google
- ▶ ... and many more languages / frameworks!

# Timsort: People



▶ Tim Peters

# Timsort: People



- ▶ Tim Peters
- ▶ Sorting Algorithm Designer

# Timsort: People



- ▶ Tim Peters
- ▶ Sorting Algorithm Designer
- ▶ Python Guru

# Timsort: People



- ▶ Tim Peters
- ▶ Sorting Algorithm Designer
- ▶ Python Guru

- ▶ Stijn de Gouw

# Timsort: People



- ▶ Tim Peters
- ▶ Sorting Algorithm Designer
- ▶ Python Guru



- ▶ Stijn de Gouw
- ▶ Postman in the NL

# Timsort: People



- ▶ Tim Peters
- ▶ Sorting Algorithm Designer
- ▶ Python Guru



- ▶ Stijn de Gouw
- ▶ Postman in the NL
- ▶ Interested in sorting for professional reasons

# Timsort: People



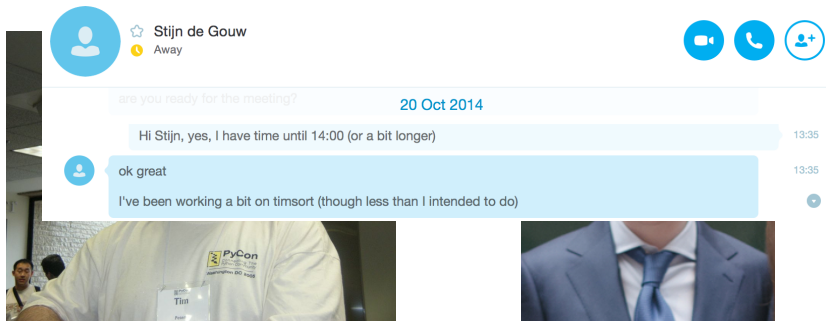
- ▶ Tim Peters
- ▶ Sorting Algorithm Designer
- ▶ Python Guru



- ▶ Stijn de Gouw
- ▶ Postman in the NL
- ▶ Interested in sorting for professional reasons
- ▶ PhD in Computer Science



# Timsort: People



- ▶ Tim Peters
- ▶ Sorting Algorithm Designer
- ▶ Python Guru

- ▶ Stijn de Gouw
- ▶ Postman in the NL
- ▶ Interested in sorting for professional reasons
- ▶ PhD in Computer Science

# Timsort: People

The screenshot shows a WhatsApp chat interface. At the top left is a profile card for 'Stijn de Gouw' with a star icon and 'Away' status. To the right are icons for video call, voice call, and a group of people. The chat history shows a message from Stijn: 'are you ready for the meeting?' dated '20 Oct 2014'. A reply from Richard says: 'Hi Stijn, yes, I have time until 14:00 (or a bit longer)' at 13:35. Stijn replies: 'ok great' at 13:35. Richard then says: 'I've been working a bit on timsort (though less than I intended to do)'. A horizontal separator indicates the date '27 Oct 2014'. Richard sends a long message at 08:52: 'morning richard', 'don't want to keep this from you, but please keep it to yourself for now... as you know I was working on proving correctness of timsort (the soring algorithm used in the jdk)', and 'I figured that the jdk was probably pretty thoroughly tested so went right ahead with specifying rather than debugging the algorithm... but I actually discovered a bug 😊'. Stijn replies at 09:07: 'Cool 😊' and 'Good morning!'. On the left side of the chat, there are three blue triangle navigation icons.

► PhD in Computer Science

# Timsort: People

The screenshot shows a WhatsApp chat interface. At the top left is a profile card for 'Stijn de Gouw' with a star icon and 'Away' status. To the right are icons for video call, voice call, and a group of people. The chat history shows a message from Stijn: 'are you ready for the meeting?' dated '20 Oct 2014'. A reply from Richard says: 'Hi Stijn, yes, I have time until 14:00 (or a bit longer)' at 13:35. Stijn replies: 'ok great' at 13:35. Richard then says: 'I've been working a bit on timsort (though less than I intended to do)'. A horizontal separator indicates the date '27 Oct 2014'. Richard's next message is: 'morning richard' at 08:52. The main body of the message reads: 'don't want to keep this from you, but please keep it to yourself for now... as you know I was working on proving correctness of timsort (the soring algorithm used in the jdk)'. The following line says: 'I figured that the jdk was probably pretty thoroughly tested so went right ahead with specifying rather than debugging the algorithm ... but I actually discovered a bug 😊'. The phrase 'but I actually discovered a bug' is highlighted with a red box. A reply from Stijn says: 'Cool 😊' at 09:07. The final message is: 'Good morning!'.

► PhD in Computer Science

# Major Case Studies with KeY

## Found Bug in Java Libraries' main Sorting Method using KeY

- ▶ `java.util.Collections.sort` and `java.util.Arrays.sort` implement **Timsort**
- ▶ KeY verification of **OpenJDK** implementation revealed *bug*.
- ▶ **Same bug** present in **Android** SDK, **Oracle's** JDK, **Phyton** library, **Haskell** library

# Major Case Studies with KeY

## Found Bug in Java Libraries' main Sorting Method using KeY

- ▶ `java.util.Collections.sort` and `java.util.Arrays.sort` implement **Timsort**
- ▶ KeY verification of **OpenJDK** implementation revealed *bug*.
- ▶ **Same bug** present in **Android** SDK, **Oracle's** JDK, **Phyton** library, **Haskell** library

## Verified Fix using KeY

- ▶ Fixing the implementation
- ▶ Verified new version with KeY

# Major Case Studies with KeY

## Found Bug in Java Libraries' main Sorting Method using KeY

- ▶ `java.util.Collections.sort` and `java.util.Arrays.sort` implement **Timsort**
- ▶ KeY verification of `Arrays.sort` revealed *bug*.
- ▶ **Same bug** previously found in `Python`, `Perl`, `Ruby`, `PHP`, `JavaScript`, `Java`, `C++`, `C`, `Python` library, `Haskell` and with corrected code shown in

## Verified

- ▶ Fixing
- ▶ Verified with KeY

Some researchers found an error in the logic of merge\_collapse, explained here, and with corrected code shown in

It should be fixed anyway, and their suggested fix looks good to me.

**Tim Peters via Python-Bugtracker**



# Tool Support is Essential

## Some Reasons for Using Tools

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



# Tool Support is Essential

## Some Reasons for Using Tools

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

## Tools used in this course:

**SPIN** to verify PROMELA programs against Temporal Logic specs

**SPIN web interface** developed for this course!

**JSPIN** A Java interface for SPIN

**KeY** to verify Java programs against contracts in JML

All are free and run on Windows/Unixes/Mac.

# Tool Support is Essential

## Some Reasons for Using Tools

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

## Tools used in this course:

**SPIN** to verify PROMELA programs against Temporal Logic specs

**SPIN web interface** developed for this course!

**JSPIN** A Java interface for SPIN

**KeY** to verify Java programs against contracts in JML

All are free and run on Windows/Unixes/Mac.

**Install first SPIN and JSPIN on your computer,  
or make sure the SPIN web interface works.**

# 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  
(Access to e-version via Chalmers Library)
- KeY** R. Hähnle: *Quo Vadis Formal Verification*. In: W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich, editors. Vol 10001 of *LNCS*, Springer, 2016  
(E-book at [link.springer.com](http://link.springer.com))
- SPIN** Gerard J. Holzmann: *A Verification Model of a Telephone Switch*. In: *The Spin Model Checker*, Chapter 14, Addison Wesley, 2004

# You will gain experience in ...

- ▶ Modelling, and modelling languages

# You will gain experience in ...

- ▶ Modelling, and modelling languages
- ▶ Specification, and specification languages

# You will gain experience in ...

- ▶ Modelling, and modelling languages
- ▶ Specification, and specification languages
- ▶ In depth analysis of possible system behaviour

# You will gain experience in ...

- ▶ Modelling, and modelling languages
- ▶ Specification, and specification languages
- ▶ In depth analysis of possible system behaviour
- ▶ Typical types of errors

# You will gain experience in ...

- ▶ Modelling, and modelling languages
- ▶ Specification, and specification languages
- ▶ In depth analysis of possible system behaviour
- ▶ Typical types of errors
- ▶ Reasoning about system (mis)behaviour



# You will gain experience in ...

- ▶ Modelling, and modelling languages
- ▶ Specification, and specification languages
- ▶ In depth analysis of possible system behaviour
- ▶ Typical types of errors
- ▶ Reasoning about system (mis)behaviour
- ▶ ...

# Learning Outcomes—Knowledge and Understanding

- ▶ Explain the potential and limitations of using logic based verification methods for assessing and improving software correctness
- ▶ Identify what can and what cannot be expressed by certain specification/modeling formalisms
- ▶ Identify what can and cannot be analyzed with certain logics and proof methods

# Learning Outcomes—Skills and Abilities

- ▶ Express safety and liveness properties of (concurrent) programs in a formal way
- ▶ Describe the basics of verifying safety and liveness properties via model checking
- ▶ Successfully employ tools which prove or disprove temporal properties
- ▶ Write formal specifications of object-oriented system units, using the concepts of method contracts and class invariants
- ▶ Describe how the connection between programs and formal specifications can be represented in a program logic
- ▶ Verify functional properties of simple Java programs with a verification tool

# Learning Outcomes—Judgment and Approach

- ▶ Judge and communicate the significance of correctness for software development
- ▶ Employ abstraction, modelling, and rigorous reasoning when approaching the development of correctly functioning software

This course is successor of course:

*Software Engineering using Formal Methods*

Major differences:

- ▶ Logic (first-order and temporal) is now prerequisite
  - ▶ Logic(s) introduced with higher speed
  - ▶ Some more advanced topics
- ▶ Oral (individual) examination

# 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  
(Access to e-version via Chalmers Library)
- KeY** R. Hähnle: *Quo Vadis Formal Verification*. In: W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich editors. Vol 10001 of *LNCS*, Springer, 2016  
(E-book at [link.springer.com](http://link.springer.com))
- SPIN** Gerard J. Holzmann: *A Verification Model of a Telephone Switch*. In: *The Spin Model Checker*, Chapter 14, Addison Wesley, 2004