

# Formal Specification and Verification

Viorica Sofronie-Stokkermans

e-mail: [sofronie@uni-koblenz.de](mailto:sofronie@uni-koblenz.de)

Some of the slides are based on or inspired by material by  
Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Andreas Podelski

# Motivation

---

**Small faults in technical systems can have catastrophic consequences**

# Motivation

---

**Small faults in technical systems can have catastrophic consequences**

In particular, this is true for software systems

# Motivation

---

**Small faults in technical systems can have catastrophic consequences**

In particular, this is true for software systems

- Operating systems
- Ariane 5
- Mars Climate Orbiter, Mars Sojourner
- Electricity Networks
- Health/devices
- Banks
- Airplanes
- ...

# Motivation

---

Software these days is inside just about anything:

- Cars, Planes, Trains
- Smart cards
- Mobile phones

**Software defects can cause failures everywhere**

# Motivation

---

## Complexity of systems makes verification difficult

- Computer hardware change of scale

In the 25 last years, computer hardware has seen its performances multiplied by  $10^4$  to  $10^6/10^9$ :

- ENIAC (5000 FLOPS) “Floating-Point Operations per Second”
- Intel/Sandia Teraflops System (1996:  $10^{12}$  FLOPS)

# Motivation

---

## Complexity of systems makes verification difficult

- Computer hardware change of scale

In the 25 last years, computer hardware has seen its performances multiplied by  $10^4$  to  $10^6/10^9$ :

- ENIAC (5000 FLOPS) “Floating-Point Operations per Second”
  - Intel/Sandia Teraflops System (1996:  $10^{12}$  FLOPS)
- The size of the programs executed by these computers has grown up in similar proportions

# 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 Work For Software

---

- Single bit-flip may change behaviour completely
- Redundancy as replication does not 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?

---

## Testing/Simulation

Testing against inherent SW errors (“bugs”)

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

Testing against external faults

- Inject faults (memory, communication) by simulation

# Limitations of Testing

---

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

# Formal Methods

---

- Rigorous methods used in system design and development
- Mathematics and symbolic logic
  - precise language / reliable correctness proofs
- Increase confidence in a system

# Formal Methods

---

- Rigorous methods used in system design and development
- Mathematics and symbolic logic
  - precise language / reliable correctness proofs
- Increase confidence in a system

## Make formal model of:

- System implementation
- System requirements

**Prove mechanically that formal execution model satisfies formal requirements**

# Specification

---

## Properties of a system

- Simple properties
  - Safety properties  
“Nothing bad will happen”
  - Liveness properties  
“Something good will eventually happen”

# Specification

---

## Properties of a system

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

# Specification

---

## Properties of a system

- 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
  - Refinement relation



# Formal Methods

---

**Main aim in formal methods is not ...**

# Formal Methods

---

**Main aim in formal methods is not ...**

- To prove “correctness” of entire systems  
(What is correctness in general? We verify specific properties)

# Formal Methods

---

**Main aim in formal methods is not ...**

- To prove “correctness” of entire systems  
(What is correctness in general? We verify specific properties)
- To replace testing entirely

# Formal Methods

---

**Main aim in formal methods is not ...**

- To prove “correctness” of entire systems  
(What is correctness in general? We verify specific properties)
- To replace testing entirely
- To replace good design practices

# Formal Methods

---

**The aim in formal methods is not ...**

- To prove “correctness” of entire systems  
(What is correctness in general? We verify specific properties)
- To replace testing entirely
- To replace good design practices

One cannot formally verify messy code with unclear specifications

Correctness guarantees - only for clear requirements and good design

# Formal Methods

---

- Formal proof can replace (infinitely) many test cases
- Formal methods can be used in automatic test case generation
- Formal methods improve the quality of specifications/programs (even without formal verification):
  - better written software (modularization, information hiding)
  - better and more precise understanding of model/implementation
- Formal methods guarantee specific properties of a specific system model

# Formal Methods

---

## Problems:

- Formalisation of system requirements is hard

Oversimplification when modeling

- 0 delays
- missing requirements
- wrong modeling

(e.g. in the case of programs:  $\mathbb{R}$  vs. FLOAT,  $\mathbb{Z}$  vs int)

# Formal Methods

---

## Problems:

- Proving properties of systems can be hard



# Level of System 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;  
positive results in special cases; **active area of research**

# 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! positive results in special cases; **active area of research**

# Main approaches

---

- Concrete programs/Complex properties
- Concrete programs/Simple properties
- Abstract programs/Complex properties
- Abstract programs/Simple properties

# Limitations of Formal Methods

---

## Possible reasons for errors:

- Program is not correct (does not satisfy the specification)  
Formal verification proved absence of this kind of error
- Program is not adequate (error in specification)  
Formal specification/verification avoid or find this kind of error
- Error in operating system, compiler, hardware  
Not avoided (unless compiler, operating system, hardware specified/verified)

# Limitations of Formal Methods

---

## Possible reasons for errors:

- Program is not correct (does not satisfy the specification)  
Formal verification proved absence of this kind of error
- Program is not adequate (error in specification)  
Formal specification/verification avoid or find this kind of error
- Error in operating system, compiler, hardware  
Not avoided (unless compiler, operating system, hardware specified/verified)

In general it is not feasible to fully specify and verify large software systems.  
Then formal methods are restricted to:

- Important parts/modules
- Important properties/requirements

# History

---

Some of the most important moments in the history of program verification:

# History

---

The idea of proving the correctness of a program in a mathematical sense dates back to the early days of computer science with John von Neumann and Alan Turing.



John von Neumann



Alan Turing

# History

---

- R. Floyd and P. Naur introduced the “partial correctness” specification togetherwith the “invariance proof method”
- R. Floyd also introduced the “variant proof method” to prove program termination



Robert Floyd



Peter Naur



# History

---

- C.A.R. Hoare formalized the Floyd/Naur partial correctness proof method in a logic (so-called “Hoare logic”) using an Hilbert style inference system;
- Z. Manna and A. Pnueli extended the logic to “total correctness” (i.e. partial correctness + termination).



C.A.R. Hoare



Z. Manna



A. Pnueli

# History

---

Edsger W. Dijkstra introduced predicate transformers (weakest liberal precondition, weakest precondition) and defined a predicate transformer calculus.

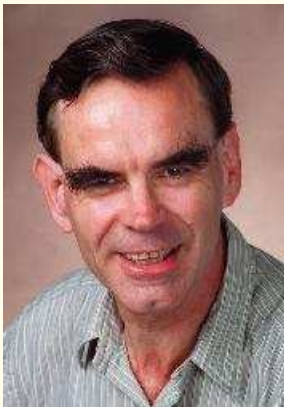


Edsger W. Dijkstra

# History

---

Dynamic logic was developed by Vaughan Pratt in 1974 (in notes for a class on program verification) as an approach to assigning meaning to Hoare logic by expressing the Hoare formula  $p\{a\}q$  as  $p \rightarrow [a]q$ . The approach was later published in 1976 as a logical system in its own right.



Vaughan Pratt

The system parallels Edsger Dijkstra's notion of weakest-precondition predicate transformer  $wp(a, p)$ , with  $[a]p$  corresponding to Dijkstra's  $wlp(a, p)$ , weakest liberal precondition.

# History

---

## First attempts towards automation

- James C. King, a student of Robert Floyd, produced the first automated proof system for numerical programs, in 1969.
- The use of automated theorem proving in the verification of symbolic programs (à la LISP) was pioneered, a.o., by Robert S. Boyer and J. Strother Moore

# History

---

Nowadays many theorem provers, many of which are being used for verification: ACL2, COQ, Simplify, SPIN, Key

Model checkers: BLAST, ...

SMT solvers used for verification (Z3, Yices, CVC, ...)

# Course Structure

---

- **Introduction**

- **Specification**

- Logic (propositional logic, first-order logic)
- Algebraic specification
- Transition systems/Abstract-state-machines/Timed automata/Hybrid automata

- **Verification**

- Temporal logic; Model Checking
- Basics of deductive verification
  - Hoare Logic and Dynamic Logic
  - Decision procedures for data types.
- Verification by Abstraction/Refinement (if sufficient time)

# Logic

---

## Formal logic:

- **Syntax:** a formal language (formula expressing facts)
- **Semantics:** to define the meaning of the language, that is which facts are valid)
- **Deductive system:** made of axioms and inference rules to formally derive theorems, that is facts that are provable
- Propositional logic (new: BDD/OBDD)
- First-order logic
- Decidability/undecidability results

# Formal specification

---

- **Specification languages for describing programs/processes/systems**

Model based specification

transition systems, abstract state machines, specifications based on set theory

Axiom-based specification

algebraic specification

Declarative specifications

logic based languages (Prolog)

functional languages,  $\lambda$ -calculus (Scheme, Haskell, OCaml, ...)

rewriting systems (very close to algebraic specification): ELAN, SPIKE, ...

- **Specification languages for properties of programs/processes/systems**

Temporal logic



# Algebraic specification

---

- appropriate for specifying the interface of a module or class
- enables verification of implementation w.r.t. specification
- for every ADT operation: argument and result types (sorts)
- semantic equations over operations (axioms) e.g. for every combination of “defined function” (e.g. top, pop) and constructor with the corresponding sort (e.g. push, empty)
- problem: consistency?, completeness?

# Example: Algebraic specification

---

```
fmod NATSTACK is
  sorts Stack .
  protecting NAT .
  op empty : -> Stack .
  op push : Nat Stack -> Stack .
  op pop : Stack -> Stack .
  op top : Stack -> Nat .
  op length : Stack -> Nat .

  var S S2 : Stack .
  var X Y : Element .
  eq pop(push(X,S)) = S .
  eq top(push(X,S)) = X .
  eq length(empty) = 0 .
  eq length(push(X,S)) =
      1 + length(S) .
endfm
```

# Example: Algebraic specification

---

reduce  $\text{pop}(\text{push}(X,S)) == S$  .

reduce  $\text{top}(\text{pop}(\text{push}(X,\text{push}(Y,S)))) == Y$  .

reduce  $S == \text{push}(X,S2)$  implies  $\text{push}(\text{top}(S),\text{pop}(S)) == S$  .

reduce  $S == \text{push}(X,S2)$  implies  $\text{length}(\text{pop}(S)) + 1 == \text{length}(S)$  .

- the equations can be used as term rewriting rules
- this allows proving properties of the specification

# Transition systems

---

- model to describe the behaviour of systems
- digraphs where nodes represent states, and edges model transitions
- state:
  - the current colour of a traffic light
  - the current values of all program variables + the program counter
  - the current value of the registers together with the values of the input bits
- transition: (“state change”)
  - a switch from one colour to another
  - the execution of a program statement
  - the change of the registers and output bits for a new input

# Generalizations of transition systems

---

- More detailed description of states: Abstract state machines
- Emphasis on processes and their interdependency: CSP
  
- Durations: Timed automata
- Continuous evolution + discrete control: Hybrid automata

# Temporal logic

---

The purpose of temporal logic (TL) is:

- reasoning about time (in philosophy), and
- reasoning about the behaviour of systems evolving over time (in computer science).

Special language for doing so:

Logical connectives  $\wedge, \vee, \neg$

temporal operators:  $\bigcirc F$  and  $FUG$

# Model Checking

---

In computer science, model checking refers to the following problem:

Given a model of a system, test automatically whether this model meets a given specification.

Typically, the systems one has in mind are hardware or software systems, and the specification contains safety requirements such as the absence of deadlocks and/or critical states that can cause the system to crash (which can be expressed in temporal logic).

Model checking is a technique for automatically verifying correctness properties of **finite-state systems**.

# Deductive verification

---

- **Model checking:**

Finite transition systems / CTL properties

States are “entities” (no precise description, except for labelling functions)

No precise description of actions (only  $\rightarrow$  important)

## Extensions in two possible directions:

- More precise description of the actions/events
  - Hoare logic
  - Propositional Dynamic Logic
- More precise description of states (and possibly also of actions)
  - succinct representation: formulae represent a set of states
  - deductive verification



# Hoare Logic

---

Hoare logic (also known as Floyd-Hoare logic) is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs. It was proposed in 1969 by C. A. R. Hoare.

**Central feature:** Hoare triple.

A triple describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form

$$\{P\} C \{Q\}$$

where  $P$  and  $Q$  are assertions and  $C$  is a command.

$P$  is named the precondition and  $Q$  the postcondition: when the precondition is met, the command establishes the postcondition.

Assertions are formulae in predicate logic.

# Hoare Logic

---

Hoare logic (also known as Floyd-Hoare logic) is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs. It was proposed in 1969 by C. A. R. Hoare.

**Central feature:** Hoare triple  $\{P\} C \{Q\}$  ( $P$  precondition/ $Q$  postcondition)

Hoare logic provides axioms and inference rules for all the constructs of a simple imperative programming language.

Standard Hoare logic proves only **partial correctness**; termination needs to be proved separately.

**Intuitive reading of a Hoare triple:**

Whenever  $P$  holds of the state before the execution of  $C$ , then  $Q$  will hold afterwards, or  $C$  does not terminate.

# Dynamic logic

---

## Dynamic logic of programs

Dynamic logic is an extension of modal logic originally intended for reasoning about computer programs and later applied to more general complex behaviors arising in linguistics, philosophy, AI, and other fields.

Operators:

$[\alpha]A$ :  $A$  holds after every run of the (non-deterministic) process  $\alpha$

$\langle\alpha\rangle A$ :  $A$  holds after some run of the (non-deterministic) process  $\alpha$

Dynamic logic permits compound actions built up from smaller actions

- $\alpha \cup \beta$
- $\alpha; \beta$
- $\alpha^*$

# Decision Procedures

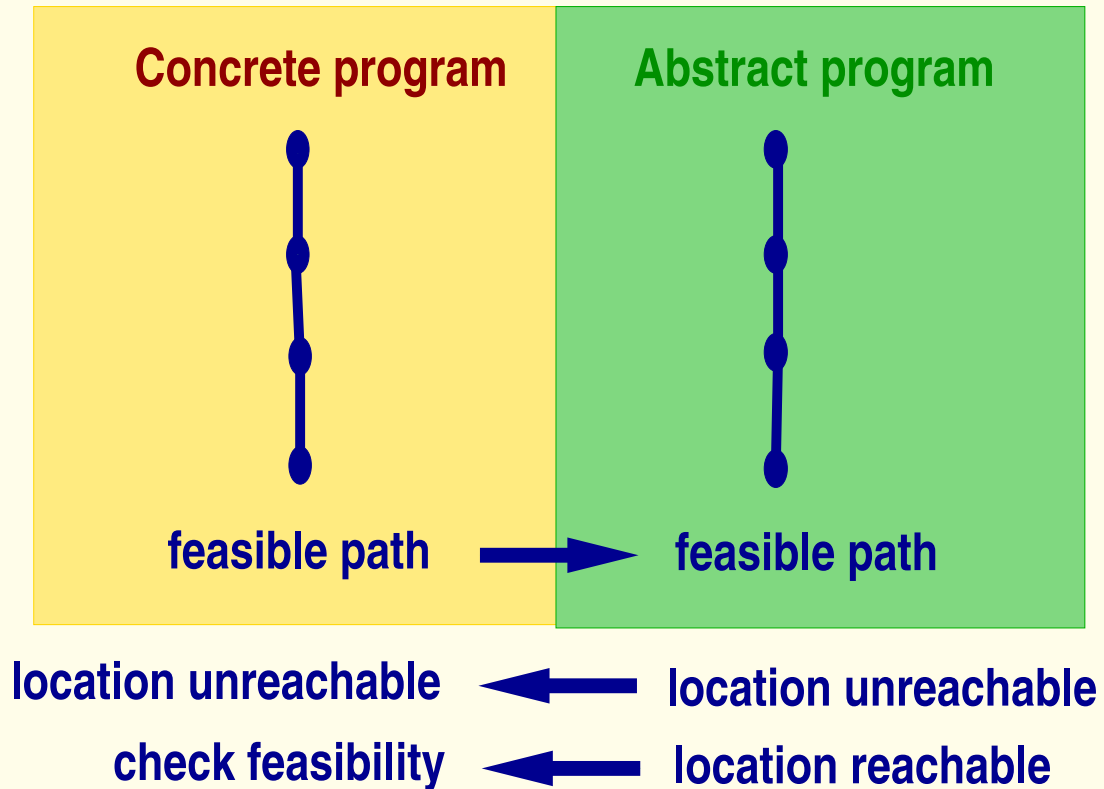
---

**Example:** Does BUBBLESORT return  
a sorted array?

```
int [] BUBBLESORT(int[] a) {  
  int i, j, t;  
  for (i := |a| - 1; i > 0; i := i - 1) {  
    for (j := 0; j < i; j := j + 1) {  
      if (a[j] > a[j + 1]) { t := a[j];  
                           a[j] := a[j + 1];  
                           a[j + 1] := t };  
    }  
  } return a}
```

# Abstraction/Refinement

---



# Organisational Info

---

## Lecturer:

Viorica Sofronie-Stokkermans

[sofronie@uni-koblenz.de](mailto:sofronie@uni-koblenz.de)

# Organisational Info

---

## Course Home Page

[www.uni-koblenz.de/~sofronie/lecture-formal-specif-verif-ss-2018/](http://www.uni-koblenz.de/~sofronie/lecture-formal-specif-verif-ss-2018/)

Will contain all the information about the course:

- slides
- exercises
- additional information

## Passing Criteria

- Written or oral exam (depending on the number of participants)