Probabilistic verification and synthesis

Marta Kwiatkowska

Department of Computer Science, University of Oxford

KTH, Stockholm, August 2015

What is probabilistic verification?

- Probabilistic verification (aka probabilistic/ quantitative model checking)…
  - is a formal verification technique for modelling and analysing systems that exhibit probabilistic behaviour
- Formal verification…
  - is the application of rigorous, mathematics-based techniques to establish the correctness of computerised systems
- Synthesis…
  - is an automatic method to generate system components that are correct-by-construction

Why must we verify?

"Testing can only show the presence of errors, not their absence."

To rule out errors need to consider all possible executions often not feasible mechanically!

- need formal verification…

"In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, computers are without precedent in the cultural history of mankind."

Edsger Dijkstra

1930–2002

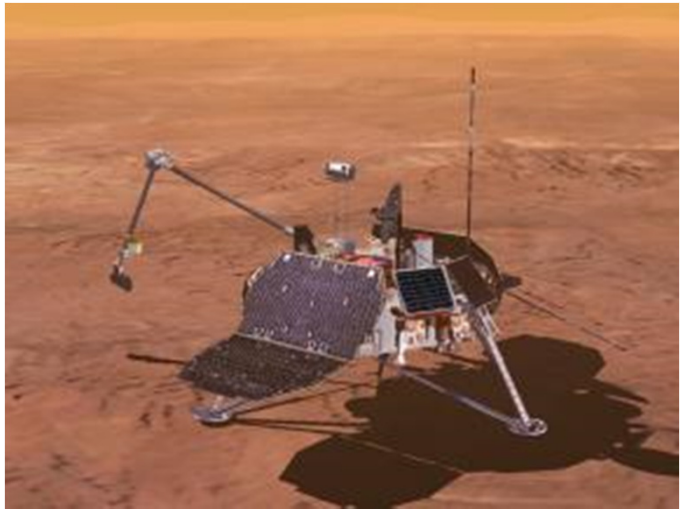
But my program works!

- True, there are many successful large-scale complex computer systems…
  - online banking, electronic commerce
  - information services, online libraries, business processes
  - supply chain management
  - mobile phone networks
- Yet many new potential application domains with far greater complexity and higher expectations
  - autonomous driving, self-parking cars
  - medical sensors: heart rate & blood pressure monitors
  - intelligent buildings and spaces, environmental sensors
- Learning from mistakes costly…

The NASA Mars space mission

Mars Climate Orbiter
Launched 11th December 1998

LOST 23rd September 1999
Conversion error from English units to metric in navigation software
Cost: $125 million



Mars Polar Lander
Launched 3rd January 1999

LOST 3rd December 1999
Engine shutdown due to spurious signals that gave false indication that spacecraft had landed

Source: http://mars.jpl.nasa.gov/msp98/

Infusion pumps

F.D.A. Steps Up Oversight of Infusion Pumps

The New York Times Published: April 23, 2010

Pump producers now typically conduct 'simulated' testing of its devices by users.

Over the last five years, […] 710 patient deaths linked to problems with the devices.

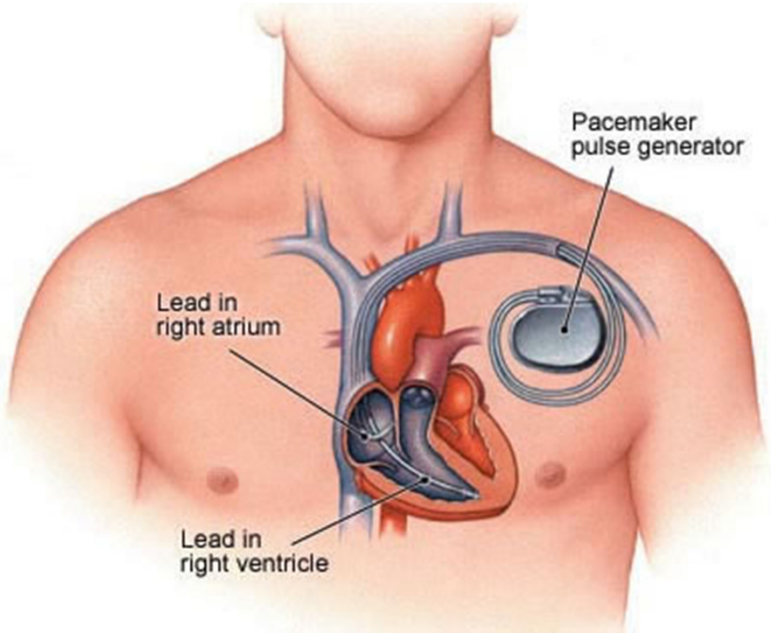
Some of those deaths involved patients who suffered drug overdoses accidentally, either because of incorrect dosage entered or because the device's software malfunctioned.

Manufacturers […] issued 79 recalls, among the highest for any medical device.

Source: http://www.nytimes.com/2010/04/24/business/24pump.html?_r=0

Cardiac pacemakers

- The Food and Drug Administration (FDA)
  - issued 23 recalls of defective pacemaker devices during the first half of 2010
  - classified as "Class I," meaning there is "reasonable probability that use of these products will cause serious adverse health consequences or death"
  - six of those due to software defects



- "Killed by code" report
  - many similar medical devices
  - wireless, implantable, e.g. glucose monitors

Source: https://www.softwarefreedom.org/resources/2010/transparent-medical-devices.html

Toyota

- February 2010
  - unintended acceleration
  - resulted in deaths

- Engine Control Module
  - source code found defective
  - no mirroring: stack overflow , recursion was used

- "Killed by firmware"
  - millions of cars recalled, at huge cost
  - handling of the incident prompted much criticism, bad publicity
  - fined $1.2 billion for concealing safety defects

Source: http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences

What do these stories have in common?

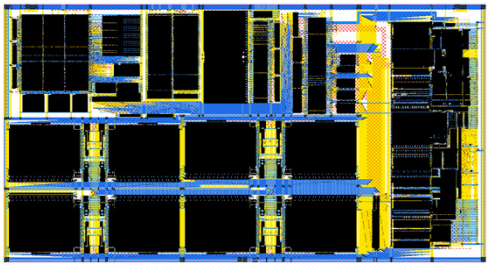
- Programmable computing devices
  - conventional computers and networks
  - software embedded in devices
    - airbag controllers, mobile phones, medical devices, etc
- Programming error direct cause of failure

- Software critical
  - for safety
  - for business
  - for performance
- High costs incurred: not just financial

- Failures avoidable…

Automatic verification

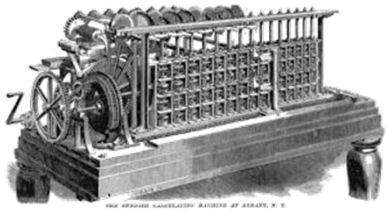
- Formal verification…
  - the application of rigorous, mathematics-based techniques to establish the correctness of computerised systems
  - essentially: proving that a program satisfies it specification
  - many techniques: manual proof, automated theorem proving, static analysis, model checking, …



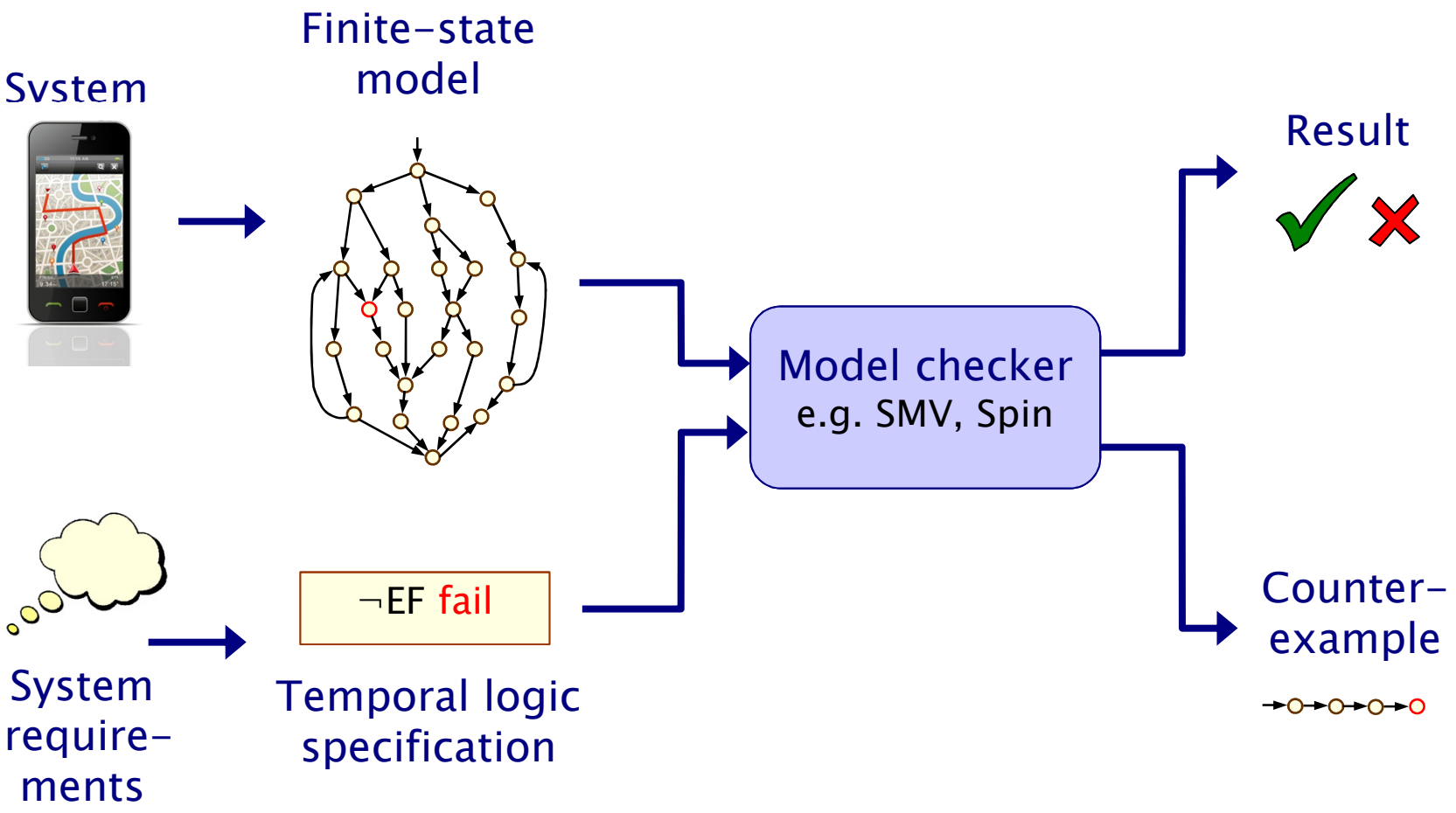
$10^{500,000}$ states

10^{70} atoms

- Automatic verification =
  - mechanical, push-button technology
  - performed without human intervention



Verification via model checking



Verification… or falsification?

- More value in showing property violation?
  - model checkers used as debugging tool!
  - can we synthesise directly from specification?
- Widely accepted in industrial practice
  - Intel, Cadence, Bell Labs, IBM, Microsoft, ...
- Many software tools, including commercial
  - CProver/CBMC, NuSMV, FDR2, UPPAAL, ...
  - hardware design, protocols, software, …

Much progress since 1981! But...

New challenges for verification

- Devices, ever smaller
  - laptops, phones, sensors…
- Networking, wireless, wired & global
  - wireless & internet everywhere
- New design and engineering challenges
  - adaptive computing, ubiquitous/pervasive computing, context-aware systems
  - DNA computing and biosensing
  - trade-offs between e.g. performance, security, power usage, battery life, …

New challenges for verification

- Many properties other than correctness are important
- Need to guarantee…
  - safety, reliability, performance, dependability
  - resource usage, e.g. battery life
  - security, privacy, trust, anonymity, fairness
  - and much more…
- Quantitative, as well as qualitative requirements:
  - "how reliable is my car's Bluetooth network?"
  - "how efficient is my phone's power management policy?"
  - "how secure is my bank's web-service?"
- This course: probabilistic verification and synthesis

Why probability?

- Some systems are inherently probabilistic…
- Randomisation, e.g. in distributed coordination algorithms
  - as a symmetry breaker, in gossip routing to reduce flooding
- Examples: real-world protocols featuring randomisation:
  - Randomised back-off schemes
    - CSMA protocol, 802.11 Wireless LAN
  - Random choice of waiting time
    - IEEE1394 Firewire (root contention), Bluetooth (device discovery)
  - Random choice over a set of possible addresses
    - IPv4 Zeroconf dynamic configuration (link-local addressing)
  - Randomised algorithms for anonymity, contract signing, …

Why probability?

- Some systems are inherently probabilistic…
- Randomisation, e.g. in distributed coordination algorithms
  - as a symmetry breaker, in gossip routing to reduce flooding
- To model uncertainty and performance
  - to quantify rate of failures, express Quality of Service
- Examples:
  - computer networks, embedded systems
  - power management policies
  - nano-scale circuitry: reliability through defect-tolerance

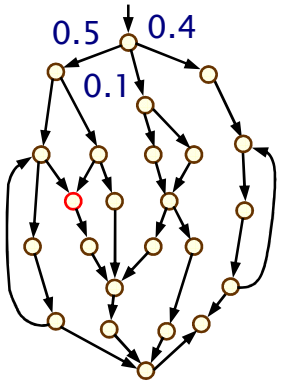
Why probability?

- Some systems are inherently probabilistic…
- Randomisation, e.g. in distributed coordination algorithms
  - as a symmetry breaker, in gossip routing to reduce flooding
- To model uncertainty and performance
  - to quantify rate of failures, express Quality of Service
- To model biological processes
  - reactions occurring between large numbers of molecules are naturally modelled in a stochastic fashion

Probabilistic model checking

System

Probabilistic model
e.g. Markov chain



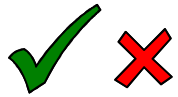
System require-ments

$P_{<0.1}$ [F fail]

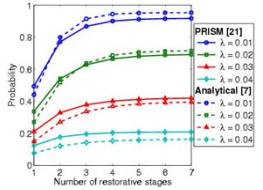
Probabilistic temporal logic specification
e.g. PCTL, CSL, LTL

Probabilistic model checker
e.g. PRISM

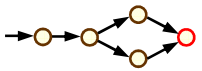
Result



Quantitative results



Counter-example



Probabilistic models

	Fully probabilistic	Nondeterministic
Discrete time	Discrete-time Markov chains (DTMCs)	Markov decision processes (MDPs)
		Simple stochastic games (SMGs)
Continuous time	Continuous-time Markov chains (CTMCs)	Probabilistic timed automata (PTAs)
		Interactive Markov chains (IMCs)

Probabilistic models

	Fully probabilistic	Nondeterministic
Discrete time	Discrete-time Markov chains (DTMCs)	Markov decision processes (MDPs)
		Simple stochastic games (SMGs)
Continuous time	Continuous-time Markov chains (CTMCs)	Probabilistic timed automata (PTAs)
		Interactive Markov chains (IMCs)

NB One can also consider continuous space…

Lecture plan

- Course slides and lab session
  - http://www.prismmodelchecker.org/courses/kth15/
- 5 sessions: lectures 9–12noon, labs 2.30–5pm
  - 1 – Introduction
  - 2 – Discrete time Markov chains (DTMCs)
  - 3 – Markov decision processes (MDPs)
  - 4 – LTL model checking for DTMCs/MDPs
  - 5 – Probabilistic timed automata (PTAs)
- For extended versions of this material
  - and an accompanying list of references
  - see: http://www.prismmodelchecker.org/lectures/

Course material

- Reading
  - [DTMCs/CTMCs] Kwiatkowska, Norman and Parker. Stochastic Model Checking. LNCS vol 4486, p220-270, Springer 2007.
  - [MDPs/LTL] Forejt, Kwiatkowska, Norman and Parker. Automated Verification Techniques for Probabilistic Systems. LNCS vol 6659, p53-113, Springer 2011.
  - [SMGs] Chen, Forejt, Kwiatkowska, Parker and Simaitis. Automatic Verification of Competitive Stochastic Systems. FMSD 43(1), 61-92, 2013.
  - [PTAs] Norman, Parker and Sproston. Model Checking for Probabilistic Timed Automata. FMSD 43(2), 164-190, 2013.
  - [DTMCs/MDPs/LTL] Principles of Model Checking by Baier and Katoen, MIT Press 2008
- See also PRISM website
  - www.prismmodelchecker.org

Part 1

Introduction to model checking

Overview (Part 1)

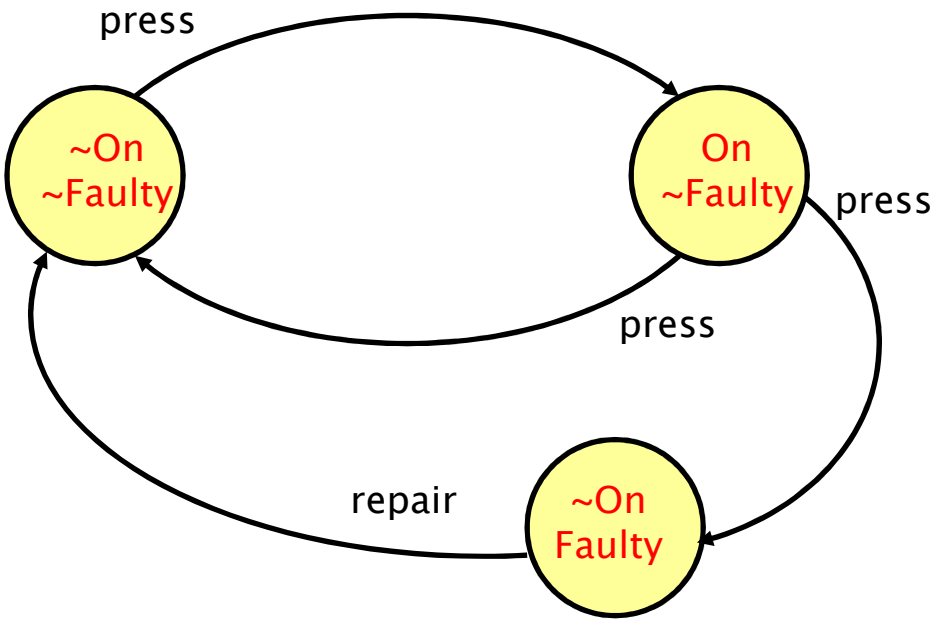
- Introduction
- Transition systems
- Temporal logic
- Model checking
  - Reachability
  - CTL model checking
- PRISM: overview
  - Probability example
  - Modelling language
  - Properties
  - GUI, etc
- Summary

Modelling reactive systems

- Reactive systems
  - keep interacting with their environment without terminating
  - e.g. protocols, operating systems, monitoring devices
  - termination is not relevant
- Graphical notations based on automata
  - based on finite-state automata (DFA, NFA) and formal languages, e.g. regular languages
  - usually no accepting state
- System is modelled as
  - states, i.e. snapshots of the system's variables at some point in time
  - transitions, which cause state changes in response to stimuli
  - computation proceeds through invoking state changes from some initial state, possibly ad infinitum

Modelling with automata

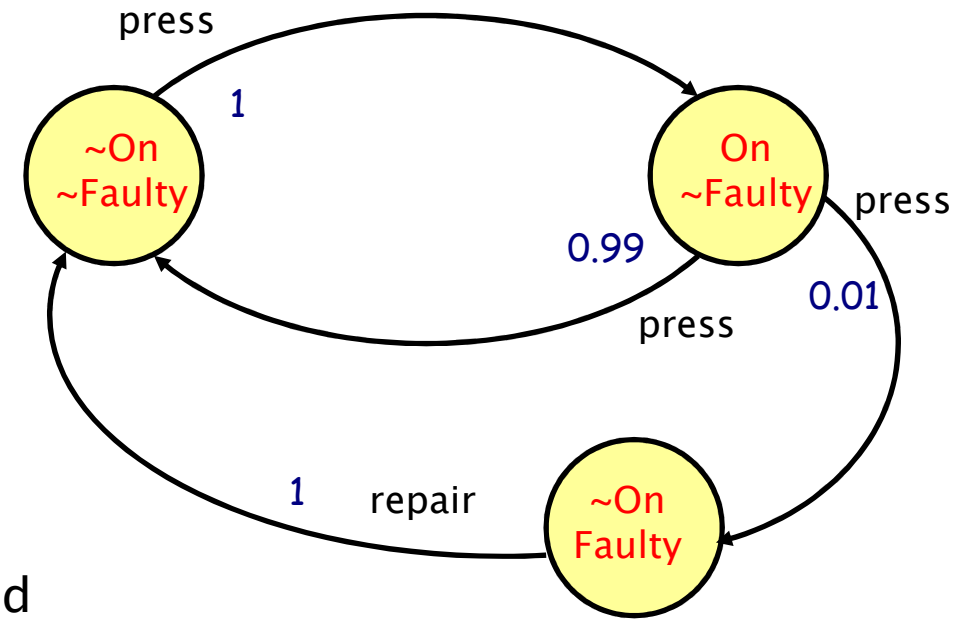
- Simple light switch



- Automaton
  - usually finite state
- States
  - atomic propositions
  - values of variables
- Transitions
  - actions/commands
  - e.g. on/off button
- Properties
  - If light is Off, then sometime in future it is On

Modelling with probabilistic automata

- As automata except
  - add probability
- States
  - values of atomic propositions
  - can have clocks
- Transitions
  - actions, possibly guarded
  - probabilistic choice of target state
- Properties
  - If light is Faulty then with probability 1 it becomes ~Faulty



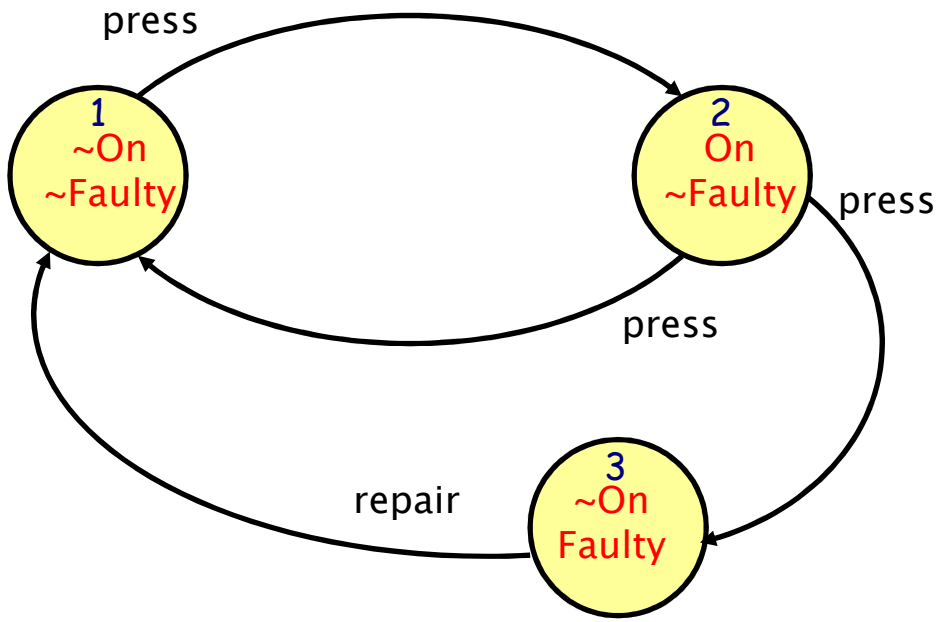
Transition systems

- A labelled transition system (LTS) is a tuple $M = (S, s_{init}, \alpha, T, L)$ where
  - S is a set of states ("state space")
  - $s_{init} \in S$ is the initial state
  - α is an alphabet of action labels
  - $T \subseteq S \times \alpha \times S$ is the transition relation
  - $L : S \rightarrow 2^{AP}$ is a labelling with atomic propositions

- Note
  - the state space is not necessarily finite
  - we sometimes omit state names or proposition labels
  - T is nondeterministic
  - since we model reactive systems, no accepting state

Example

- $S = \{1,2,3\}$
- $s_{init} = 1$
- $\alpha = \{press,repair\}$
- $T = \{(1,press,2),$
  $(2,press,1),$
  $(2,press,3),$
  $(3,repair,1)\}$
- $AP = \{On, Faulty\}$
- $L = \{1 \mapsto \{\},$
  $2 \mapsto \{On\},$
  $3 \mapsto \{Faulty\}\}$



Paths & execution runs

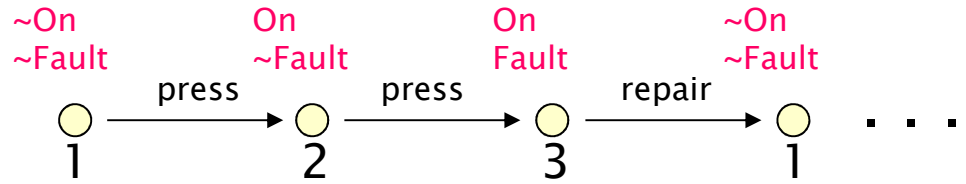
- A path ω of transition system $M = (S, s_{init}, \alpha, T, L)$ is a finite or infinite sequence of transitions $(s_i, a_i, s'_i) \in T$ such that, for all $i \geq 0$, $s'_i = s_{i+1}$

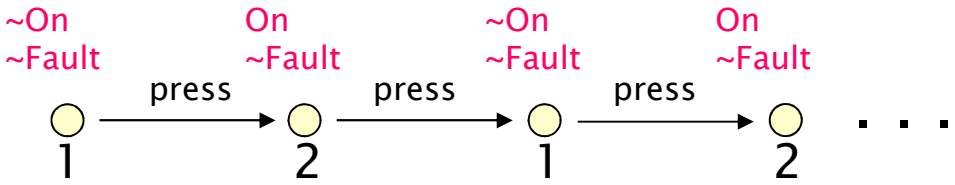
- Paths are denoted $s_0 a_0 s_1 a_1 \ldots$
- Or, if transition labels are omitted, $s_0 s_1 s_2 \ldots$

- A path ω is a partial execution run if it starts in the initial state s_{init}
- An execution run is complete if it is maximal, i.e. cannot be extended (ends in a deadlocking state)

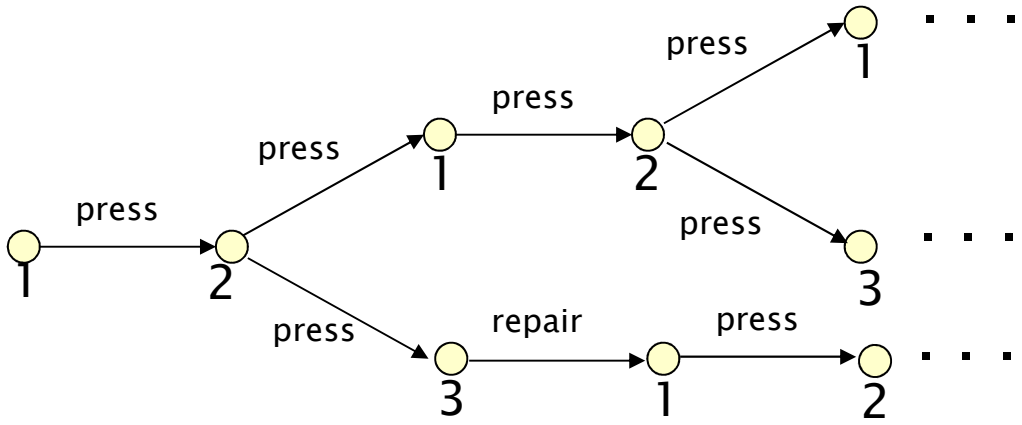
- Transition systems can be unfolded into execution trees

Executions and unfoldings





Execution runs

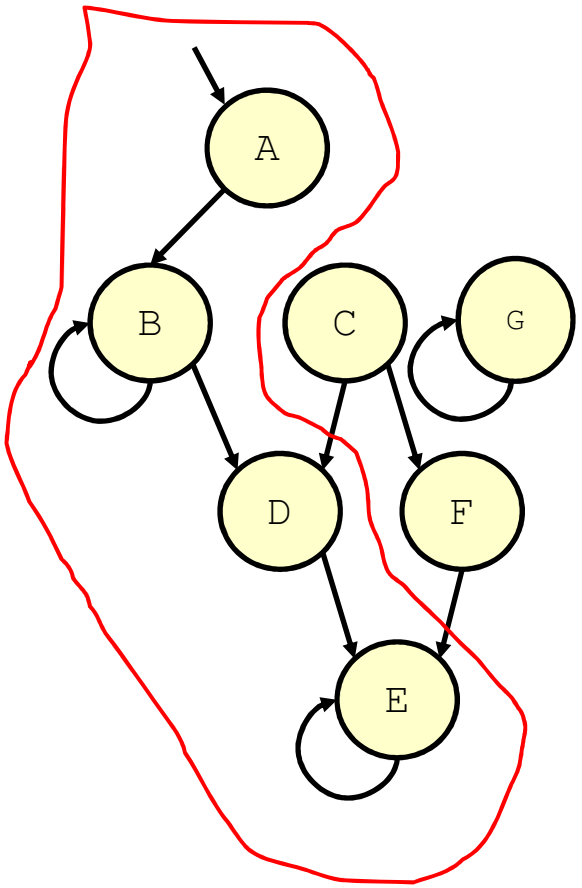


Execution tree

(some detail omitted)

Reachability graph

- Let $M = (S, s_{init}, \alpha, T, L)$ be a finite transition system
- A state s is reachable if there exists an execution leading to s
- The reachable states are
  - $Reach(M) = \{s \in S \mid s \text{ is reachable}\}$
- The reachability graph is the subgraph of M obtained by restricting to Reach(M)



Reachability & invariance properties

- Given transition system $M = (S, s_{init}, \alpha, T, L)$ and set of states $Y \subseteq S$, define the following:

- Reachability: is it possible to reach a state in Y?
  - Is $Reach(M) \cap Y \neq \{\}$?
  - A witness is an execution leading to a state in Y

- Invariance: is every reachable state of S also in Y?
  - Is $Reach(M) \subseteq Y$?
  - Y is an invariant for M

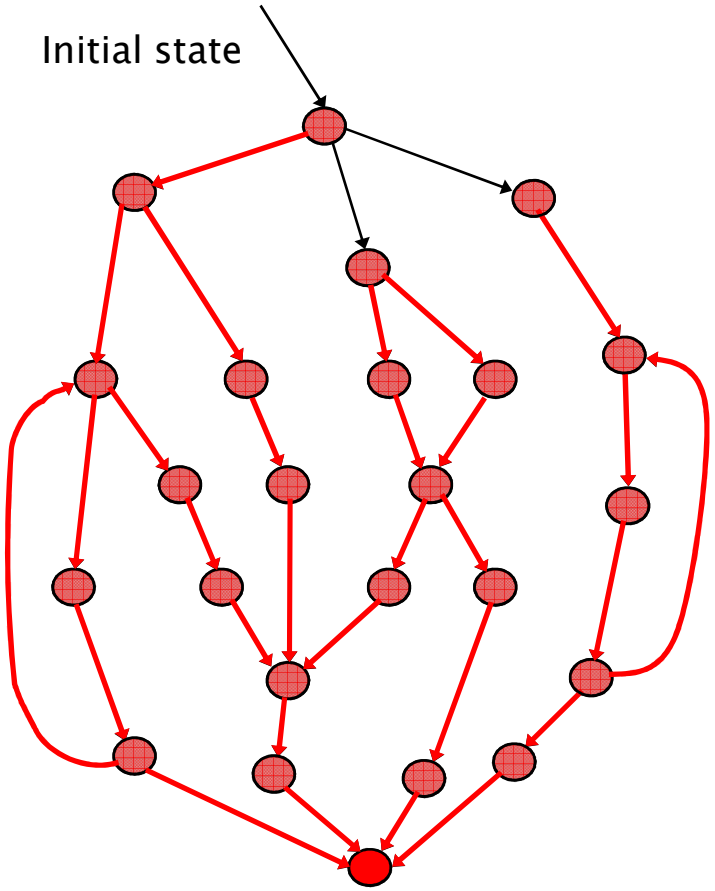
- Safety: is a state in Y never reachable?
  - Is $Reach(M) \setminus Y = \{\}$?
  - NB the dual of reachability – a witness for reachability of ~Y is an error trace for Y

Reachability analysis

- Represent the graph in an appropriate data structure – adjacency list
  - e.g. a list of states, each associated with a list of successors
  - memory optimisation possible, not discussed
- Employ graph traversal – traverse the graph one edge at a time
  - iterate computation of successors (forward)
  - iterate computation of predecessors (backward)
- Known as enumerative or explicit state
  - since states explored one at a time
  - symbolic analysis processes sets of states at a time

Backward safety checking

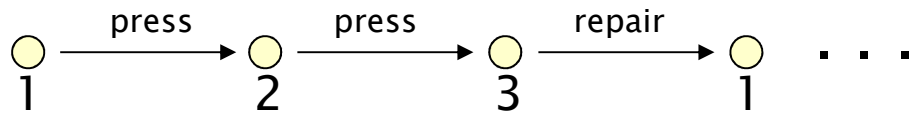
- Start from the set ~Y of unsafe states
- Is it possible to reach an unsafe state from the initial state?
- Model finite-state, hence termination assured, but can visit unreachable states
- Can also generate witness
  - how?



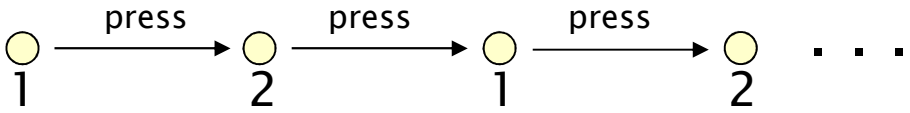
Temporal logic basics

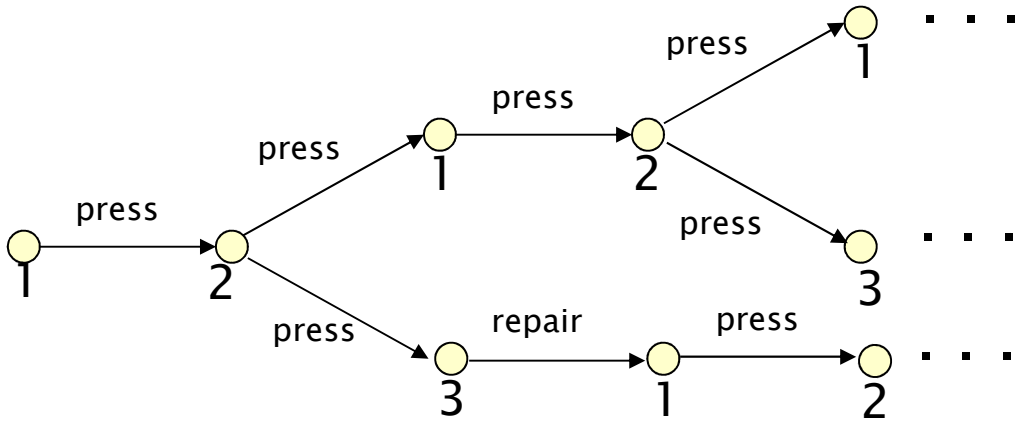
- Temporal logic expresses statements about the temporal order of events
  - e.g. sent → F received
- Consider individual executions, i.e. infinite sequences of states
- Atomic propositions are elementary statements about states appearing in executions
  - e.g. true, ~Closed
- Boolean combinators
  - negation ¬
  - conjunction (∧), disjunction (∨), implication (→)
- Propositional formulas
  - built from atomic propositions and Boolean connectives

Unfold into executions or trees



Linear time





Branching time

Temporal combinators

- For linear time, a model is an infinite sequence

- The temporal combinators express statements about order of events along a sequence
  - X, next
  - F, future
  - G, always
  - U, until

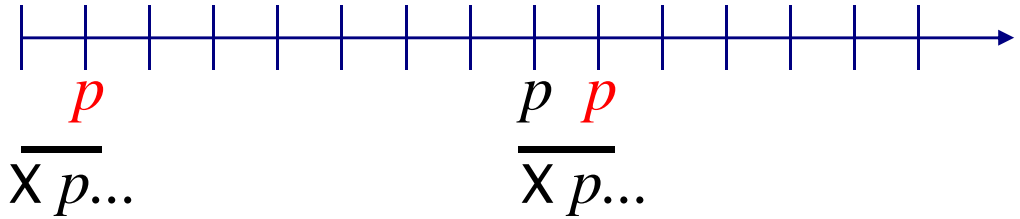
- NB, each of these refers to a particular execution

Linear time temporal logic (LTL)

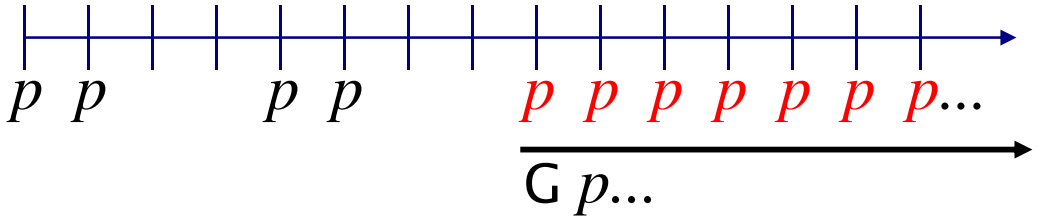
- For LTL, a model is an infinite state sequence

$$\omega = s_0, s_1, s_2 \ldots$$

- Temporal operators
  - "Next-time": X p at t iff p at $t+1$

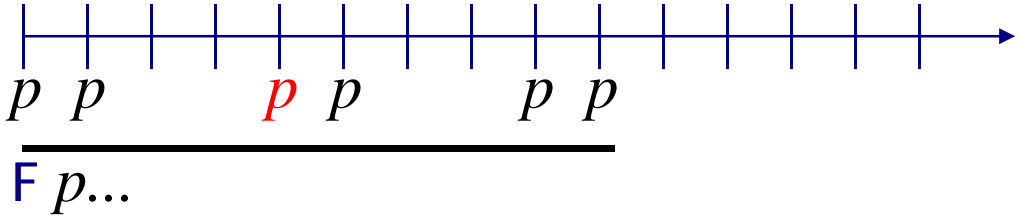


  - "Globally": G p at t iff p for all $t' \geq t$.

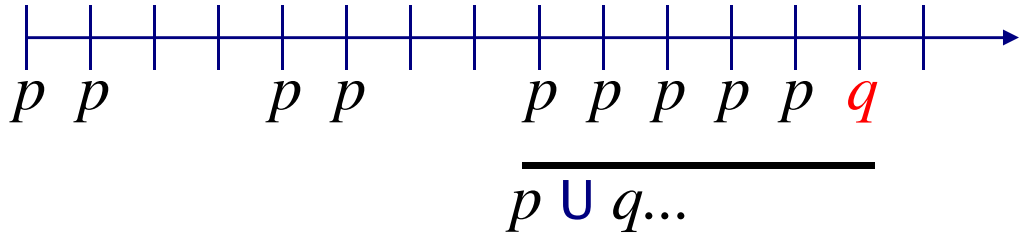


Temporal operators...

- "Future": F *p* at *t* iff *p* for some *t'* ≥ *t.*



- "Until": *p* U *q* at *t* iff
  - *q* for some *t'* ≥ *t* and
  - *p* occurs between (and incl.) *t* and (not incl.) *t'*



Examples

- "p now and in the next two states"

$$p \wedge Xp \wedge XXp$$

atomic prop

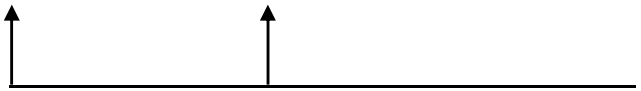
- "no signal until light is on"

$$\neg \text{signal } U \text{ on}$$

must occur

- "if sent, then eventually received"

$$G\ (\text{sent} \Rightarrow F\ \text{received})$$



temporal operator

Linear time temporal logic

- PLTL (Propositional Linear Time Logic)
- Models are infinite sequences of states $\omega = s_0, s_1, s_2 \ldots$
- LTL syntax (path formulae only)
  - $\psi ::= \text{true} \mid a \mid \psi \wedge \psi \mid \neg\psi \mid X\,\psi \mid \psi\,U\,\psi$
  - where $a \in AP$ is an atomic proposition
- Derived formulas
  - $F\,\phi \equiv \text{true}\,U\,\phi$
  - $G\,\phi \equiv \neg(F\,\neg\phi)$

LTL semantics

- LTL semantics (for a path ω)
  - $\omega \models \text{true}$ always
  - $\omega \models a \Leftrightarrow a \in L(\omega(0))$
  - $\omega \models \psi_1 \wedge \psi_2 \Leftrightarrow \omega \models \psi_1$ and $\omega \models \psi_2$
  - $\omega \models \neg\psi \Leftrightarrow \omega \not\models \psi$
  - $\omega \models X\,\psi \Leftrightarrow \omega[1\ldots] \models \psi$
  - $\omega \models \psi_1\ U\ \psi_2 \Leftrightarrow \exists k \geq 0$ s.t. $\omega[k\ldots] \models \psi_2 \wedge \forall i<k\ \omega[i\ldots] \models \psi_1$

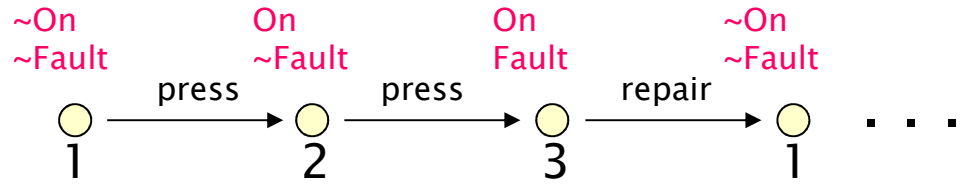
  where $\omega(i)$ is i^{th} state of ω, and $\omega[i\ldots]$ is suffix starting at $\omega(i)$

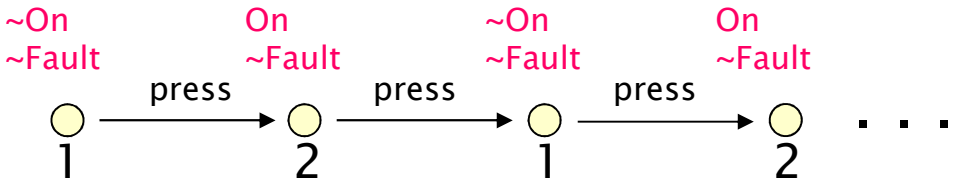
- How to define LTL for an LTS?

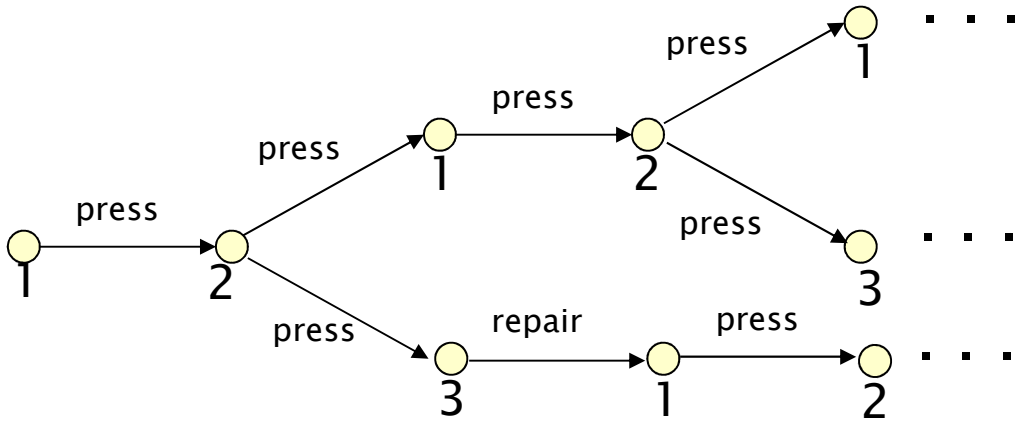
Path quantifiers

- So far, the operators pertain to a single execution
- Branching time logics allow to quantify over paths possible from a given state
- Path quantifiers allow to express:
  - Aψ: all executions out of the state satisfy ψ
  - Eψ: there exists an execution satisfying ψ
- Path quantifiers and temporal operators often used in pairs, e.g.
  - AG ¬deadlock (invariant)
  - EF p (reachability of p)
- Do not confuse A (all paths from the given state) with G (all states of the given path)

Linear vs branching time logic







F On

GF On

(implied A for LTS)

AF On

AGF On

(state dependent)

Computation Tree Logic

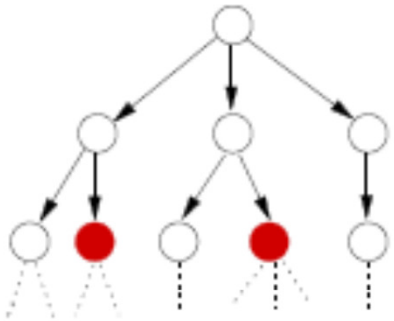
- Variants CTL* (more expressive) and CTL (simpler and easier to model check)

- CTL* composed from
  - propositional logic
  - two types of formulas, state and path formulas
  - path formulas are as in LTL
  - state formulas allow quantification over paths
    - e.g. Aψ for path formula ψ
  - arbitrary nesting

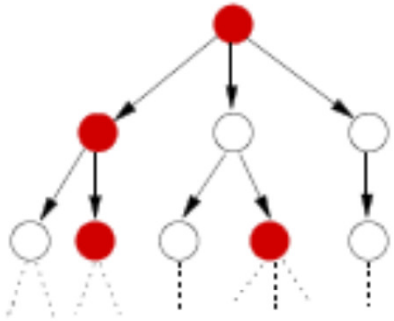
- CTL is a syntactic restriction of CTL*
  - every operator F, G, X, U is preceded by A or E

CTL semantics

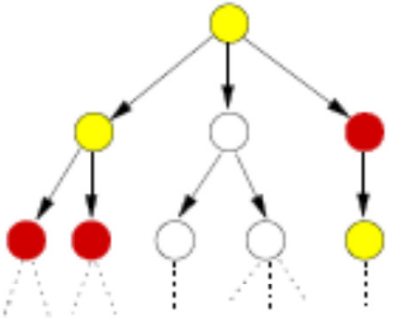
- Intuitive semantics:
  - of quantifiers (A/E) and temporal operators (F/G/U)



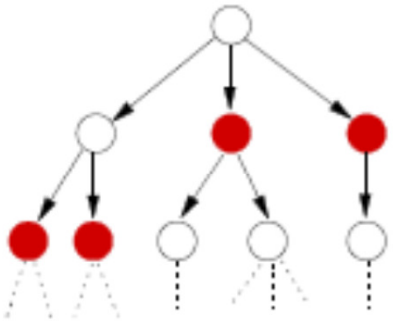
EF red



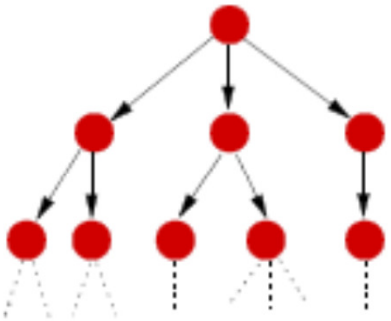
EG red



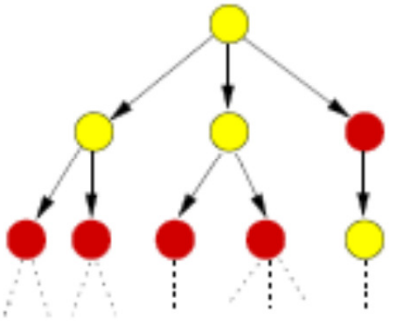
E [yellow U red]



AF red



AG red



A [yellow U red]

CTL semantics

- Semantics of state formulae:
  - $s \models \phi$ denotes "s satisfies ϕ" or "ϕ is true in s"

- For a state s of an LTS $M = (S, s_{init}, \alpha, T, L)$:

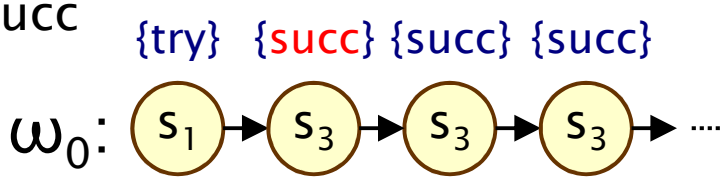
  - $s \models \text{true}$ always
  - $s \models a \Leftrightarrow a \in L(s)$
  - $s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1$ and $s \models \phi_2$
  - $s \models \neg\phi \Leftrightarrow s \not\models \phi$
  - $s \models A\,\psi \Leftrightarrow \omega \models \psi$ for all $\omega \in Path(s)$
  - $s \models E\,\psi \Leftrightarrow \omega \models \psi$ for some $\omega \in Path(s)$

CTL semantics

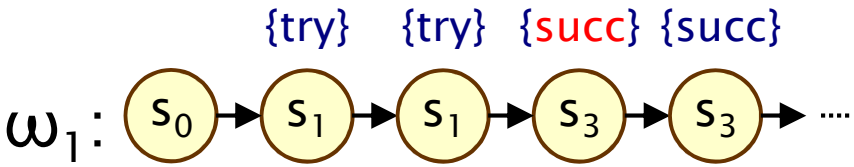
- Semantics of path formulae:
  - $\omega \models \psi$ denotes "ω satisfies ψ" or "ψ is true along ω"
- For a path ω of an LTS $(S, s_{init}, \rightarrow, L)$:
  - $\omega \models X\,\phi \quad \Leftrightarrow \quad \omega(1) \models \phi$
  - $\omega \models F\,\phi \quad \Leftrightarrow \quad \exists k \geq 0$ s.t. $\omega(k) \models \phi$
  - $\omega \models G\,\phi \quad \Leftrightarrow \quad \forall i \geq 0\ \omega(i) \models \phi$
  - $\omega \models \phi_1\ U\ \phi_2 \quad \Leftrightarrow \quad \exists k \geq 0$ s.t. $\omega(k) \models \phi_2$ and $\forall i < k\ \omega(i) \models \phi_1$

CTL examples

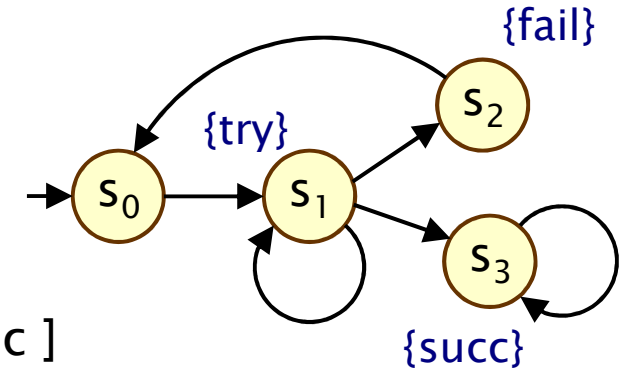
- Some examples of satisfying paths:
  - $\omega_0 \models$ X succ



  - $\omega_1 \models \neg$fail U succ



- Example CTL formulas:
  - $s_1 \models$ try $\wedge \neg$fail
  - $s_1 \models$ E [X succ] and $s_3 \models$ A [X succ]
  - $s_0 \models$ E [$\neg$fail U succ] but $s_0 \not\models$ A [$\neg$fail U succ]

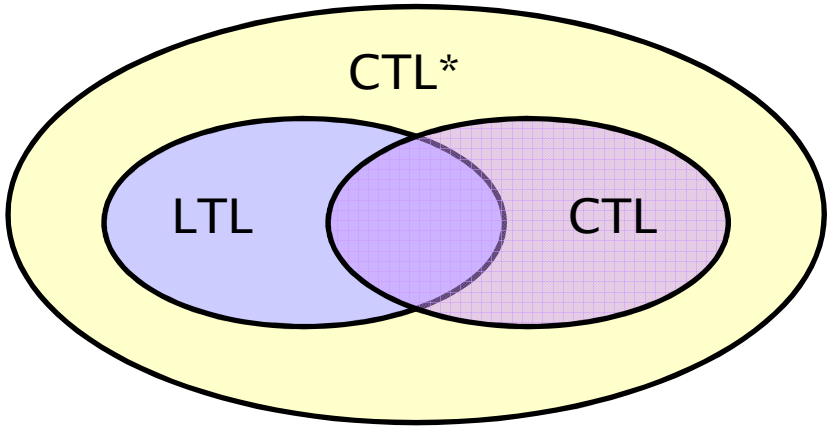


CTL examples

- AG ($\neg(crit_1 \wedge crit_2)$)
  - mutual exclusion
- AG EF initial
  - for every computation, it is always possible to return to the initial state
- AG (request → AF response)
  - every request will eventually be granted
- AG AF $crit_1$ ∧ AG AF $crit_2$
  - each process has access to the critical section infinitely often

Expressiveness

- LTL less expressive than CTL*
  - EF ϕ not LTL-expressible
- CTL sublogic of CTL*
  - FG ϕ not CTL-expressible



- LTL and CTL not comparable
  - FG ϕ is LTL- but not CTL-expressible
  - EF ϕ is CTL- but not LTL-expressible

CTL model checking

- Given
  - a finite-state labelled transition system $M = (S, s_{init}, \alpha, T, L)$:
    - where AP are atomic propositions
    - $L: S \rightarrow 2^{AP}$ is a labelling of states with propositions
  - and a CTL formula ϕ

- Find all states in M that satisfy ϕ :

  $\{s \in S \mid M,s \models \phi\}$

  and check that this set includes all initial states

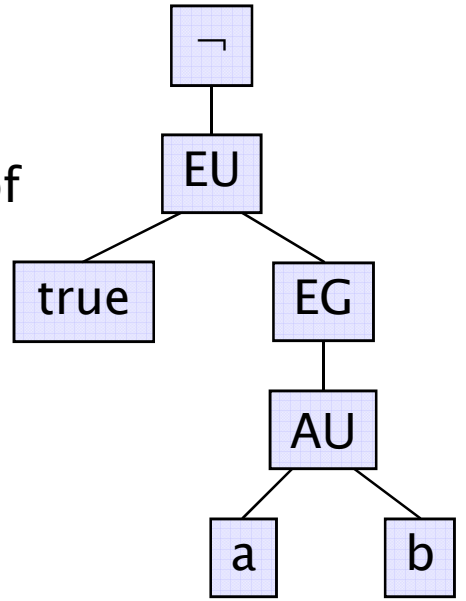
- Model checking much more efficient than LTL and CTL*

CTL model checking idea

- Convert formula to ENF
- Build parse tree of the formula

- Proceed recursively, bottom-up (from leaves upwards) labelling states for each subformula
  - if subformula is true in $s \in S$, add it to the set of labels for s
  - continue, going up the formula parse tree
  - stop when root of the parse tree is checked

- When the algorithm terminates
  - $M \models \phi$ iff the initial state is labelled with ϕ



Complexity

- CTL model checking (EG is worst case)
  - Partition the state space into strongly connected components (subgraphs where every state can reach every other state), $O(|S|+|T|)$
  - Traverse the transition graph, $O(|S|+|T|)$
  - Label for each subformula, $|\phi|$ of them
- The overall complexity is

  $O(|\phi|*(|S|+|T|))$
- In contrast, LTL/CTL* model checking
  - is $O(2^{|\phi|}*(|S|+|T|))$, i.e. exponential in size of formula (PSPACE)
  - Linear in size of model, as is CTL
  - Proceeds by automata-theoretic methods (product with the LTS)

Overview (Part 1)

- Introduction
- Transition systems
- Temporal logic
- Model checking
  - reachability
  - CTL model checking
- PRISM: overview
  - Probability example
  - Modelling language
  - Properties
  - GUI, etc
- Summary

PRISM

- PRISM: Probabilistic symbolic model checker
  - developed at Birmingham/Oxford University, since 1999
  - free, open source software (GPL), runs on all major OSs

- Construction/analysis of probabilistic models…
  - discrete-time Markov chains, continuous-time Markov chains, Markov decision processes, probabilistic timed automata, stochastic multi-player games, …
- Simple but flexible high-level modelling language
  - based on guarded commands; see later…
- Many import/export options, tool connections
  - in: (Bio)PEPA, stochastic π-calculus, DSD, SBML, Petri nets, …
  - out: Matlab, MRMC, INFAMY, PARAM, …

PRISM…

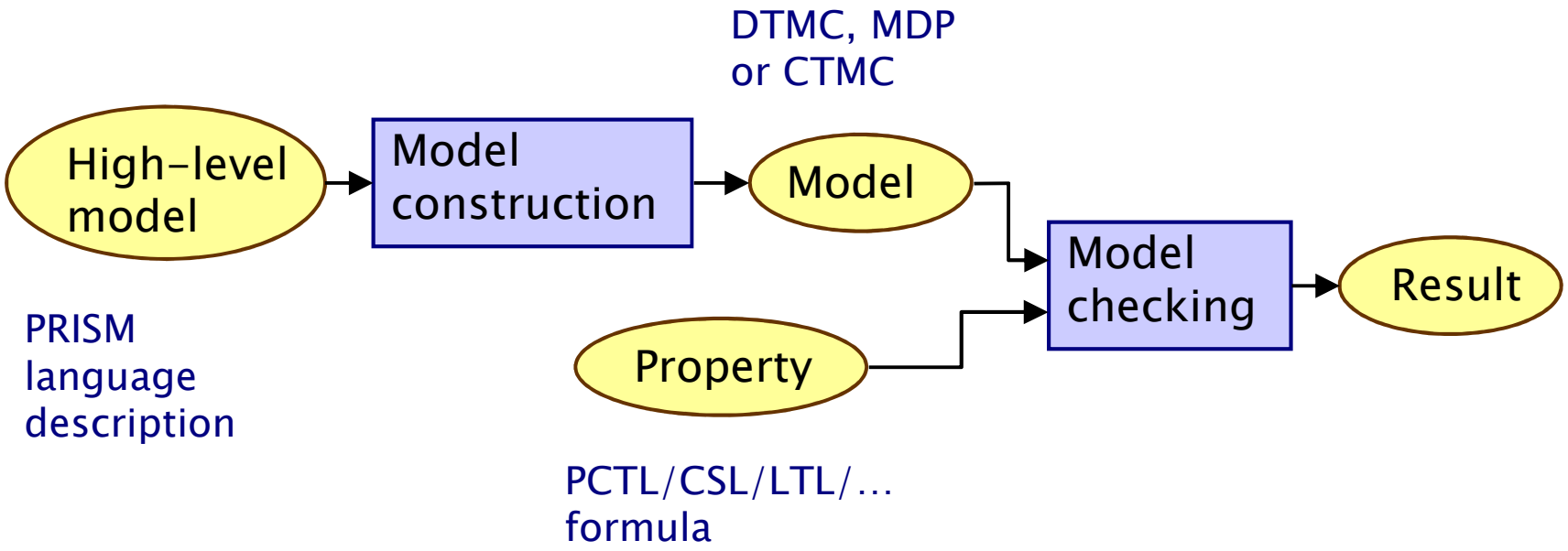
- Model checking for various temporal logics…
  - probabilistic/reward extensions of CTL/CTL*/LTL
  - PCTL, CSL, LTL, PCTL*, rPATL, CTL, …
- Various efficient model checking engines and techniques
  - symbolic methods (binary decision diagrams and extensions)
  - explicit-state methods (sparse matrices, etc.)
  - statistical model checking (simulation-based approximations)
  - and more: symmetry reduction, quantitative abstraction refinement, fast adaptive uniformisation, ...
- Graphical user interface
  - editors, simulator, experiments, graph plotting
- See: http://www.prismmodelchecker.org/
  - downloads, tutorials, case studies, papers, …

PRISM functionality

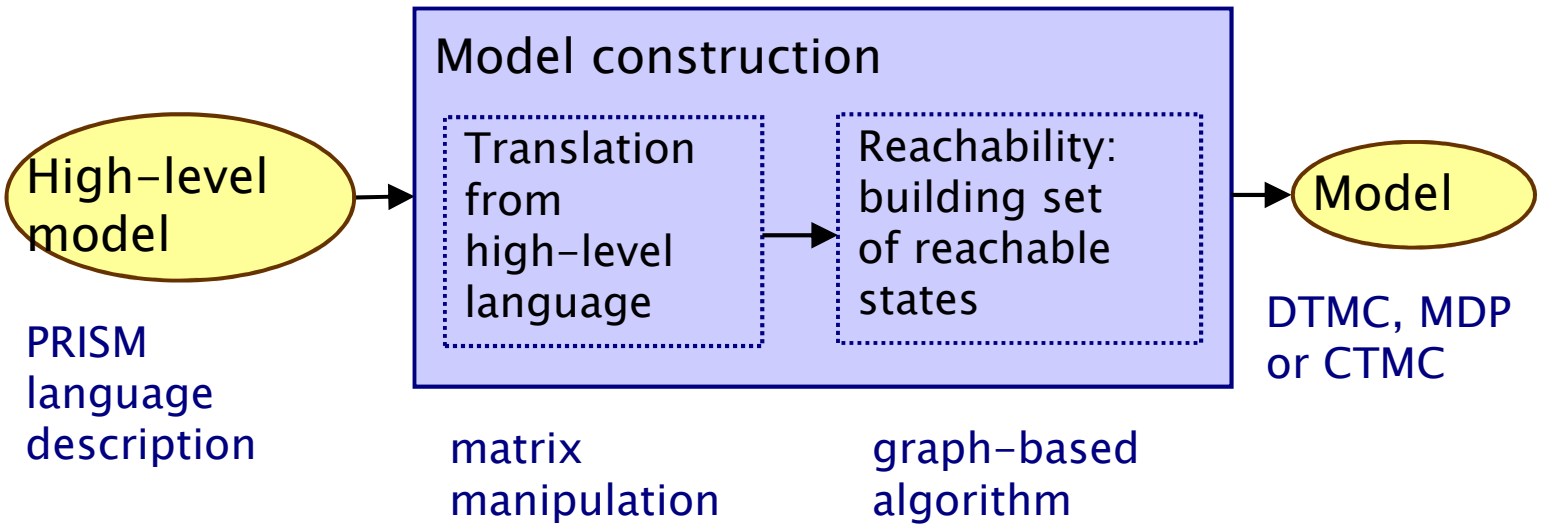
- High-level modelling language
- Wide range of model analysis methods
  - efficient symbolic implementation techniques
  - also: approximate verification using simulation + sampling
- Graphical user interface
  - model/property editor
  - discrete-event simulator – model traces for debugging, etc.
  - easy automation of verification experiments
  - graphical visualisation of results
- Command-line version
  - same underlying verification engines
  - useful for scripting, batch jobs

Probabilistic model checking

- Overview of the probabilistic model checking process
  - two distinct phases: model construction, model checking



Model construction



PRISM modelling language

- Simple, textual, state-based language
  - modelling of DTMCs, CTMCs, MDPs, …
  - based on Reactive Modules [AH99]

- Basic components…
- Modules:
  - components of system being modelled
  - composed in parallel
- Variables
  - finite (integer ranges or Booleans)
  - local or global
  - all variables public: anyone can read, only owner can modify

PRISM modelling language

- Guarded commands
  - describe behaviour of each module
  - i.e. the changes in state that can occur
  - labelled with probabilities (or, for CTMCs, rates)
  - (optional) action labels

[send] (s=2) -> p_{loss} : (s'=3)&(lost'=lost+1) + (1-p_{loss}) : (s'=4);

action guard probability update probability update

PRISM modelling language

- Parallel composition
  - model multiple components that can execute independently
  - for DTMC models, mostly assume components operate synchronously, i.e. move in lock-step
- Synchronisation
  - simultaneous transitions in more than one module
  - guarded commands with matching action-labels
  - probability of combined transition is product of individual probabilities for each component
- More complex parallel compositions can be defined
  - using process-algebraic operators
  - other types of parallel composition, action hiding/renaming

Simple example

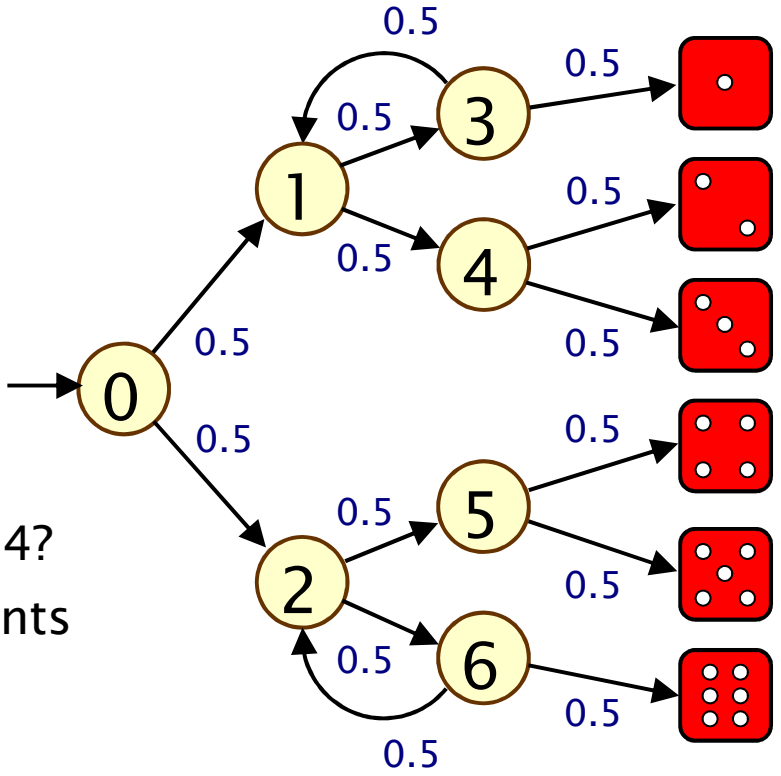
```
dtmc

module M1
    x : [0..3] init 0;
    [a] x=0 -> (x'=1);
    [b] x=1 -> 0.5 : (x'=2) + 0.5 : (x'=3);
endmodule

module M2
    y : [0..3] init 0;
    [a] y=0 -> (y'=1);
    [b] y=1 -> 0.4 : (y'=2) + 0.6 : (y'=3);
endmodule
```

Probability example

- Modelling a 6-sided die using a fair coin
  - algorithm due to Knuth/Yao:
  - start at 0, toss a coin
  - upper branch when H
  - lower branch when T
  - repeat until value chosen

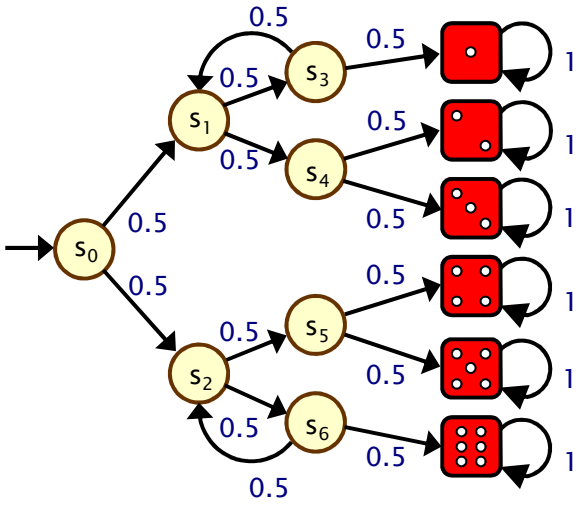


- Is this algorithm correct?
  - e.g. probability of obtaining a 4?
  - obtain as disjoint union of events
  - THH, TTTHH, TTTTTHH, …
  - Pr("eventually 4")

    $= (1/2)^3 + (1/2)^5 + (1/2)^7 + \ldots = 1/6$

Example…

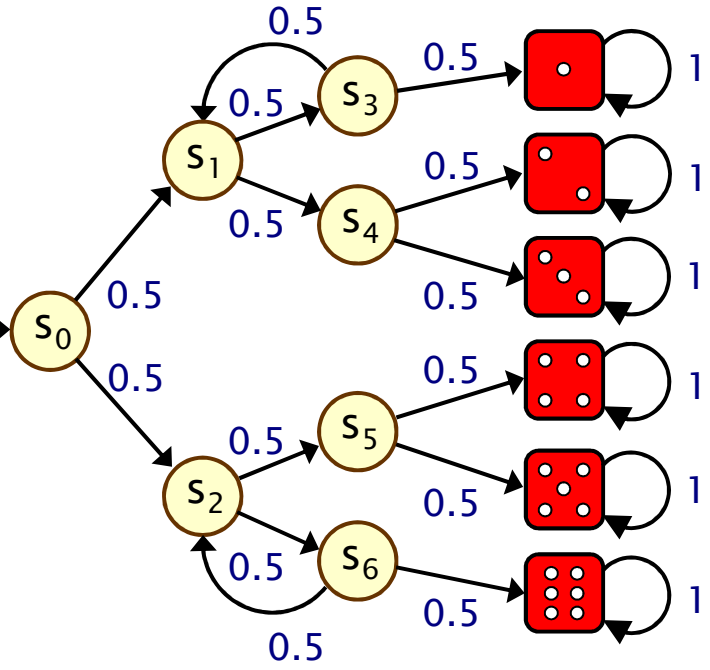
- Other properties?
  - "what is the probability of termination?"
- e.g. efficiency?
  - "what is the probability of needing more than 4 coin tosses?"
  - "on average, how many coin tosses are needed?"



- Probabilistic model checking provides a framework for these kinds of properties…
  - modelling languages
  - property specification languages
  - model checking algorithms, techniques and tools

Probabilistic models

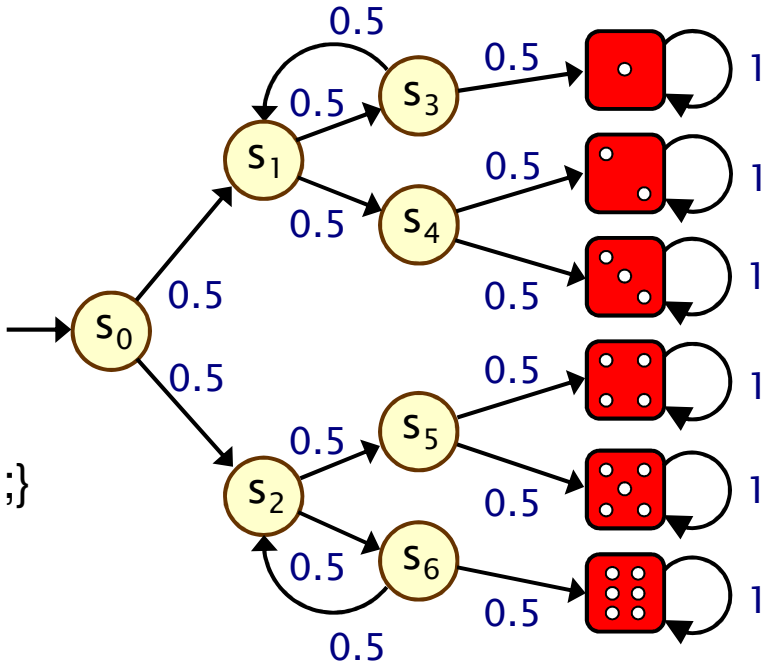
```
dtmc
module die
// local state s : [0..7] init 0;
// value of the dice d : [0..6] init 0;
[] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
…
[] s=3 ->
   0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
[] s=4 ->
   0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
…
[] s=7 -> (s'=7);
endmodule
rewards "coin_flips"
[] s<7 : 1;
endrewards
```



Given in PRISM's guarded commands modelling notation

Probabilistic models

```
int s, d;
s = 0; d = 0;
while (s < 7) {
   bool coin = Bernoulli(0.5);
   if (s = 0)
      if (coin) s = 1 else s = 2;
...
  else if (s = 3)
      if (coin) s = 1 else {s = 7; d = 1;}
  else if (s = 4)
      if (coin) {s = 7; d = 2} else {s = 7; d = 3;}
…
}
return (d)
```



Given as a probabilistic program

Costs and rewards

- We augment models with rewards (or, conversely, costs)
  - real-valued quantities assigned to states and/or transitions
  - these can have a wide range of possible interpretations
- Some examples:
  - elapsed time, power consumption, size of message queue, number of messages successfully delivered, net profit, …
- Costs? or rewards?
  - mathematically, no distinction between rewards and costs
  - when interpreted, we assume that it is desirable to minimise costs and to maximise rewards
  - we consistently use the terminology “rewards” regardless
- Properties (see later)
  - reason about expected cumulative/instantaneous reward

Rewards in the PRISM language

```
rewards "total_queue_size"
  true : queue1+queue2;
endrewards
```

(instantaneous, state rewards)

```
rewards "time"
  true : 1;
endrewards
```

(cumulative, state rewards)

```
rewards "dropped"
  [receive] q=q_max : 1;
endrewards
```

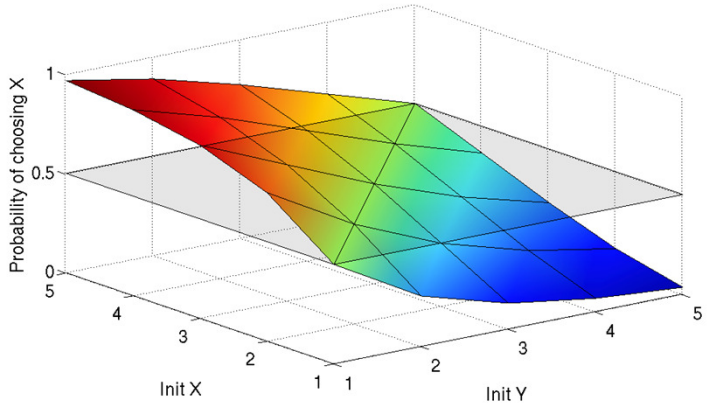
(cumulative, transition rewards)
(q = queue size, q_max = max. queue size, receive = action label)

```
rewards "power"
  sleep=true : 0.25;
  sleep=false : 1.2 * up;
  [wake] true : 3.2;
endrewards
```

(cumulative, state/trans. rewards)
(up = num. operational components, wake = action label)

PRISM – Property specification

- Temporal logic-based property specification language
  - subsumes PCTL, CSL, probabilistic LTL, PCTL*, …
- Simple examples:
  - $P_{\leq 0.01}$ [F "crash"] – "the probability of a crash is at most 0.01"
  - $S_{>0.999}$ ["up"] – "long-run probability of availability is >0.999"
- Usually focus on quantitative (numerical) properties:
  - $P_{=?}$ [F "crash"]
    "what is the probability of a crash occurring?"
  - then analyse trends in quantitative properties as system parameters vary

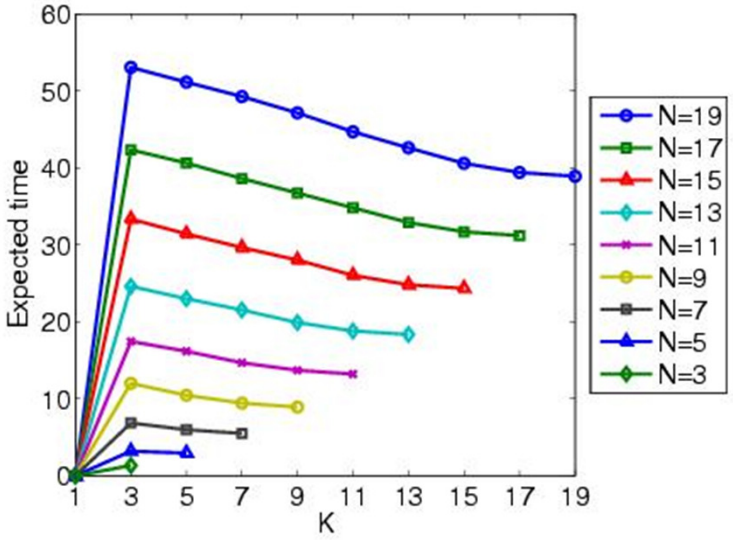


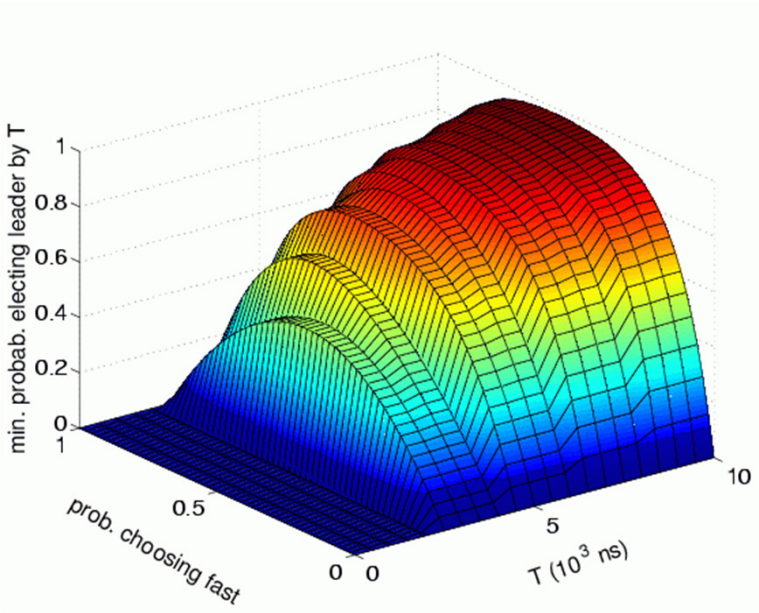
PRISM – Property specification

- Properties can combine numerical + exhaustive aspects
  - $P_{max=?}$ [$F^{\leq 10}$ "fail"] – "worst-case probability of a failure occurring within 10 seconds, for any possible scheduling of system components"
  - $P_{=?}$ [$G^{\leq 0.02}$!"deploy" {"crash"}{max}] – "the maximum probability of an airbag failing to deploy within 0.02s, from any possible crash scenario"
- Reward-based properties (rewards = costs = prices)
  - $R_{\{\text{"time"}\}=?}$ [F "end"] – "expected algorithm execution time"
  - $R_{\{\text{"energy"}\}max=?}$ [$C^{\leq 7200}$] – "worst-case expected energy consumption during the first 2 hours"
- Properties can be combined with e.g. arithmetic operators
  - e.g. $P_{=?}$ [F $fail_1$] / $P_{=?}$ [F $fail_{any}$] – "conditional failure prob."

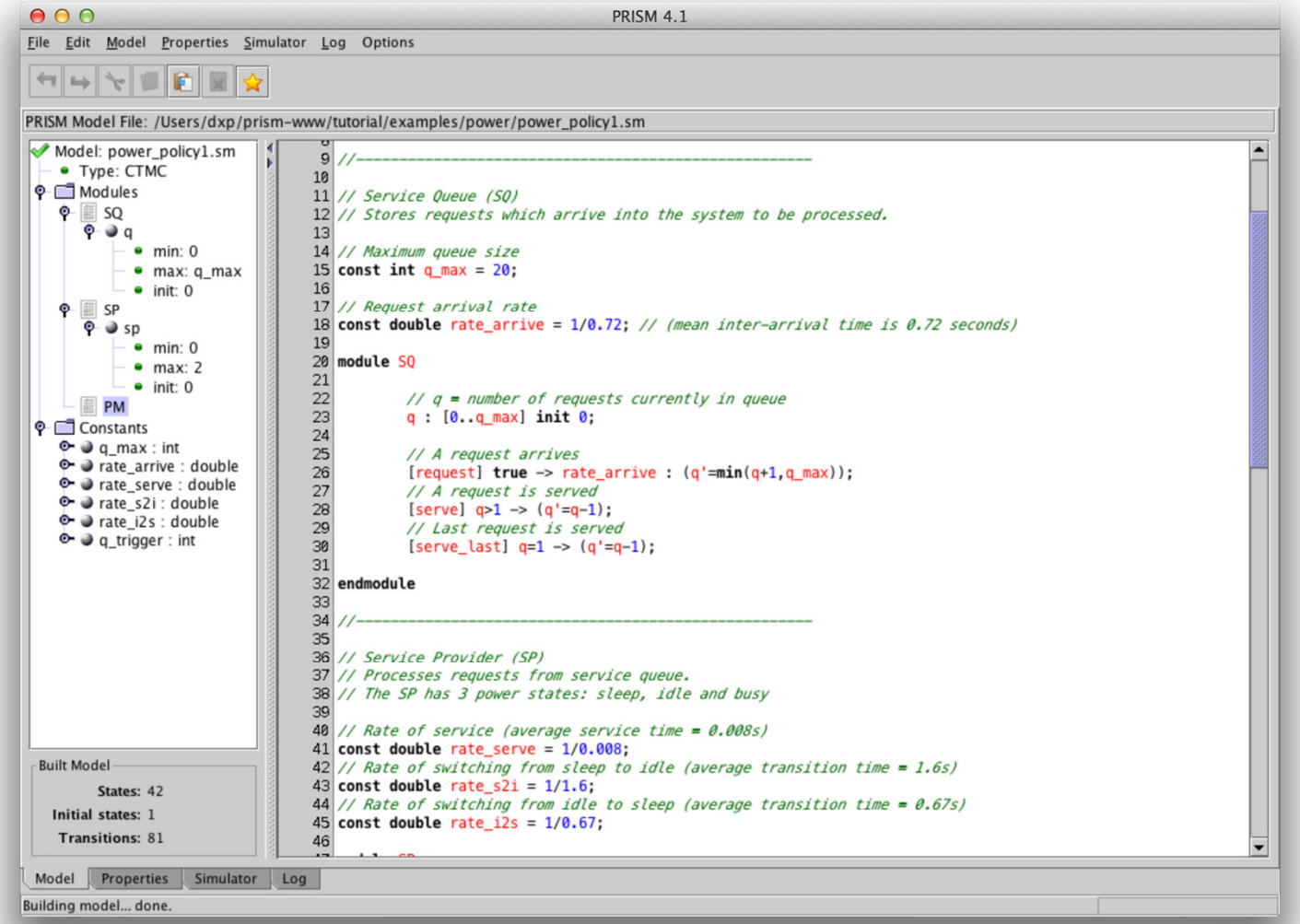
PRISM property specifications

- Experiments:
  - ranges of model/property parameters
  - e.g. $P_{=?}$ [$F^{\leq T}$ error] for N=1..5, T=1..100
    where N is some model parameter and T a time bound
  - identify patterns, trends, anomalies in quantitative results

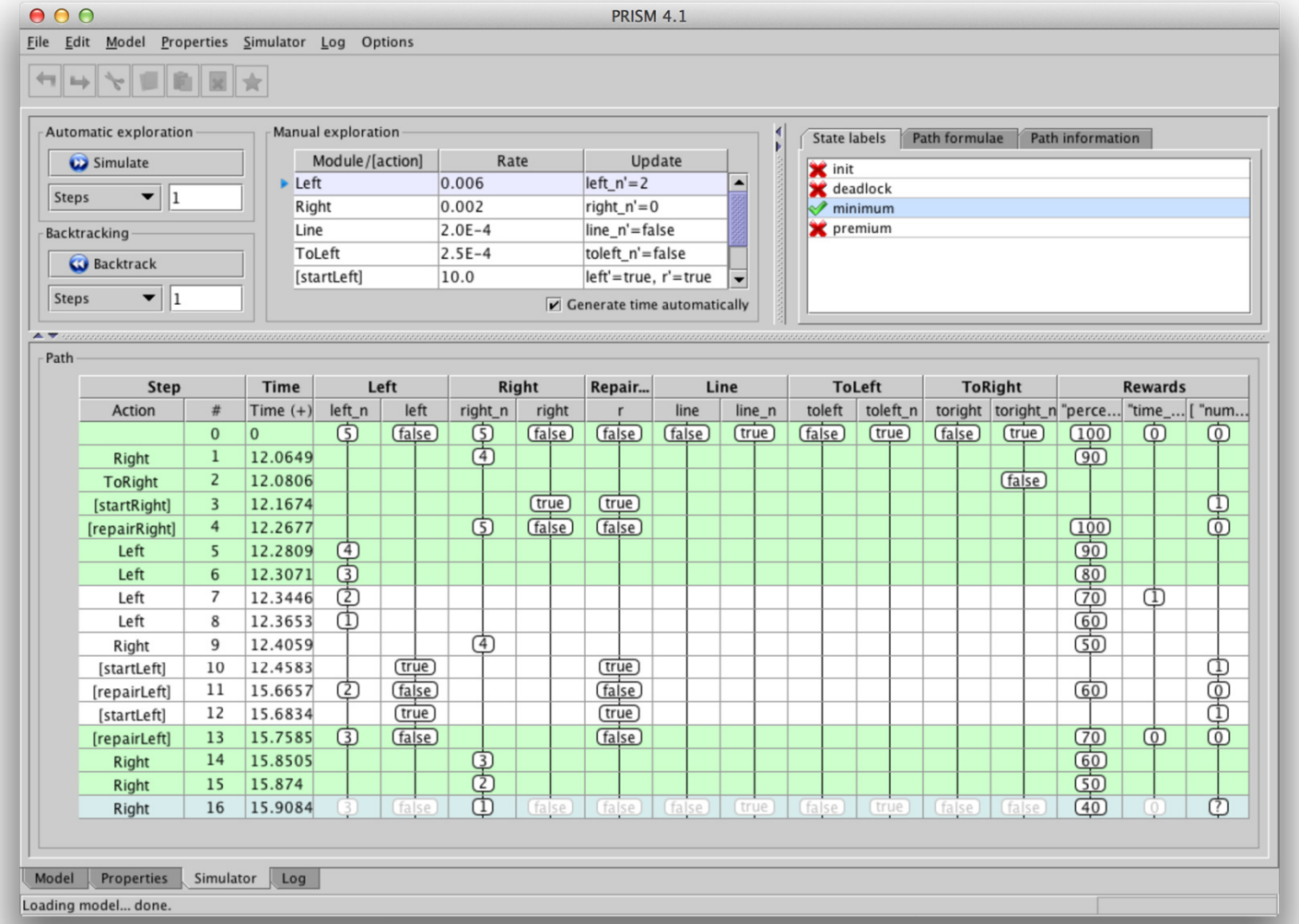




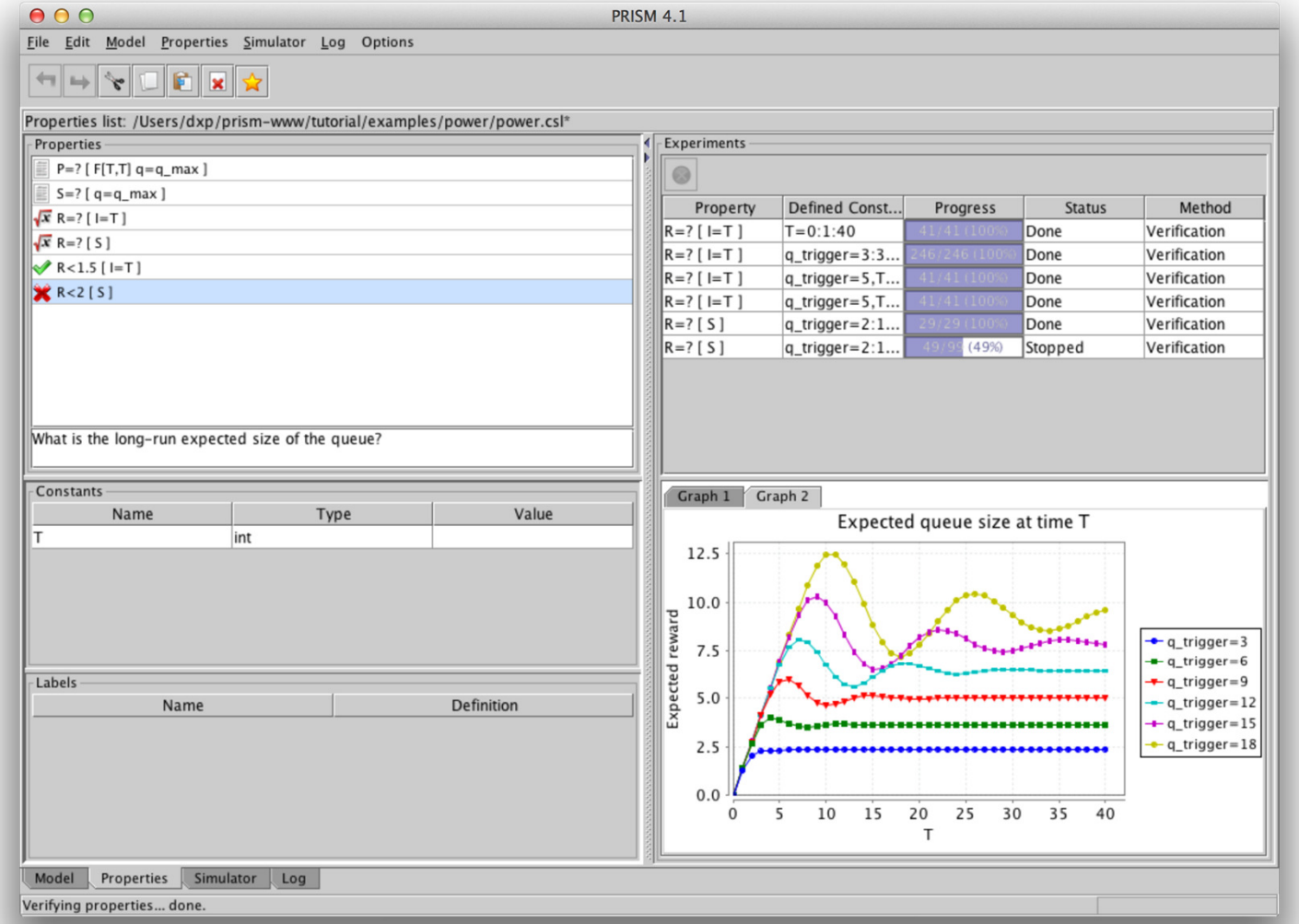
PRISM GUI: Editing a model



PRISM GUI: The Simulator



PRISM GUI: Model checking and graphs



PRISM – Case studies

- Randomised distributed algorithms
  - consensus, leader election, self-stabilisation, …
- Randomised communication protocols
  - Bluetooth, FireWire, Zeroconf, 802.11, Zigbee, gossiping, …
- Security protocols/systems
  - contract signing, anonymity, pin cracking, quantum crypto, …
- Biological systems
  - cell signalling pathways, DNA computation, …
- Planning & controller synthesis
  - robotics, dynamic power management, …
- Performance & reliability
  - nanotechnology, cloud computing, manufacturing systems, …

- See: www.prismmodelchecker.org/casestudies

Summary (Part 1)

- Introduced reactive systems
  - modelled by labelled transition systems
  - unfolded into execution paths or trees
- Property specifications
  - expressed in temporal logic, e.g. CTL, LTL
- Model checking algorithms
  - graph-based algorithms
  - automata constructions

- PRISM: Probabilistic Symbolic Model Checker

- Next: discrete-time Markov chains (DTMCs)