# Proseminar *Einführung in die Programmanalyse*

**Einführungsveranstaltung**
**Wintersemester 2017/18; 20. Oktober 2017**

**Thomas Noll**
**Software Modeling and Verification Group**
**RWTH Aachen University**

`https://moves.rwth-aachen.de/teaching/ws-1718/epa/`

Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

## Übersicht

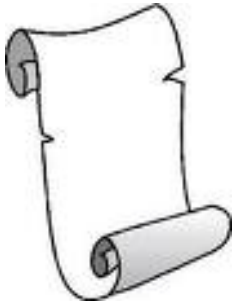
## Traum der Programmanalyse

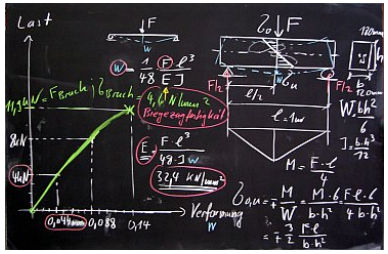
**Programm**

```
for h3 in page.findAll("h3"):
    value = (h3.contents[0])
    if value != "Afdeling":
        print >> txt, value
        import codecs
        f = codecs.open("alle.txt", "r", encoding="utf-8")
        text = f.read()
        f.close()
        # open the file again for writing
        f = codecs.open("alle.txt", "w", encoding="utf-8")
        f.write(value+"\n")
        # write the original contents
```

**Analyse**

**Ergebnis**





**Spezifikation**

## Seminarthema

### Thema des Proseminars

Einführung in Grundkonzepte der (automatischen) Programmanalyse

- Modelle:
  - (abstrakte) Darstellung des Systemverhaltens
  - Spezifikation des erwarteten Verhaltens
- Logiken:
  - formale Beschreibung von (Korrektheits-)Eigenschaften
- Statische Verfahren:
  - automatische Analyse von Programmeigenschaften
  - basierend auf Quellcode der Software
  - üblicherweise vollständige Abdeckung
- Dynamische Verfahren:
  - automatische Analyse von Programmeigenschaften
  - basierend auf beobachtetem Ausführungsverhalten
  - üblicherweise unvollständige Abdeckung

## Zielsetzung

### Ziele des Proseminars

- Selbstständiges Einarbeiten in ein neues Thema
- Literaturrecherche
- Darstellen des Inhalts in einer wissenschaftlichen Ausarbeitung
- Verständliches Präsentieren

## Zielsetzung

### Ziele des Proseminars

- Selbstständiges Einarbeiten in ein neues Thema
- Literaturrecherche
- Darstellen des Inhalts in einer wissenschaftlichen Ausarbeitung
- Verständliches Präsentieren

### Bearbeitung in Zweiergruppen

- Gemeinsame Anfertigung der Ausarbeitung
- Zwei separate Vorträge

## Anforderungen Ausarbeitung

### Ausarbeitung

- Selbstständiges Verfassen einer Ausarbeitung von $\geq$ 10 Seiten
- Vollständiges Literaturverzeichnis
- Korrektes Zitieren
- Plagiarismus:
  Die nicht gekennzeichnete übernahme fremder Inhalte führt zum sofortigen Ausschluss.
- Schriftgröße 12pt, übliche Seitenränder
- Titelseite mit Thema, Titel Proseminar, Semester, Name, Datum
- Vorlage wird zur Verfügung gestellt
- Sprache Deutsch oder Englisch
- Korrekte Sprache wird vorausgesetzt:
  $\geq$ 10 Fehler pro Seite $\Longrightarrow$ Abbruch der Korrektur

## Anforderungen Vortrag

### Vortrag

- 20-minütiger Vortrag
- Zielgruppengerechte Präsentation der Inhalte
- übersichtliche Folien:
  - $\leq$ 15 Textzeilen
  - sinnvoller Einsatz von Farben
- Vortrag in Deutsch oder Englisch

## Übersicht

## Themenauswahl

### Verfahren

- Themenliste wurde/wird ausgehändigt
- Priorisierte Auswahl
- ggf. Angabe Wunschpartner(in)
- Abgabe bis (spätestens) Montag, 23. Oktober per Mail/im Sekretariat
- Wir bemühen uns (ohne Garantie) um ein „optimales“ Matching
- Zuordnung der Themen und Betreuer bis Mitte nächster Woche online

### Rücktritt vom Proseminar

- Bis zu drei Wochen nach Einführung: ohne Folgen
- Danach: Fehlversuch

## Bibliothekseinführung

### Einführung in die Literaturrecherche

- Einweisung in themenspezifische Literaturrecherche
- Dauer: ca. zwei Stunden
- Teilnahme für BSc-Studierende verpflichtend
- Bedarf bitte auf Themenblatt vermerken
- Termine zur Auswahl:
  - Montag, 27.11., 15:30
  - Dienstag, 28.11., 17:00
  - Dienstag, 05.12., 12:00
  - Freitag, 08.12., 11:30
- Mögliche Termine auf Themenliste vermerken, Bestätigung per Mail

## Deadlines

### Deadlines

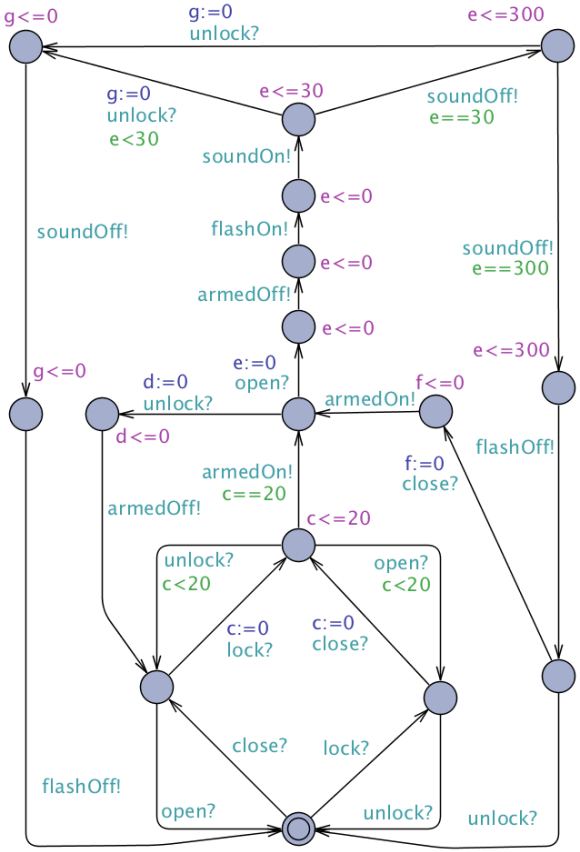
Folgende Termine sind einzuhalten:

- 10.11.2017: letzte Rücktrittsmöglichkeit
- 13.11.2017: Vorlage der detaillierten Inhaltsübersicht
- 11.12.2017: vollständige Fassung der Ausarbeitung
- 15.01.2017: vollständige Fassung der Folien
- 01./02.02.2017 (?): Blockseminar

## Übersicht

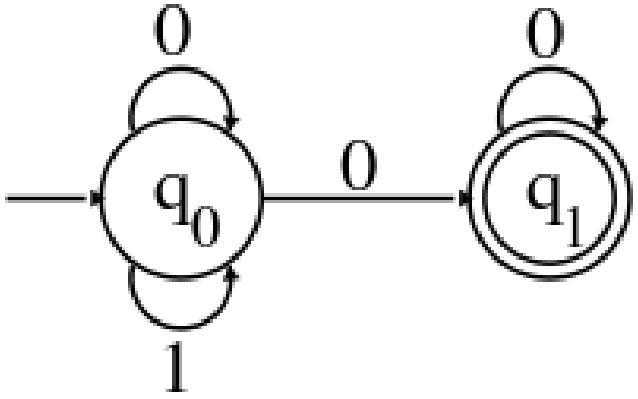
# Modelle

## 1. Timed Automata




- Timed automaton = finite automaton + real-valued clocks
- Clock values increase at same speed
- Can be reset in transitions
- Can be tested in states (invariants) and transitions (guards)
- Enables modelling and analysis of time-dependent system behaviour

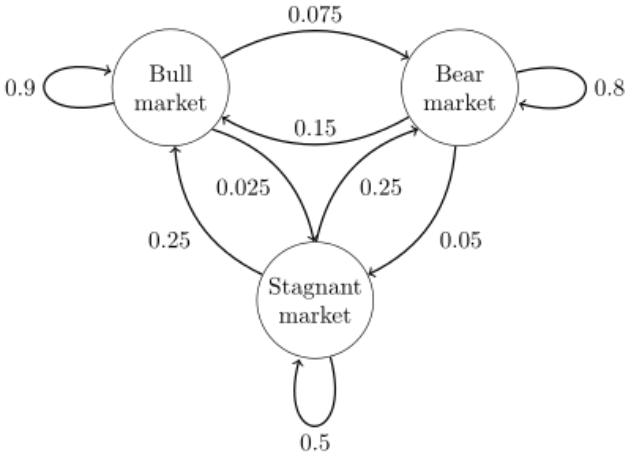
## 2. Büchi Automata




BA for $(0|1)^*0^\omega$)

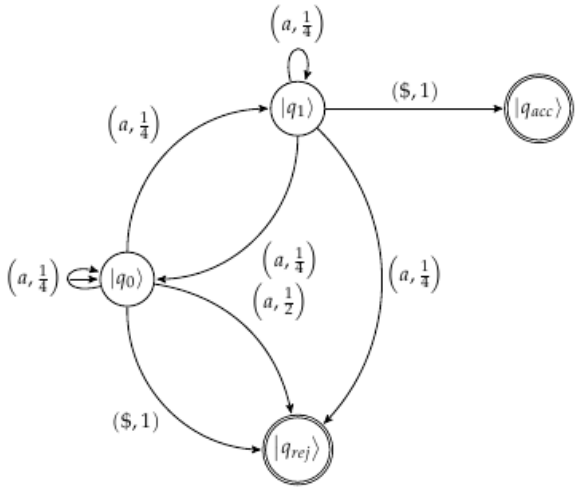
- Syntax identical to DFA/NFA
- But define languages of infinite words
- Accepts an infinite input sequence if there exists a run that visits one of the final states infinitely often
- Enables analysis of non-terminating system behaviour
- Non-deterministic variant more expressive

## 3. Markov Chains



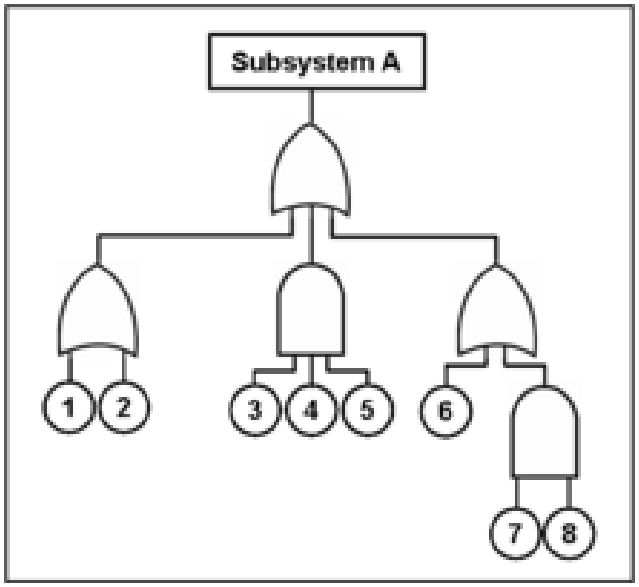

- States + probabilistic transitions
- Models „memoryless“ stochastic behaviour
- Two variants:
  - discrete time (one step per time unit, outgoing probabilities sum up to 1)
  - continuous time (outgoing transitions labelled with exit rates)
- Numerous applications

## 4. Probabilistic Automata




- Generalisation of
  - finite automata: + transition probabilities
  - Markov chain: + transition labels
- Define stochastic languages as sets of words that are recognised with a certain (minimal) probability
  - regular languages are proper subset

## 5. (Dynamic) Fault Trees




- Visualise possible failure behaviours of HW/SW system
- Represented by tree structure with gates (OR, AND, Priority-AND, ...)
- Applications in safety/reliability engineering

## Übersicht

## 6. Hoare Logic

$$\frac{\dfrac{\{A \wedge b\}\, c\, \{B\} \qquad \dfrac{(A \wedge \neg b) \Rightarrow B}{\{(A \wedge \neg b)\}\ \texttt{skip}\ \{B\}}}{\{A\}\ \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else skip}\ \{B\}}}{\{A\}\ \texttt{if}\ b\ \texttt{then}\ c\ \{B\}}$$

- Formal system for reasoning about correctness of computer programs
- Goal: establish partial (or total) correctness properties of the form

$$\{A\}\, c\, \{B\}$$

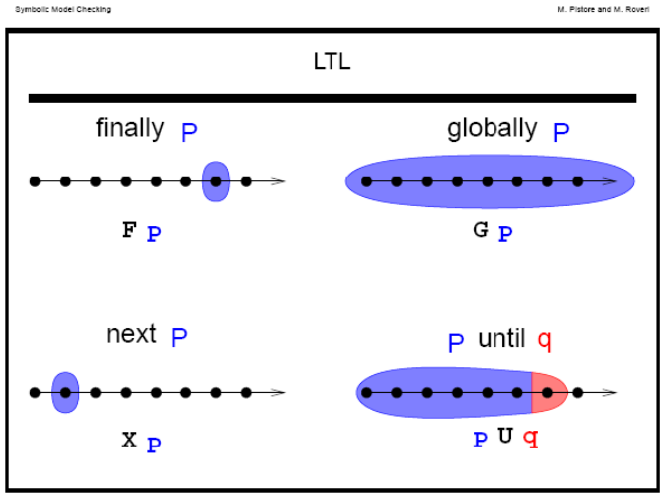
  with assertions $A$, $B$ and program $c$
- Systematic construction of proof using logical rules of the form

$$\text{(Name)}\ \frac{\text{Premise(s)}}{\text{Conclusion}}$$

- Critical part: deriving loop invariants

$$\text{(while)}\ \frac{\{A \wedge b\}\, c\, \{A\}}{\{A\}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\ \{A \wedge \neg b\}}$$

## 7. Linear-Time Temporal Logic (LTL)



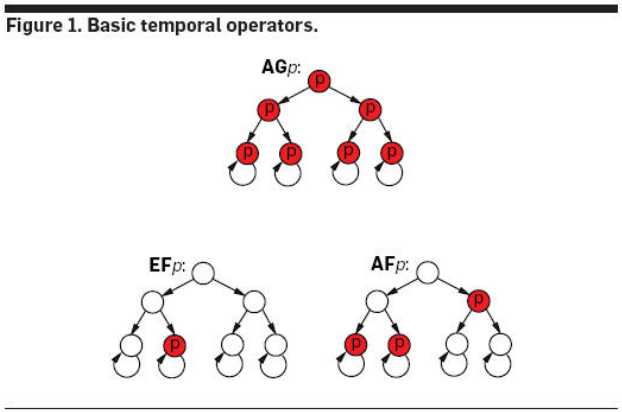

- Modal temporal logic
- Formulae specify properties of infinite system traces
- Safety: something bad never happens

$$G(\neg Bad)$$

- Liveness: something good will happen

$$G(Request \implies F\ Response)$$

## 8. Computation Tree Logic (CTL)




- Similar to LTL but supports branching time
- Time is tree structured with several possible futures
- Path operators:
  - $A\varphi$ (all): $\varphi$ has to hold on all paths starting from current state
  - $E\varphi$ (exists): there exists at least one path starting from current state where $\varphi$ holds
- State operators:
  - $X\varphi$ (next): $\varphi$ has to hold at next state
  - $G\varphi$ (globally): $\varphi$ has to hold on entire subsequent path
  - $F\varphi$ (finally): $\varphi$ has to hold somewhere on subsequent path
  - $\varphi U\psi$ (until): $\varphi$ has to hold until $\psi$ holds

## 9. Separation Logic

Frame rule of SL:

$$\text{(frame)}\ \frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

if $\mathrm{mod}(c) \cap \mathrm{free}(R) = \emptyset$

- Extension of Hoare logic to reason about programs that manipulate pointer data structures
  - dereferencing invalid pointers
  - creation of memory leaks
  - invalidation of data structures
- Additional operator $*$ (separating conjunction): expresses that heap can be split into two disjoint parts where its two arguments respectively hold

## 10. Hennessy–Milner Logic

- Modal logic used to specify properties of a labelled transition systems
- Constructs:

| | |
|---|---|
| tt | satisfied by all states |
| ff | satisfied by no state |
| $\varphi \wedge \psi$ | satisfied by all states that satisfy both $\varphi$ and $\psi$ |
| $\varphi \vee \psi$ | satisfied by all states that satisfy either $\varphi$ or $\psi$ or both |
| $\langle\alpha\rangle\varphi$ | satisfied by all states that afford an $\alpha$-labelled transition to a state satisfying $\varphi$ (possibility) |
| $[\![\alpha]\!]\varphi$ | satisfied by all states such that all their $\alpha$-labelled transitions lead to a state satisfying $\varphi$ (necessity) |

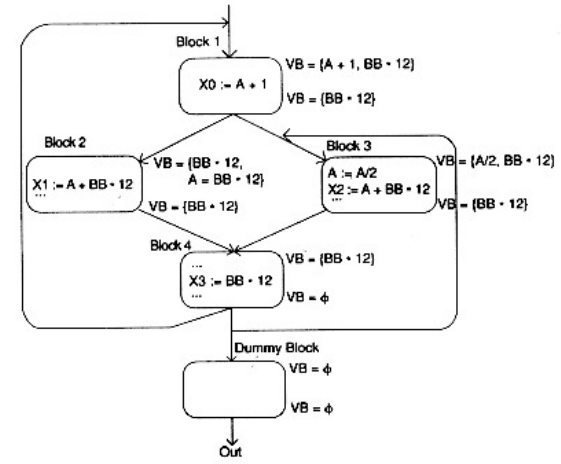
- Example: responsiveness

$$[\mathit{request}]\langle \mathit{reply}\rangle \mathrm{tt}$$

- Extension by recursion (least and greatest fixed points) to support reasoning about arbitrarily long computations (e.g., „no deadlock state reachable“)

## Übersicht

# 11. Data-Flow Analysis




- Technique for associating information with control points of computer program
  - values of variables
  - available expressions
  - live variables, ...
- Distinctions:

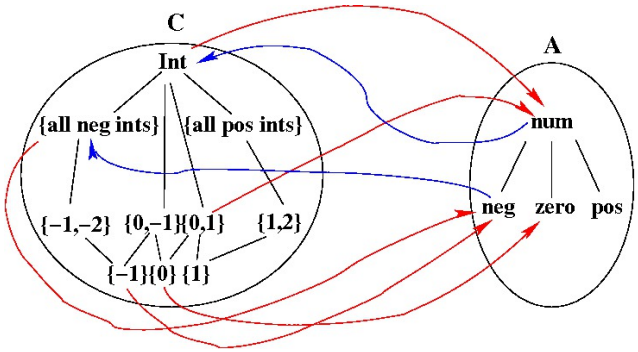
  dependence on statement order:
  flow-sensitive vs. flow-insensitive analyses

  direction of flow:
  forward vs. backward analyses

  quantification over paths:
  may (union) vs. must (intersection) analyses

  procedures:
  interprocedural vs. intraprocedural analyses
- Approach: solution of data-flow equation system by fixpoint iteration

# 12. Abstract Interpretation



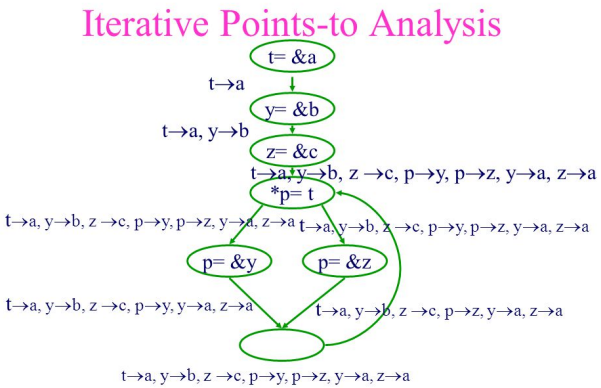

- Theory of (sound) approximation of the semantics of computer programs
  - integer values $\rightsquigarrow$ signs
  - integer values $\rightsquigarrow$ value intervals (array bounds checking)
  - concrete values $\rightsquigarrow$ types (JVM byte code verifier)
- Formalisation by abstraction and concretisation mappings that form a Galois connection
- Soundness: all concrete computations captured by abstraction

## 13. Type Systems

$$\text{(add)}\ \frac{e_1 : \text{int} \qquad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

- Type system: collection of rules that assign a type property to constructs of computer program
  - variables, expressions, functions, modules, ...
- Main purpose: reduce possibilities for bugs in computer programs
  - definition of interfaces between program parts
  - checking that parts have been connected in a consistent way
- Static vs. dynamic
- „Strong“ vs. „weak“

## 14. Points-to Analysis

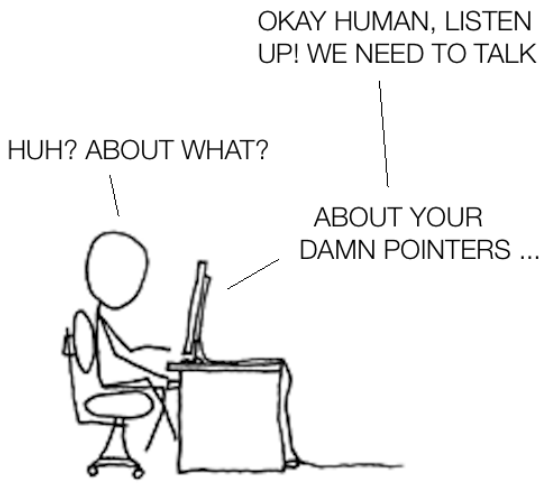



- Static code analysis technique that establishes which pointers (heap references) can point to which variables (storage locations)
- Often a component of more complex analyses such as escape analysis (topic 16)
- Example:

```
int x;
int y;
int* p = unknown() ? &x : &y;
```

yields $\{x, y\}$ as points-to set of `p`

## 15. Alias Analysis




- Determines whether or not separate memory references point to the same area of memory
- Allows compiler to determine what variables in the program will be affected by a statement
- Example:

```
p.foo = 1;
q.foo = 2;
i = p.foo + 3;
```

1. `p` and `q` cannot alias (i.e., never point to the same memory location) $\Longrightarrow$ `i` = 4
2. `p` and `q` must alias (i.e., always point to the same memory location) $\Longrightarrow$ `i` = 5
3. it cannot be conclusively determined at compile time if `p` and `q` alias or not $\Longrightarrow$ `i` $\in \{4, 5\}$

# 16. Escape Analysis

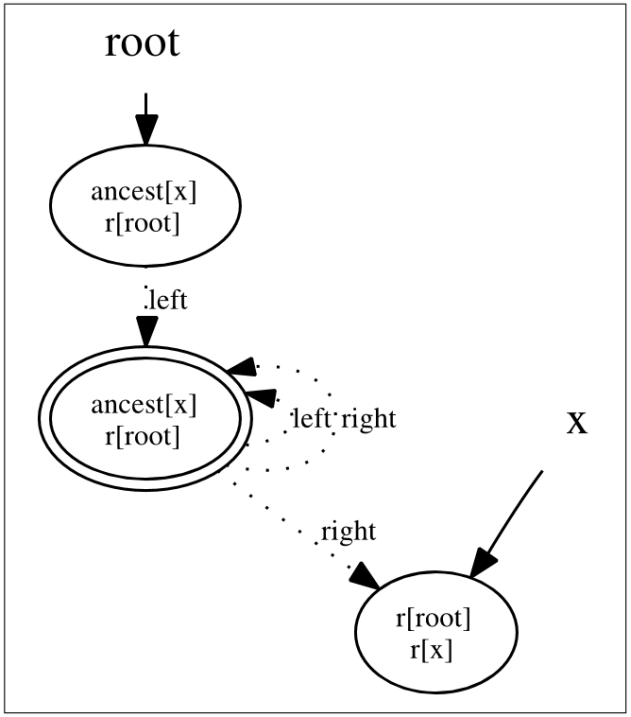
**Escape analysis example**

```
// Sum returns the sum of the numbers 1 to 100.
func Sum() int {
        numbers := make([]int, 100)
        for i := range numbers {
                numbers[i] = i + 1
        }
        var sum int
        for _, i := range numbers {
                sum += i
        }
        return sum
}
```

numbers never escapes Sum()

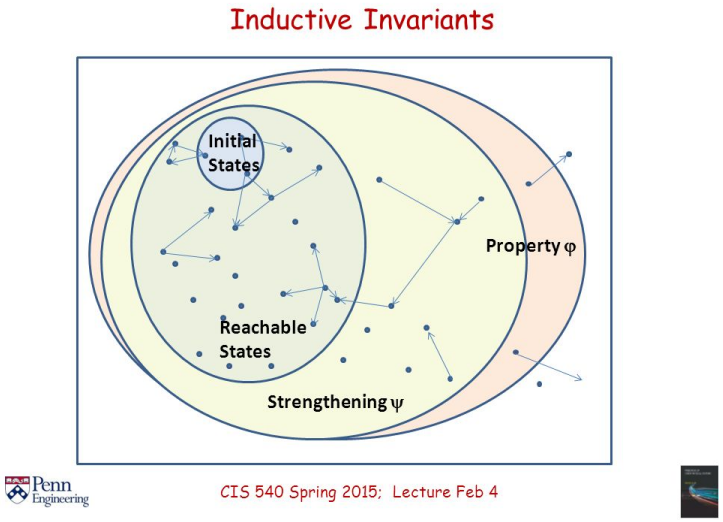
- Method for determining the dynamic scope of pointers
- When an object is allocated in a subroutine, a pointer to that object can escape to other contexts
- Non-escaping results can be used as basis for (compiler) optimisation:
  - converting heap allocations to stack allocations (avoids garbage collection)
  - removal of redundant synchronisation operations (if object accessible from one thread only)

## 17. Shape Analysis



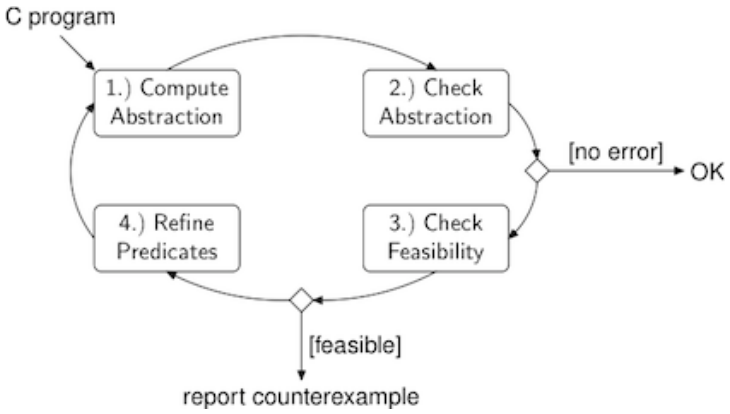

- Technique that discovers and verifies properties of linked, dynamically allocated data structures
- Used at compile time to find elementary bugs or to verify high-level correctness properties of programs
  - absence of memory leaks
  - correctness of list-sorting method
- Abstraction of state space through summary nodes

# 18. Inductive Invariants




- Technique to establish invariant safety properties: „it is never the case that ...“
  - the value of variable *x* becomes zero
  - more than one process is in the critical section
  - ...
- Inductive:
  - satisfied by initial state(s) of program
  - closed under execution steps

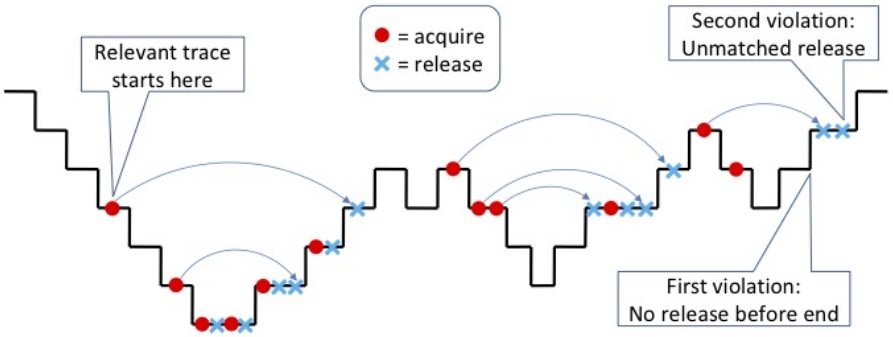
## 19. Counterexample-Guided Abstraction Refinement (CEGAR)




- Iterative procedure for checking safety properties
- To cope with state explosion problem in state-space exploration
- Start with simple abstraction of system with only few states
- In each iteration, check whether abstract system satisfies property
  - if yes, system is safe
  - if no, check feasibility of counterexample
    - if feasible, system is unsafe
    - otherwise, refine abstraction

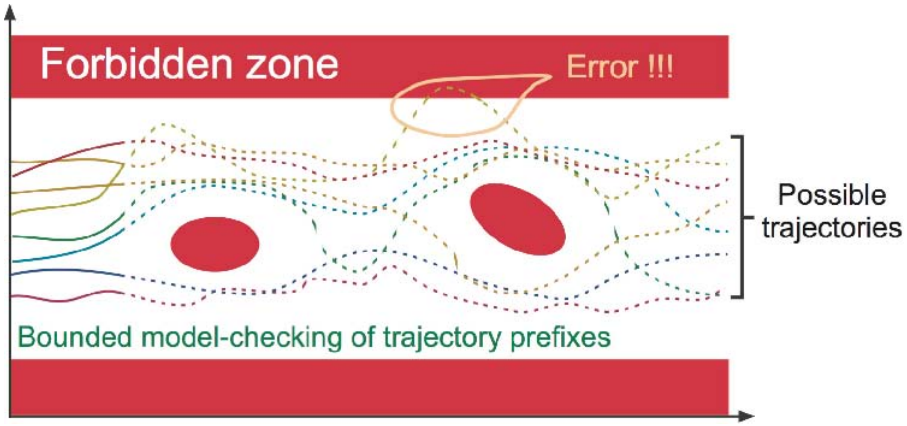
## Übersicht

## 20. Runtime Verification



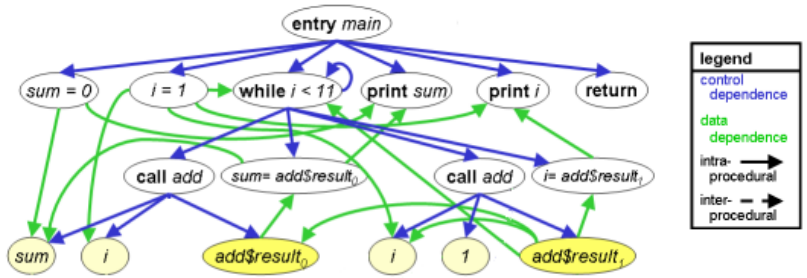

- Observation of running system to detect (and possibly react to) behaviours violating certain properties
  - deadlocks
  - data races
  - improper use of synchronisation mechanisms
  - ...
- Desired properties specified as predicates over execution traces
  - finite automata/regular expressions
  - context-free grammars
  - linear temporal logics (LTL, ...)
  - ...

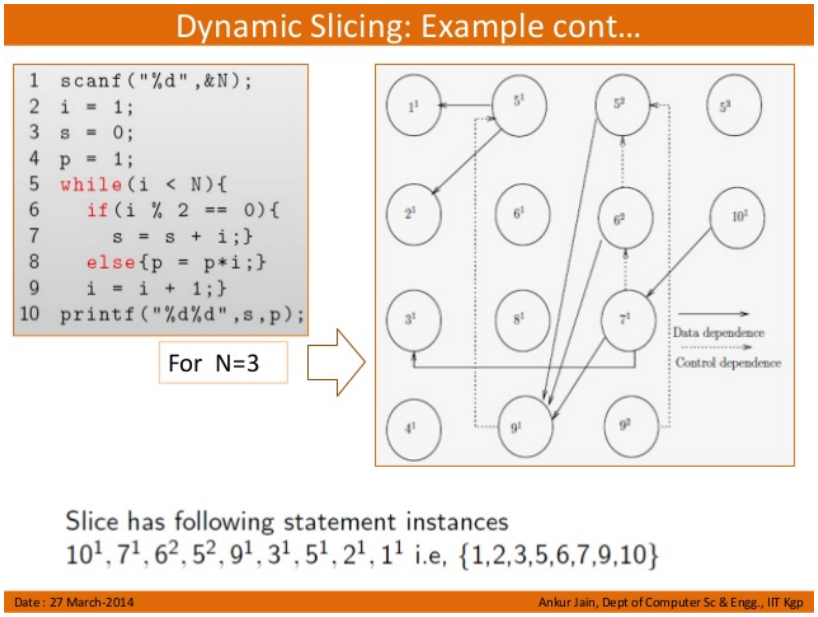
## 21. Bounded Model Checking




- Model checking: exploration of state space of a system to match against specification
- Usually exhaustive
  $\Longrightarrow$ state-space explosion problem
- Bounded model checking: fast exploration of bounded fragment of state space

## 22. Program Slicing




- Computation of the part of program („program slice“) that may affect the values at some point of interest („slicing criterion“)
- Static slicing:
  - no assumptions regarding input
  - based on program dependence graph
  - applications: software maintenance (regression testing), information flow control
- Dynamic slicing:
  - assumes fixed input for program
  - based on execution trace
  - applications: debugging