



Proseminar *Einführung in die Programmanalyse*

Einführungsveranstaltung

Wintersemester 2016/17; 21. Oktober 2016

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ws-1617/epa/>

Einführung

Übersicht

Einführung

Termine

Modelle

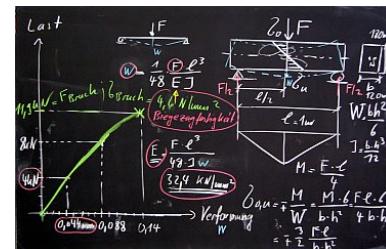
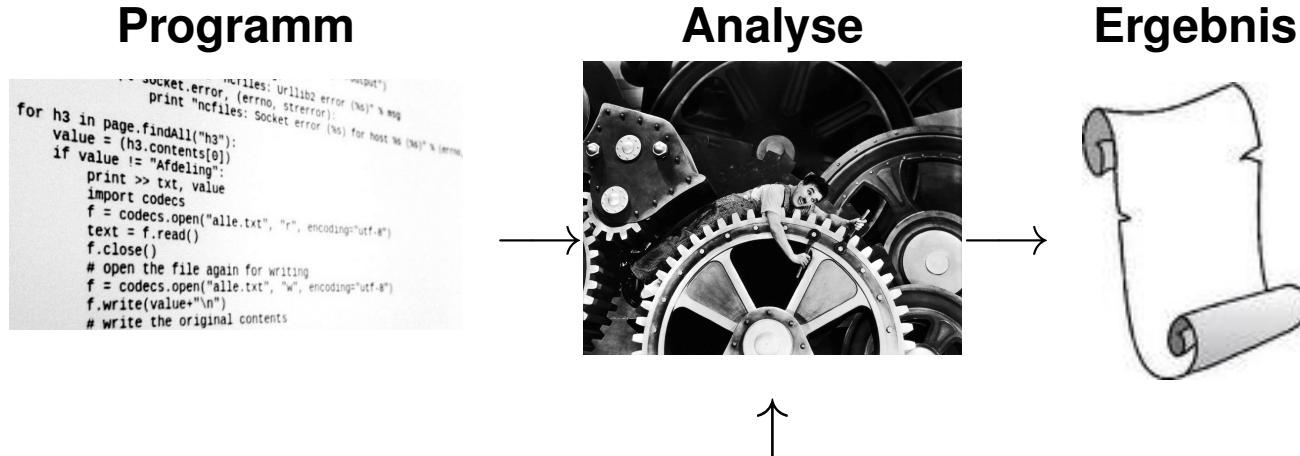
Logiken

Statische Analyseverfahren

Dynamische Analyseverfahren

Hybride Analyseverfahren

Traum der Programmanalyse



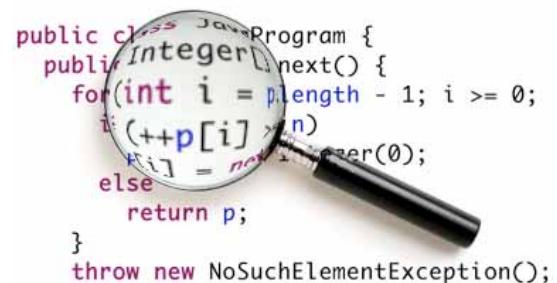
Spezifikation

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
- Dynamische Verfahren:
 - automatische Analyse von Programmeigenschaften
 - basierend auf beobachtetem Ausführungsverhalten



Einführung

Zielsetzung

Ziele des Proseminars

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

Einführung

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 **ca. 15-seitigen** Ausarbeitung
- **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
- **Sprache** Deutsch oder Englisch
- **Korrekte Sprache** wird vorausgesetzt:
 ≥ 10 Fehler pro Seite \Rightarrow Abbruch der Korrektur

Anforderungen Vortrag

Vortrag

- **30-minütiger** Vortrag (25 + 5 min)
- **Zielgruppengerechte** Präsentation der Inhalte
- **übersichtliche** Folien:
 - \leq 15 Textzeilen
 - sinnvoller Einsatz von Farben
- Vortrag in **Deutsch oder Englisch**

Übersicht

Einführung

Termine

Modelle

Logiken

Statische Analyseverfahren

Dynamische Analyseverfahren

Hybride Analyseverfahren

Termine

Themenauswahl

Verfahren

- Themenliste wurde/wird ausgehändigt
- Priorisierte Auswahl
- ggf. Angabe Wunschpartner(in)
- Abgabe bis (spätestens) Montag 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 werden per Mail abgeklärt

Deadlines

Deadlines

Folgende Termine sind **einzuhalten**:

- 11.11.2016: letzte Rücktrittsmöglichkeit
- 28.11.2016: Vorlage der detaillierten Inhaltsübersicht
- 09.01.2017: endgültige Fassung der Ausarbeitung
- 30.01.2017: endgültige Fassung der Folien
- 06./07.02.2017: Blockseminar

Übersicht

Einführung

Termine

Modelle

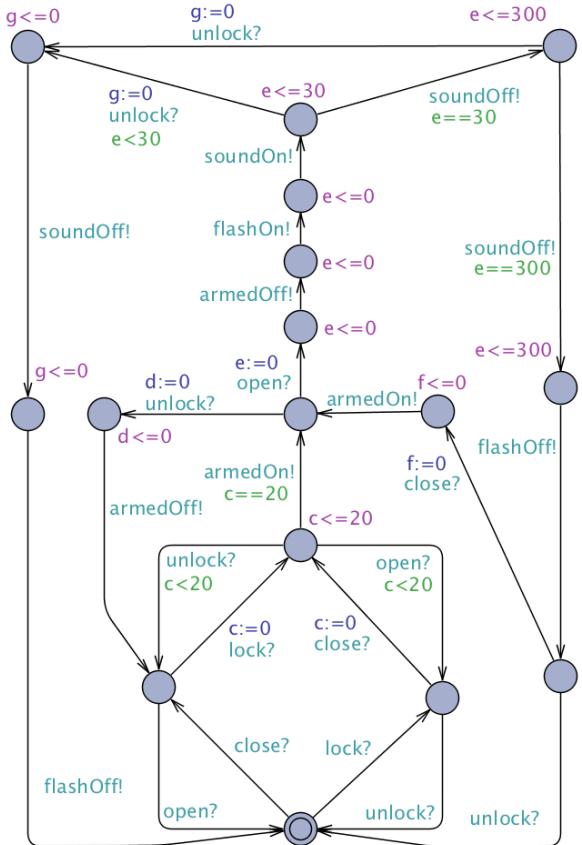
Logiken

Statische Analyseverfahren

Dynamische Analyseverfahren

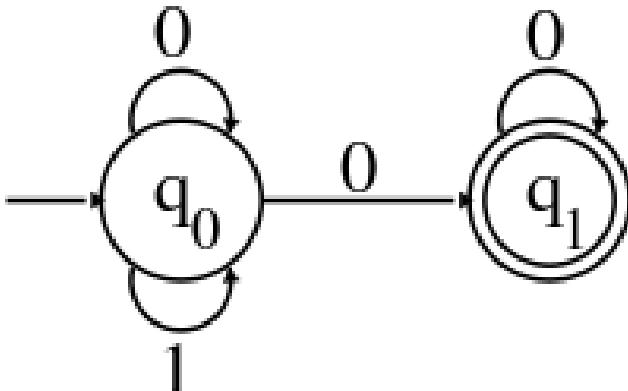
Hybride Analyseverfahren

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)

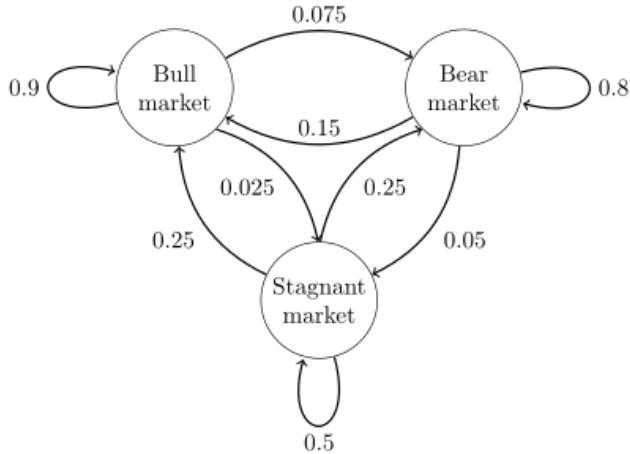
2. Büchi Automata



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

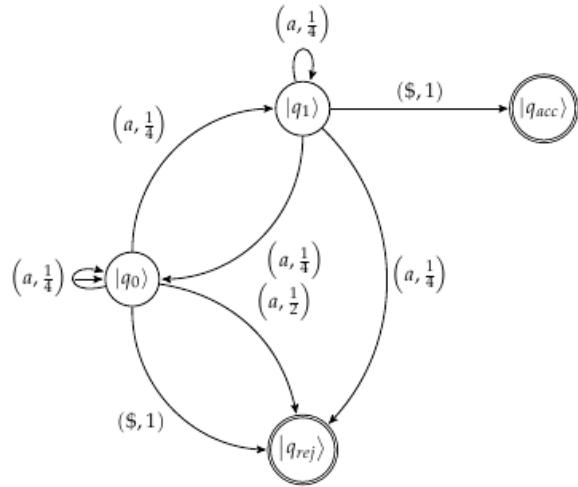
- Syntax identical to DFA/NFA
- But define languages of infinite words
- Accepts an infinite input sequence if there exists a run that visits one of the final states infinitely often
- Non-deterministic variant more expressive

3. Markov Chains



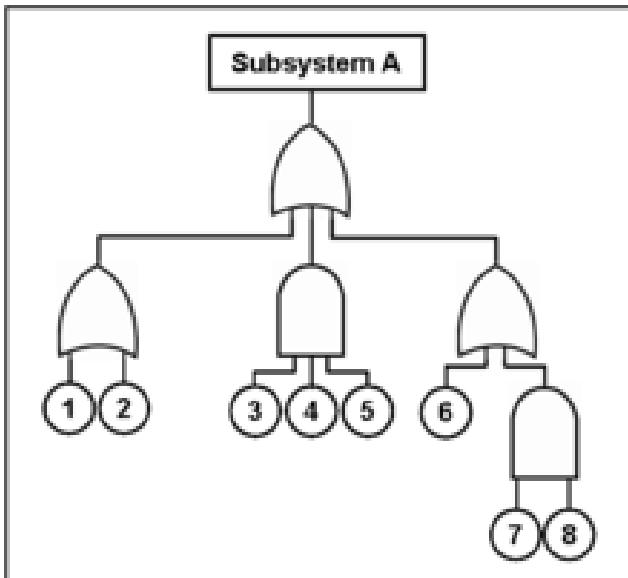
- States + probabilistic transitions
- Models „memoryless“ behaviour
- Two variants:
 - discrete time (one step per time unit, outgoing probabilities sum up to 1)
 - continuous time (outgoing transitions labelled with exit rates)
- Numerous applications

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

Einführung

Termine

Modelle

Logiken

Statische Analyseverfahren

Dynamische Analyseverfahren

Hybride Analyseverfahren

6. Hoare Logic

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

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

$$\frac{\{A \wedge b\} c \{B\} \quad \frac{(A \wedge \neg b) \Rightarrow B}{\{(A \wedge \neg b)\} \text{ skip } \{B\}}}{\begin{array}{c} \{A\} \text{ if } b \text{ then } c \text{ else skip } \{B\} \\ \{A\} \text{ if } b \text{ then } c \{B\} \end{array}}$$

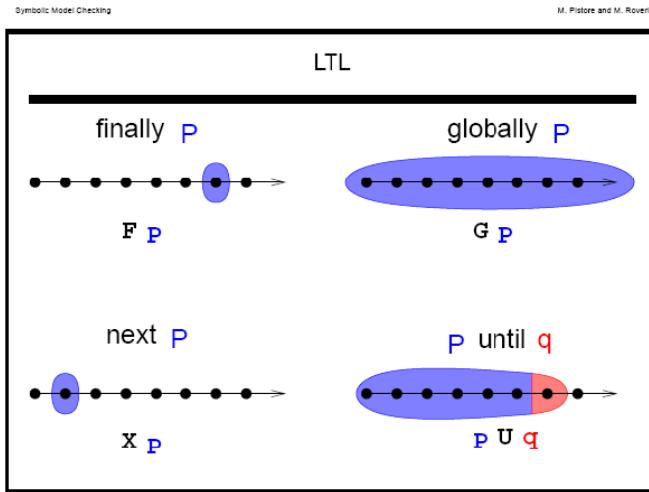
- Systematic construction of proof using logical rules of the form

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

- Critical part: deriving loop invariants

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

7. Linear-Time Temporal Logic (LTL)



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

$$G(\neg \text{Bad})$$

- Liveness: something good will happen

$$G(\text{Request} \implies F \text{ Response})$$

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): φ has to hold on all paths starting from current state

$E\varphi$ (exists): there exists at least one path starting from current state where φ holds

- State operators:

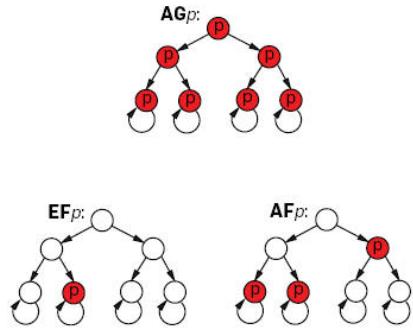
$X\varphi$ (next): φ has to hold at next state

$G\varphi$ (globally): φ has to hold on entire subsequent path

$F\varphi$ (finally): φ has to hold somewhere on subsequent path

$\varphi U \psi$ (until): φ has to hold until ψ holds

Figure 1. Basic temporal operators.



9. Separation Logic

Frame rule of SL:

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

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

- 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 φ and ψ
 - $\varphi \vee \psi$ satisfied by all states that satisfy either φ or ψ or both
 - $\langle \alpha \rangle \varphi$ satisfied by all states that afford an α -labelled transition to a state satisfying φ (possibility)
 - $[\alpha] \varphi$ satisfied by all states such that all their α -labelled transitions lead to a state satisfying φ (necessity)
- Example: responsiveness
 - $[\text{request}] \langle \text{reply} \rangle \text{tt}$
- Extension by recursion (least and greatest fixed points) to support reasoning about arbitrarily long computations (e.g., „no deadlock state reachable“)

Statische Analyseverfahren

Übersicht

Einführung

Termine

Modelle

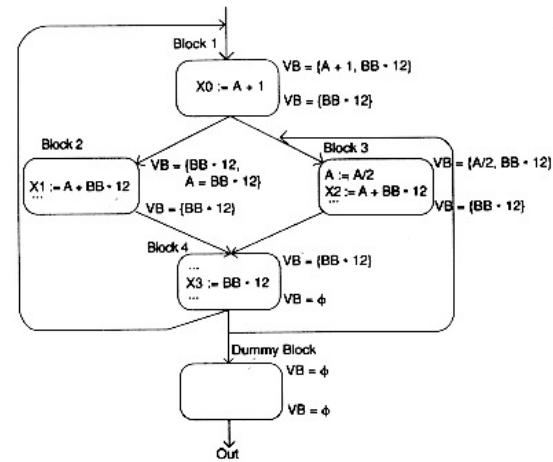
Logiken

Statische Analyseverfahren

Dynamische Analyseverfahren

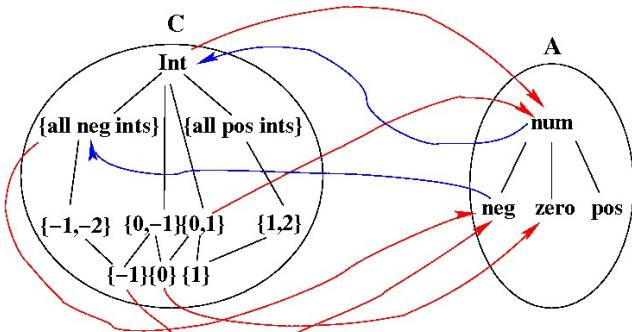
Hybride Analyseverfahren

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

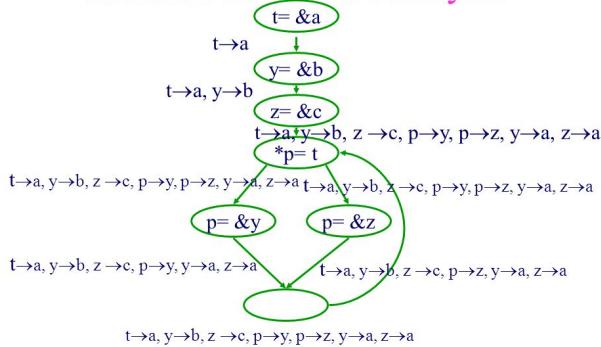
13. Type Systems

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

- 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

Iterative Points-to Analysis

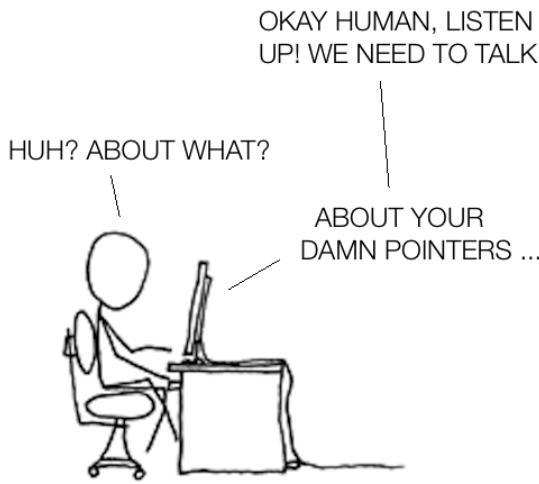


- Static code analysis technique that establishes which pointers (heap references) can point to which variables (storage locations)
- Often a component of more complex analyses such as escape analysis (topic 16)
- Example:

```
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)
 2. p and q must alias (i.e., always point to the same memory location)
 3. it cannot be conclusively determined at compile time if p and q alias or not

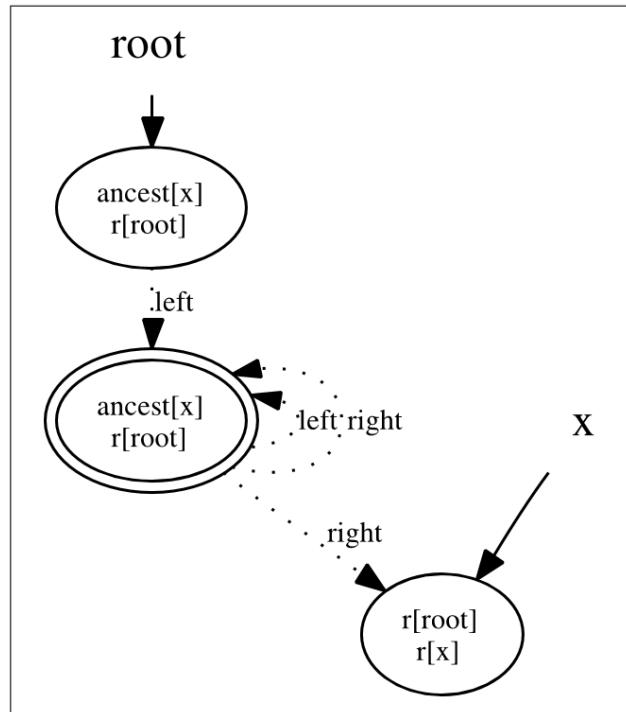
16. Escape Analysis

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
}
```

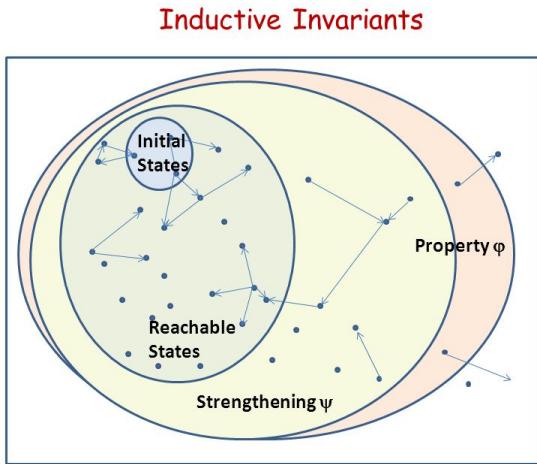
- Method for determining the dynamic scope of pointers
- When an object is allocated in a subroutine, a pointer to that object can escape to other contexts
- Results can be used as basis for (compiler) optimisation:
 - converting heap allocations to stack allocations (avoids garbage collection)
 - removal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

17. Shape Analysis



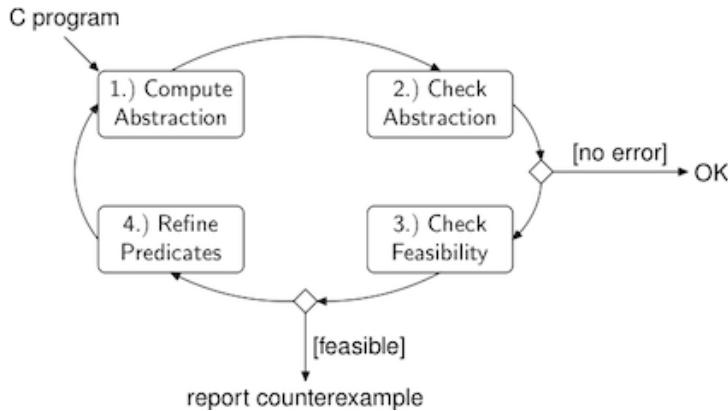
- Technique that discovers and verifies properties of linked, dynamically allocated data structures
- Used at compile time to find elementary bugs or to verify high-level correctness properties of programs
 - absence of memory leaks
 - correctness of list-sorting method
- Abstraction of state space through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

Dynamische Analyseverfahren

Übersicht

Einführung

Termine

Modelle

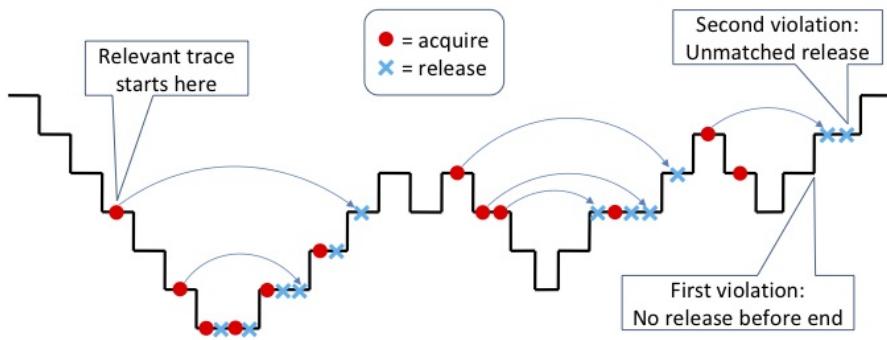
Logiken

Statische Analyseverfahren

Dynamische Analyseverfahren

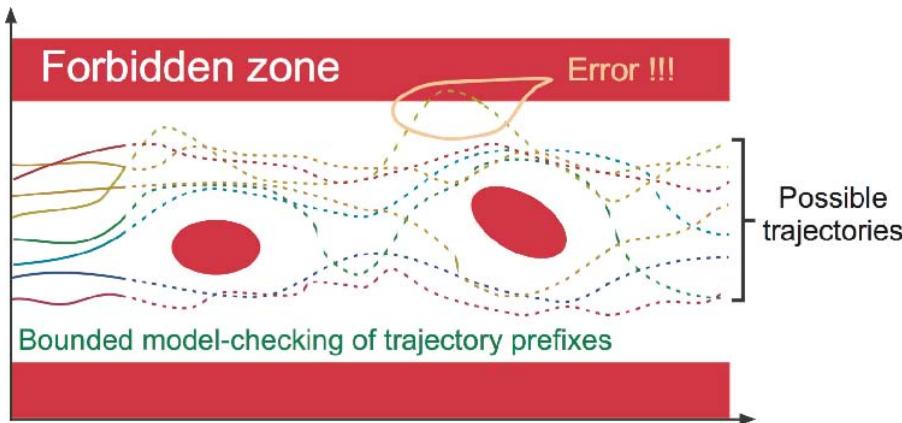
Hybride Analyseverfahren

20. Runtime Verification



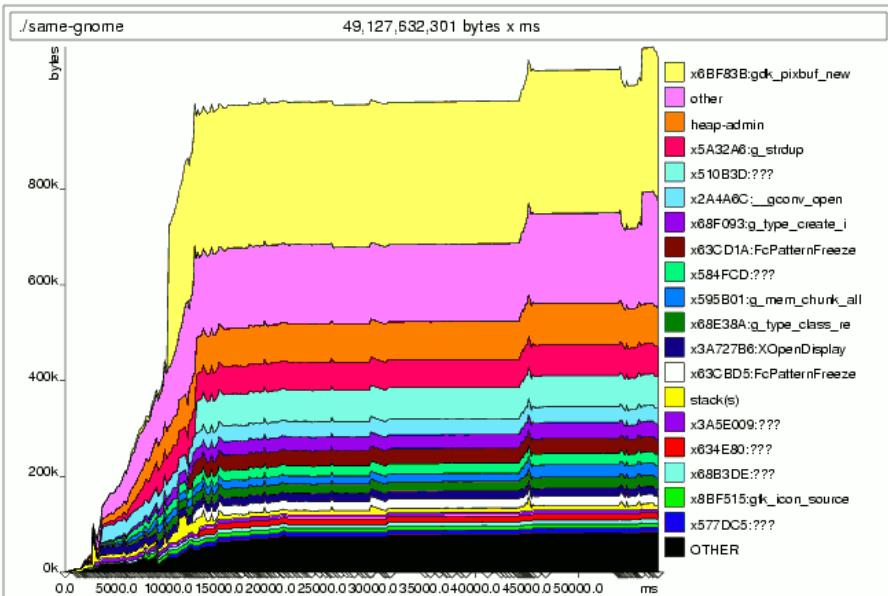
- Observation of running system to detect (and possibly react to) behaviours violating certain properties
 - deadlocks
 - data races
 - improper use of synchronisation mechanisms
 - ...
- Desired properties specified as predicates over execution traces
 - finite automata/regular expressions
 - context-free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

21. Bounded Model Checking



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

22. Profiling



- Collection of information of running program for optimisation purposes
 - memory usage
 - runtime (of specific parts)
 - usage of particular instructions
 - frequency of function calls
- Technique: instrumentation of source code or binary executable, possibly resulting in
 - performance changes
 - inaccurate results/heisenbugs

Hybride Analyseverfahren

Übersicht

Einführung

Termine

Modelle

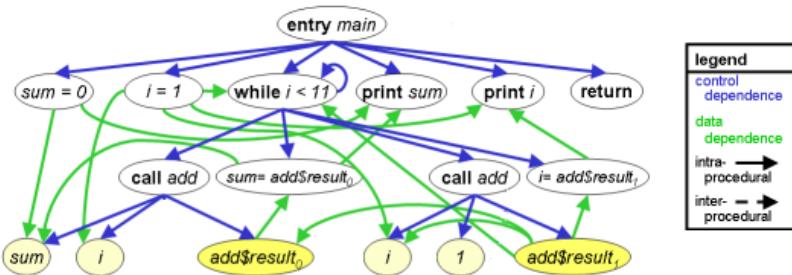
Logiken

Statische Analyseverfahren

Dynamische Analyseverfahren

Hybride Analyseverfahren

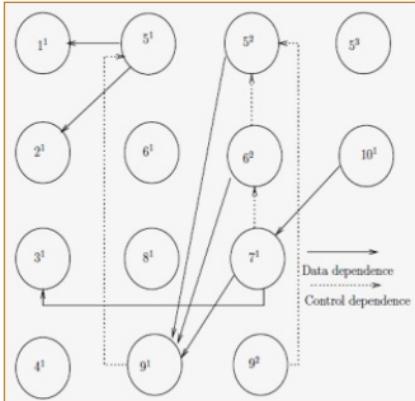
23. Program Slicing



Dynamic Slicing: Example cont...

```
1 scanf ("%d", &N);
2 i = 1;
3 s = 0;
4 p = 1;
5 while(i < N){
6   if(i % 2 == 0){
7     s = s + i;};
8   else{p = p*i;};
9   i = i + 1;};
10 printf ("%d%d", s, p);
```

For N=3



Slice has following statement instances
10¹, 7¹, 6², 5², 9¹, 3¹, 5¹, 2¹, 1¹ i.e., {1,2,3,5,6,7,9,10}

- 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