Formal Methods for Software Development

Introduction

Wolfgang Ahrendt

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

03 September 2019

Course Team

Teachers

- Wolfgang Ahrendt (WA) examiner, lecturer
- Oskar Abrahamsson (OA) teaching assistant
- Andreas Lööw (AL) teaching assistant

Course Team

Teachers

- Wolfgang Ahrendt (WA) examiner, lecturer
- Oskar Abrahamsson (OA) teaching assistant
- Andreas Lööw (AL) teaching assistant

course assistant activities include:

- giving exercise classes
- correcting lab hand-ins
- student support via:
  - e-mail
  - meetings on e-mail request
    - Oskar, room 5453
    - Andreas, room 5461

Information Channels

Course Home Page

On Canvas, via Chalmers and GU.
Also used for online news and discussions.

Course Structure

Course Structure

Topic	# Lectures	# Exercises	Lab
Intro	1	✘	✘
Modeling & Model Checking with PROMELA & SPIN	6	3	✔
Specification & Verification with JML & KeY	6 (+1?)	3	✔

PROMELA & SPIN abstract programs, model checking, automated

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

. . . more on this later!

Lectures

Lectures

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

Lectures

Lectures

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

Lectures

Lectures

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

Exercises

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)

Passing Criteria

- Oral examination in exam week
- Two lab hand-ins
- (No written end-exam)
- Oral exam and labs can be passed separately

Oral Exam

- individual, oral examination

Oral Exam

- individual, oral examination
- 30 min per student

Oral Exam

- individual, oral examination
- 30 min per student
- slots between 28 October and 1 November

Oral Exam

- individual, oral examination
- 30 min per student
- slots between 28 October and 1 November
- see course page for more information

Labs

Labs

- 2 Lab handins: PROMELA/SPIN 04 Oct, JML/KeY 28 Oct
- 2 Lab Questions Sessions
- Submission via Fire, linked from course home page
- If submission is returned, roughly one week for correction

Labs

Labs

- 2 Lab handins: PROMELA/SPIN 04 Oct, JML/KeY 28 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![a]
  You pair up by either:
  1. talk to people
  2. post request via Canvas
  3. participate in pairing at first exercise session

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

[a] Only PhD students have to work alone.

Web Presence

- Canvas
- Web Pages (linked from Canvas)
- Fire System (for lab submissions)

Web Presence

- Canvas
- Web Pages (linked from Canvas)
- Fire System (for lab submissions)

(inspect course schedule)

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:

- Anna Brunzell
- David Hagerman Olzon
- Gabriel Lindeby
- Ramkumar Venkatesh
- Yonca Yunatci

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:

- Anna Brunzell
- David Hagerman Olzon
- Gabriel Lindeby
- Ramkumar Venkatesh
- Yonca Yunatci

GU students: please consider volunteering

Course Literature

- In part I, we partly use:

  Ben-Ari Mordechai Ben-Ari
  Principles of the Spin Model Checker
  Springer, 2008
  Ben-Ari received ACM award for outstanding contributions to CS education. Recommended by G. Holzmann. Excellent student text book.
  (E-book at `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`)

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
- Clouds
- Smart cards
- Smart devices
- Cars
- Blockchains
- ...

Motivation: Software Defects cause OMNIPRESENT Failures

Ubiquitous Computing results in Ubiquitous Failures

Software is almost everywhere:

- Mobiles
- Clouds
- Smart cards
- Smart devices
- Cars
- Blockchains
- ...

software/specification quality is a growing commercial and legal issue

Achieving Reliability in Engineering

Well-known strategies from mechanical and civil engineering

- Precise calculations (or accurate estimations) of forces, stress, etc.

Achieving Reliability in Engineering

Well-known strategies from mechanical and civil engineering

- Precise calculations (or accurate estimations) of forces, stress, etc.
- Redundancy ("make it a bit stronger than necessary")

Achieving Reliability in Engineering

Well-known strategies from mechanical and civil engineering

- Precise calculations (or accurate estimations) of forces, stress, etc.
- 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 (or accurate estimations) of forces, stress, etc.
- 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 (or accurate estimations) of forces, stress, etc.
- 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 is this So Difficult for Software?

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

Why is this So Difficult for Software?

- Software systems compute non-continuous functions.
  Single bit-flip may change behaviour completely.
- Redundancy as replication does not help against bugs.
  Redundant SW development only viable in special cases.

Why is this So Difficult for Software?

- Software systems compute non-continuous functions.
  Single bit-flip may change behaviour completely.
- Redundancy as replication does not help against bugs.
  Redundant SW development only viable in special cases.
- Insufficient separation of subsystems.
  Seemingly correct sub-systems may together behave incorrectly.

Why is this So Difficult for Software?

- Software systems compute non-continuous functions.
  Single bit-flip may change behaviour completely.
- Redundancy as replication does not 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 is this So Difficult for Software?

- Software systems compute non-continuous functions.
  Single bit-flip may change behaviour completely.
- Redundancy as replication does not 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 is this So Difficult for Software?

- Software systems compute non-continuous functions.
  Single bit-flip may change behaviour completely.
- Redundancy as replication does not 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 is this So Difficult for Software?

- Software systems compute non-continuous functions.
  Single bit-flip may change behaviour completely.
- Redundancy as replication does not 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
- Formalise 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 (e.g., green light mutual exclusion)
  - Liveness properties
    Something good will happen eventually

Specification — What a System Should Do

- Simple properties
  - Safety properties
    Something bad will never happen (e.g., green light 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 (e.g., green light 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 (e.g., green light 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 (e.g., green light 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 (e.g., green light 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 (e.g., green light 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

But ...

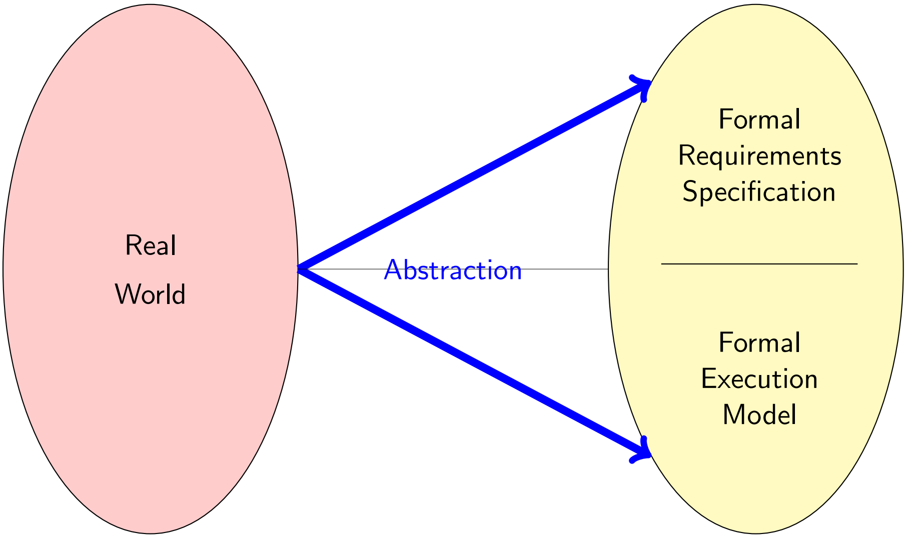
- Formal proof can replace (infinitely) many test cases
- Formal methods improve the quality of specs
  (even without formal verification)
- Formal methods guarantee specific properties of system (model)

A Fundamental Fact

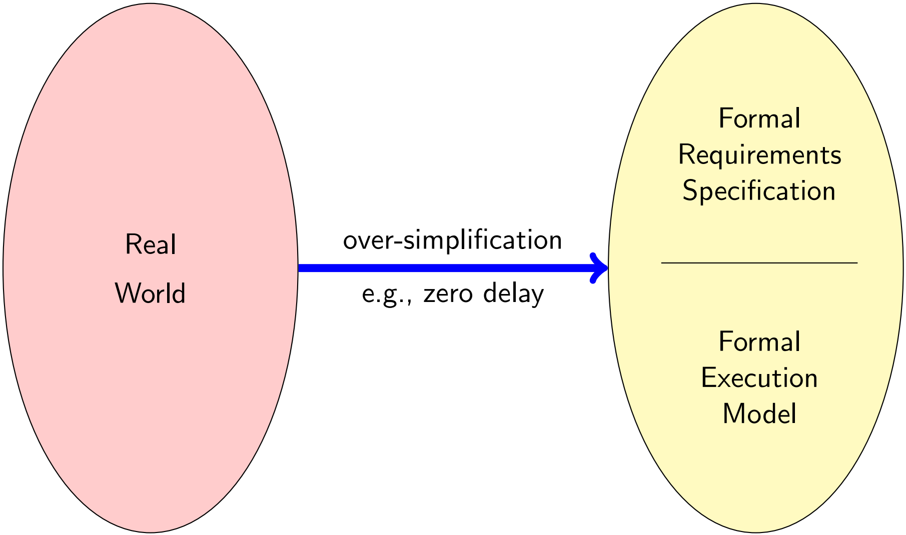
Formalisation of system requirements is hard

Let's see why . . .

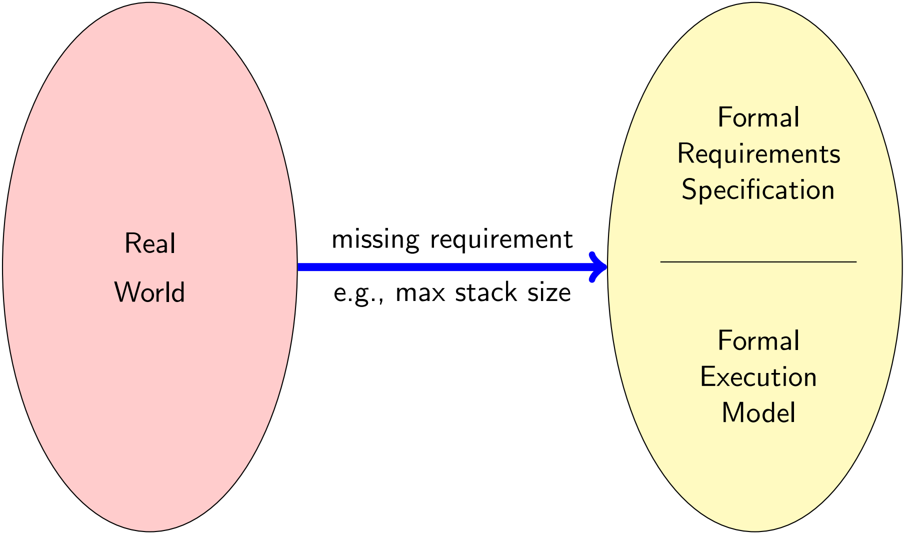
Difficulties in Creating Formal Models



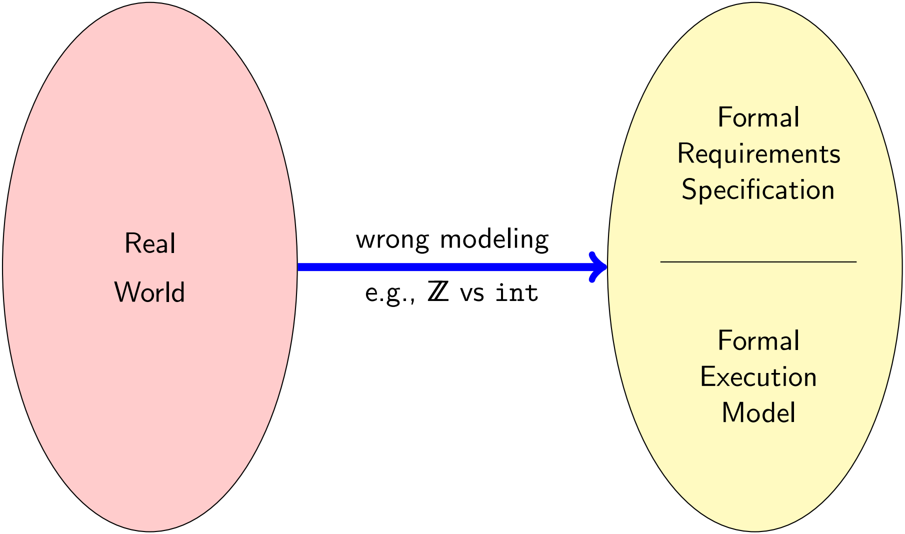
Difficulties in Creating Formal Models



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

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

Formalization Helps to Find Bugs in Specs

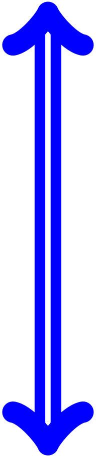
Errors in specifications are as common as errors in code,
but their discovery gives deep insights in (mis)conceptions of the system.

- 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)
  - Simplification, unfaithful modeling inevitable
  - Automated proofs are (in principle) possible
- 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

Model Checking, 1st part of course

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

Main Approaches

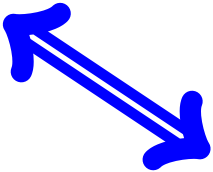
Model Checking, 1st part of course

…stract programs, Simple properties	Abstract programs, Complex properties
Concrete programs, Simple properties	Concrete programs… Complex proper…

Deductive Verification, 2nd part of course

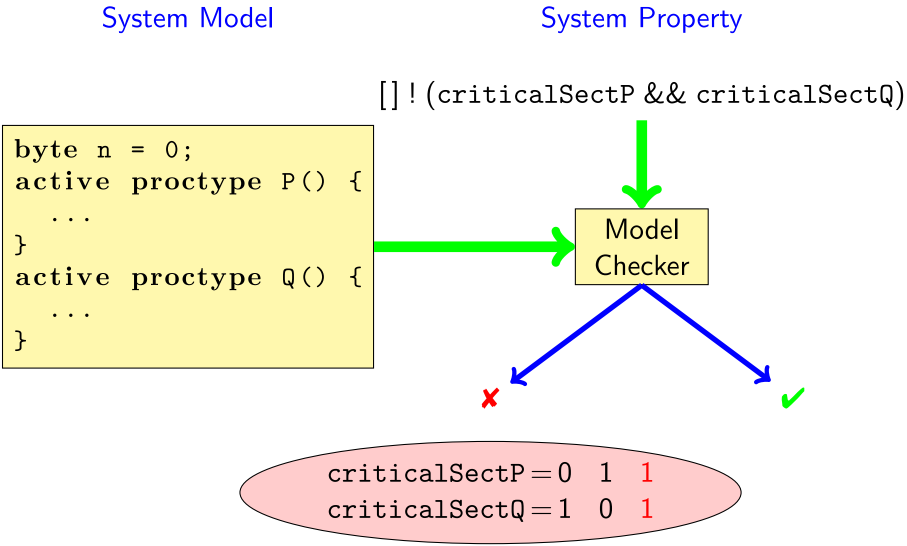
Proof Automation

- "Automated" Proof
  ("batch-mode")
  - No interaction (or lemmas) necessary
  - 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



Model Checking in Industry—Examples

- Hardware verification
  - Good match between limitations of methods 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© 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

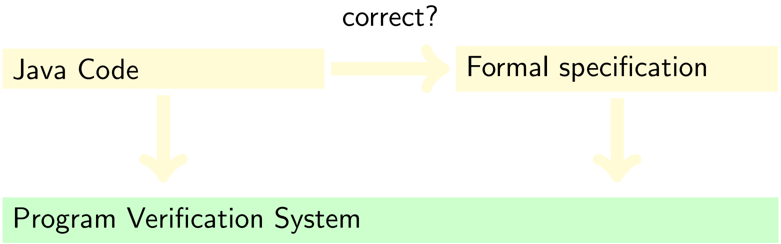
Formal specification

Deductive Verification with KeY

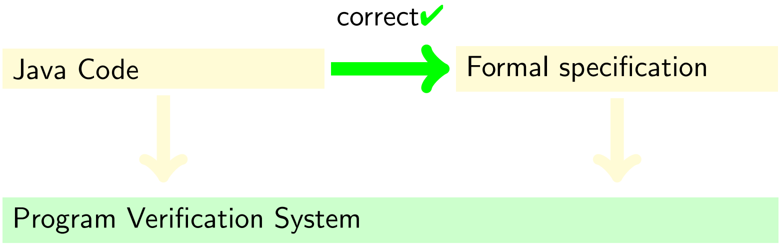
correct?

Java Code → Formal specification

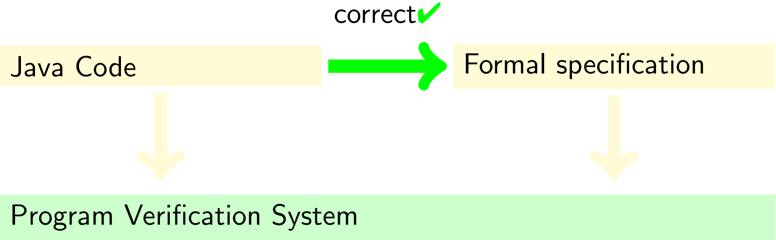
Deductive Verification with KeY



Deductive Verification with KeY



Deductive Verification with KeY



Proof rules establish relation “implementation conforms to specs”

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

Major Case Studies with KeY

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 libraries OpenJDK, 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
- Assistant Professor

Timsort: People

- Tim Peters
- Sorting Algorithm Designer
- Python Guru

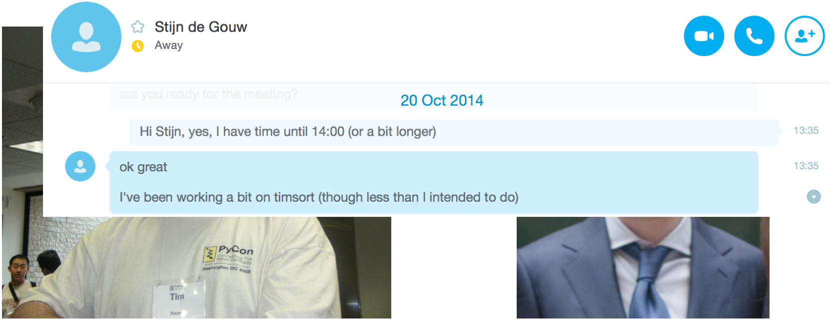
- Stijn de Gouw
- Assistant Professor
- Formerly postman in the NL

Timsort: People

- Tim Peters
- Sorting Algorithm Designer
- Python Guru

- Stijn de Gouw
- Assistant Professor
- Formerly postman in the NL
- Interested in sorting for professional reasons

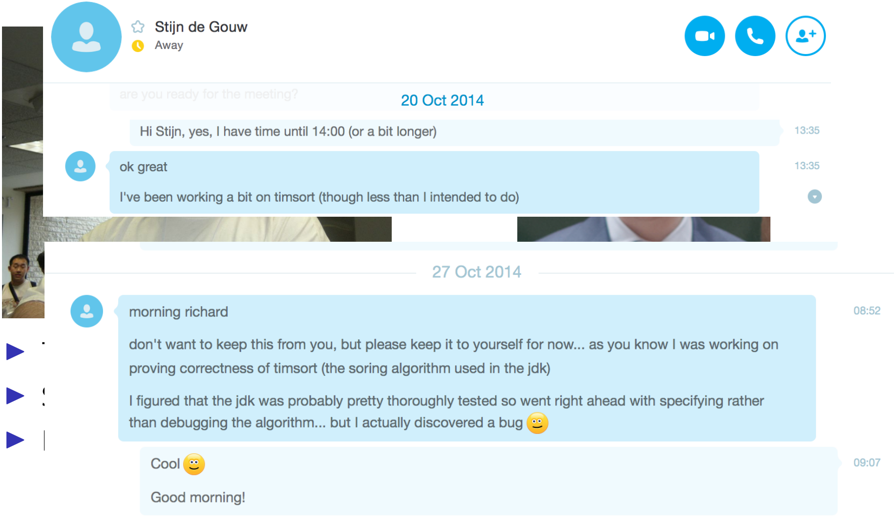
Timsort: People



- Tim Peters
- Sorting Algorithm Designer
- Python Guru

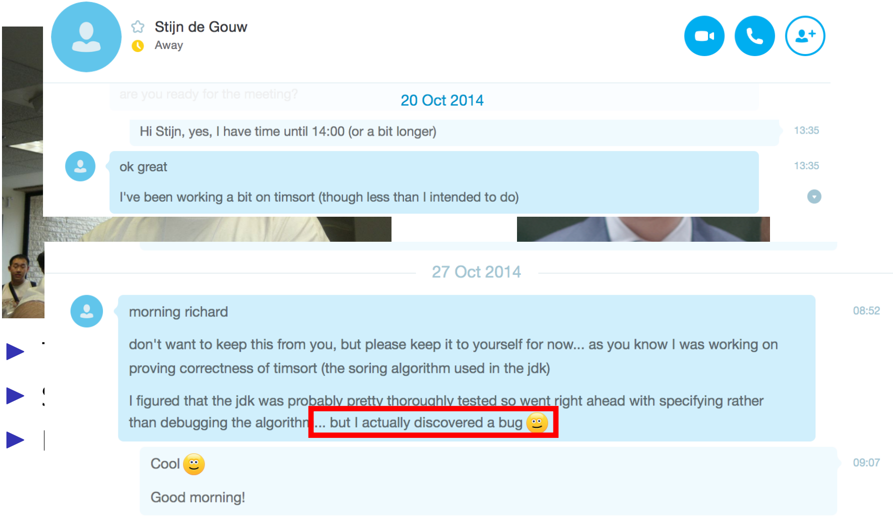
- Stijn de Gouw
- Assistant Professor
- Formerly postman in the NL
- Interested in sorting for professional reasons

Timsort: People



professional reasons

Timsort: People



professional reasons

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, 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, 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 So... Method using KeY

- java.util.Collections.s... util.Arrays.sort implement Timsort
- KeY verificatio... vealed *bug*.
- Same ... , Haskell library

Verified ...

- Fixing ...
- Verified ... ith 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**

Major Case Studies with KeY

Found Bug in Java Libraries' main So… Method using KeY

- `java.uti… ollections.s… util.Arrays.sort` impleme…
- KeY v… vealed *bug*.
- Same … , Haskell library

Verified …

- Fixing …
- Verified … ith KeY

Some r… d an error in the … xplained here, … own in … logic of m… and with co… heir sug- …

It should be fixed … gested fix looks goo…

Tim Peters via … acker

Congratulations to Stijn de Gouw et al. for finding and fixing a bug in TimSort using formal methods!

Joshua Bloch via Twitter

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 front-end 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 front-end 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.

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

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`)

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