

Anwendung formaler Verifikation

Organisatorisches

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2010

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK



Webseite zur Vorlesung

`http://formal.iti.kit.edu/teaching/
AnwendungFormalerVerifikation/`

Alle für die Vorlesung relevanten Informationen und Materialien:

- Termine und aktuelle Informationen
- Folien
- Tools und weitere Materialien

Webseite zur Vorlesung

`http://formal.itl.kit.edu/teaching/
AnwendungFormalerVerifikation/`

Alle für die Vorlesung relevanten Informationen und Materialien:

- Termine und aktuelle Informationen
- Folien
- Tools und weitere Materialien

Zielgruppe

Diplom Informatik, Master Informatik

Vertiefungsfächer

- Theoretische Grundlagen
- Softwaretechnik und Übersetzerbau

Modul *Formale Methoden*

Zielgruppe

Diplom Informatik, Master Informatik

Vertiefungsfächer

- Theoretische Grundlagen
- Softwaretechnik und Übersetzerbau

Modul *Formale Methoden*

Zielgruppe

Diplom Informatik, Master Informatik

Vertiefungsfächer

- Theoretische Grundlagen
- Softwaretechnik und Übersetzerbau

Modul *Formale Methoden*

Umfang und Struktur

- **3 SWS = 21 Doppelstunden im Semester**
- aufgeteilt in sieben Einheiten mit je zwei Doppelstunden Vorlesung und einer Doppelstunde Übung
- Übungen: Verfahren werden anhand konkreter Verifikationssysteme praktisch erprobt
- Vorlesung und Übung gehen fließend ineinander über

Umfang und Struktur

- 3 SWS = 21 Doppelstunden im Semester
- aufgeteilt in sieben Einheiten mit je zwei Doppelstunden Vorlesung und einer Doppelstunde Übung
- Übungen: Verfahren werden anhand konkreter Verifikationssysteme praktisch erprobt
- Vorlesung und Übung gehen fließend ineinander über

Umfang und Struktur

- 3 SWS = 21 Doppelstunden im Semester
- aufgeteilt in sieben Einheiten mit je zwei Doppelstunden Vorlesung und einer Doppelstunde Übung
- **Übungen: Verfahren werden anhand konkreter Verifikationssysteme praktisch erprobt**
- Vorlesung und Übung gehen fließend ineinander über

Umfang und Struktur

- 3 SWS = 21 Doppelstunden im Semester
- aufgeteilt in sieben Einheiten mit je zwei Doppelstunden Vorlesung und einer Doppelstunde Übung
- Übungen: Verfahren werden anhand konkreter Verifikationssysteme praktisch erprobt
- **Vorlesung und Übung gehen fließend ineinander über**

Termine

- Donnerstags, 11:30-13:00
- Freitags, 11:30-13:00

An welchen 21 der 28 möglichen Termine eine Vorlesung/Übung stattfindet steht auf der Webseite!

Vorlesungseinheiten

1	Funktionaler Eigenschaften imperativer und objekt- orientierter Programme	Java Modeling Language Dynamische Logik
2	Nebenläufige Programme (Concurrency)	Rely/Guarantee
3	Temporallogische Eigenschaften	Model Checking
4	Hybride Systeme	Model Checking, Dynami- sche Logik
5	Echtzeiteigenschaften	Timed Automata, UPPAAL
6	Informationsfluss- Eigenschaften	Dynamische Logik, Typsys- teme
7	Protokollverifikation	Termersetzungverfahren (Rewriting)

- Formale Systeme (Beckert/Schmitt)
- Formale Systeme II (Schmitt/Beckert)
- Spezifikation und Verifikation von Software / Formaler Entwurf und Verifikation von Programmen (Schmitt/Beckert)
- Theorembeweiser und ihre Anwendungen (Snelting/Wasserrab)
- Model Checking (Sinz/Tveretina)
- Modellgetriebene Software-Entwicklung (Reussner/Becker)

Applications of Formal Verification

Introduction

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2010

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK



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
- EC-Karten Bug

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
- Aviation

⇒ software—and specification—quality is a growing legal issue

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
- Aviation

⇒ software—and specification—quality is a growing legal issue

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
- Aviation

⇒ *software—and specification—quality is a growing legal issue*

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**
- Design practice for reliable software in **immature** state
for complex, particularly, distributed systems
- Cost efficiency favoured over reliability
- Extremely short innovation cycles

- 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

- 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

- 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

Various Properties

(Require Different Verification Techniques)

- 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
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

- **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

Some Reasons for Using Tools

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