Proseminar *Einführung in die Programmanalyse*

Einführung

Sommersemester 2020; April 2020

Thomas Noll et al.
Software Modeling and Verification Group
RWTH Aachen University

`https://moves.rwth-aachen.de/teaching/ss-20/ipa/`

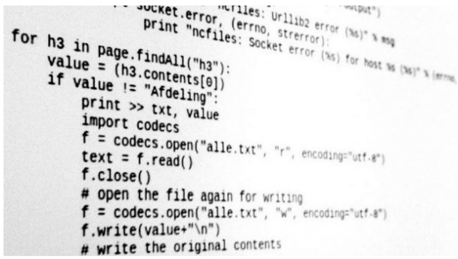
Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

Übersicht

Traum der Programmanalyse

Programm

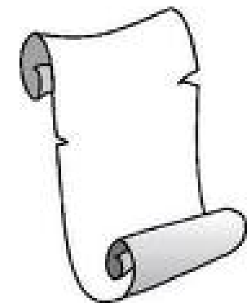


$\longrightarrow$

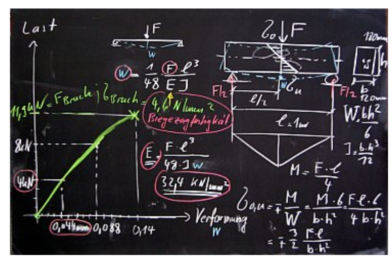
Analyse

$\longrightarrow$

Ergebnis



$\uparrow$



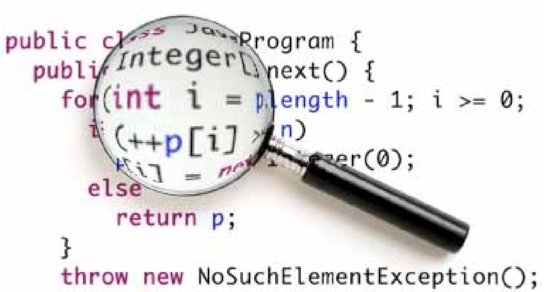
Spezifikation

Seminarthema

Thema des Proseminars

Einführung in Grundkonzepte der (automatisierten) 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/hybride 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

Bearbeitung in Zweiergruppen

- Gemeinsame Anfertigung der Ausarbeitung
- Zwei separate Vorträge (mit nicht notwendigerweise verschiedenen Foliensätzen)

Anforderungen Ausarbeitung

Ausarbeitung

- Selbstständiges Verfassen einer Ausarbeitung von ≥ 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
- LaTeX-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
- LaTeX/`beamer`-Vorlage wird zur Verfügung gestellt
- Vortrag in Deutsch oder Englisch

Übersicht

Einführung

Termine

Modelle

Logiken

Statische Analyseverfahren

Dynamische und Hybride Analyseverfahren

Themenauswahl

Verfahren

- Foodle-Umfrage unter `https://terminplaner4.dfn.de/AIP8MJyxqHvJsncj`
- Bitte mindestens drei Ja-Stimmen ✓
- Möglichst weitere Vielleicht-Stimmen (✓)
- Als Kommentar angeben:
  - Themenpriorität (falls gewünscht)
  - Wunschpartner(in) (falls vorhanden)
  - Sprache Ausarbeitung und Vortrag (Deutsch/Englisch)
- Ausfüllen bis (spätestens) Freitag, 17. April
- Wir bemühen uns (ohne Garantie) um ein „optimales“ Matching
- Zuordnung der Themen und Betreuer bis Mitte KW 17 online

Rücktritt vom Proseminar

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

Bibliothekseinführung

Einführung in die Literaturrecherche

- Einweisung in themenspezifische Literaturrecherche
- Teilnahme für BSc-Studierende verpflichtend
- Video wird per RWTHmoodle bereitgestellt

Deadlines

Deadlines

Folgende Termine sind verpflichtend:

- 17.04.2020: Frist für Themenauswahl
- 11.05.2020: letzte Rücktrittsmöglichkeit
- 18.05.2020: Vorlage der detaillierten Inhaltsübersicht
- 08.06.2020: Vollständige Fassung der Ausarbeitung
- 06.07.2020: Vollständige Fassung der Folien
- 15./16.07.2020 (?): Blockseminar

Übersicht

Einführung

Termine

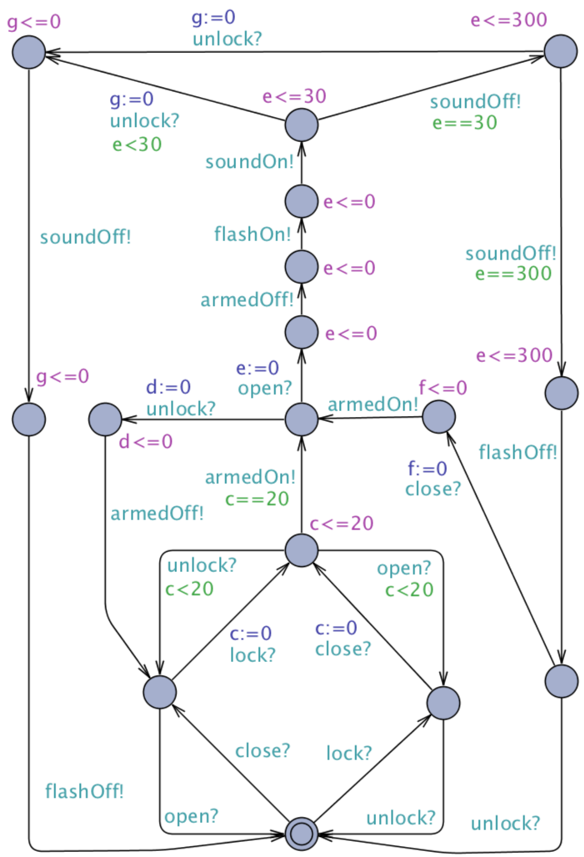
Modelle

Logiken

Statische Analyseverfahren

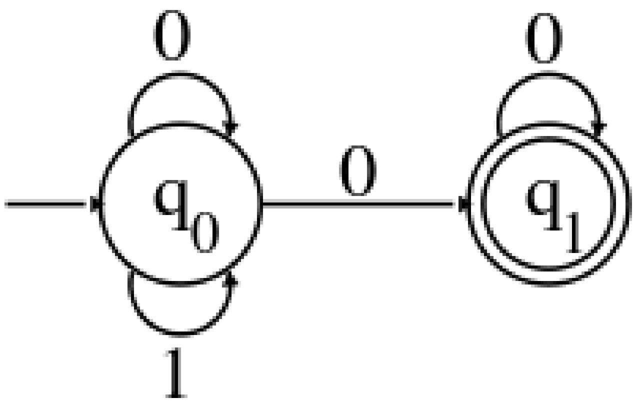
Dynamische und Hybride Analyseverfahren

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)
- Enables modelling and analysis of time-dependent system behaviour

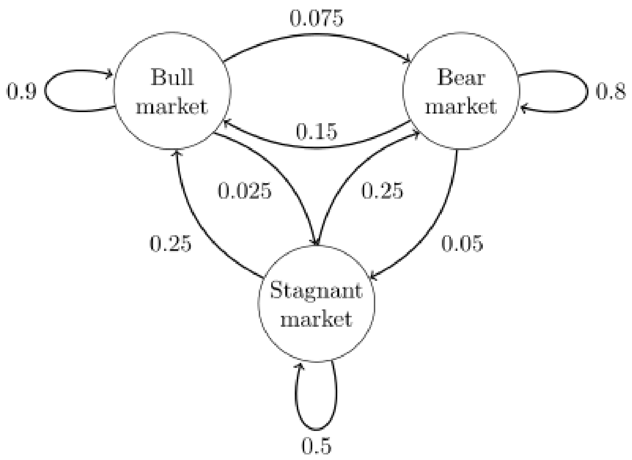
2. Büchi Automata



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

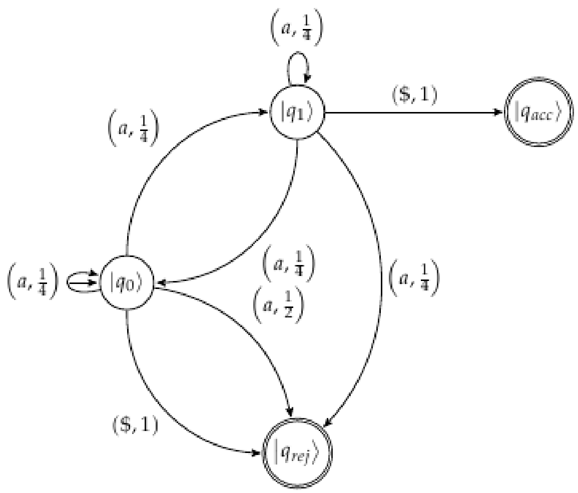
- Syntax identical to DFA/NFA
- But define languages of infinite words
- Accepts an infinite input sequence if there exists a run that visits one of the final states infinitely often
- Enables analysis of non-terminating system behaviour
- Non-deterministic variant more expressive

3. Markov Chains



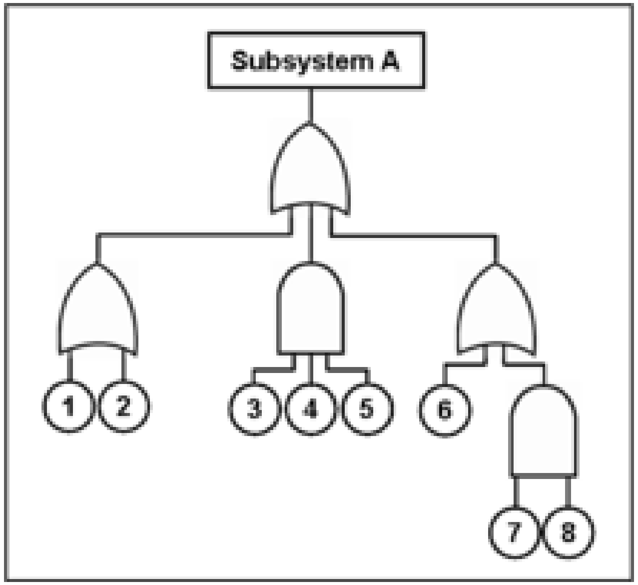
- States + probabilistic transitions
- Models „memoryless“ stochastic behaviour
- Two variants:
  - discrete time (one step per time unit, outgoing probabilities sum up to 1)
  - continuous time (outgoing transitions labelled with exit rates)
- Numerous applications

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 subclass

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

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

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

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

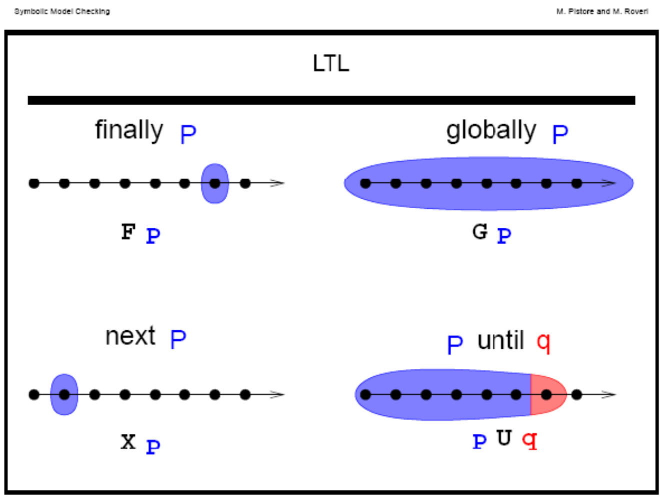
  with assertions A, B and program c
- Systematic construction of proof using logical rules of the form

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

- 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

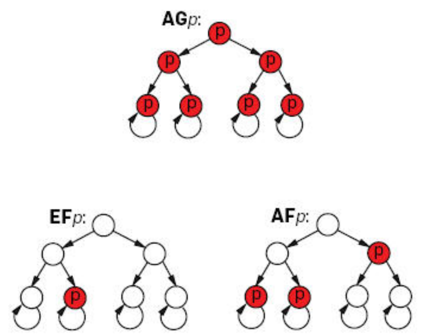
$$G(\neg Bad)$$

- Liveness: something good will happen

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

8. Computation Tree Logic (CTL)

Figure 1. Basic temporal operators.



- 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

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 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)

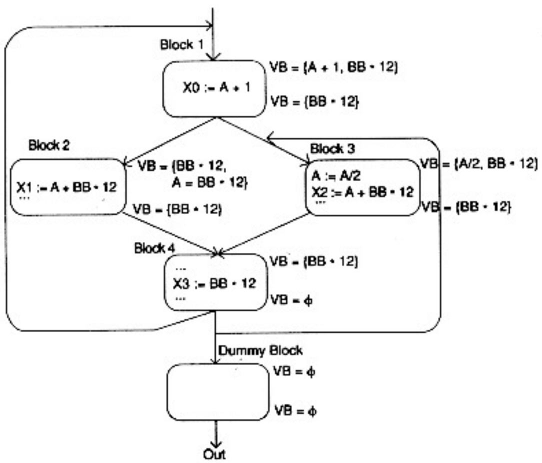
- Example: responsiveness

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

- 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

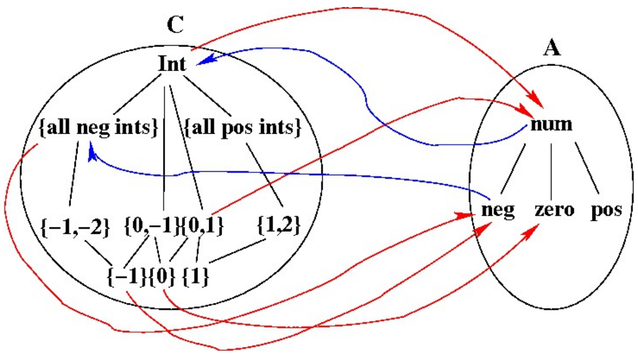
  quantification over paths:

  may (union) vs. must (intersection) analyses

  procedures:

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

12. Abstract Interpretation



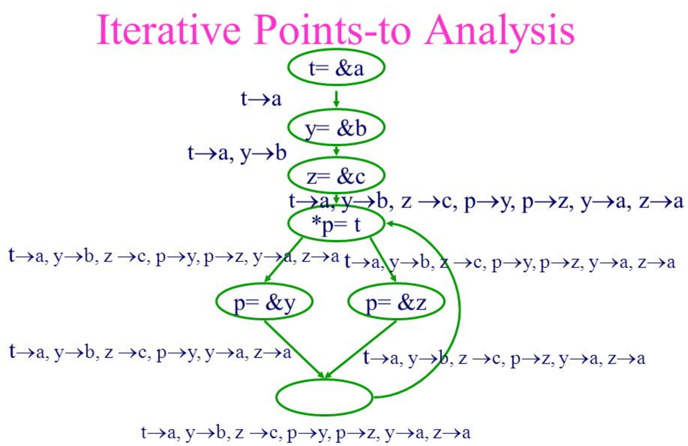
- Theory of (sound) approximation of the semantics of computer programs
  - integer values $\rightsquigarrow$ signs
  - integer values $\rightsquigarrow$ value intervals (array bounds checking)
  - concrete values $\rightsquigarrow$ types (JVM byte code verifier)
- Formalisation by abstraction and concretisation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

13. Type Systems

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

- 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. Pointer Analysis



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

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

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

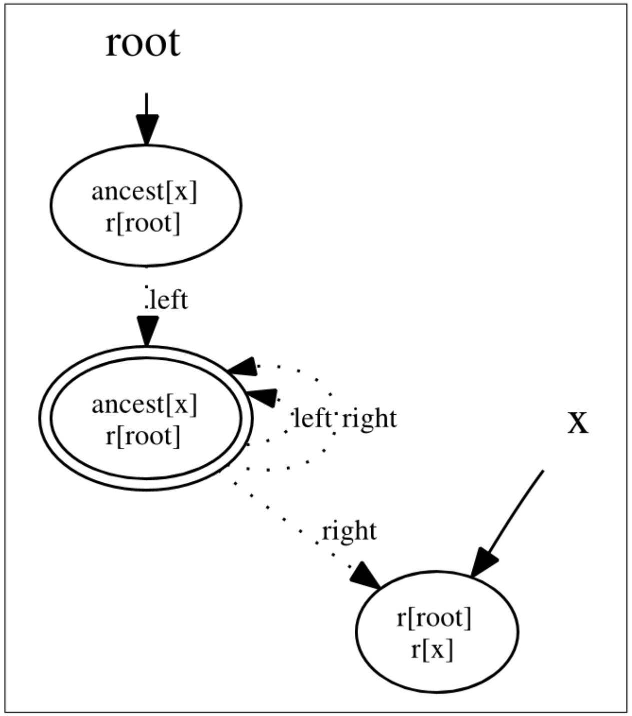
15. Escape Analysis

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
}
```

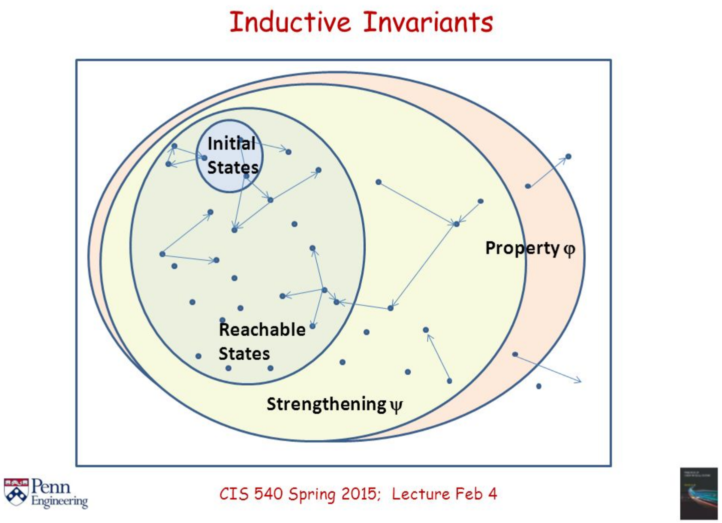
numbers never escapes Sum()

- 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)



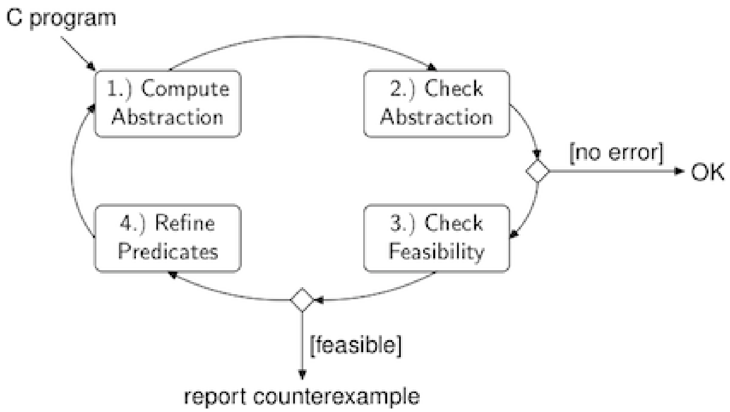
- Technique that discovers and verifies properties of linked, dynamically allocated data structures
- Used at compile time to find elementary bugs or to verify high-level correctness properties of programs
  - absence of memory leaks
  - non-destructive traversal
  - correctness of list-sorting method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

17. Inductive Invariants



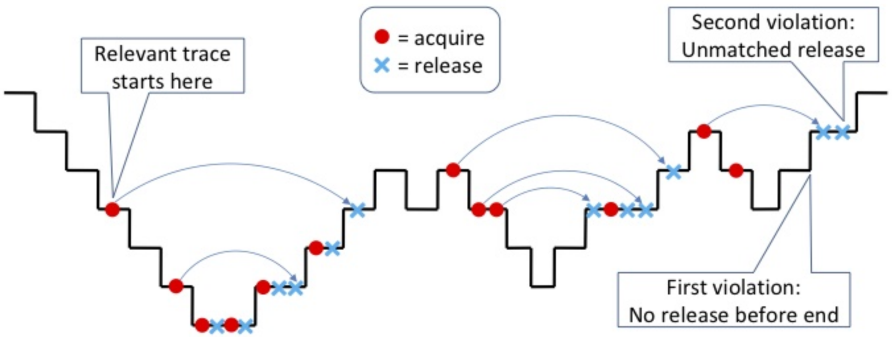
- Technique to establish invariant safety properties: „it is never the case that ...“
  - the value of variable *x* becomes zero
  - more than one process is in the critical section
  - ...
- Inductive:
  - satisfied by initial state(s) of program
  - closed under execution steps

18. Counterexample-Guided Abstraction Refinement (CEGAR)



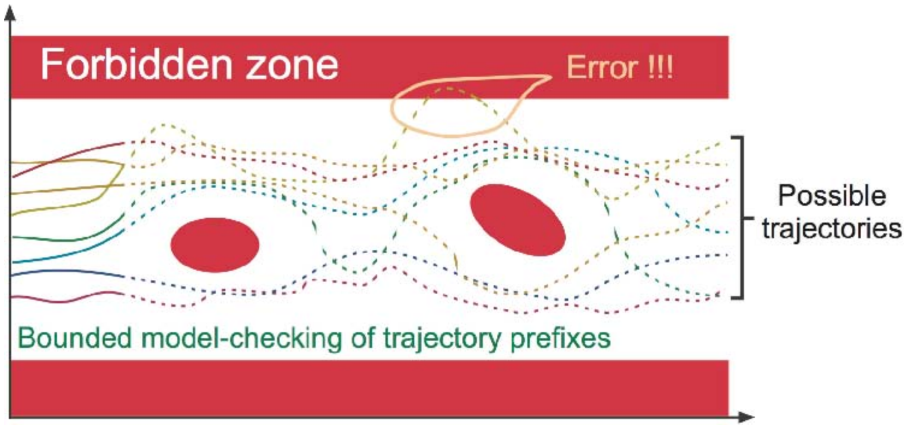
- Iterative procedure for checking safety properties
- To cope with state explosion problem in state-space exploration
- Start with simple abstraction of system with only few states
- In each iteration, check whether abstract system satisfies property
  - if yes, system is safe
  - if no, check feasibility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

Übersicht



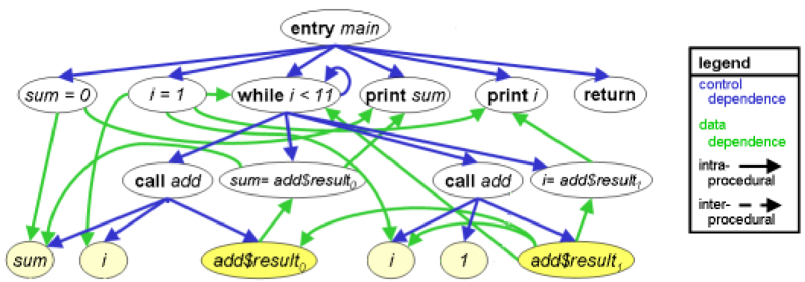
- Observation of running system to detect (and possibly react to) behaviours violating certain properties
  - deadlocks
  - data races
  - improper use of synchronisation mechanisms
  - ...
- Desired properties specified by predicates over execution traces
  - finite automata/regular expressions
  -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

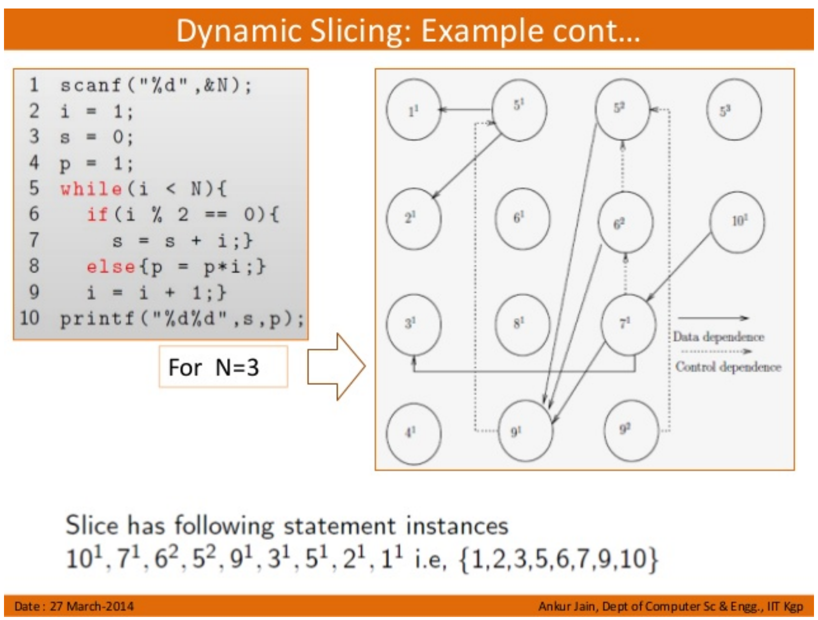
20. Bounded Model Checking



- 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

21. 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