15-819M: Data, Code, Decisions
01: Introduction

André Platzer

aplatzer@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, PA
Outline

1. Organisation
2. Motivation
3. Formalisation
Outline

1. Organisation

2. Motivation

3. Formalisation
Organisational Stuff

Course Web

http://symbolaris.com/course/dcd.html

Passing Criteria

- Homework assignments
- Midterm
- Project (e.g., practical programming, applications, theory, seminar)

Homework assignments / Exercise

- Two weeks for homework assignments
- Assignments include practical part
Course Structure

- Introduction
- Propositional Logic
- First-Order Logic
- Modeling & Verification with JML & KeY
- Decision Procedures
- Real arithmetic
- Integer arithmetic
Suggested Course Literature


*Author received ACM award for outstanding Contributions to CS Education.*


Acknowledgment

Slides based on Reiner Hähnle’s course “Software Engineering using Formal Methods” at Chalmers University
Outline

1. Organisation
2. Motivation
3. Formalisation
Motivation: Software Defects cause Big Failures

Tiny faults in technical systems can have catastrophic consequences

Especially for software bugs

- Ariane 5
- Mars Climate Orbiter, Mars Sojourner
- Denver Airport Luggage Handling System
- Pentium-Bug
- USS Yorktown
- F-22 jet crash
Reliability means in 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 **discontinuous** functions
  Single bit-flip may change behavior 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 favored over reliability
- Design practice for reliable software in **immature** state
  for complex systems
How to Ensure Software Correctness/Compliance?

A Central Strategy: **Testing**
(others: SW processes, reviews, libraries, sandboxing, . . .)

**Testing against inherent SW errors / bugs**
- design test configurations that are hopefully representative and
- ensure that the system behaves as intended on these tests

**Testing against external faults**
- Inject faults (memory, communication) by simulation or radiation
A Central Strategy: Testing
(others: SW processes, reviews, libraries, sandboxing, . . .)

Testing against inherent SW errors / bugs
- design test configurations that are hopefully representative and
- ensure that the system behaves as intended on these tests

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

Issues: In both cases spot error by
How to Ensure Software Correctness/Compliance?

A Central Strategy: Testing
(others: SW processes, reviews, libraries, sandboxing, . . .)

**Testing against inherent SW errors / bugs**
- design test configurations that are hopefully representative and
- ensure that the system behaves as intended on these tests

**Testing against external faults**
- Inject faults (memory, communication) by simulation or radiation

**Issues: In both cases spot error by**
- Visual inspection of output
How to Ensure Software Correctness/Compliance?

A Central Strategy: **Testing**
(others: SW processes, reviews, libraries, sandboxing, ...)

**Testing against inherent SW errors / bugs**
- design test configurations that are hopefully representative and
- ensure that the system behaves as intended on these tests

**Testing against external faults**
- Inject faults (memory, communication) by simulation or radiation

**Issues:** In both cases spot error by
- Visual inspection of output
- Expected result (input 9, then output 3)
## How to Ensure Software Correctness/Compliance?

### A Central Strategy: Testing
(others: SW processes, reviews, libraries, sandboxing, ...)

<table>
<thead>
<tr>
<th><strong>Testing against inherent SW errors / bugs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• design test configurations that are hopefully representative and</td>
</tr>
<tr>
<td>• ensure that the system behaves as intended on these tests</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Testing against external faults</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Inject faults (memory, communication) by simulation or radiation</td>
</tr>
</tbody>
</table>

### Issues: In both cases spot error by

| • Visual inspection of output |
| • Expected result (input 9, then output 3) |
| • Test procedure for output (output $o = \sqrt{i}$ good if $o^2 = i$) |
How to Ensure Software Correctness/Compliance?

A Central Strategy: Testing
(others: SW processes, reviews, libraries, sandboxing, . . .)

**Testing against inherent SW errors / bugs**
- design test configurations that are hopefully representative and
- ensure that the system behaves as intended on these tests

**Testing against external faults**
- Inject faults (memory, communication) by simulation or radiation

**Issues: In both cases spot error by**
- Visual inspection of output
- Expected result (input 9, then output 3)
- Test procedure for output \( o = \sqrt{i} \) good if \( o^2 = i \)
- Assertions throughout the code (it’s getting formal)
## How to Ensure Software Correctness/Compliance?

A Central Strategy: **Testing**  
(others: SW processes, reviews, libraries, sandboxing, ...)

### Testing against inherent SW errors / bugs
- design test configurations that are hopefully representative and  
- ensure that the system behaves as intended on these tests

### Testing against external faults
- Inject faults (memory, communication) by simulation or radiation

### Issues: In both cases spot error by
- Visual inspection of output
- Expected result (input 9, then output 3)
- Test procedure for output (output $o = \sqrt{i}$ good if $o^2 = i$)
- Assertions throughout the code (it’s getting formal)
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 behavior
  - Random test data (no real guarantees, but can find bugs)
  - Intelligent test data (by hand, expensive)
  - Automatic test data (need formal spec)
- What about other inputs? (test coverage)
- What about the observation? (test oracle)

Challenges can be solved using formal methods

- Automatic (model-based) test case generation
Specification: What a System Should Do

- Simple properties
  - Safety properties
    - *Something bad will never happen* (e.g., simultaneous access)
  - Liveness properties
    - *Something good will happen eventually* (e.g., finally answer request)

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

- Non-functional properties
  - Runtime, memory, usability, ...

- Full behavioral 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
  - Program refinement
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
  - 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 $\mu$-processors
  - Fault tolerant software

- **Saving human lives**
  - Avionics
  - X-by-wire
A Fundamental Fact

Formalisation of system requirements is hard

Let’s see why …
Difficulties in Creating Formal Models

Real World

Abstraction

Formal Requirements Specification

Formal Execution Model
Difficulties in Creating Formal Models

Real World → oversimplification → e.g., zero delay → Formal Requirements Specification → Formal Execution Model
Difficulties in Creating Formal Models

Real World

missing requirement

e.g., max stack size

Formal Requirements Specification

Formal Execution Model
Difficulties in Creating Formal Models

Real World

Formal Requirements Specification

Formal Execution Model

wrong modeling

e.g., $\mathbb{Z}$ vs int
Formalization Helps to Find Bugs in Specs

- Wellformedness and consistency of formal specs machine-checkable
- 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
- 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.
Wellformedness and consistency of formal specs machine-checkable

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.

Proving properties of systems can be hard for non-trivial systems
Level of System (Implementation) Description

- **Abstract level**
  - Finitely many states (finite data)
  - Tedious to program and maintain
  - Over-simplification, unfaithful modeling
    sometimes inevitable
  - Relationship to concrete implementation
  - Automatic proofs are (in principle) possible

- **Concrete level**
  - Infinite data
    (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 behavioral specification
  - Quantification over infinite domains
  - High precision, tight modeling
  - Automatic proofs (in general) impossible!
# Main Approaches

<table>
<thead>
<tr>
<th>Concrete programs, Complex properties</th>
<th>Concrete programs, Simple properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract programs, Complex properties</td>
<td>Abstract programs, Simple properties</td>
</tr>
</tbody>
</table>
Main Approaches

<table>
<thead>
<tr>
<th>Concrete programs, Complex properties</th>
<th>Concrete programs, Simple properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract programs, Complex properties</td>
<td>Abstract programs, Simple properties</td>
</tr>
</tbody>
</table>

SPIN
Main Approaches

KeY

<table>
<thead>
<tr>
<th>Concrete programs, Complex properties</th>
<th>Concrete programs, Simple properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract programs, Complex properties</td>
<td>Abstract programs, Simple properties</td>
</tr>
</tbody>
</table>

SPIN
Main Approaches

KeY

Concrete programs, Complex properties

Concrete programs, Simple properties

Abstract programs, Complex properties

Abstract programs, Simple properties

SPIN

Abstract Interpretation
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

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

System Property

\[\square \neg (\text{criticalSectP} \land \text{criticalSectQ})\]

Model Checker

- criticalSectP = 0 1 1
- criticalSectQ = 1 0 1

*André Platzer (CMU)*

15-819M/01: Data, Code, Decisions
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 abstraction
  - Bell Labs, Ericsson, Microsoft
Proof rules establish relation “implementation conforms to specs”

Computer support essential for verification of real programs

Java code:
```java
synchronized java.lang.StringBuffer append(char c)
```

5.000 proof steps
200 case distinctions
2 human interactions, 1 minute computing time
Deductive Verification

correct?

Java Code \rightarrow Formal specification
Deductive Verification

Java Code → Formal specification

Program Verification Tool
Deductive Verification

Java Code \rightarrow \text{Formal specification} \rightarrow \text{correct}

Program Verification Tool

\text{proof rules establish relation "implementation conforms to specs"}

Computer support essential for verification of real programs

\text{synchronized java.lang.StringBuffer append(char)}

5.000 proof steps
200 case distinctions
2 human interactions, 1 minute computing time
Deductive Verification

Java Code → Formal specification

Program Verification Tool

Proof rules establish relation “implementation conforms to specs”

Computer support essential for verification of real programs

`synchronized java.lang.StringBuffer append(char c)`

- 5,000 proof steps
- 200 case distinctions
- 2 human interactions, 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™ 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
- Model checking on 16 computers, daily with 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
### 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

KeY to verify Java (Card) programs against contracts in JML
Spin to verify PROMELA programs against Temporal Logic specs
jSpin as a Java interface for Spin

Both are free and run on Windows/Unixes/Mac)
Install 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
- Formal verification for cyber-physical, embedded, real-time, hybrid systems
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

Responsible software/system management should consider formal methods, especially for safety-critical / security-critical / cost-intensive
You will gain experience in ...

... more than Formal Methods

- modeling, and modeling languages
- specification, and specification languages
- in depth analysis of possible system behavior
- typical types of errors
- reasoning about system (mis)behavior
- reasoning principles and logic
- decision procedures
- ...

André Platzer (CMU)
15-819M/01: Data, Code, Decisions