Formal Specification and Verification

Introduction

Bernhard Beckert

Based on a lecture by Wolfgang Ahrendt and Reiner Hähnle at Chalmers University, Göteborg

Organisational Stuff

Course Home Page

`http://www.uni-koblenz.de/~beckert/Lehre/Formale-Verifikation/`

Also linked from KLIPS

Passing Criteria

- Written or oral exam
- Two lab hand-ins

Organisational Stuff: Course Structure

Course Structure

- Intro
- Propositional & Temporal Logic
- First-Order Logic
- Modeling & Verification with PROMELA & SPIN
- Modeling & Verification with JML & KeY

PROMELA/SPIN	abstract programs, model checking, automatic
JML/KeY	executable Java, deductive verification, semi-automatic

... more on this later!

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, Mars Sojourner
- London Ambulance Dispatch System
- Denver Airport Luggage Handling System
- Pentium-Bug
- NEDAP Voting Computer Attack

Motivation: Software Defects cause OMNIPRESENT Failures

Ubiquitous Computing results in Ubiquitous Failures

Software these days is inside just about anything:

- Mobiles
- Smart devices
- Smart cards
- Cars

Motivation:
Software Defects cause OMNIPRESENT Failures

Ubiquitous Computing results in Ubiquitous Failures

Software these days is inside just about anything:

- Mobiles
- Smart devices
- Smart cards
- Cars

Motivation:
Software Defects cause OMNIPRESENT Failures

Ubiquitous Computing results in Ubiquitous Failures

Software these days is inside just about anything:

- Mobiles
- Smart devices
- Smart cards
- Cars

$\Rightarrow$ software—and specification—quality is a growing legal issue

Achieving Reliability in Engineering

Some well-known strategies from civil engineering

- Precise calculations/estimations of forces, stress, etc.
- Hardware redundancy (“make it a bit stronger than necessary”)
- Robust design (single fault not catastrophic)
- Clear separation of subsystems
  Any air plane flies with dozens of known and minor defects
- Design follows patterns that are proven to work

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn't help against bugs
  Redundant SW development only viable in extreme cases
- No clear separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Most SW engineers untrained to address correctness
- Cost efficiency favoured over reliability
- Design practice 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, ...)

Testing against inherent SW errors ("bugs")

- design test configurations that hopefully are representative and
- ensure that the system behaves intentionally on them

Testing against external faults

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

Limitations of Testing

- Testing shows the presence of errors, in general not their absence (exhaustive testing viable only for trivial systems)
- Representativeness of test cases/injected faults subjective
  How to test for the unexpected? Rare cases?
- Testing is labor intensive, hence expensive

Formal Methods: The Scenario

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

Formal Methods: The Vision

- Complement other analysis and design methods
- Are good at finding bugs
  (in code and specification)
- Reduce development (and test) time
- Can *ensure* certain properties of the system model
- Should ideally be as automatic as possible

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 formalized spec)
- What about other inputs? (test coverage)
- What about the observation? (test oracle)

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
  - Program equivalence
  - Refinement relation

The Main Point of Formal Methods is Not

- To show "correctness" of entire systems
  What IS correctness? Always go for specific properties!
- To replace testing entirely
  - Formal methods work on models, on source code, or, at most, on bytecode level
  - Many non-formalizable properties
- To replace good design practices

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 can be used in automatic test case generation
- Formal methods improve the quality of specs
  (even without formal verification)
- Formal methods guarantee specific properties of a specific system model

Formal Methods Aim at:

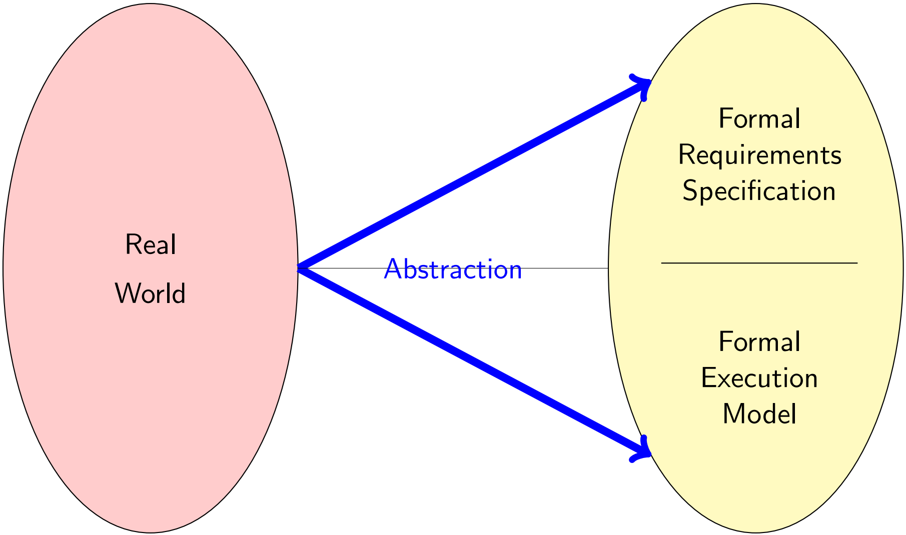
- Saving money
  Intel Pentium bug
  Smart cards in banking
- Saving time
  otherwise spent on heavy testing and maintenance
- More complex products
  Modern μ-processors
  Fault tolerant software
- Saving human lives
  Avionics, X-by-wire
  Washing machine

A Fundamental Fact

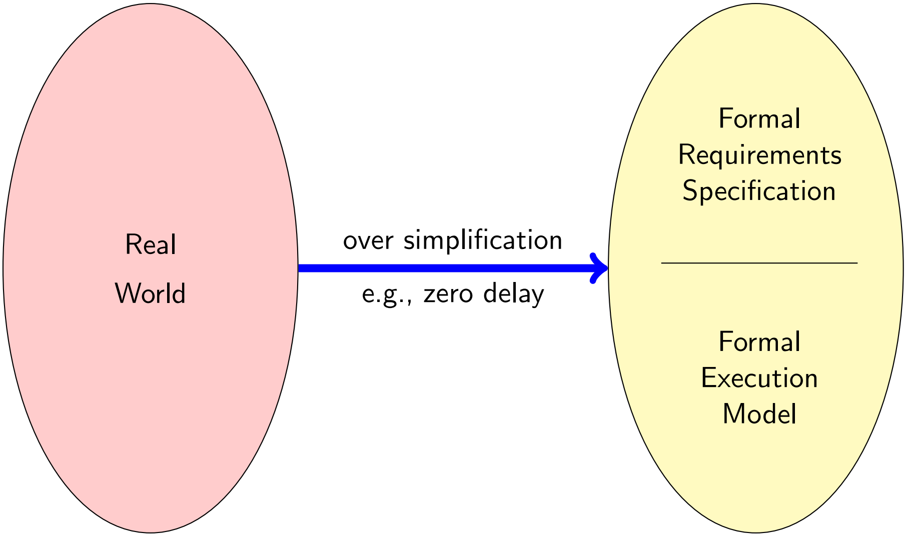
Formalisation of system requirements is hard

Let's see why . . .

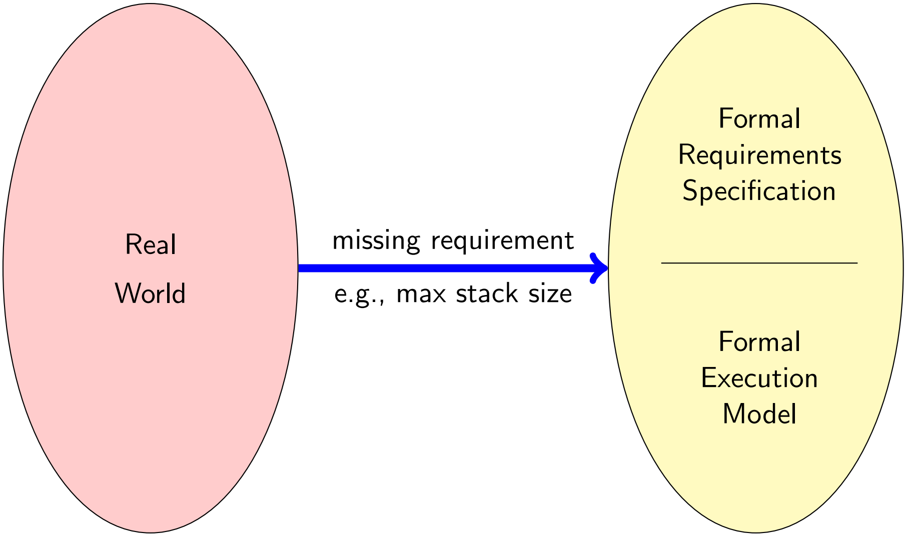
Difficulties in Creating Formal Models



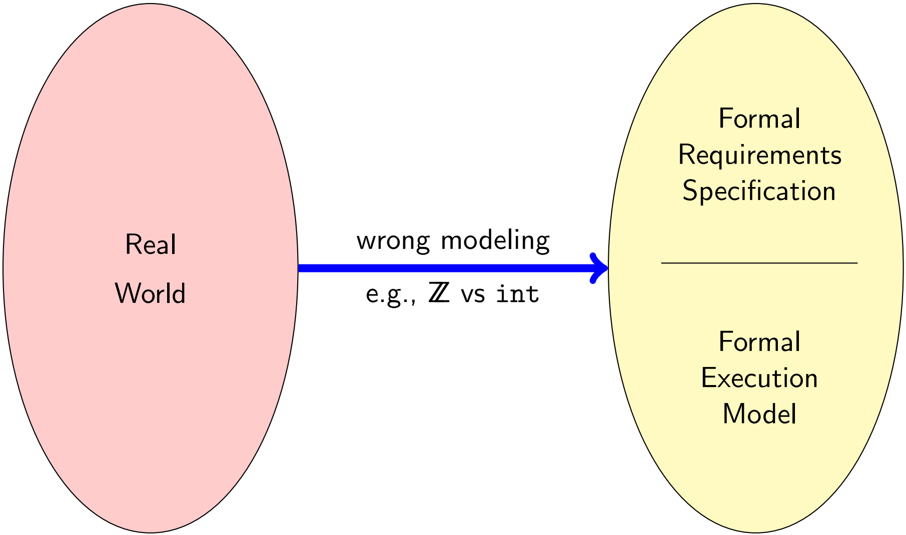
Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Difficulties in Creating Formal Models



Formalization Helps to Find Bugs in Specs

- Wellformedness and consistency of formal specs 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 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)
  - Tedious to program, worse to maintain
  - Over-simplification, unfaithful modeling inevitable
  - Automatic proofs are (in principle) possible
- Concrete level
  - Infinite datatypes
    (pointer chains, dynamic arrays, streams)
  - Complex datatypes and control structures, general programs
  - Realistic programming model (e.g., Java)
  - Automatic proofs (in general) impossible!

Expressiveness of Specification

- Simple
  - Simple or general properties
  - Finitely many case distinctions
  - Approximation, low precision
  - Automatic proofs are (in principle) possible
- Complex
  - Full behavioural specification
  - Quantification over infinite domains
  - High precision, tight modeling
  - Automatic proofs (in general) impossible!

Main Approaches

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

Main Approaches

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

SPIN
2nd part
of course

Main Approaches

KeY
1st part
of course

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

SPIN
2nd part
of course

Proof Automation

- "Automatic" Proof
  Perhaps better called "batch-mode" proof
  - No interaction during verification necessary
  - Proof may fail or result inconclusive
    Tuning of tool parameters necessary
  - Formal specification still "by hand"
- "Semi-Automatic" Proof
  Perhaps better called "interactive" proof
  - Interaction may be required during proof
  - Need certain knowledge of tool internals
    Intermediate inspection can be helpful, too
  - Proof is checked by tool

Model Checking

System Model

System Property

`[] ! (criticalSectP && criticalSectQ)`

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

Model Checker

✘ ✔

`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 sourse code, but of abstraction
  - Bell Labs, Ericsson, Microsoft

Deductive Verification

Java Code

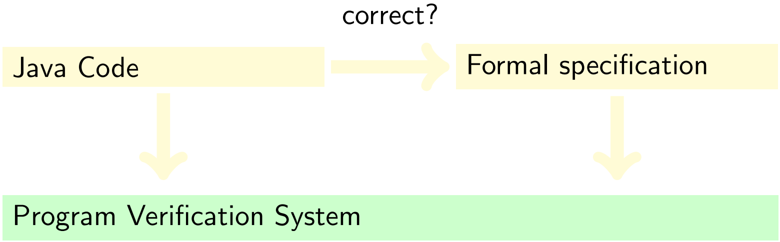
Formal specification

Deductive Verification

correct?

Java Code → Formal specification

Deductive Verification



Deductive Verification

correct✔

Java Code → Formal specification

Program Verification System

Deductive Verification

correct✔

Java Code → Formal specification

Program Verification System

Proof rules establish relation "implementation conforms to specs"

Computer support essential for verification of real programs

```
synchronized java.lang.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 driverless metro (Meteor)
    - Emergency closing system in North Sea
  - Libraries
  - Implementations of Protocols

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

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 card
- Full functional verification
- Total effort 4 person months
- With correct invariants: proofs fully automatic
- 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

Tools are Used in this Course in Both Parts:

SPIN to verify PROMELA programs against Temporal Logic specs

jSPIN as a Java interface for SPIN

KeY to verify Java (Card) programs against contracts in JML

Both are free and run on Windows/Unixes/Mac
(will be available via course webpage)
Install them on your computer!

Future Trends

- Design for formal verification
- Combining semi-automatic methods with SAT, theorem provers
- Combining static analysis of programs
  with automatic methods and with theorem provers
- Combining test and formal verification
- Integration of formal methods into SW development process
- Integration of formal method tools into CASE tools
- Applying formal methods to dependable systems design
- Scaling formal methods to open, distributed, adaptive systems

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.

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

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.

Summary

Formal Methods ...

- Are (more and more) used in practice
- Can shorten development time
- Can push the limits of feasible complexity
- Can increase quality/reliability of systems dramatically

Summary

Formal Methods . . .

- Are (more and more) used in practice
- Can shorten development time
- Can push the limits of feasible complexity
- Can increase quality/reliability of systems dramatically

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

You will gain experience in ...

... more than Formal Methods (in the strict sense)

- modelling, and modelling languages
- specification, and specification languages
- in depth analysis of possible system behaviour
- typical types of errors
- reasoning about system (mis)behaviour
- ...