Software Engineering using Formal Methods

Introduction

Wolfgang Ahrendt

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

3 September 2013

Course Team

Teachers

- Wolfgang Ahrendt (WA) examiner, lecturer
- Ramona Enache (RE) teaching assistant
- Bart van Delft (BvD) teaching assistant
- Moa Johansson (MJ) additional lecturer
- Laura Kovács (LK) additional lecturer

Course Team

Teachers

- Wolfgang Ahrendt (WA) examiner, lecturer
- Ramona Enache (RE) teaching assistant
- Bart van Delft (BvD) teaching assistant
- Moa Johansson (MJ) additional lecturer
- Laura Kovács (LK) additional lecturer

course assistant activities include:

- giving exercise classes
- correcting lab hand-ins
- student support via:
  - e-mail
  - meetings on e-mail request
    - Ramona, room 6105
    - Bart, room 5483

Organisational Stuff

Course Home Page

www.cse.chalmers.se/edu/course/TDA293/
Also linked from course portal

Google News Group

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

Passing Criteria

- Written exam 25 October, 2013; re-exam January (date t.b.a.)
- Two lab hand-ins
- Exam and labs can be passed separately

Course Structure

Course Structure

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

PROMELA & SPIN abstract programs, model checking, automated

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

. . . more on this later!

Lectures

Lectures

- You are welcome to ask questions during lecture
- Please respond to my questions
- Slides appear online shortly after each lecture

Exercises

Exercises

- One exercise web page each week (6 in total)
- Solutions discussed in exercise class on Friday
- Try to solve before coming to the class!
- Exercises not obligatory, but highly recommended
- Bring laptops if you have
  (ideally w. installed tools or browser interface working)

Labs

Labs

- 2 lab handins: PROMELA/SPIN 1 Oct, JML/KeY 22 Oct
- Submission via Fire, linked from course home page
- If submission is returned, roughly one week for correction

Labs

Labs

- 2 lab handins: PROMELA/SPIN 1 Oct, JML/KeY 22 Oct
- 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 to the Google group
  3. participate in pairing at first exercise session

  In case all that is not sufficient, contact Ramona by e-mail

[a] Only PhD student have to work alone.

Schedule

see course homepage

Course Evaluation

1. course evaluation group:
   - at least two students + teacher
   - one/two meetings during the course, one after
2. web questionnaire after the course

Course Evaluation

1. course evaluation group:
   - at least two students + teacher
   - one/two meetings during the course, one after
2. web questionnaire after the course

Students for evaluation group needed. Please volunteer.

Course Literature

- The Course Book:

  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.

- further reading:

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

  KeYbook B. Beckert, R. Hähnle, and P. Schmitt, editors. Verification of Object-Oriented Software: The KeY Approach, vol 4334 of *LNCS*. Springer, 2006.
  Chapters 1 and 10 only. Written by course developers.
  (Download via Chalmers library → E-books → Lecture Notes in Computer Science)

Connection to other Courses

Skills in object-oriented programing (like Java) assumed.

Connection to other Courses

Skills in object-oriented programing (like Java) assumed.

Knowledge corresponding to the following courses can further help:

- Concurrent Programming
- Finite Automata
- Testing, Debugging, and Verification
- Logic in Computer Science

Connection to other Courses

Skills in object-oriented programing (like Java) assumed.

Knowledge corresponding to the following courses can further help:

- Concurrent Programming
- Finite Automata
- Testing, Debugging, and Verification
- Logic in Computer Science

if you took any of those: nice

Connection to other Courses

Skills in object-oriented programing (like Java) assumed.

Knowledge corresponding to the following courses can further help:

- Concurrent Programming
- Finite Automata
- Testing, Debugging, and Verification
- Logic in Computer Science

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

Some well-known strategies from civil engineering

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

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")

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)

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

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
- 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 extreme 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 extreme cases
- No clear separation of subsystems
  Seemingly correct sub-systems cause global failures

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 extreme cases
- No clear separation of subsystems
  Seemingly correct sub-systems cause global failures
- 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 extreme cases
- No clear separation of subsystems
  Seemingly correct sub-systems cause global failures
- 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 extreme cases
- No clear separation of subsystems
  Seemingly correct sub-systems cause global failures
- 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 extreme cases
- No clear separation of subsystems
  Seemingly correct sub-systems cause global failures
- 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, particularly 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")

- design (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")

- design (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 used in system design and development
- 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
  - mechanical proof that implementation satisfies requirements, or
  - exhaustive search for failing scenario

What are Formal Methods for

- Complement other analysis and design methods
- Good at finding bugs
  (in code and specification)
- Reduce overall development time (testing/maintenance included)
- *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
- Good at finding bugs
  (in code and specification)
- Reduce overall development time (testing/maintenance included)
- *Ensure* certain properties of the system model
- Should ideally be as automated as possible

and

- Training in Formal Methods increases high quality development skills

Formal Methods: Relation with Testing

- Run the system at chosen inputs and observe its behaviour
  - Randomly chosen (no guarantees, but can find bugs)
  - Intelligently chosen (by hand: expensive!)
  - Automatically chosen (need formalised specification)
- What about other inputs? (test coverage)
- How to interpret output? (test oracle)

Formal Methods: Relation with Testing

- Run the system at chosen inputs and observe its behaviour
  - Randomly chosen (no guarantees, but can find bugs)
  - Intelligently chosen (by hand: expensive!)
  - Automatically chosen (need formalised specification)
- What about other inputs? (test coverage)
- How to interpret output? (test oracle)

Challenges can be addressed by/require formal methods

- Automated (model-based) test case generation
- Not the focus of this course, but see *Testing, Debugging, and Verification* (TDA566/DIT082)

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
  - Runtime, 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
  - Runtime, memory, usability, ...
- Full behavioural specification
  - Code satisfies a contract that describes its functionality

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
  - Runtime, memory, usability, ...
- Full behavioural specification
  - Code satisfies a contract that describes its functionality
  - 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
  - Runtime, memory, usability, ...
- Full behavioural specification
  - Code satisfies a contract that describes its functionality
  - 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
  - Runtime, memory, usability, ...
- Full behavioural specification
  - Code satisfies a contract that describes its functionality
  - 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 entirely
  - Formal methods work on models, on source code, or, at most, on bytecode level
  - Many non-formalizable properties
- 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 . . .

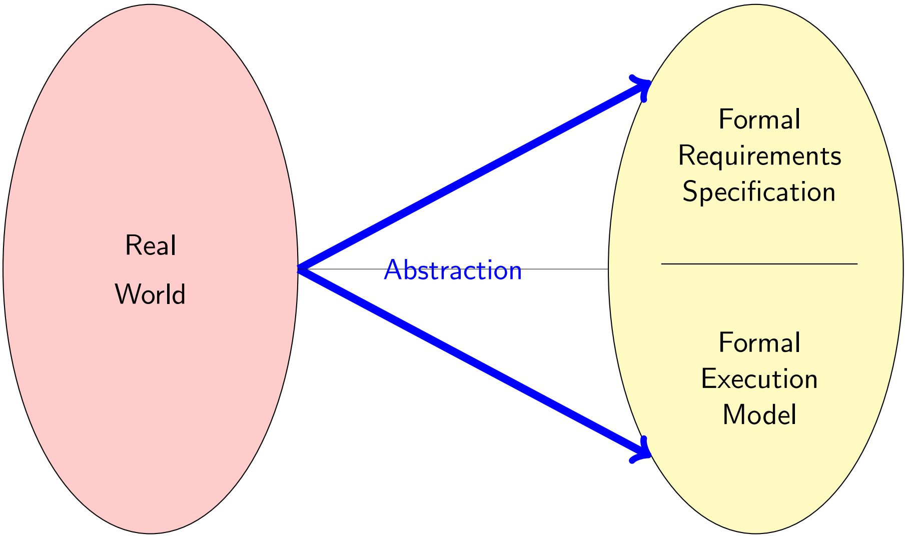
- Formal proof can replace (infinitely) many test cases
- Formal methods improve the quality of specs
  (even without formal verification)
- Formal methods guarantee specific properties of system model

A Fundamental Fact

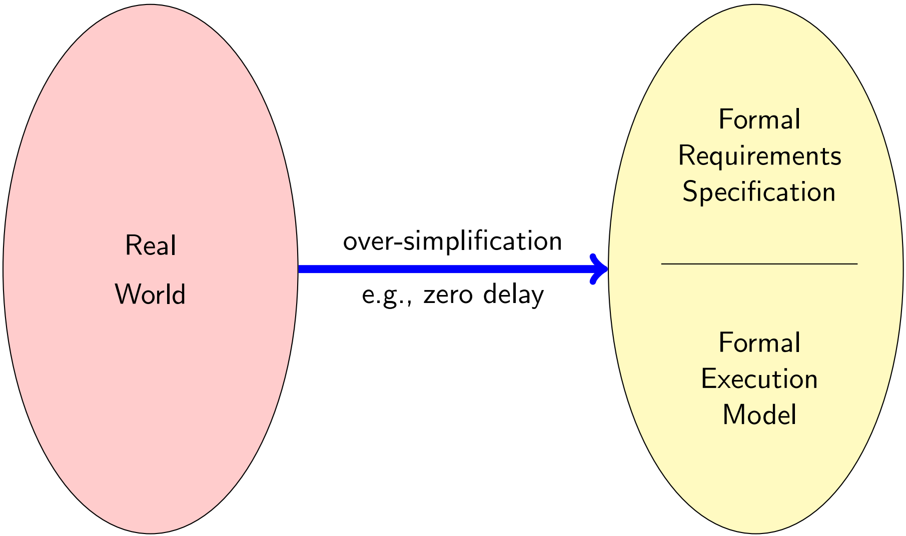
Formalisation of system requirements is hard

Let's see why . . .

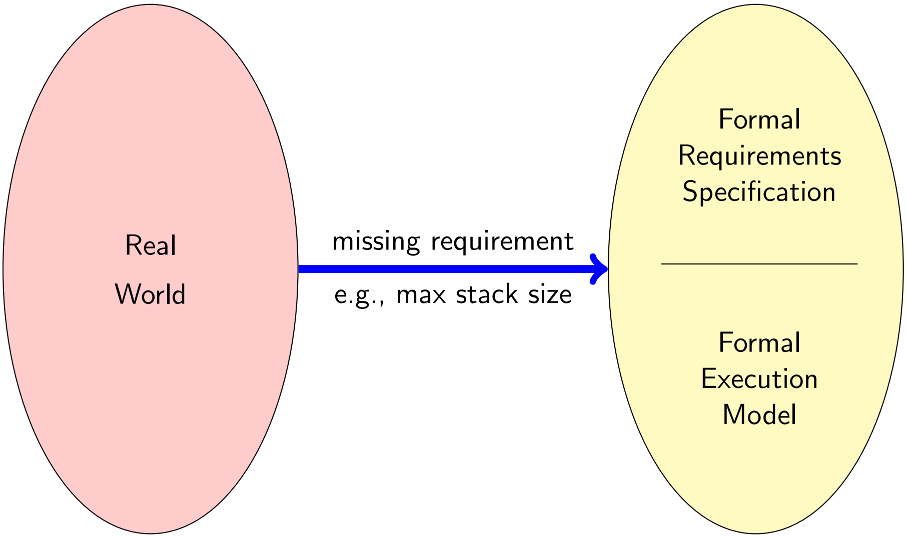
Difficulties in Creating Formal Models



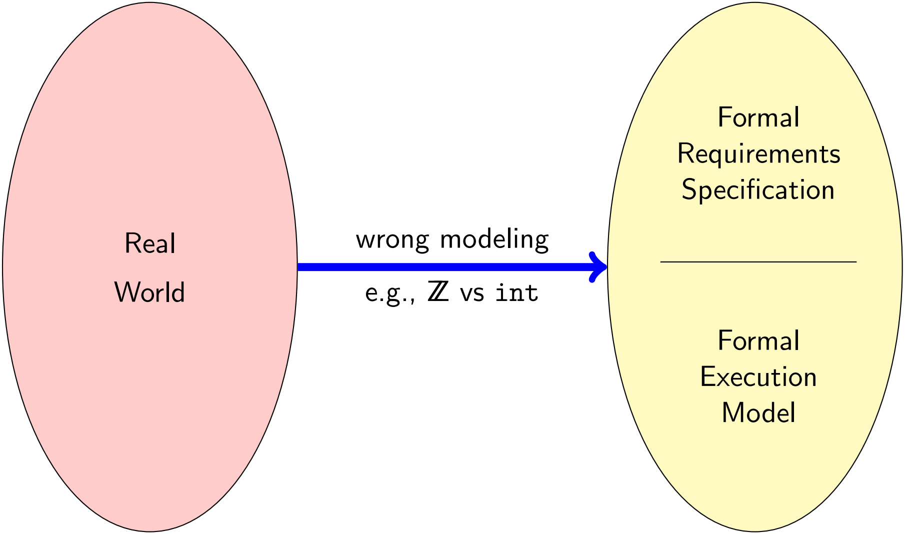
Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Formalization Helps to Find Bugs in Specs

- 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

Formalization Helps to Find Bugs in Specs

- 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

Errors in specifications are at least as common as errors in code

Formalization Helps to Find Bugs in Specs

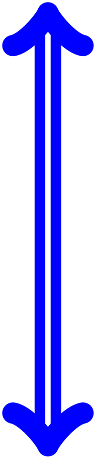
- Wellformedness and consistency of formal specs partly machine-checkable
- Declared signature (symbols) helps to spot incomplete specs
- Failed verification of implementation against spec gives feedback on erroneous formalization

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 properties of systems can be hard

Level of System (Implementation) Description



- Abstract level
  - Finitely many states (finite datatypes)
  - Automated proofs are (in principle) possible
  - Simplification, unfaithful modeling inevitable

- Concrete level
  - Infinite datatypes
    (pointer chains, dynamic arrays, streams)
  - Complex datatypes and control structures,
    general programs
  - Realistic programming model (e.g., Java)
  - Automated proofs (in general) 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 domains
  - High precision, tight modeling
  - Automated proofs (in general) 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

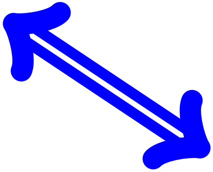
SPIN
1st part
of course

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"



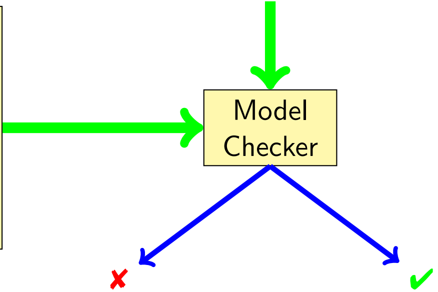
- "Semi-Automated" Proof
  ("interactive")
  - Interaction (or lemmas) may be required
  - Need certain knowledge of tool internals
    Intermediate inspection can help
  - Proof is checked by tool

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, ...
- Software verification
  - Specialized software: control systems, protocols
  - Typically no checking of executable source code, but of abstractions
  - Bell Labs, 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
- 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

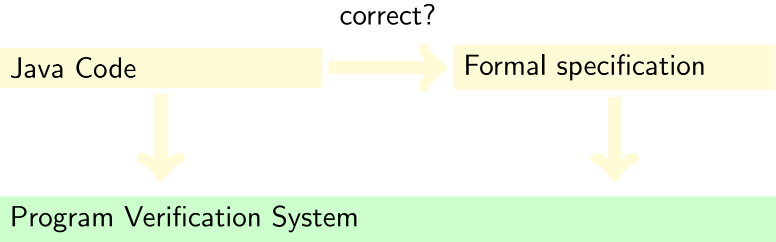
Formal specification

Deductive Verification with KeY

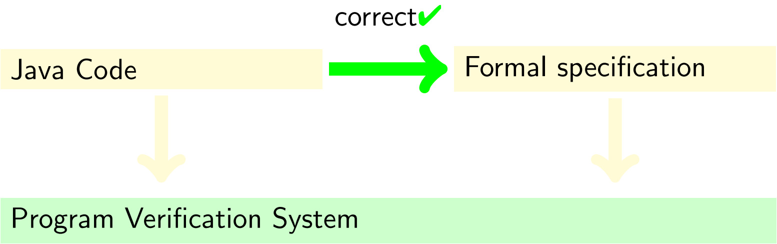
correct?

Java Code → Formal specification

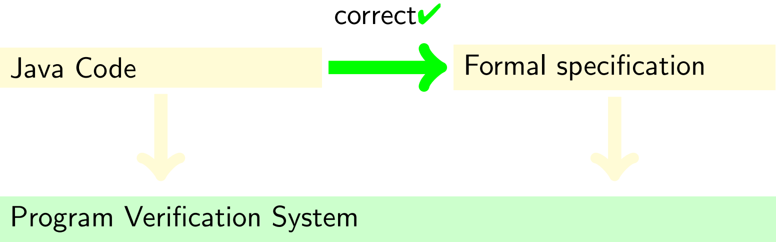
Deductive Verification with KeY



Deductive Verification with KeY



Deductive Verification with KeY



Proof rules establish relation “implementation conforms to specs”

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

- 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
  - Implementations of Protocols

A Major Case Study with KeY

Mondex Electronic Purse

- Specified and implemented by NatWest ca. 1996
- Original formal specs in **Z** and proofs by hand
- Reformulated specs in JML, implementation in Java Card
- Can be run on actual smart cards
- Full functional verification
- Total effort 4 person months
- With correct invariants: proofs fully automated
- Main challenge: loop invariants, getting specs right

Tool Support is Essential

Some Reasons for Using Tools

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

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 are Used in this Course in Both Parts:

SPIN to verify PROMELA programs against Temporal Logic specs

SPIN **web interface** Developped by Bart 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 clerical errors, etc.
- Cope with large/complex programs
- Make verification certifiable

Tools are Used in this Course in Both Parts:

SPIN to verify PROMELA programs against Temporal Logic specs

SPIN **web interface** Developped by Bart 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: A New Look at Formal Methods for Software Construction. In: B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, pp 1–18, vol 4334 of *LNCS*. Springer, 2006.
(Access to e-version via Chalmers Library)

SPIN Gerard J. Holzmann: A Verification Model of a Telephone Switch. In: *The Spin Model Checker*, pp 299–324, 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
- ...