Faculty of Computer Systems & Software Engineering

Formal methods.

Verification by model checking

Vitaliy Mezhuyev

Introduction

- Verification: checking correctness of computer systems
  - hardware, software or its combination
- It is most needed for
  - safety-critical systems
  - mission critical
  - commercially critical

Verification techniques

- A formal verification techniques includes

1. A specification language
   - for describing properties to be verified
2. A verification method
   - To check if the description of the system satisfies its specification
3. A framework for modeling
   - To support a user in a system specification and verification

Classification

- Approaches to verification can be classified in several ways:
  - Proof-based vs. model-based
  - Degree of automation
    - From fully automated to fully manual
  - Full- vs. property- verification
    - The specification may describe a single property of a system, or it may describe its full behavior
  - Domain of application
  - Stage of use at development cycle
    - Pre- vs. post- verification

Domain of application

Types of systems:

- Hardware, software
- Sequential, concurrent
- Reactive, terminating
  - Reactive: reacts to its environment, and not intended to terminate (e.g. operating system for embedded domain)

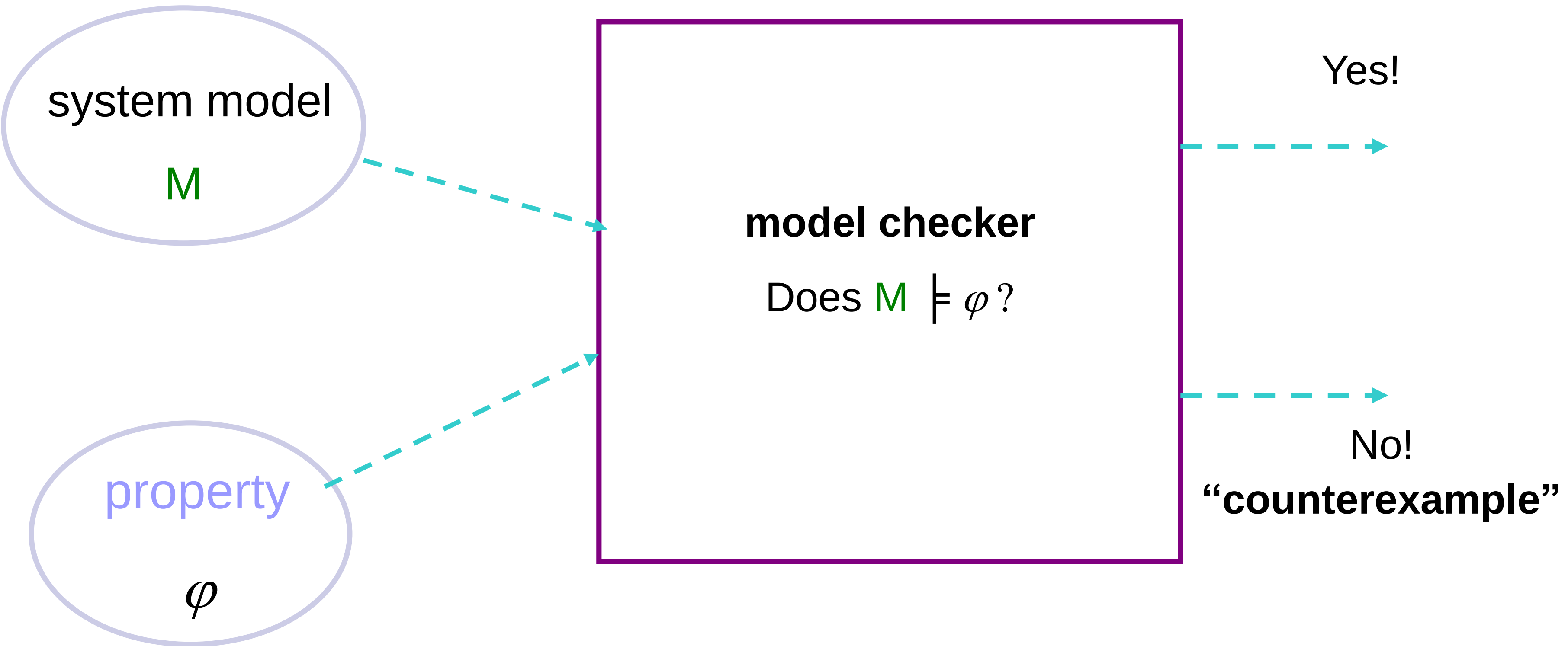
Proof-based verification

- A system description is a set of formulas Γ in some logic
- A specification is another formula φ
- The verification method is finding a proof that $\Gamma \vdash \varphi$
  - $\vdash$ means ***deduction***
- Application of a proof based verification needs high user expertise

Model-based verification

- The system is represented by a model $\mathcal{M}$ in some logic
- The specification is also represented by a formula φ
- The verification method is ***computing*** whether a model $\mathcal{M}$ satisfies φ
  - $\mathcal{M}$ satisfies φ : $\mathcal{M} \models \varphi$
- The computation is usually **automatic** for *finite* models

Basic schema for model checking



Model checking

- Model checking is an *automatic*, *model-based*, *property-verification* approach
- It is intended to be used first of all for *concurrent* and *reactive* systems
- Possible to check concurrency bugs
  - not covered by test cases
  - non reproducible by testing methods

How to build a model of a system?

hardware

e.g., Verilog or VHDL, source code

abstraction & other (semi-)automated transformations

software

e.g., C, C++ , Java, etc. source code

hand-built design models

System model

M

(typically, state machine-based)

Development of a system model: what do we want to model?

- systems have a **<u>state</u>** that evolves over time.
- they manipulate data, accessed through **<u>variables</u>**, whose values change as the state changes.
- **<u>concurrency</u>**: systems have interacting **<u>processes</u>**
  - <u>asynchronous / synchronous,</u>
  - message passing or shared data <u>communication</u>**.**
- dynamic memory allocation, process creation, procedure call stack, clocks and real time, etc.

Models have to be:

- show as many relevant aspects of real systems as possible
- be amenable to efficient algorithmic analysis.

Where do we get the properties?

requirements documentation

+

insights

Formal Properties

(typically based on temporal logic and state automata formalism)

standard properties - safety (type invariance), absence of deadlock, livelock etc.

Temporal Logic

- The idea is that a formula is not *statically* true or false in a model
- A model has many states
  - a formula can be true in some state and false in the others
- The static notion of truth is replaced by a *dynamic* one
  - the formulas may change their truth values as the system evolves from state to state

Why use temporal logic to specify properties?

- [Pnueli'77] and others recognized that correctness assertions for **reactive** systems are best phrased in terms of occurrence of events during the entire, potentially indefinite, execution of the system. Not just what it outputs when it halts.

- **Example**: "Until event stop occurs, every occurrence of event request is eventually followed by an occurrence of event response"

What is temporal logic?

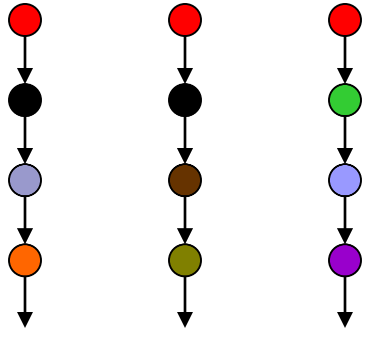
- It is a language for describing relationships between the occurrence of events over time.
- It comes is several dialects, e.g., Linear vs. Branching time.
- We focus on propositional **Linear Temporal Logic** (**LTL**).

Linear vs. Branching

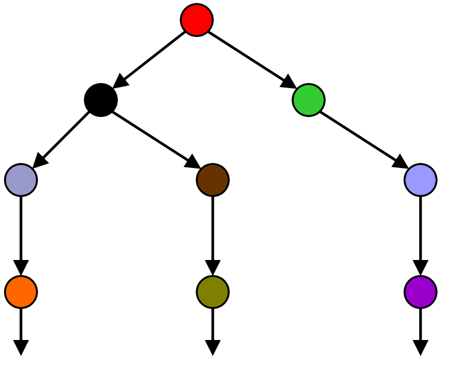
- Linear-time logics think of time as a **set of paths**
  - path is a sequence of states (time instances)
- Branching time logics represent time as a **tree**
  - it is rooted at the present moment and branches out into the future
- Many logics were suggested during last years that fit into one of above categories

Linear vs. Branching (cont.)

- Linear Time
  - Every moment has a unique successor
  - Infinite sequences of states
  - Linear Time Temporal Logic (LTL)



- Branching Time
  - Every moment has several successors
  - Infinite tree
  - Computation Tree Logic (CTL)



LTL: **L**inear-time **T**emporal **L**ogic

- It models time as a sequence of states, extending infinitely to future
  - computation path
- The future is not determined, we should consider *several paths* for different futures
  - Any one of the paths can be the *actual path* that is realized

Models in Temporal Logic

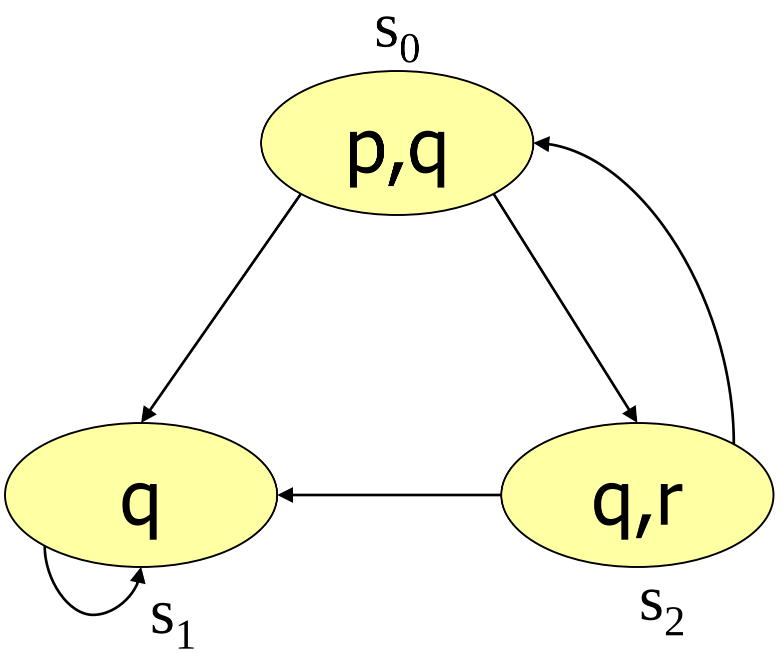
- In model checking:
  - The model $\mathcal{M}$ is a state transition system
    - e.g. HourClock
  - The properties φ are formulas in temporal logic
    - e.g. []HCini or TypeInvariant
- Model checking steps:
  1. Model $\mathcal{M}$ the system using description language (e.g. Temporal Logic of Actions)
  2. Write a property φ using the specification language (TLA also can be used here)
  3. Run the model checker with the inputs $\mathcal{M}$ and φ

Model in Temporal Logic - Transition System

- A transition system is a structure $\mathcal{M} = (S, \rightarrow, L)$ where
  - S: a finite set of states
  - $\rightarrow$: a binary relation on S, such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$
  - L : a labeling function $L: S \rightarrow \mathcal{P}(\mathit{Atoms})$
    - $\mathcal{P}(\mathit{Atoms})$ means the power set of Atoms

- The interpretation of the labeling function is that each state s has a set of atomic propositions $L(s)$ which are true at that particular state

Example



$S = \{s_0, s_1, s_2\}$

transitions = $s_0 \longrightarrow s_1$, $s_1 \longrightarrow s_1$, $s_2 \longrightarrow s_1$, $s_2 \longrightarrow s_0$, $s_0 \longrightarrow s_2$

$L(s_0) = \{p,q\}$

$L(s_1) = \{q\}$

$L(s_2) = \{q,r\}$

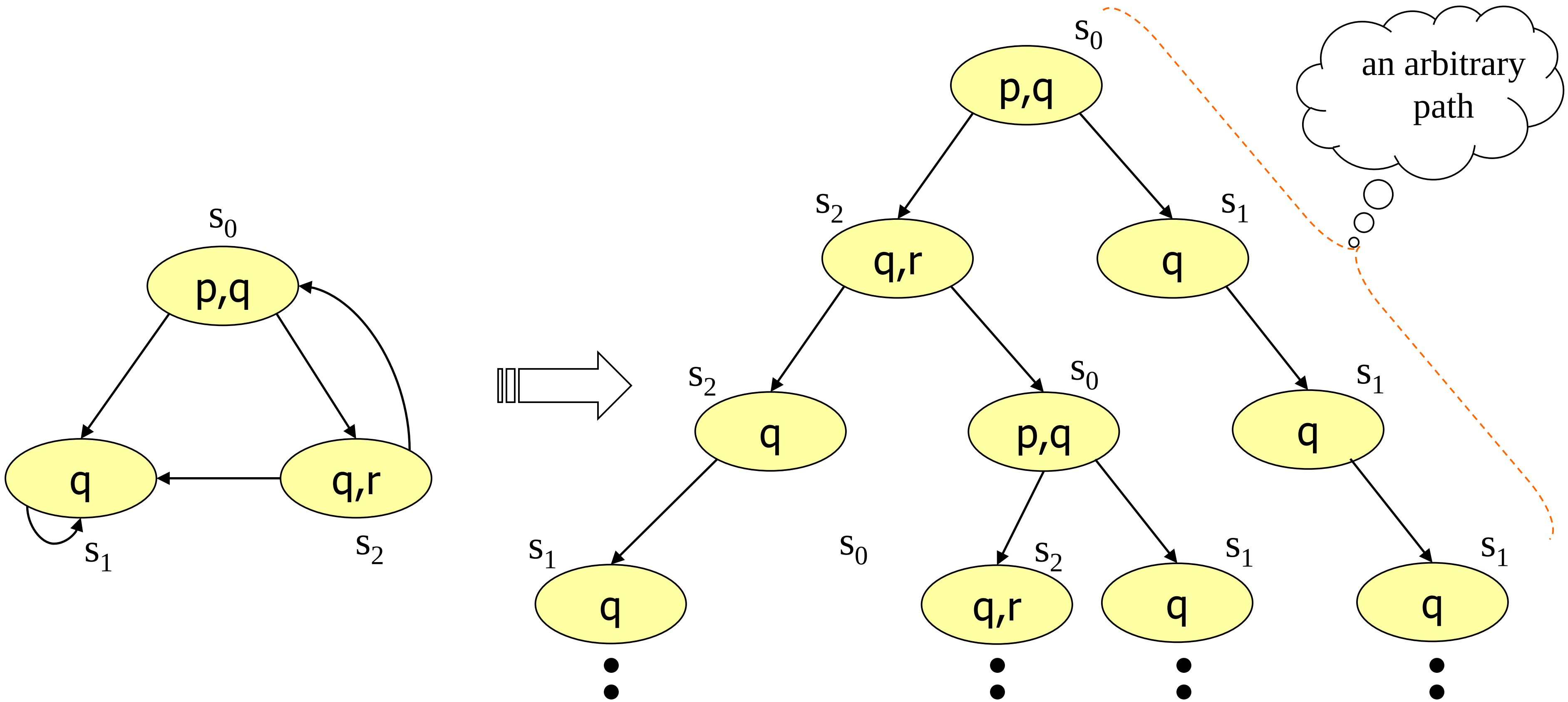
Path as a model of behavior

- A path in a model $M = (S, \to, L)$ is an infinite sequence of states $s_1, s_2, s_3,\ldots$ in S such that, for each $i \geq 1$, $s_i \to s_{i+1}$.
- We write paths as $s_1 \to s_2 \to s_3 \to \ldots$
- Each arbitrary path (e.g. $\pi = s_1 \to s_2 \to \ldots$) represents a possible *behavior* of a system
  - first it is in s_1, then s_2 and so on

Building a computation tree (unwinding)

- To check the model M we will unwind the transition system to obtain an infinite computation tree
- The execution paths of a model M are explicitly represented in the unwinded tree

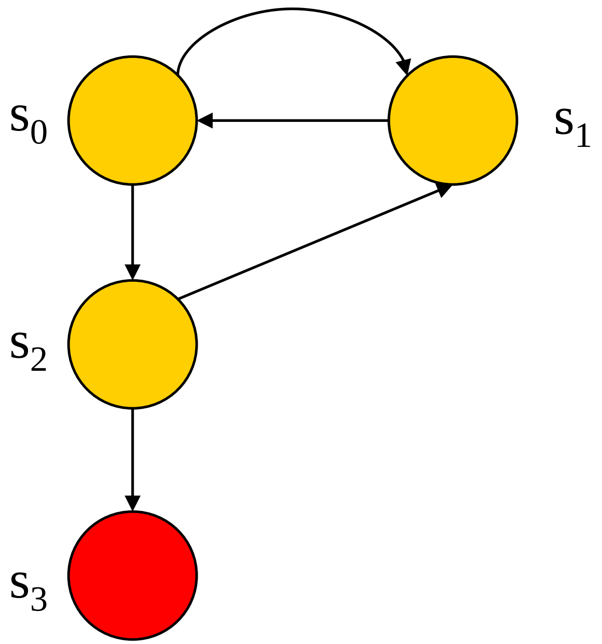
Unwinding: example

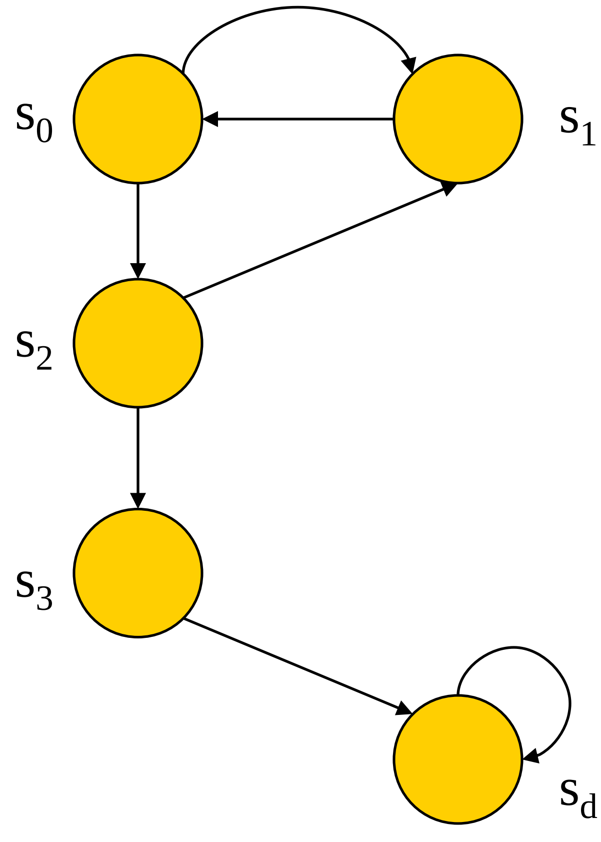


Model of deadlock

- We will call a transition system simply a *model*
- According to the definition of a model, for each $s \in S$ there is at least one $s' \in S$
- I.e. there is should not be a deadlock state of a system
- If a system *intentionally* has a deadlock, lets add an extra state s_d representing it

Deadlock state





s_3 doesn’t have any further transitions

adding a deadlock state s_d

Thank you for your attention!
Please ask questions