Formal Methods for Software Development

Introduction to Model Checking

Srinivas Pinisetty [1]

IIT Bhubaneswar

April 8, 2024

[1]Acknowledgement: Based on material from Prof. Wolfgang Ahrendt.

Motivation: Software Defects cause BIG failures

Software is used widely in many applications where a bug in the system can cause large damage:

- Economically critical systems: e-commerce systems, Internet, microprocessors, etc.
- Safety critical systems: airplane control systems, medical care, train signalling systems, air traffic control, power plants, etc.

Price of software defects

Two very expensive software bugs:

- Intel Pentium FDIV bug (1994, approximately $500 million)
- Ariane 5 floating point overflow (1996, approximately $500 million)

Pentium FDIV — software bug in HW

Image © CPU-World.com

The floating point division algorithm uses an array of constants with 1066 elements. However, only 1061 elements of the array were correctly initialised.

Ariane 5

Exploded 37 seconds after takeoff - the reason was an overflow in a conversion of a 64 bit floating point number into a 16 bit integer.

Motivation:
Software Defects cause OMNIPRESENT 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.
- 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.
- 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, . . .)

How to Ensure Software Correctness/Compliance?

A central strategy: testing
(others: SW processes, reviews, . . .)

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, . . .)

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), e.g., by simulation
- 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
- Non-functional properties
  - Execution time, memory, usability, ...
- Full behavioural specification
  - Code functionality described by contracts (in particular for efficient, i.e., redundant, data representations)

The Main Point of Formal Methods is Not

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

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

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

e.g., SPIN

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

Main Approaches

e.g., SPIN

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

e.g., JML
+ KeY
Dafny..

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

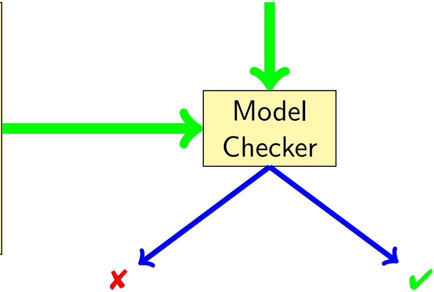
Model Checking (with SPIN)

System Model

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

System Property

`[] ! (criticalSectP && criticalSectQ`



criticalSectP =	0	1	1
criticalSectQ =	1	0	1

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© 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

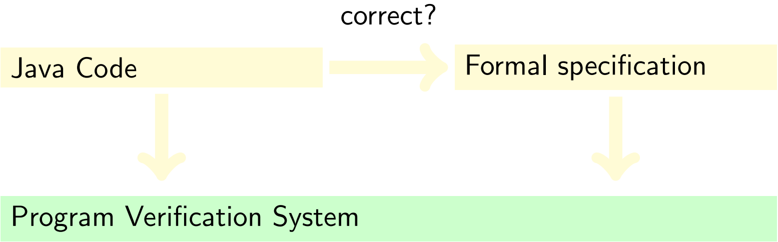
Deductive Verification with KeY

correct?

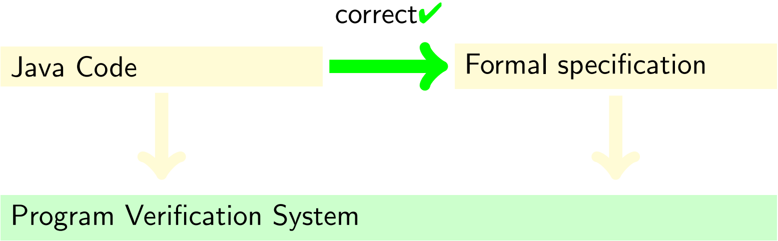
Java Code

Formal specification

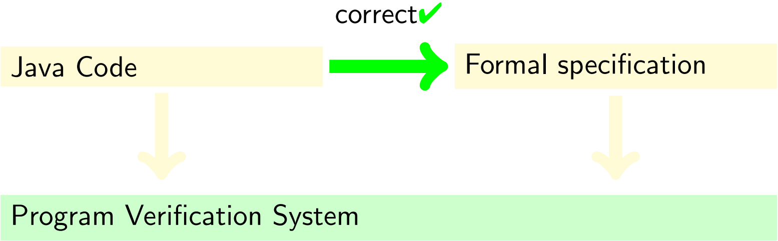
Deductive Verification with KeY



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

Main topics of the module

The rest of the module will concentrate especially on:

- modelling of systems,
- specifying properties, and
- using model checkers to verify them,

Theoretical background of model checking (anyone interested can contact me).

Additional Literature

BaierKatoen	Christel Baier, Joost-Pieter Katoen Principles of Model Checking MIT Press, 2008
HuthRyan	Michael Huth, Mark Ryan Logic in Computer Science
Holzmann	Gerard J. Holzmann The Spin Model Checker Addison Wesley, 2004
Ben-Ari	Mordechai Ben-Ari Principles of the Spin Model Checker Springer, 2008