Model Checking, Counterexamples, and Questions Regarding Visualization

Stefan Leue

University of Konstanz
Chair for Software Engineering

Stefan.Leue@uni-konstanz.de
http://www.inf.uni-konstanz.de/~soft

16 September 2004

Universität
Konstanz

Software Systems

- **Systems**
  - software is a determining, if not the most determining, technology in most system development projects
  - current software artifacts are large and highly complex
    - carrier-class switches: > 50 mio loc
    - operating systems: > 50 mio loc
    - voice over IP switches: 250+ kloc
    - automobile engine control: 150+ kloc
    - medical software (defibrilator)
      - 60 kloc embedded plus 200 kloc in external control device
  - in particular in embedded systems sofware is increasingly important
    - an S class Mercedes has more than 100 processors built in

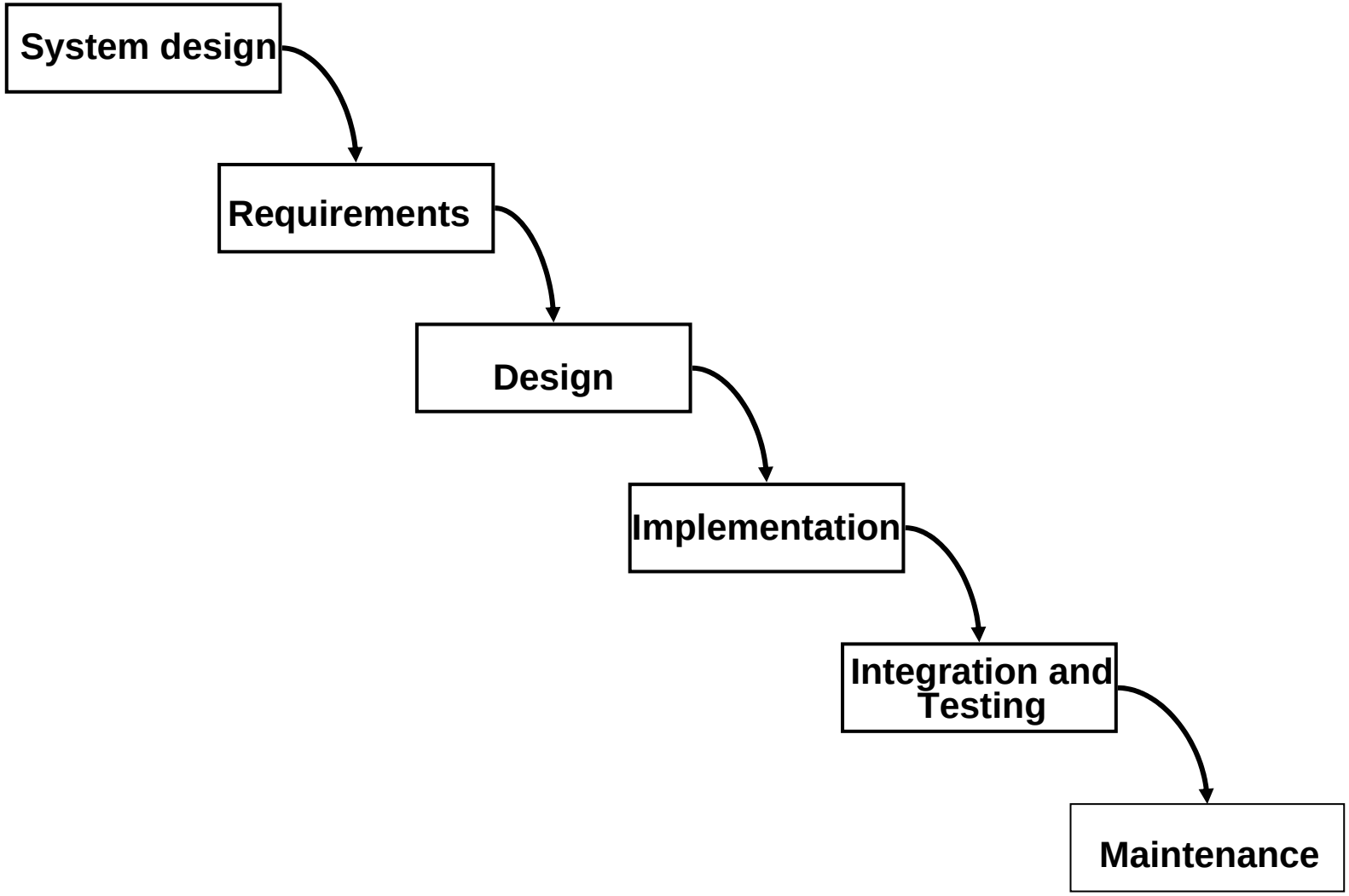
Software Systems

- **Transformational Systems**
  - finite, terminating input-output computation
    - compiler, batch data processing, scientific computation,...
- **Reactive Systems**
  - ongoing, non-terminating stimulus-response interaction with environment
- **Embedded Systems**
  - mostly reactive systems
  - sw tightly integrated with the software that it controls
    - telephone switches, cell phone sw, ...
- **Real-Time Systems**
  - reactive systems with time bounds on stimulus-response events
    - hard real-time systems: may not violate bounds
      - medical systems, engine control, ...
    - soft real-time systems: may violate bounds

Software Crisis

- **Survey by PC Week, 1995**
  - query of 365 information systems professionals on success of software development projects
    - 16% successful
    - 53% operational, but less than successful
    - 31% cancelled
- **Survey by Standish Group 1994**
  - Analysis of 350 companies with over 8000 software projects
    - 31 % of projects were aborted prior to completion
    - in large development companies, only 9% of all projects were completed within projected budget and time limits (16% in small companies)
- **Survey by IEEE Software, 1995**
  - 30% of all software procects are cancelled
  - 50% of all projects are 150% over budget
  - only 60% of functionality is in the final product

Life-cycle Model: Waterfall



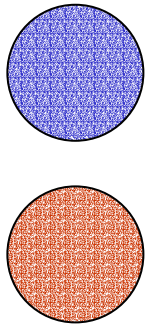
A Software Verification Method

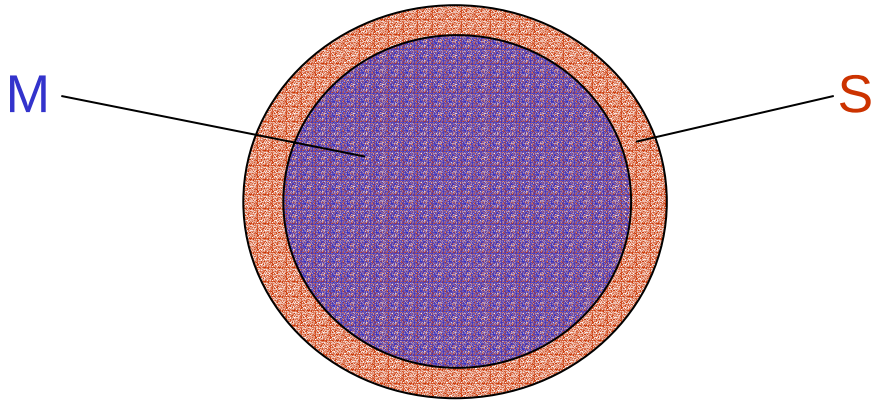
- **Requirements**
  - verification method should possess a formal foundation
    - enables tool implementation
  - verification method should be capable of relating artifacts from different stages of the software design cycle, e.g.,
    - formal requirements vs. informal requirements
    - design vs. requirements
    - code vs. design or code vs. requirements
  - verification method should be easy to integrate in software design cycle
    - high degree of automation ("push-button")
    - low degree of interaction
  - highly scalable
    - deal with complex software artifacts
- **Model Checking**
  - ... is a method meeting many of the above requirements

Model Checking

- **What is Model Checking?**
  - basic principle
    - given
      - (software-) model M
      - property specification S
    - does M satisfy S?

      M |= S
      - that is the case if every behaviour of M is also a behaviour of S
      - i.e., the model M does not reveal behaviour violating the specification S



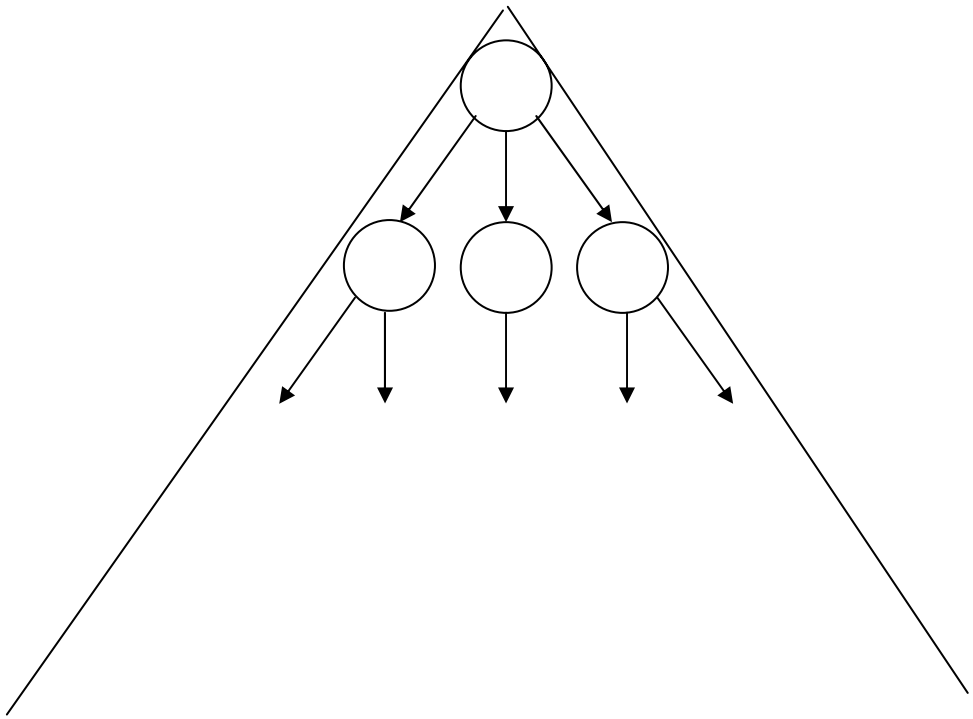


Model Checking

- **Principle**
  - systematic analysis of the state space of the software system
    - work on model (abstraction) of software
      - "classical" model checking
    - work directly on source code of the software system
      - "software" model checking
- **Challenges**
  - combinatorial explosion of the state space size
    - data
    - concurrent composition
- **Principal Approaches**
  - symbolic model checking
    - represent state space symbolically using binary decision diagrams (BDDS)
    - symbolic fixed point calculation
  - explicit (state) model checking
    - state space explicitly enumerated
    - state space search

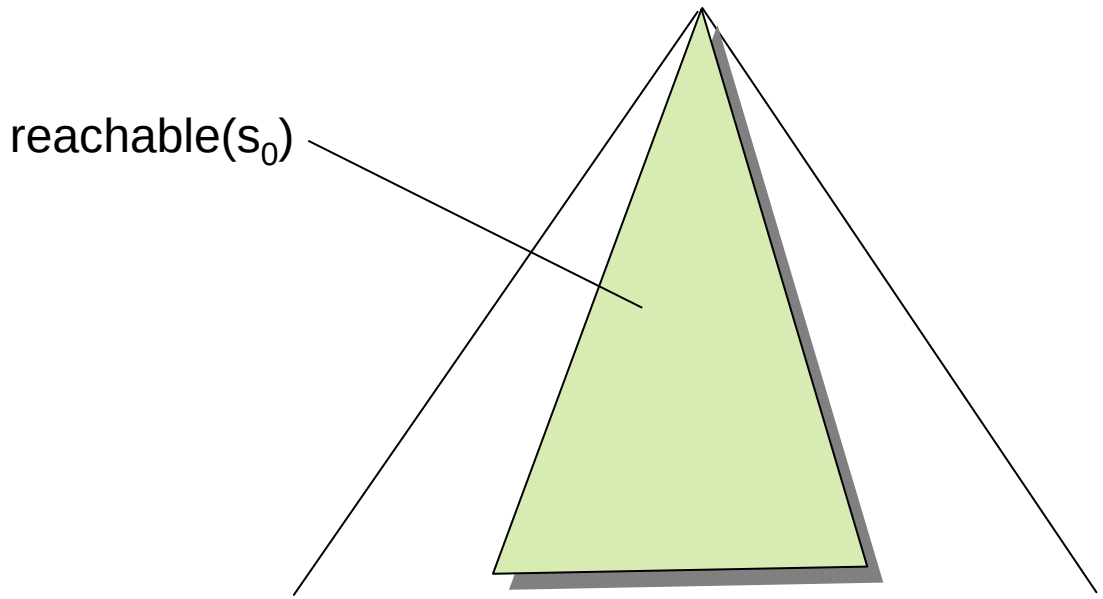
Model Checking

- **Explicit-State Model Checking**
  - generate a state space
  - states in the state space are related by a successor relation
  - the number of distinct state can be bounded or unbounded
    - most model checking procedures assume finiteness



Model Checking

- **Next State Relation**
  - `nextstate(s)`
    - computes all possible successor states s' to a state s
    - implements the semantics of the underlying modeling language
    - note that usually only a portion of the theoretical state space is reachable from initial system state s_0
      - $(s_1, s_2) \in succ$ if $s_2 \in nextstate(s_1)$
      - $reachable(s) := \{s' \mid (s, s') \in succ^*\}$



      - still, sizes of reachable state spaces can easily be in the order of 2^{25} or more, if finite

Model Checking

- **Explicit-State Model Checking**
  - is state space search
  - most efficient: depth first search

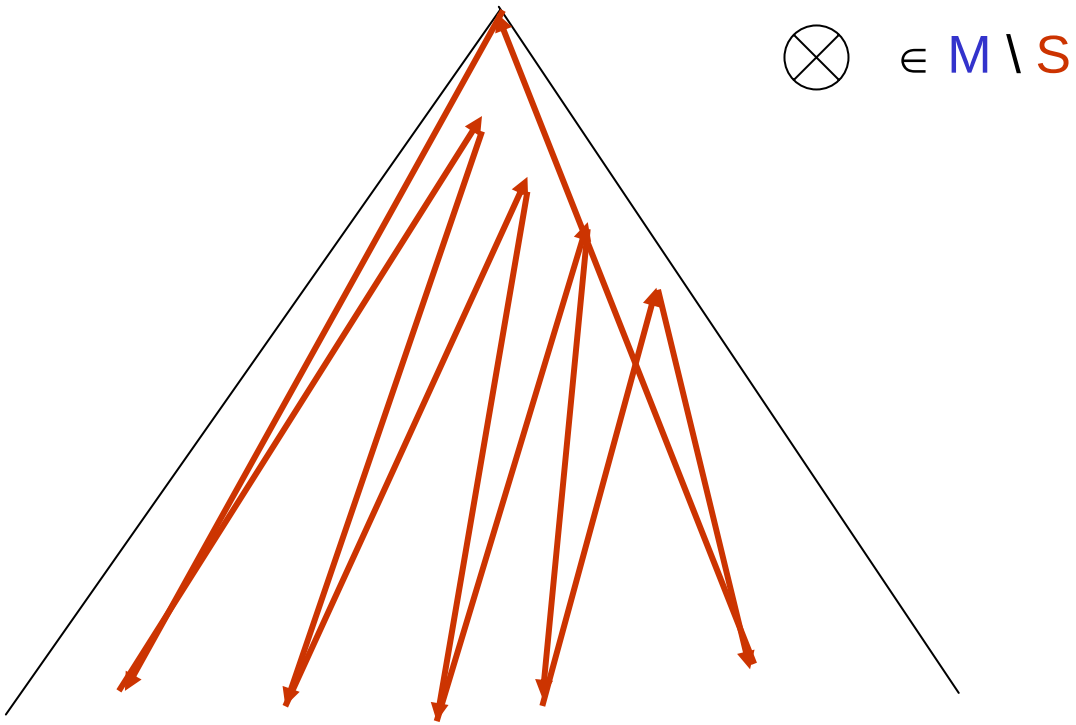
```
main
        s0 := initial system state;
        DFS (s0);

DFS(s)
        hash (s);
        if property violated
                then break;
        for all s' ∈ nextstate(s) do
                if s' not in hashtable
                        then DFS(s');
```

  - usually implemented "on-the-fly"
    - state space will not be entirely generated, stored and then analyzed, but instead
    - successor states only generated when needed for `nextstate()`

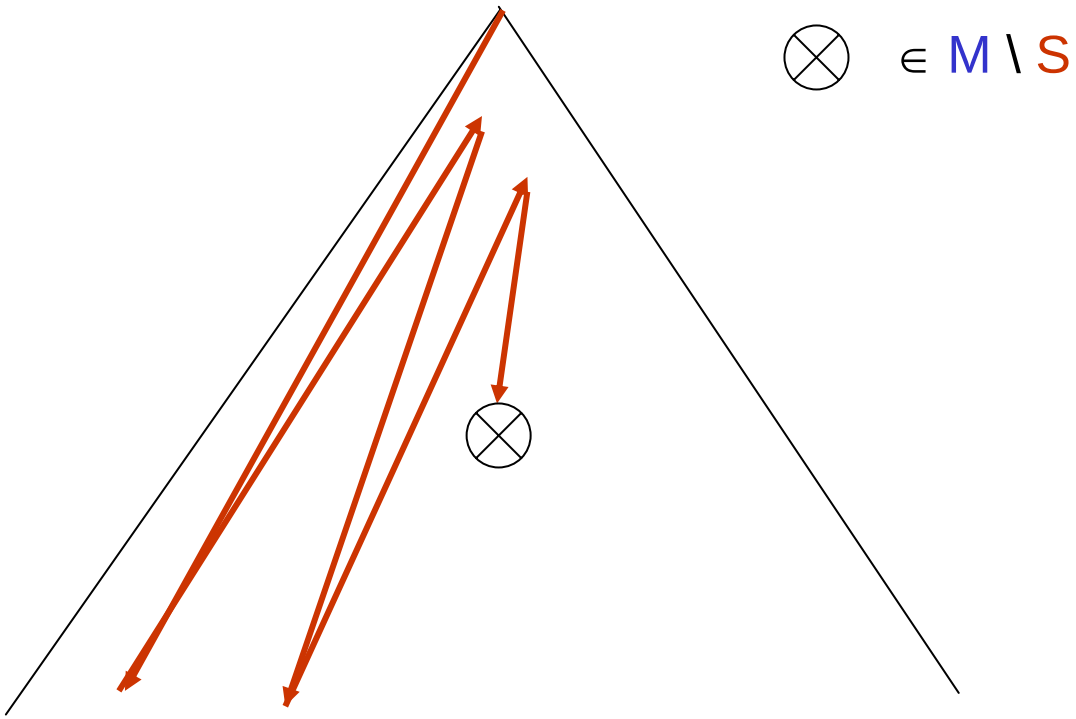
Model Checking

- **Explicit-State Model Checking**
  - classical use
    - show correctness of M wrt. S by completely traversing the state space and thereby establishing that there are no error states



Model Checking

- **Explicit-State Model Checking**
  - more recent use
    - software debugging
      - find property violating states, if present (will be present...)
      - interpret witness as error path explaining the fault

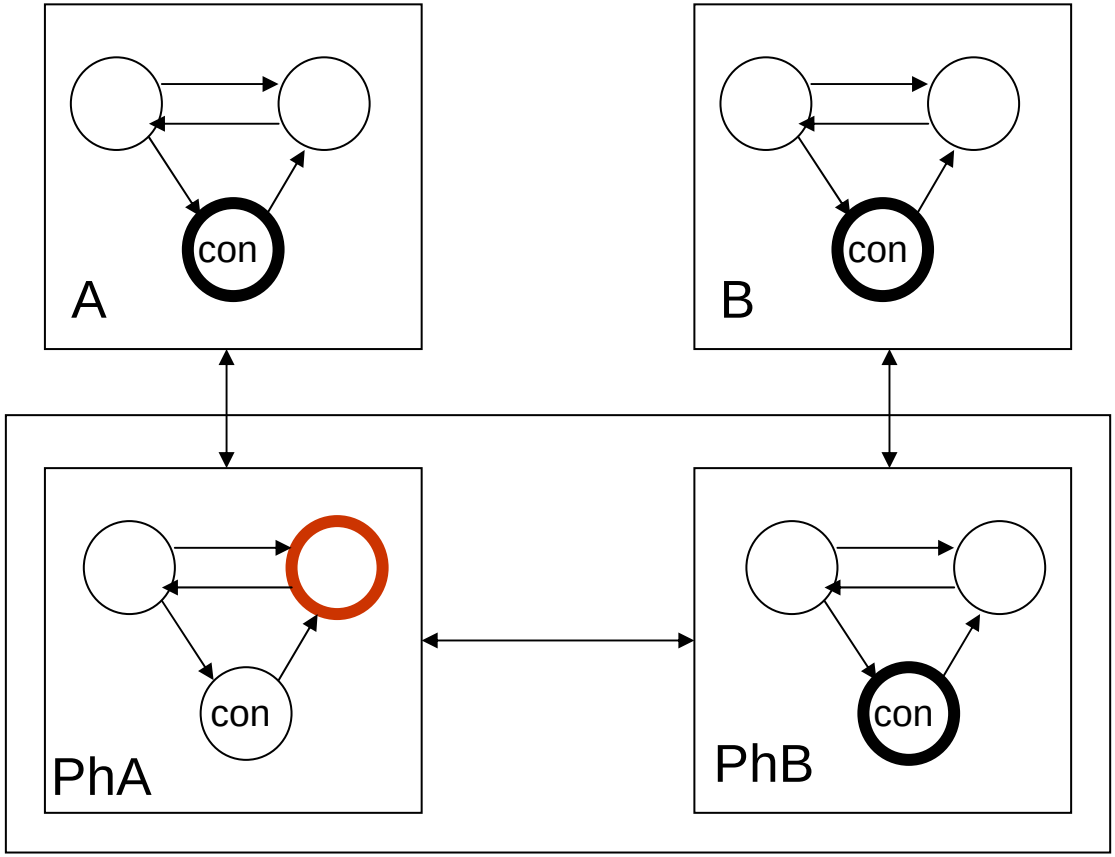


Model Checking of Safety Properties

- **Safety properties**
  - invariants, absence of deadlock, reachability of states,...
  - DFS in the system's global state space
    - if no property violating state found
      - termination after complete state space exploration
    - if property violating state found
      - output counterexample
        - search stack contains path from initial system state into property violating state
- **Use of Model Checker for Debugging Purposes**
  - let p an undesired state property
  - claim $\neg p$
    - model checker will try to disprove this claim and find a trail into a state satisfying the undesired property
  - error explanation

Model Checking of Safety Properties

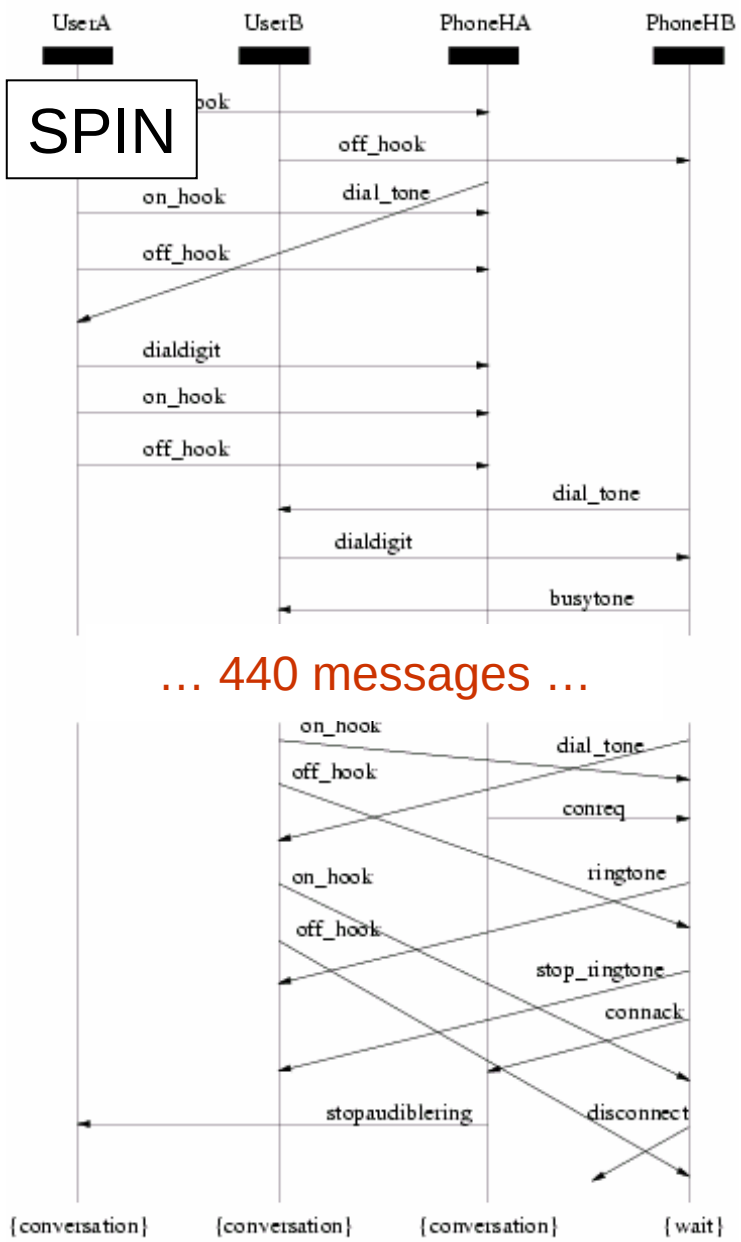
- **Example: Debugging of POTS**

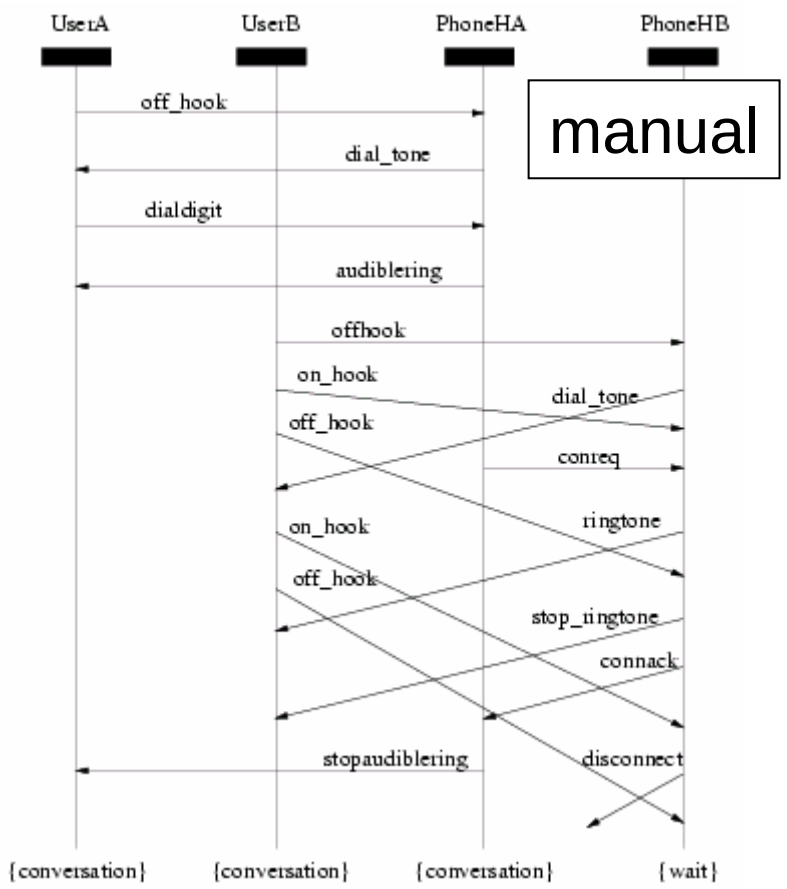


Invariant Property

$\neg$(A@con $\wedge$ $\neg$PhA@con $\wedge$ B@con $\wedge$ PhB@con)

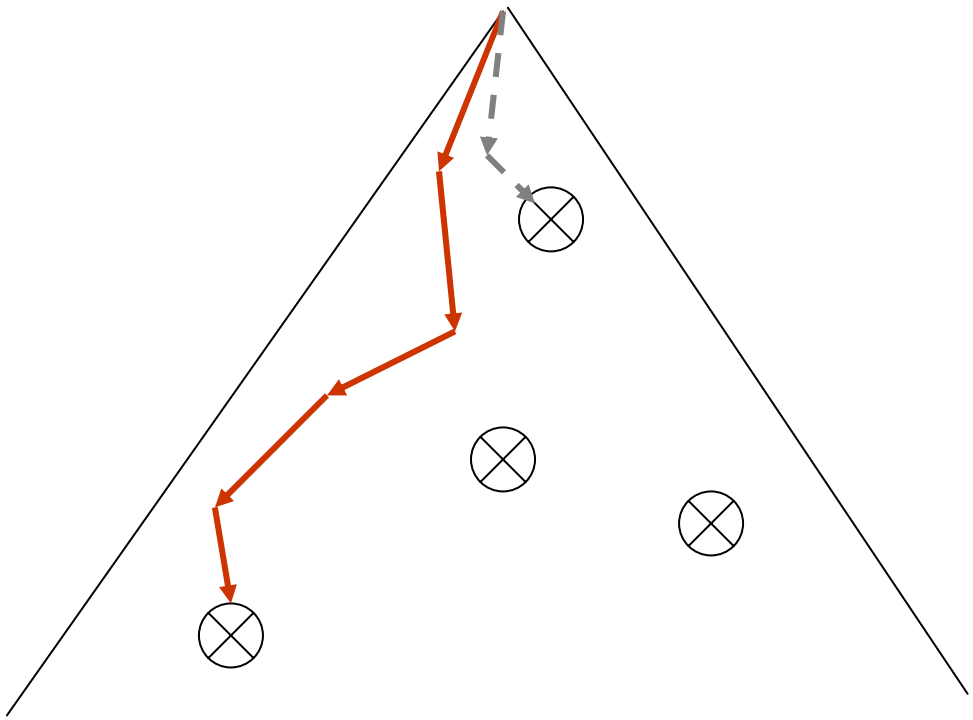
Counterexamples





total of 16 messages

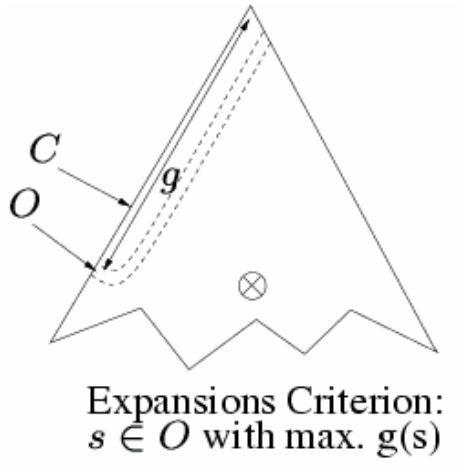
Counterexample with Depth-First Search (DFS)



- **Shorter Counterexamples**
  - shorter path to **the same** property-violating state
  - shorter path to **some** property-violating state
- **State Space Exploration Strategies?**

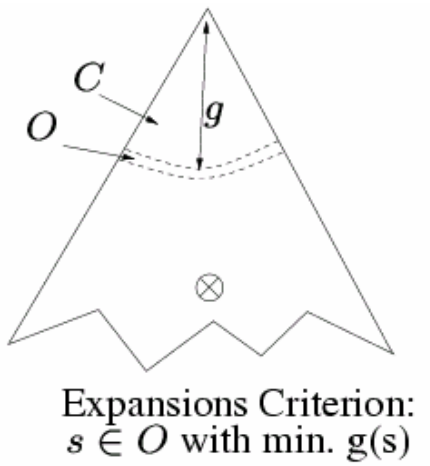
Uninformed Search

Depth-First Search (DFS)



- memory efficient
- potentially long error trails
- stack-based

Breadth-first Search (BFS)

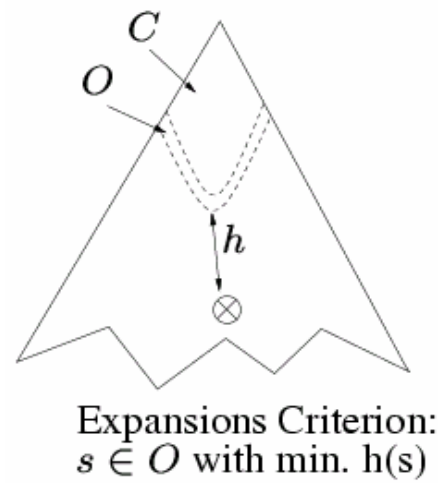


- memory inefficient
- shortest error trails
- no stack

Graphiken von A. Lluch-Lafuente

Informed Search

Best-First Search (BF)

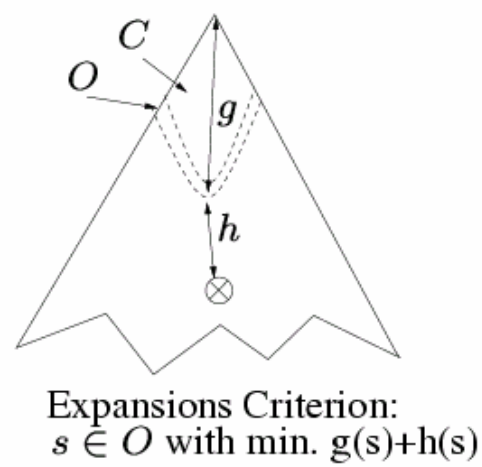


- informed search
  - heuristic estimate h(S): estimate of length of path to goal state S
- no guarantee for optimally short trails (sub-optimal solution)
- complete algorithm
  - goal state, if present, will be found
- often improved error trails

Graphiken von A. Lluch-Lafuente

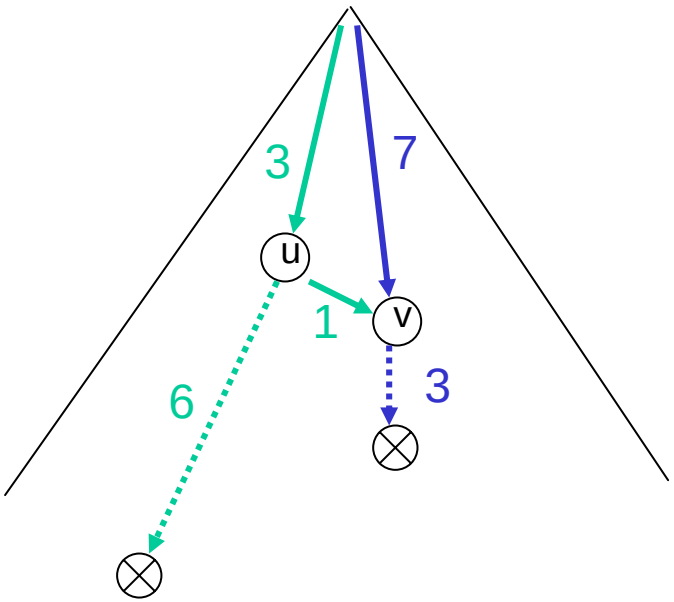
Informed Search

A*



- informed search
- optimally short error trails if h is a lower bound (admissible heuristics)
- complete
- no stack

- re-opening of nodes
  - move states from *Closed* to *Open* if they can be reached on a shorter path
    - re-open v



Graphiken von A. Lluch-Lafuente

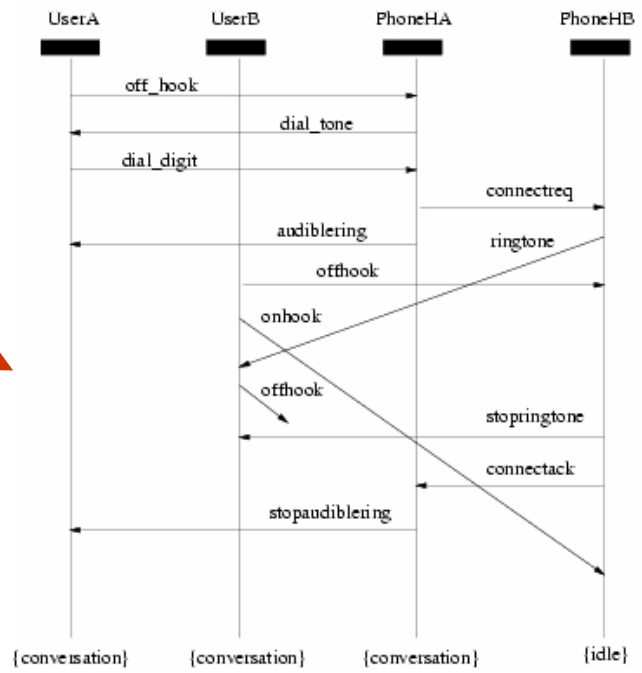
HSF-SPIN Experiments

- **Invariant Violation (H_f)**

Elevator	BFS	DFS	A*	BF	SPIN
s	38,662	279	38,598	2,753	292
e	38,564	279	38,506	2,297	292
t	160,364	356	160,208	5,960	348
l	203	510	203	421	510

states
exp.states
transitions
err. **l**ength

POTS	BFS	DFS	A*	BF	SPIN
s	24,546	-	6,654	781	148,049
e	17,632	-	3,657	209	148,049
t	99,125	-	18,742	1,067	425,597
l	81	-	81	83	2,765



12 messages

Software Visualization

- **Goal**
  - enhance the specification, documentation and understanding of complex software systems
- **Problem**
  - multi-dimensional visualization
    - variables, data-objects
    - control flow
    - temporal evolution
      - point of control
      - dynamicism (object creation/deletion, roaming in mobile systems,...)
- **State of the Art**
  - visualization of static and (limited) dynamic aspects of software systems
    - algorithm animation
  - software landscapes
    - static aspects of software architecture
      - reverse engineering
- **Challenges**
  - dynamic behaviour of complex, object-oriented systems
  - link to software verification / model checking

Objectives

- **Distance Visualization**
  - counterexample is a deviation of the actual behavior from the specified behavior
  - where did things go wrong?
    - visualize distance from actual to specified behavior
- **Interactive Visual Debugging based on Counterexamples**
  - stepwise simulation of system
    - possibly guided by counterexample
  - how to choose next step to take
    - support by heuristics, presented visually
- **Visualization of Dynamic Systems**
  - creation and termination of object/class instances
  - dynamic communication relationships
  - dynamic system configurations (mobility, ambients)
    - e.g., roaming

Objectives

- **Visualization and Abstraction**
  - complex software systems cannot be structurally understood by single person
  - reduce complexity by judicious use of abstraction
    - static analysis
      - slicing
    - data abstraction
      - predicate abstraction
      - boolean abstraction
      - cartesian abstraction
    - structural abstraction
      - symmetry reductions
      - partial order reductions
  - visualize the abstracted models
    - are the abstractions known from program analysis appropriate for visualization?

Context

- **Related Activities in Graduate School**
  - software landscapes
    - Prof. Deussen
  - analysis and visualization of large text bodies
    - Prof. Brandes
  - visual data mining
    - Prof. Keim
  - visual metadata browser
    - Prof. Reiterer
- **Projects within the Chair for Software Engineering**
  - IMCOS
    - incomplete analysis methods for concurrent, object-oriented systems
  - AVACS
    - heuristic model checking techniques for real-time and probabilistic systems