Model Checking

Wintersemester 2018/2019

Prof. Dr. Heike Wehrheim

WS 18/19

Orga I

Old exam regulations:

- area: software technology and informations systems
- 4 ECTS
- modules: III.1.1, III.1.5

New exam regulations:

- Focus area: software engineering
- 6 ECTS
- course is a module on its own

Orga II

Course consists of three parts:

- Lectures (approximately first half of term)
- Lab part (second half)
- Reading and summary writing

Orga III

Depending on version, different requirements

- old regulations:
  - all of the lecture part
  - oral examination
  - prerequisite for oral exam (Studienleistung): 70/30-rule: in 70% of the exercise sheets have 30% of the total points
  - no lab, no reading & writing
- new regulations:
  - as in old regulations (70/30-rule)
  - + 50% of lab exercises
  - + summary of 1 book chapter

Orga IV

Heike Wehrheim
Office hours: by appointment
email: wehrheim@uni-paderborn.de

Lecture & Tutorial:

Lecture:	Tue, 9 - 11, O1.258
	Wed, 9 - 11, O1.258
Tutorial:	Jürgen König, jkoenig@mail.upb.de
	Wed, 14 - 16, O1.258

Orga V

Lab part:

- starts approximately second half of term
- Manuel Töws, mtoews@mail.uni-paderborn.de
- Wed, 14 - 16, O1.258

Homework

Homework assignments every week
(until lecture part is over)

- first one on Friday in Panda
- solutions must be handed in via Panda
- submitted in groups of 2 - 4 students
- exercises discussed during tutorial

Reading

Books:

- E. Clarke, O. Grumberg, D. Peled: Model Checking, MIT Press, 1999.
- Ch. Baier, J.-P. Katoen: Principles of Model Checking, MIT Press, 2008.

Other material

- Slides (available after lecture in Panda)
- examples, (mainly) on the board,
  partly hand-out

Part I

Basics

Motivation

Software everywhere in daily life:

- mobile phones,
- cars,
- medical applications,
- banking,
- . . ..

The more software is used, the more drastic the consequences of software failures
and
with an increasing complexity of the software it gets harder to avoid software failures.

Why avoid failures?

Train accidents:

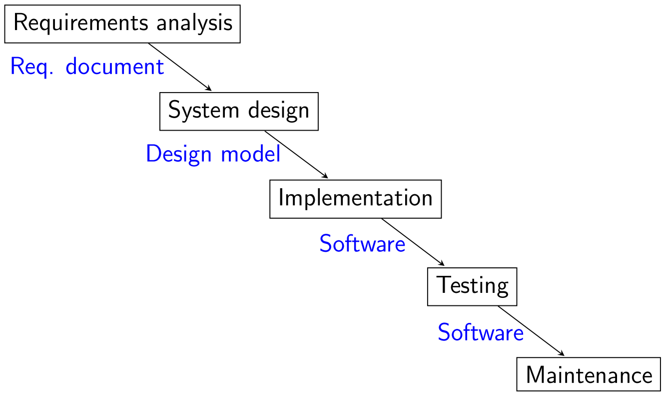
Montparnasse, 1895

Rasender Roland, 2004

Today: software controls points, gates, signalling, ...

System development

Design processes, e.g. waterfall model (old)



Validation - Testing

Correctness: System should meet requirements

Waterfall model: checked via testing

Methods: Blackbox/Whitebox testing, test coverage

Advantage:
- relatively easy (and cheap)

Disadvantage:
- errors found late (better: incremental processes)
- often not systematic
- incomplete

"Testing can only show the presence of errors, never their absence." (E. Dijkstra)

Validation - Simulation

Different option: simulation

simulate runs of the model (arbitrarily chosen)

advantage:
- on the model, thus in early phase

disadvantage:
- only some runs inspected (similar to testing)

used in hardware design

Validation - Verification

Verification:
mathematical proof of the correctness of a model/program with respect to the requirements

needs: formal description of model and requirements

$\rightarrow$ additional costs: more time, experts needed

drawback:
might be infeasible $\rightarrow$ combination with testing

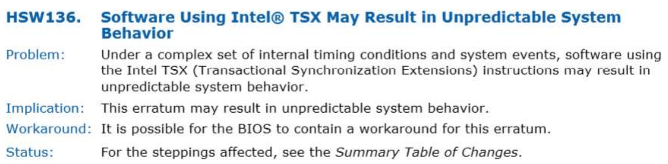
Verification - why?

Consequences of incorrect software/hardware:

1. Danger to human lifes
   airbag goes off without reason, trains collide, ...
2. High costs
   - Ariane 5: $\approx$ 500 million dollars
     overflow error in a conversion floating point to integer
   - Intel Pentium: $\approx$ 500 million dollars
     error in floating point division

Recent bug (2014)

Intel Haswell processor



HSW136. Software Using Intel® TSX May Result in Unpredictable System Behavior

Problem: Under a complex set of internal timing conditions and system events, software using the Intel TSX (Transactional Synchronization Extensions) instructions may result in unpredictable system behavior.

Implication: This erratum may result in unpredictable system behavior.

Workaround: It is possible for the BIOS to contain a workaround for this erratum.

Status: For the steppings affected, see the *Summary Table of Changes*.

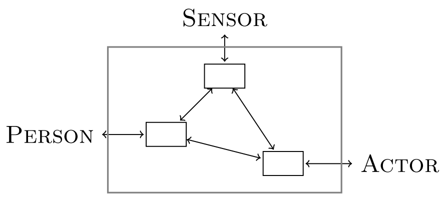
Verification - how?

A proof of correctness, how can this be achieved?

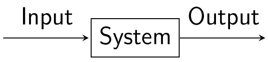
- depends on the type of system

transformational systems

reactive systems



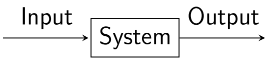
Transformational systems



Examples: compiler, sorting programs, book keeping

program is implementing a function from an input (state) into an output (state)

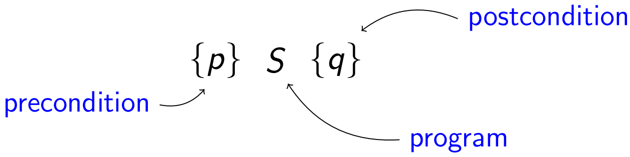
Transformational systems



Examples: compiler, sorting programs, book keeping

program is implementing a function from an input (state) into an output (state)

requirements denotable by "Hoare Triples"



Hoare Triple

Precondition - program - postcondition

$$\{p\}S\{q\}$$

"if p holds and we execute S then afterwards q holds"

Example: $\{y \geq 0\}x := y\{x \geq 0\}$

in addition: proof of termination

Verification of transf. systems

Deductive verification: axioms + proof rules

e.g. a rule for sequential composition

$$\frac{\{p\}S_1\{q\}\ ,\ \{q\}S_2\{r\}}{\{p\}S_1;\ S_2\{r\}} \qquad \begin{array}{l}\text{premise}\\ \text{conclusion}\end{array}$$

if the premises holds for the components, the conclusion can be deduced for the sequential composition of components

rules for all constructs of programs, including parallel composition
$\Rightarrow$ deductive verification

Deductive verification

Advantage:

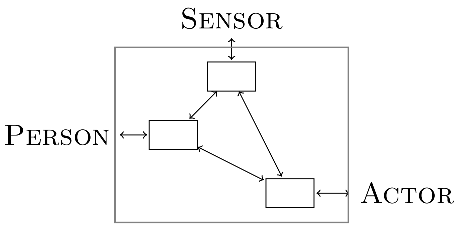
- complete (often)
- verification of programs with infinite state space

Disadvantage:

- tedious
- proofs done by hand / with help of theorem prover
- Hoare-style proofs not for reactive systems

(however, deductive verification possible for reactive systems as well)

Reactive systems (1)



Set of components, executing in parallel and communicating with each other

Reactive systems (2)

Characteristica:

- parallelism, distribution
- reactivity
- interaction with an environment, usually no termination
- high complexity, safety critical

Examples

Examples

- embedded system (automotive sector)
- telecommunication
- elevator

requirements specify the behaviour of a system in time, not its I/O behaviour

e.g. requirement on a communication protocol

> *"if process P sends a message it will not send another message until it got an acknowledgement from the receiver"*

Specifying requirements

Requirements on reactive systems specified in temporal logic (TL)

Amir Pnueli

- 1977: proposal of such a logic
- 1996: Turing Award
  "For seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification."

Example

The requirement on the communication protocol:

"if process P sends a message it will not send another message until it got an acknowledgement from the receiver"

$$\varphi = \; G \; \big(snd_p(m) \Rightarrow (\neg snd_p(nxt(m)) \; U \; rcv_p(ack))\big)$$

globally (G) — until (U)

What's now verification?

Question (correctness):
does the model meet the requirements?

formally:

Requirement (in TL) → φ

$$M \models \varphi \quad ?$$

System model → M

Verification

Proof of $M \models \varphi$

- Term model checking
  "is the system a model of the formula"
- in general undecidable: Theorem of Rice
- beginning of 80th:
  Clarke & Emerson, Quielle & Sifakis
  model checking algorithm searches the whole state space of systems
  hence: state space needs to be finite

Clarke, Emerson, Sifakis

Turing Award 2007

> *"For their role in developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries."*


Ed Clarke


Allen Emerson


Joseph Sifakis

And then?

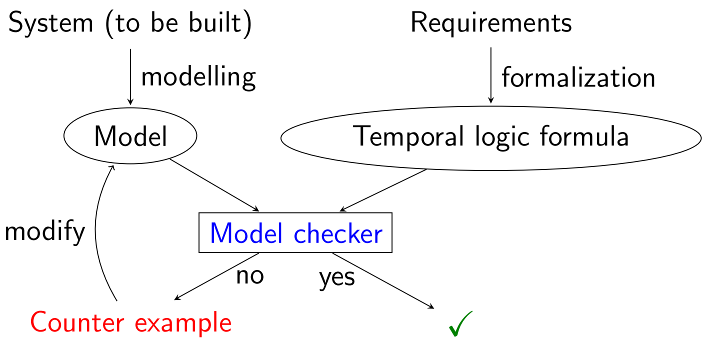
- implementation of these algorithms
  → tools (model checker)
  allow for a fully automatic correctness proof (for certain classes of systems)
- end of 80th, beginning 90th:
  research: larger systems
  efficient representation of state (BDDs)
  reduction techniques
- '90, '00
  industrial applications (in particular hardware)
  research departments (IBM, Intel, Motorola, Siemens, Microsoft)

Today

Research today:

- systems with large (or infinite state space)
- software model checking (C, Java)
- combination of different techniques:
  deductive verification, constraint-solving, static analysis, heuristic search, ...

Model checking – Big picture



What's needed ...

1. Formal description of models $\rightarrow M$
2. Temporal logic formula $\rightarrow \varphi$
3. Def. of $M \models \varphi$
4. Algorithms for checking $M \models \varphi$
5. Tools implementing these algorithms
6. Clever people being able to develop formal models and write formulae

this course: 1, 2, 3, 4, 5,
at the end you belong to 6

Success stories

Verification of the floating point unit of Pentium4 (2001)
one error found

Verification of a cache protocol in the IEEE-Futurebus+ (1992)
several errors found

SLAM/ Static Driver Verifier (2000 - 2004)
verification of Windows-XP Drivers

First example (1)

Mutual exclusion of two processes

$$\left[\begin{array}{l|l} l_0: \ \textbf{while } \textit{true} \textbf{ do} & l_1: \ \textbf{while } \textit{true} \textbf{ do} \\ \quad NC_0: \textbf{wait}\,(turn = 0); & \quad NC_1: \textbf{wait}\,(turn = 1); \\ \quad CR_0: turn := 1 & \quad CR_1: turn := 0 \\ \ \textbf{od}; & \ \textbf{od}; \\ \hat{l}_0; & \hat{l}_1; \end{array}\right]$$

$l_0, l_1, CR_O, \ldots$: labels, $||$: parallel composition
NC: non-critical section, CR: critical section
(wait: busy waiting)

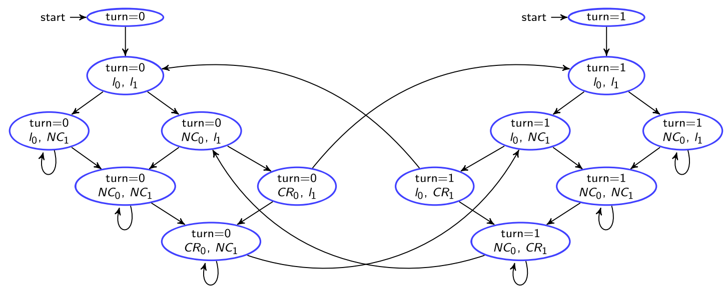
First example (2)

Semantics: Kripke structure

- describes state space
- state: evaluation of variables + program counter
- transitions: state changes

plus atomic propositions (predicates on variables)
e.g. turn = 0

A Kripke structure



Requirements

1. Mutex: process 1 and 2 never both in their critical section

   $G\ \neg(at\_CR_0 \wedge at\_CR_1) \quad \surd$

   safety property ("nothing bad happens", Lamport)

Requirements

1. Mutex: process 1 and 2 never both in their critical section

   $G\ \neg(at\_CR_0 \wedge at\_CR_1) \quad \surd$

   safety property ("nothing bad happens", Lamport)
2. Progress: every process can always eventually enter its critical section

   $G\ (F\ at\_CR_0), \quad G\ (F\ at\_CR_1)$

   does not hold (only under additional fairness assumption)
   liveness property ("something good will happen")

This course

We will learn something about

- two temporal logics: LTL and CTL
  LTL = linear time temporal logic
  CTL = computation tree logic
- model checking techniques
- a model checker: SPIN (for LTL)
  small examples
  - distributed algorithms
  - kryptographic protocols