

Computer-Aided Verification



Dave Parker

University of Birmingham

2017/18

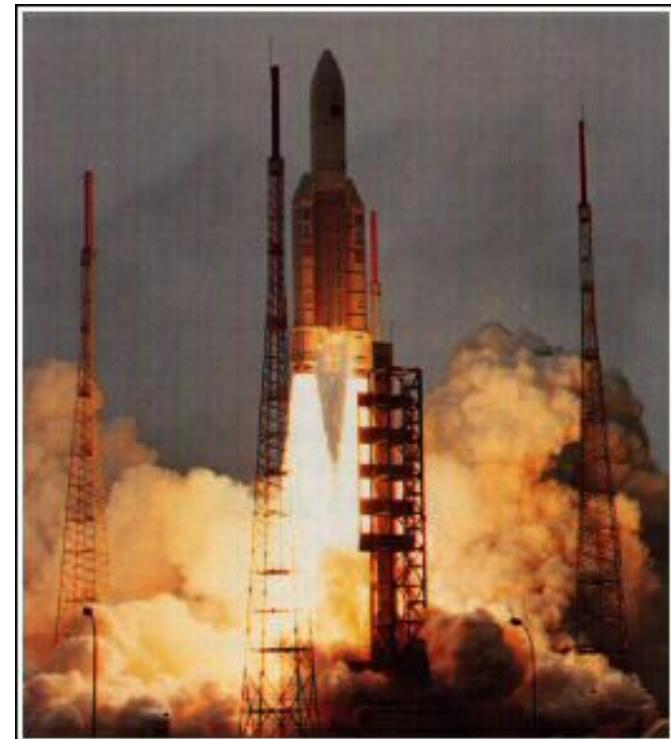
- Formal verification
 - is the application of rigorous, mathematical methods to establish the correctness of computerised systems
 - "system" = software, hardware, protocol, ...
- Computer-aided verification
 - automated verification: tools, algorithms, ...

Overview

- Motivation
 - why verify?
- Computer-aided verification
 - model checking
 - example
- This module
 - syllabus
 - aims, delivery, resources, ...

Ariane 5

- ESA (European Space Agency) Ariane 5 launcher
 - shown here in maiden flight on 4th June 1996
- 37secs later self-destructs
 - numerical overflow in a conversion routine (64-bit float to 16-bit int)
 - results in incorrect altitude sent by the on-board computer
 - exception handling disabled
- Expensive, embarrassing...
 - more than \$500 million



Toyota Prius

- Toyota Prius
 - first mass-produced hybrid vehicle
- February 2010
 - software “glitch” in anti-lock braking system
 - in response to numerous complaints/accidents
 - eventually fixed via software update
 - 185,000 cars recalled, at huge cost
 - much criticism of handling, significant bad publicity
- Further recalls in 2015
 - 625,000 Toyota hybrids recalled due to a software "glitch"
 - can cause loss of power while driving



Correctness is crucial

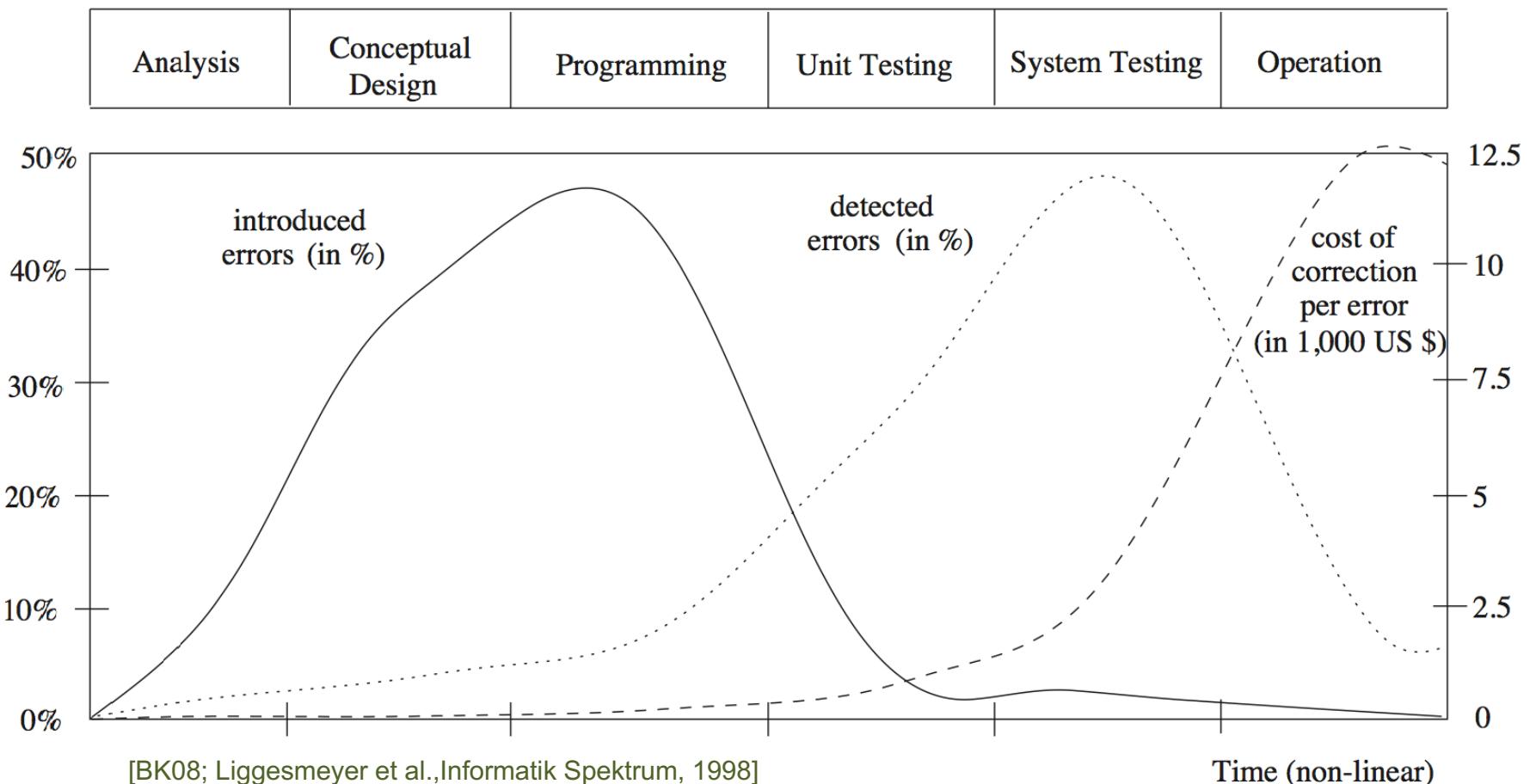
- What do the previous examples have in common?
 - failures/errors in programmable computerised devices
 - high costs incurred (and not just financial)
 - failures were avoidable
- Trends in computing
 - computerised devices are everywhere...
 - including business-, safety-critical domains
 - avionic/automotive embedded software, driverless cars, medical sensors/devices
 - software is increasingly complex
 - and increasingly interconnected



How to ensure correctness?

- Existing methods for checking (software) correctness:
- Code review
 - peer review of code by other programmers
 - static, manual analysis during/after development
- Testing
 - program execution against test suite of known results
 - dynamic analysis performed after development
- Analysis types:
 - dynamic = based on execution of system
 - static = examination of source code/design (no execution)

When to check correctness?



Formal verification

- Basic idea:
 - check system correctness using formal, logical reasoning
 - build a mathematical **model** of a system and then prove that it satisfies a formal **specification** of correctness



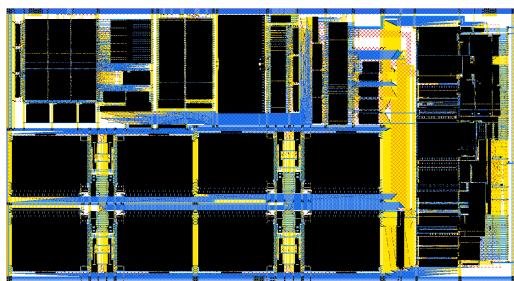
- Key idea: “Testing can only show the presence of errors, not their absence”

Edsger Dijkstra
1930-2002

- Key points:
 - static analysis (based on source code or system design)
 - opportunities for early integration in the design process
 - exhaustive (higher coverage than testing alone)
 - verification, not validation
 - or, help to identify subtle bugs/flaws

Automatic verification

- Formal verification
 - essentially: proving that a program satisfies its specification
 - i.e., that all possible system executions behave correctly
 - many techniques: manual proof, automated theorem proving, static analysis, model checking, ...



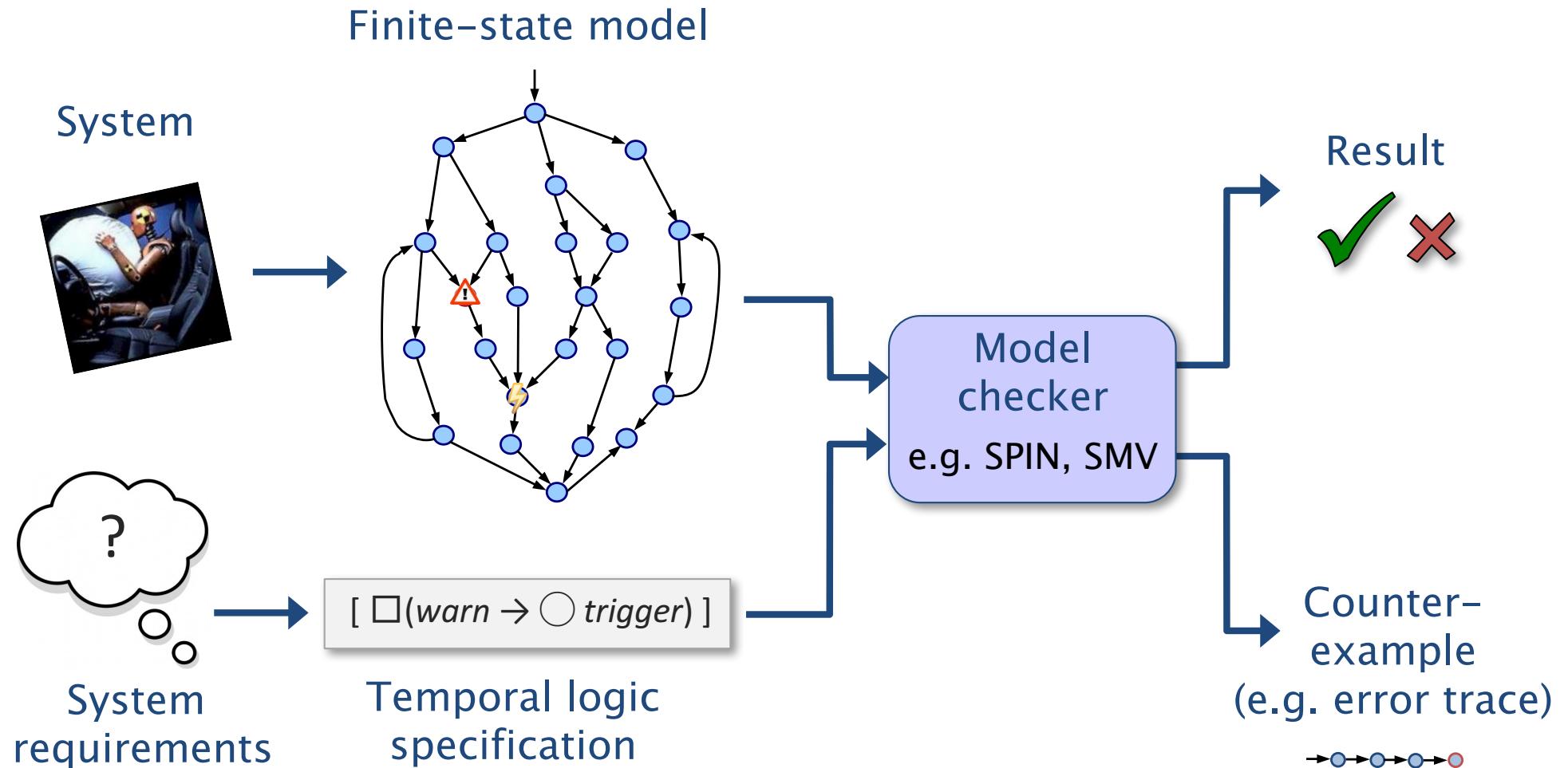
$10^{500,000}$ states



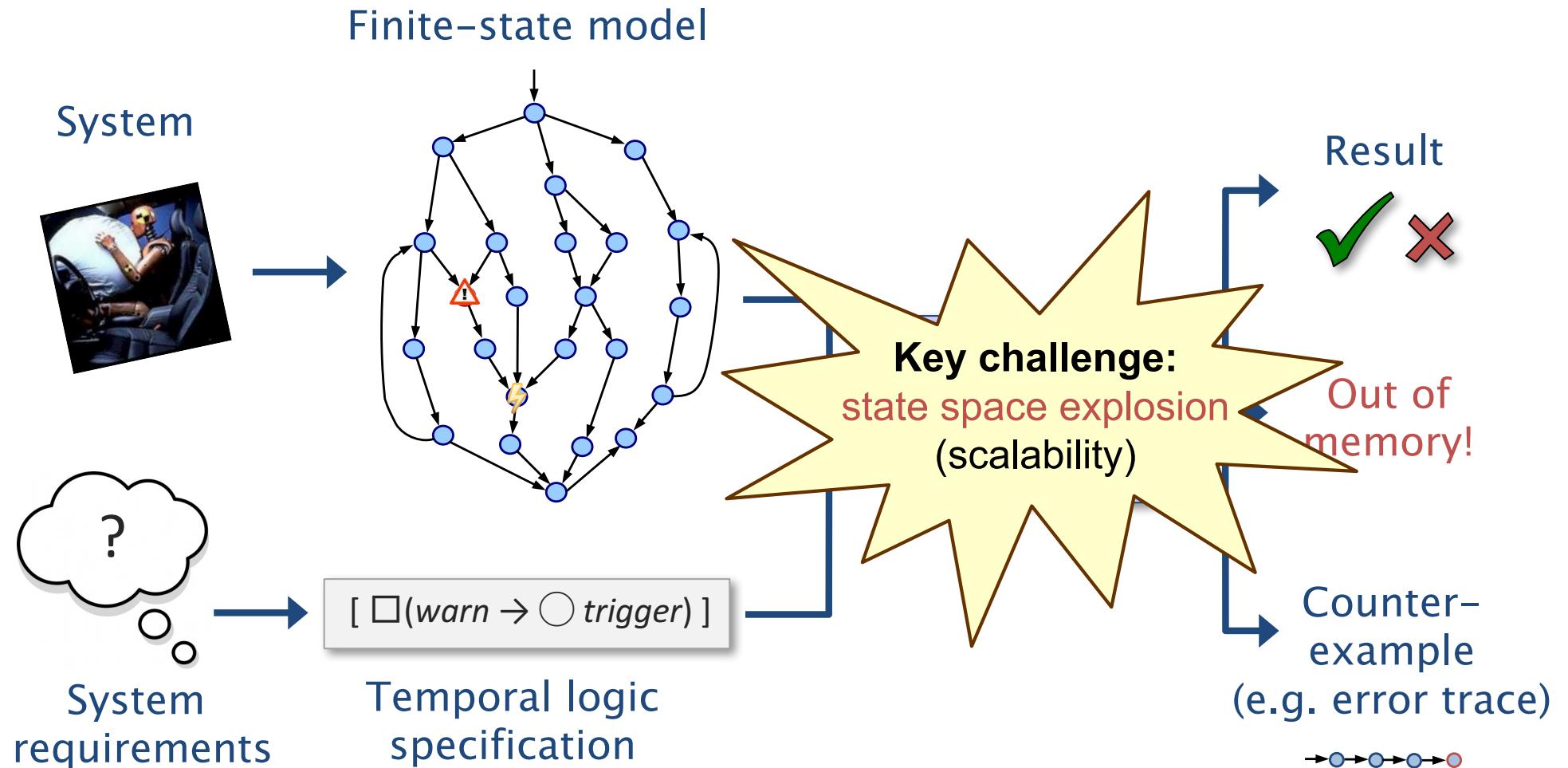
10^{70} atoms

- Computer-aided (automatic) verification
 - push-button technology: algorithms, tools
 - in particular: **model checking**

Verification via model checking



Verification via model checking



Why study verification?

- Topic of growing importance
 - increasing need for correctness + advances in technology
- Many successful applications in practice
 - NASA: Deep Space 1, Mars Pathfinder
 - Microsoft: SLAM project (device drivers)
 - security: Needham-Schroeder public key protocol (Lowe + FDR)
 - also: hardware industry, avionics, Facebook, ...
- Major research area
 - e.g. Turing award for Clarke/Emerson/Sifakis in 2007
- What's involved?
 - automata theory, graph algorithms, logic, data structures, software tools

Example

- A simple concurrent program:

```
process Inc = while true do if x < 200 then x := x + 1 od  
  
process Dec = while true do if x > 0 then x := x - 1 od  
  
process Reset = while true do if x = 200 then x := 0 od
```

- Specification:
 - variable x always remains in the range $\{0, 1, \dots, 200\}$
- Is this true?

(example from [BK08])

Example – modelling

- Modelled in PROMELA
 - (input language for SPIN model checker)

```
int x=0;
proctype Inc () {
    do :: true -> if :: (x < 200) -> x = x + 1 fi od
}
proctype Dec() {
    do :: true -> if :: (x > 0) -> x = x - 1 fi od
}
proctype Reset () {
    do :: true -> if :: (x == 200) -> x = 0 fi od
}
init {
    run Inc() ; run Dec() ; run Reset()
}
```

Example – specification

- Add a "monitor" process that checks whether $0 \leq x \leq 200$

```
int x=0;
proctype Inc () {
    do :: true -> if :: (x < 200) -> x = x + 1 fi od
}
proctype Dec() {
    do :: true -> if :: (x > 0) -> x = x - 1 fi od
}
proctype Reset () {
    do :: true -> if :: (x == 200) -> x = 0 fi od
}
proctype Check () {
    assert (x >= 0 && x <= 200)
}
init {
    run Inc(); run Dec(); run Reset(); run Check()
}
```

Example – model checking

- Analysis using the SPIN model checker
 - does the assertion always hold?
- No...

```
.....  
pan: assertion violated ((x >= 0) && (x <= 200)) (at depth 1802)  
pan: wrote pan_in.trail  
.....  
State-vector 32 byte, depth reached 3598, errors: 1  
12609 states, stored
```

Example – counterexample

- Counterexample trace produced by the model checker:

```
.....  
605: proc 1 (Inc)           [((x<200))]  
606: proc 1 (Inc)           [x = (x+1)]  
607: proc 2 (Dec)           [((x > 0))]  
608: proc 1 (Inc)           [(1)]  
609: proc 3 (Reset) line 13 "pan_in" (state 2) [((x==200))]  
610: proc 3 (Reset) line 13 "pan_in" (state 3) [x=0]  
611: proc 3 (Reset) line 13 "pan_in" (state 1) [(1)]  
612: proc 2 (Dec) line 5 "pan_in" (state 3) [x = (x-1)]  
613: proc 2 (Dec) line 5 "pan_in" (state 1) [(1)]  
spin: line 17 "pan_in", Error: assertion violated spin: text of failed  
assertion: assert(((x>=0)&&(x<=200)))
```

Module syllabus

- Modelling sequential and parallel systems
 - labelled transitions systems, parallel composition
- Temporal logic
 - LTL, CTL and CTL*, etc.
- Model checking
 - CTL model checking algorithms
 - automata-theoretic model checking (LTL)
- Verification tools: SPIN
- Advanced verification techniques
 - bounded model checking via propositional satisfiability
 - (symbolic execution), (symbolic model checking)
- Quantitative verification
 - real-time systems, probabilistic systems

Module aims & focus

- Aims of the module:
 - introduce the basic ideas of automatic verification
 - familiarise you with key **techniques & algorithms** for verification
 - illustrate **uses & applications** of automatic verification
 - give practical experience in state of the art **verification software**
 - provide a foundation for further study in the area of verification
- Mix of: theory + algorithms + tools
 - no programming, but use of modelling languages
- Prerequisites
 - background in: propositional logic, automata, graph algorithms

Module delivery

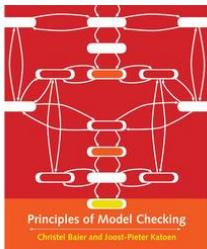
- Lectures:
 - Tue 11–12:
 - Lecture Room 7, Arts Building
 - Thur 12–1:
 - G29, Mechanical and Civil Engineering
- Tutorials (feedback on exercises) (not all weeks):
 - Thur 4–5 (surnames A–L, by default):
 - UG06, Murray Learning Centre
 - Fri 10–11 (surnames M–Z, by default):
 - Lecture Theatre 1, Sports and Exercise Sciences

Assessment

- **Split:**
 - 80% exam (1.5hr, in the summer)
 - 20% continuous assessment
- **Continuous assessment**
 - 4 exercises (1st is formative; 2–4 are assessed: 6%/8%/6%)
 - 1 week for each: due Thurs of weeks 3, 5, 8, 11
 - 1 extra assessed exercise for "extended" version (due week 10)
 - submitted through Canvas
 - solutions worked through in tutorial sessions

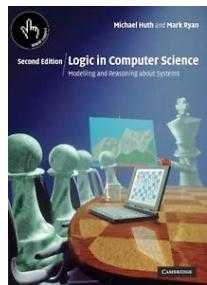
Reference material

- Useful books



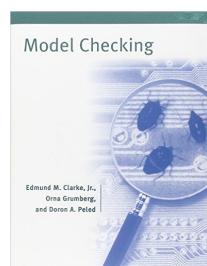
Principles of Model Checking

Christel Baier and Joost-Pieter Katoen [BK08]
The MIT Press, 2008



**Logic in Computer Science:
Modelling and reasoning about systems**

Michael Huth and Mark Ryan
Cambridge University Press, 2004



Model Checking

Edmund M. Clarke, Orna Grumberg, Doron Peled
2nd edn., MIT Press, 2000

- Links to further papers/tutorials will be added to Canvas

Resources

- Canvas
 - <https://canvas.bham.ac.uk/courses/27245>
 - lecture slides/videos, links, assessments, announcements
- Facebook group
 - <https://www.facebook.com/groups/bham.cav.1718>
 - questions, discussion, announcements
- Office hours
 - room 133 (see my door/webpage for times)

Next lecture

- Modelling sequential and parallel systems
 - labelled transition systems
 - parallel composition
 - see Chapter 2 of [BK08]

2. Modelling Sequential and Parallel Systems



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

Module aims & focus

- Aims of the module:
 - introduce the basic ideas of automatic verification
 - familiarise you with key **techniques & algorithms** for verification
 - illustrate **uses & applications** of automatic verification
 - give practical experience in state of the art **verification software**
 - provide a foundation for further study in the area of verification
- Mix of: theory + algorithms + tools
 - no programming, but use of modelling languages
- Prerequisites
 - background in: propositional logic, automata, graph algorithms

Module delivery

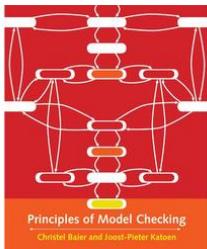
- Lectures:
 - Tue 11–12:
 - Lecture Room 7, Arts Building
 - Thur 12–1:
 - G29, Mechanical and Civil Engineering
- Tutorials (feedback on exercises) (not all weeks):
 - Thur 4–5 (surnames A–L, by default):
 - UG06, Murray Learning Centre
 - Fri 10–11 (surnames M–Z, by default):
 - Lecture Theatre 1, Sports and Exercise Sciences

Assessment

- **Split:**
 - 80% exam (1.5hr, in the summer)
 - 20% continuous assessment
- **Continuous assessment**
 - 4 exercises (1st is formative; 2–4 are assessed: 6%/8%/6%)
 - 1 week for each: due Thurs of weeks 3, 5, 8, 11
 - 1 extra assessed exercise for "extended" version (due week 10)
 - submitted through Canvas
 - solutions worked through in tutorial sessions

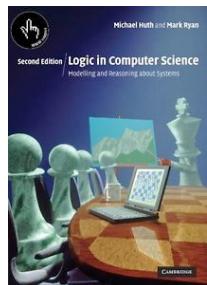
Reference material

- Useful books



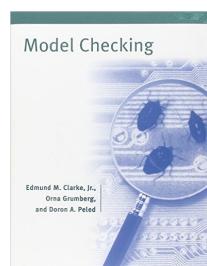
Principles of Model Checking

Christel Baier and Joost-Pieter Katoen [BK08]
The MIT Press, 2008



**Logic in Computer Science:
Modelling and reasoning about systems**

Michael Huth and Mark Ryan
Cambridge University Press, 2004



Model Checking

Edmund M. Clarke, Orna Grumberg, Doron Peled
2nd edn., MIT Press, 2000

- Links to further papers/tutorials will be added to Canvas

Resources

- Canvas
 - <https://canvas.bham.ac.uk/courses/27245>
 - lecture slides/videos, links, assessments, announcements
- Facebook group
 - <https://www.facebook.com/groups/bham.cav.1718>
 - questions, discussion, announcements
- Office hours
 - room 133 (see my door/webpage for times)

2. Modelling Sequential and Parallel Systems



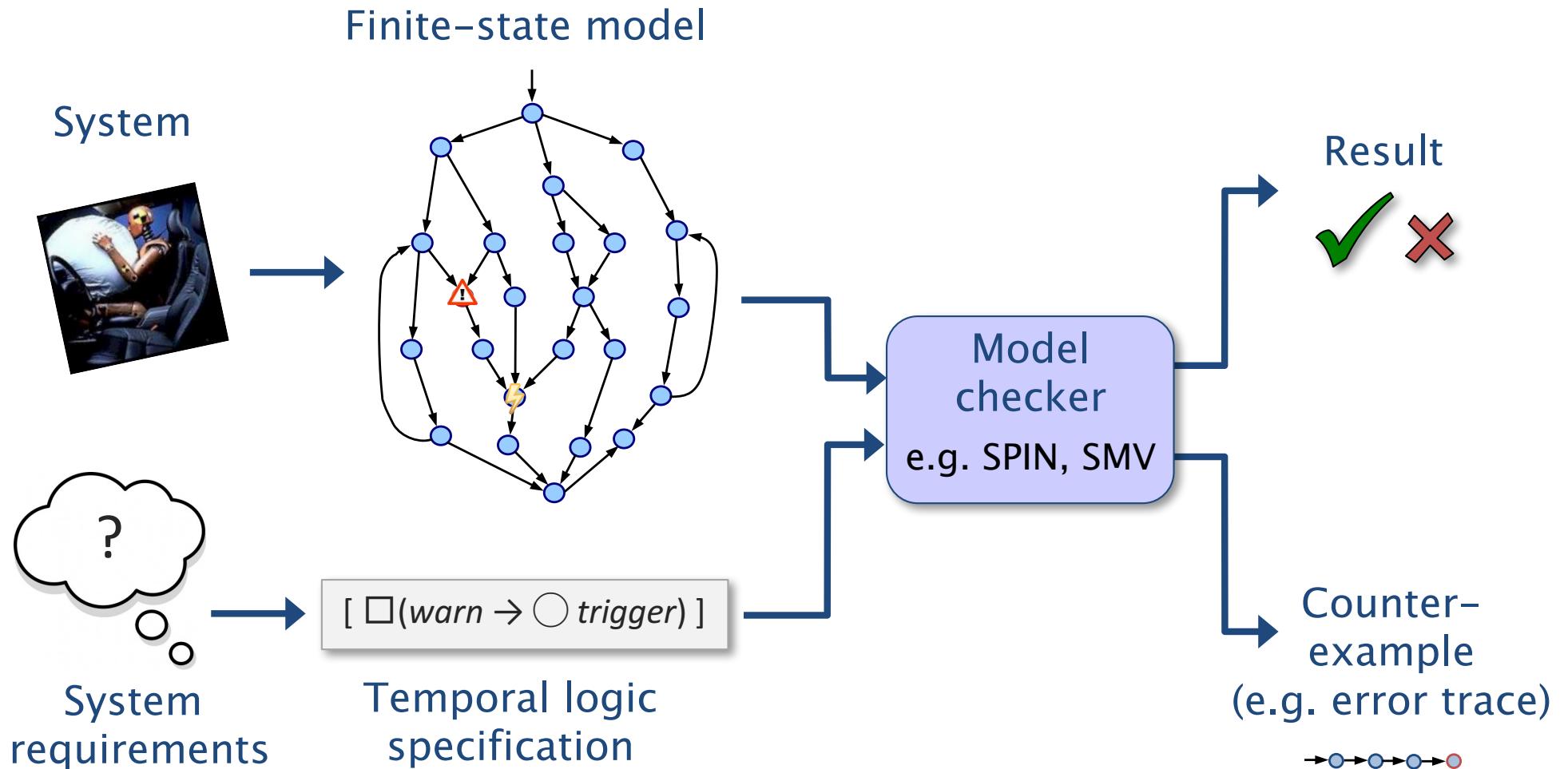
Computer-Aided Verification

Dave Parker

University of Birmingham

2016/17

Model checking



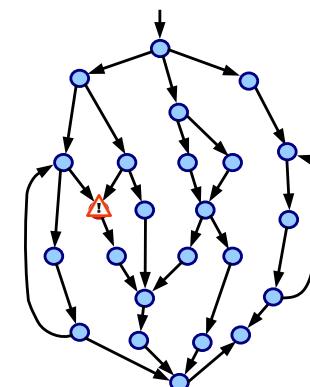
Models

- To verify computerised systems
 - we need precise mathematical **models** of their behaviour
- "All models are wrong, but some are useful" [George Box]
 - results of verification are only as good as the model
- It's all about **abstraction**

```
do {
    if (is_request) {
        size = makerequest(WRQ, name, dp, mode) - 4;
    } else {
        size = readit(file, &dp, 0);
        if (size < 0) {
            nak(errno + 100, NULL);
            break;
        }
        dp->th_opcode = _htons((u_short)DATA);
        dp->th_block  = _htons((u_short)block);
    }

    timeout = 0;
}timeout:
    if (trace)
        tpacket("sent", dp, size + 4);
```

abstraction

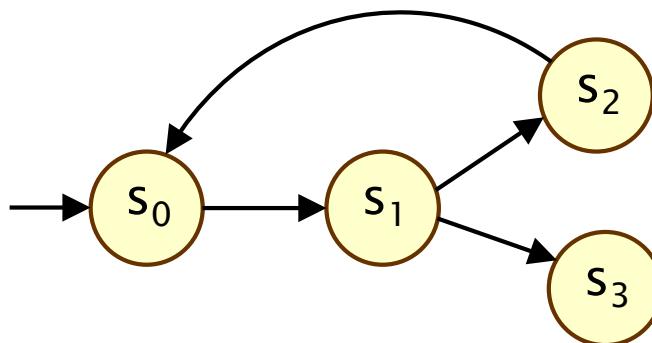


Overview

- Labelled transition systems
 - definitions, notation, ...
- Modelling sequential systems
 - e.g. simple programs
- Nondeterminism
- Parallelism and concurrency
 - interleaving, shared variables, handshaking
 - SOS-style semantics
- See [BK08] chapter 2 (specifically: 2.1–2.1.1, 2.2–2.2.3)

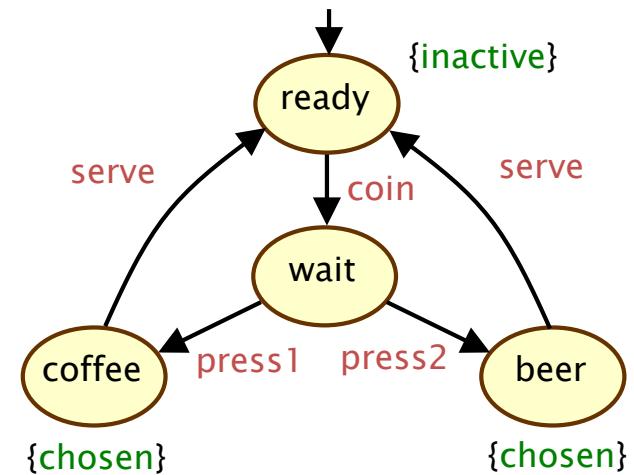
Labelled transition systems

- States = possible configurations of system
 - e.g. valuations of program variables
 - e.g. values of registers in a hardware circuit
- Transitions = possible ways system can evolve
 - e.g. execution of a single program statement
 - e.g. sequential circuit update



Labelled transition systems

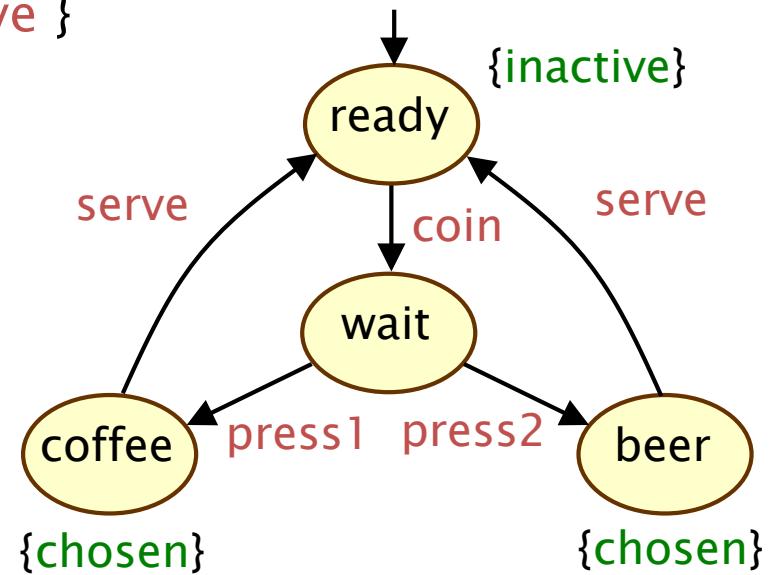
- A labelled transition system (LTS) is:
 - a tuple $(S, \text{Act}, \rightarrow, I, AP, L)$
- where:
 - S is a set of **states** (“state space”)
 - Act is a set of **actions**
 - $\rightarrow \subseteq S \times \text{Act} \times S$ is a **transition relation**
 - $I \subseteq S$ is a set of **initial states**
 - AP is a set of **atomic propositions**
 - $L : S \rightarrow 2^{AP}$ is a **labelling function**
- An LTS is also known as:
 - transition system (TS), state-transition system, Kripke structure
- Essentially: directed graph
 - where nodes/vertices = states, edges = transitions



Example: Drinks machine

- Example LTS $(S, Act, \rightarrow, I, AP, L)$:

- $S = \{ \text{ready}, \text{wait}, \text{coffee}, \text{beer} \}$
- $Act = \{ \text{coin}, \text{press1}, \text{press2}, \text{serve} \}$
- $\rightarrow = \{$
 $(\text{ready}, \text{coin}, \text{wait}),$
 $(\text{wait}, \text{press1}, \text{coffee}),$
 $(\text{wait}, \text{press2}, \text{beer}),$
 $(\text{coffee}, \text{serve}, \text{ready}),$
 $(\text{beer}, \text{serve}, \text{ready})$
}
- $I = \{\text{ready}\}$
- $AP = \{\text{inactive}, \text{chosen}\}$
- $L(\text{ready}) = \{\text{inactive}\},$
 $L(\text{wait}) = \emptyset,$
 $L(\text{coffee}) = L(\text{beer}) = \{\text{chosen}\}$



Labellings and finiteness

- State labelling
 - states are labelled with atomic propositions $a, b, \dots \in AP$
 - represent facts/observations, e.g. "failed", "size $\leq max$ ", ...
- Transition labelling
 - transitions are labelled with actions $\alpha, \beta, \dots \in Act$
 - will be used for communication between components
- Finiteness
 - an LTS is finite if S , Act and AP are finite
 - we will usually (but not always) assume finite LTSs

Transitions

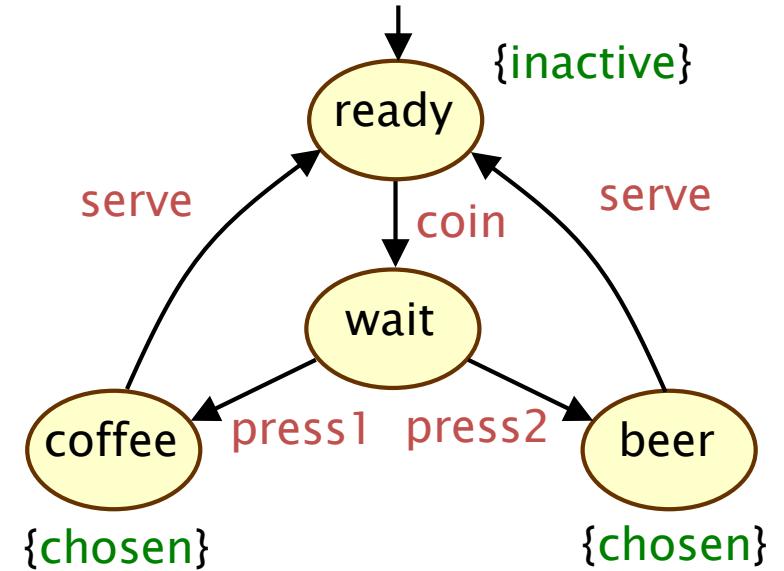
- **Transitions**
 - we write $s -\alpha \rightarrow s'$ if $(s, \alpha, s') \in \rightarrow$
- **Direct successors/predecessors**
 - $\text{Post}(s, \alpha) = \{ s' \in S \mid s -\alpha \rightarrow s' \}$ and $\text{Post}(s) = \bigcup_{\alpha \in \text{Act}} \text{Post}(s, \alpha)$
 - $\text{Pre}(s, \alpha) = \{ s' \in S \mid s' -\alpha \rightarrow s \}$ and $\text{Pre}(s) = \bigcup_{\alpha \in \text{Act}} \text{Pre}(s, \alpha)$
- **Terminal states**
 - state s is **terminal** if $\text{Post}(s) = \emptyset$, i.e., has no outgoing transitions
 - might represent termination of a program
 - often represents erroneous/undesired behaviour
 - e.g. **deadlock** (not all components have terminated)

Paths & reachability

- An **path** (or run, trajectory, execution) is
 - an alternating sequence $s_0\alpha_0s_1\alpha_1s_2\alpha_2\dots$
 - such that $s_i - \alpha_i \rightarrow s_{i+1}$ for all $i \geq 0$ and $s_0 \in I$
- i.e. one possible behaviour/execution of the system modelled
- A **finite path** is
 - a finite prefix of an (infinite) path, ending in a state
 - e.g. $s_0\alpha_0s_1\alpha_1\dots\alpha_{n-1}s_n$
- **Reachability**
 - state s' is **reachable** from s if there is a finite path from s to s'
 - s is a **reachable state** if it is reachable from some $s_0 \in I$
 - **reachability** (the process of determining reachable states) is a fundamental problem/task in model checking

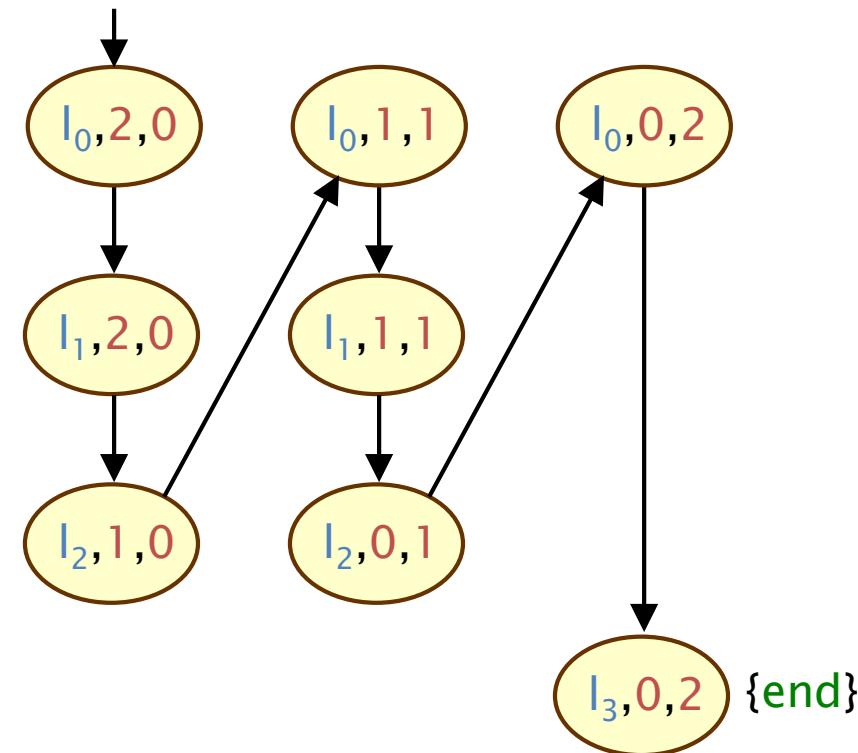
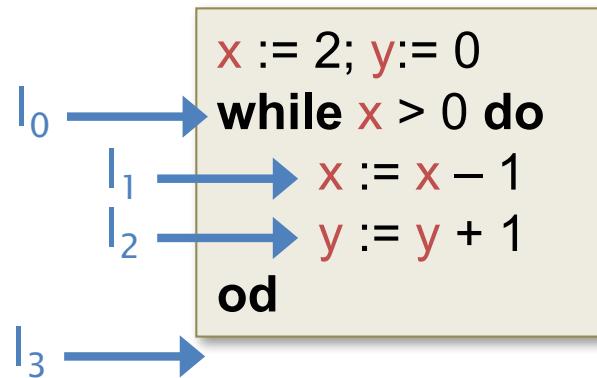
Example

- **Transitions**
 - $\text{Post}(\text{wait}, \text{press1}) = \{\text{coffee}\}$
 - $\text{Post}(\text{wait}) = \{\text{coffee}, \text{beer}\}$
 - $\text{Pre}(\text{ready}) = \{\text{coffee}, \text{beer}\}$
- **A finite path**
 - ready **coin** wait **press1** coffee
- **An infinite path/execution**
 - ready **coin** wait **press1** coffee **serve** ready (**coin** wait **press2** beer **serve** ready) $^\omega$
- **All states are reachable and non-terminal**



Programs as LTSs

- How to model a (sequential) program as an LTS?
 - states are tuples (l_i, x, y) of location & variable values



3. Modelling Sequential and Parallel Systems



Computer-Aided Verification

Dave Parker

University of Birmingham

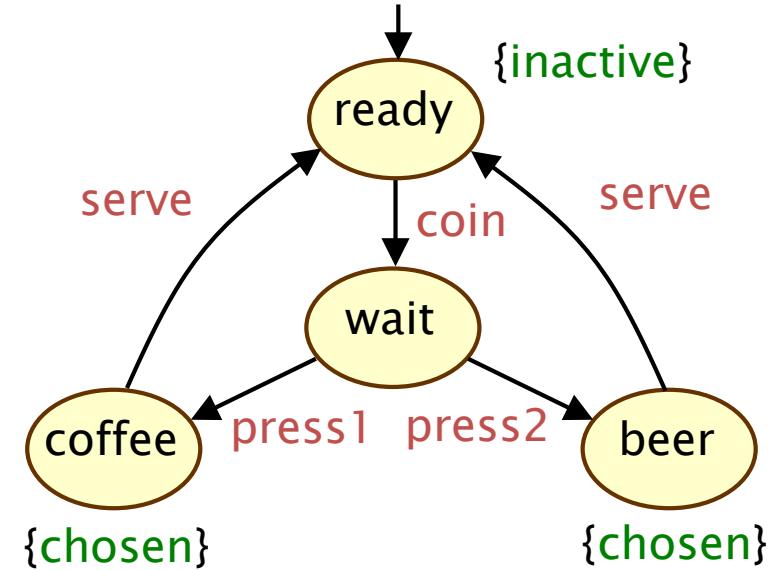
2017/18

Recap

- Formal verification & model checking
 - need precise models of system behaviour over time
 - results of verification only as good as the models used
 - need to be wary of the state space explosion problem
- Labelled transition systems (LTSs)
 - states, transitions & labels
 - states capture all information needed...
 - to determine what happens next
 - and to specify properties to be verified

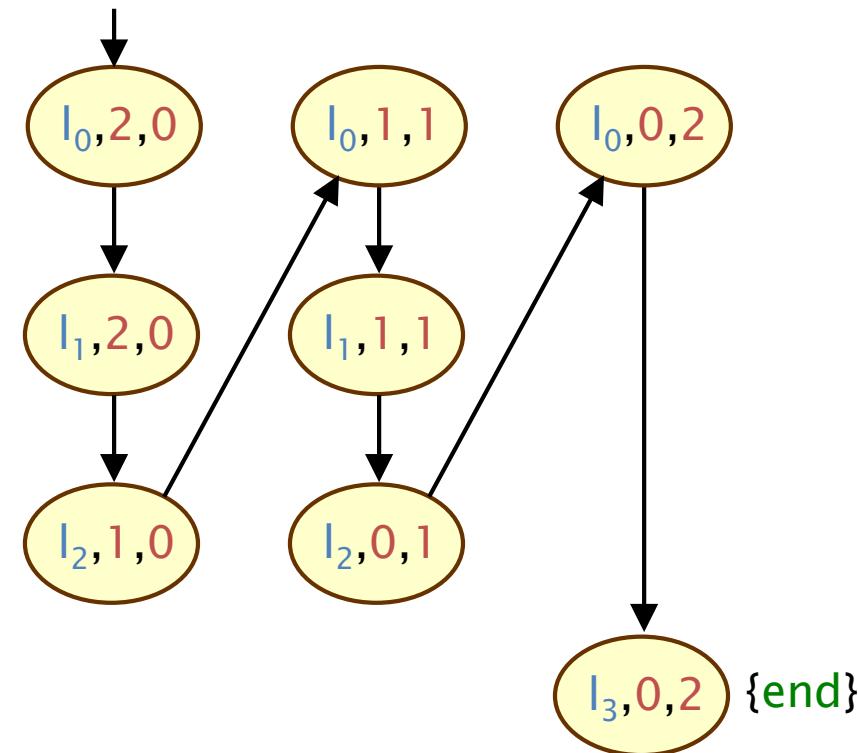
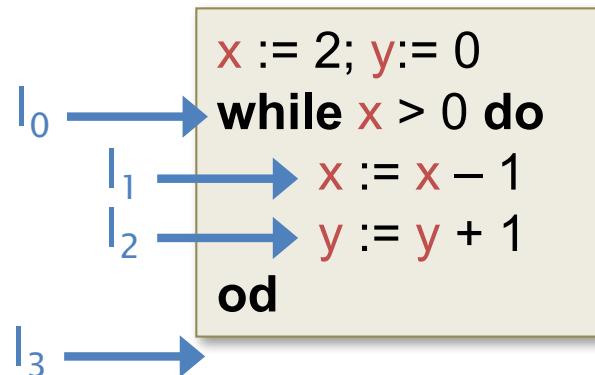
Labelled transition systems (LTSs)

- **Transitions**
 - $\text{Post}(\text{wait}, \text{press1}) = \{\text{coffee}\}$
 - $\text{Post}(\text{wait}) = \{\text{coffee}, \text{beer}\}$
 - $\text{Pre}(\text{ready}) = \{\text{coffee}, \text{beer}\}$
- **A finite path**
 - ready **coin** wait **press1** coffee
- **An infinite path/execution**
 - ready **coin** wait **press1** coffee **serve** ready (**coin** wait **press2** beer **serve** ready) $^\omega$
- **All states are reachable and non-terminal**



Programs as LTSs

- How to model a (sequential) program as an LTS?
 - states are tuples (l_i, x, y) of location & variable values

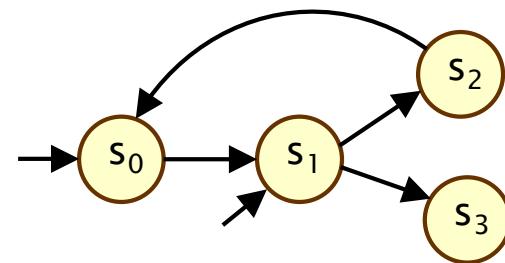


Overview

- Nondeterminism
- Parallelism and concurrency
 - interleaving, shared variables, handshaking
 - SOS-style semantics
 - see [BK08] chapter 2 (specifically: 2.1–2.1.1, 2.2–2.2.3)
- Linear-time properties of LTSs
 - see Chapter 3 of [BK08]

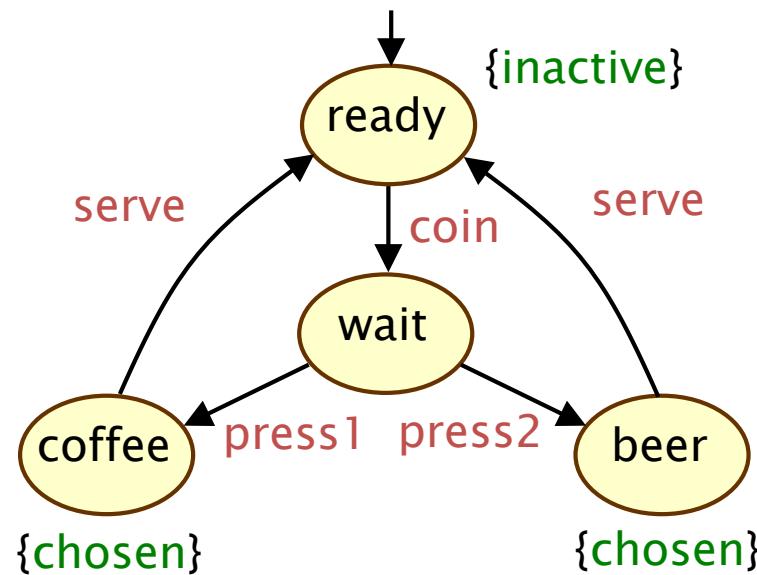
Nondeterminism

- Nondeterminism
 - the outcome of the event is not known in advance
- Nondeterminism in labelled transition systems
 1. choice of action/transition in each state
 2. choice of initial state
- Uses of nondeterminism in system modelling
 1. unknown system environment (e.g. reactive systems, user input)
 2. abstraction (omitting detail), underspecification
 3. concurrency (parallelism)



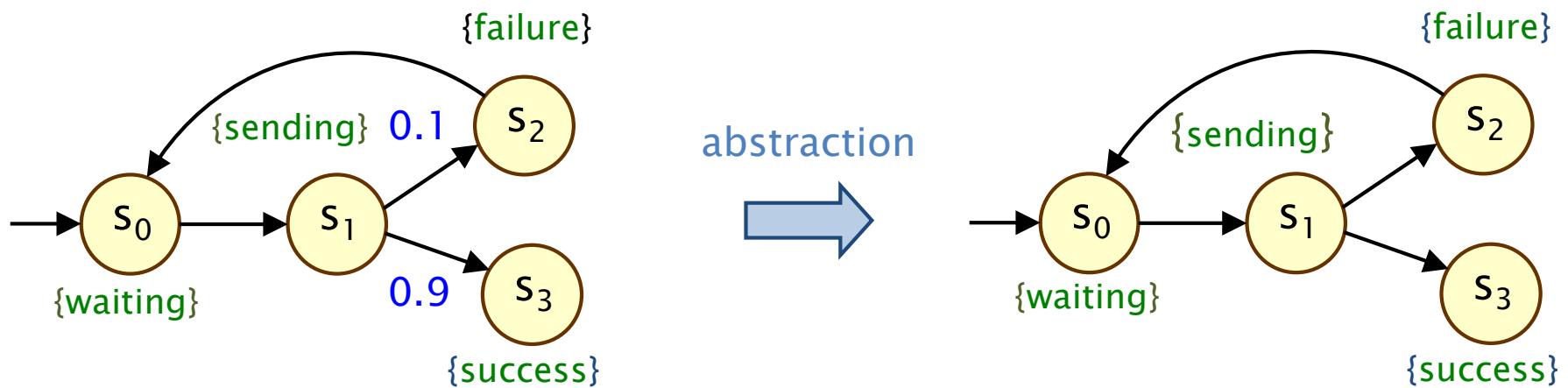
Unknown system environment

- E.g., interaction with human, controller, ...



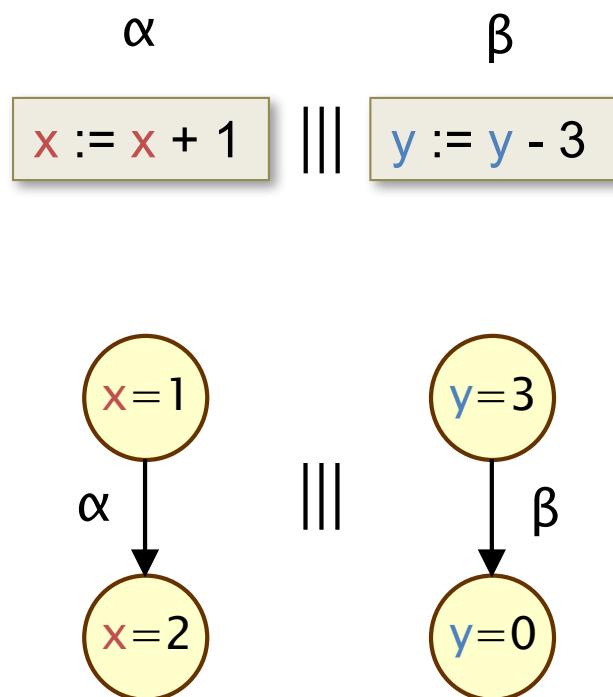
Abstraction, underspecification

- A simple model of message transmission
 - model abstracted – probability of success is unknown/ignored



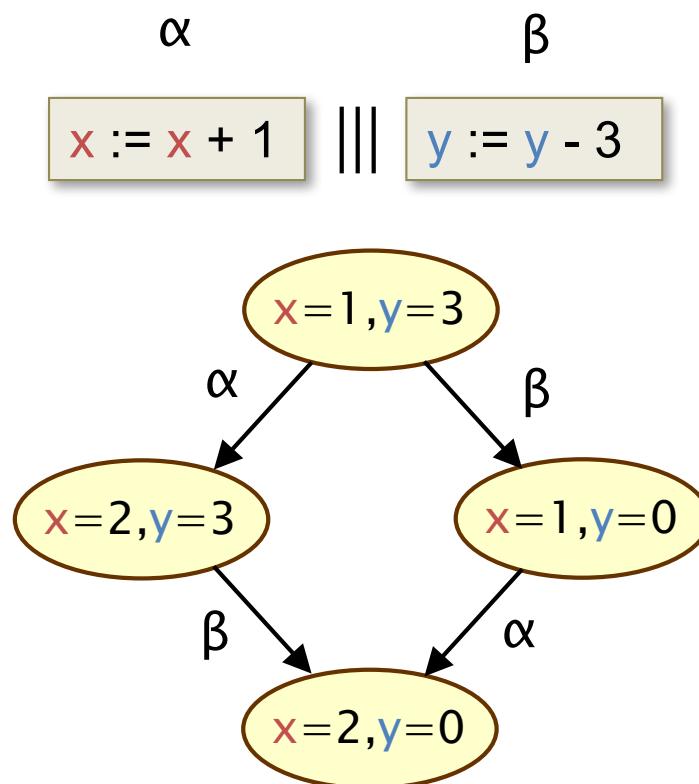
Concurrency: Parallel composition

- Consider two asynchronous components in parallel
 - parallel execution of independent actions
 - nondeterminism models interleaving (e.g. of scheduler)



Concurrency: Parallel composition

- Consider two asynchronous components in parallel
 - parallel execution of independent actions
 - nondeterminism models interleaving (e.g. of scheduler)

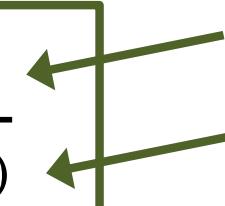


Composition (interleaving) of LTSs

- A formal definition... Let M_1 and M_2 be two LTSs:
 - $M_i = (S_i, \text{Act}_i, \rightarrow_i, I_i, AP_i, L_i)$ for $i=1,2$
- Then we define their interleaving $M_1 \parallel M_2$ as the LTS:
 - $M_1 \parallel M_2 = (S_1 \times S_2, \text{Act}_1 \cup \text{Act}_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$
- where:
 - $L((s_1, s_2)) = L(s_1) \cup L(s_2)$ for any $s_1 \in S_1, s_2 \in S_2$
- and \rightarrow is defined as follows:

$$\frac{s_1 - \alpha \rightarrow_1 s_1'}{(s_1, s_2) - \alpha \rightarrow (s_1', s_2)}$$
$$\frac{s_2 - \alpha \rightarrow_2 s_2'}{(s_1, s_2) - \alpha \rightarrow (s_1, s_2')}$$

premise
conclusion



SOS rules (structured operational semantics)

Concurrency: Shared variables

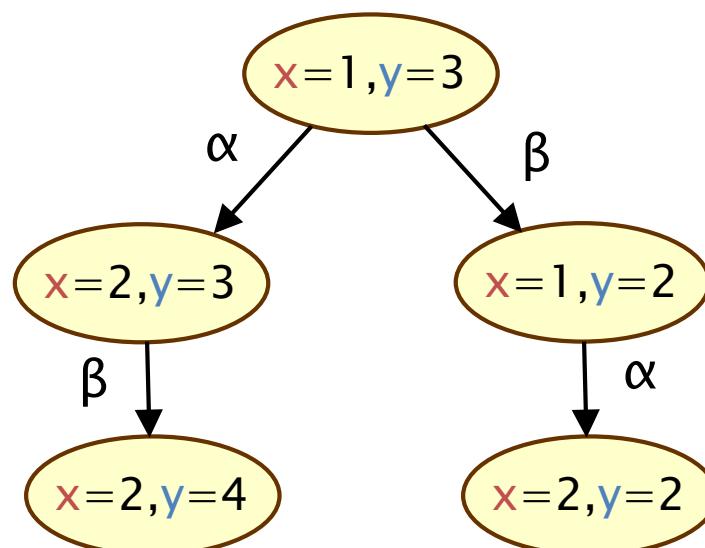
- Consider again two components in parallel
 - but with shared access to some variable
 - nondeterminism models **competition** between components

α β

$x := x + 1$

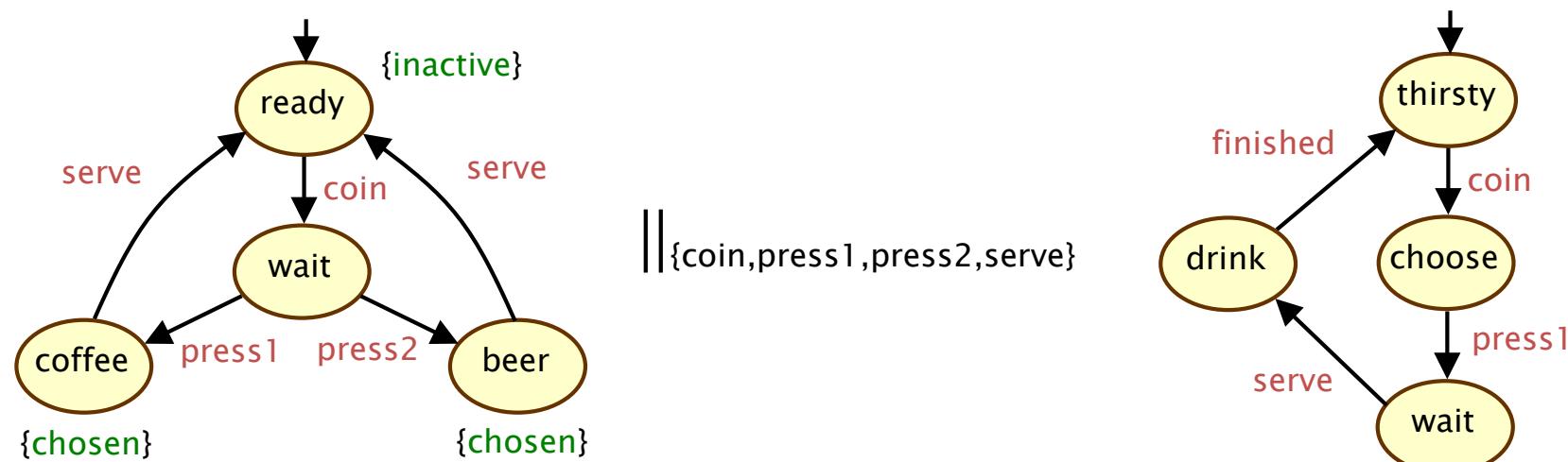
 \parallel

$y := 2 * x$

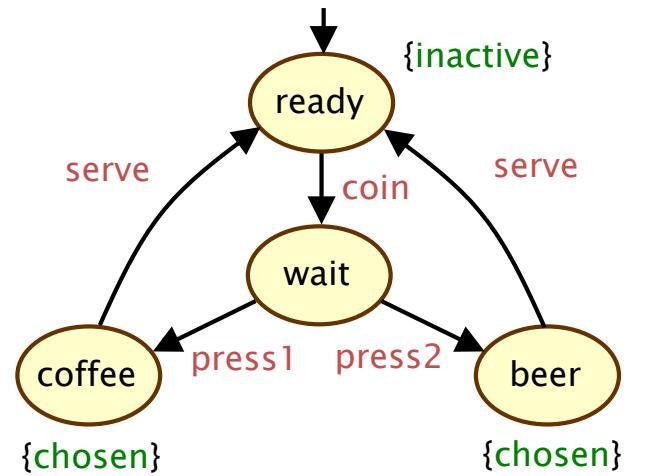


Concurrency: Synchronisation

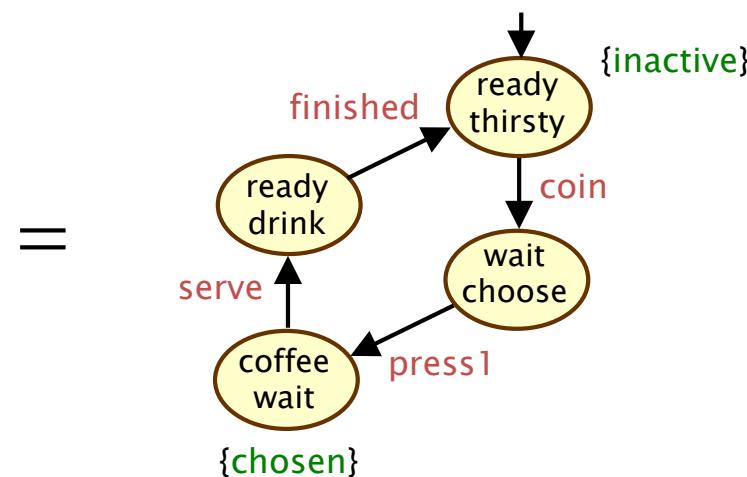
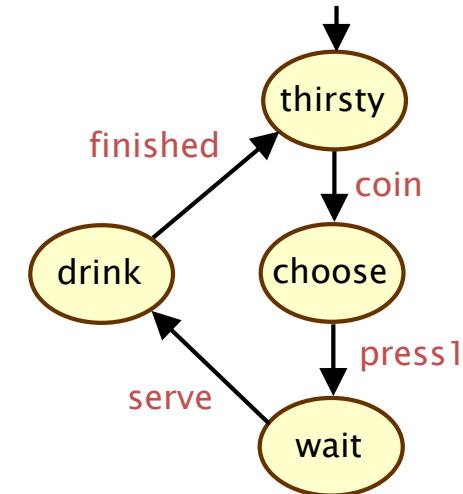
- **Synchronisation** between parallel components: **handshaking**
 - parallel composition $M_1 \parallel_H M_2$ of LTSs M_1 and M_2
 - for a set $H \subseteq \text{Act}$ of handshake actions
 - synchronise (only) on actions in H
 - (like synchronous message passing but message itself is ignored)
- Example



Example



$\parallel \{ \text{coin}, \text{press1}, \text{press2}, \text{serve} \}$



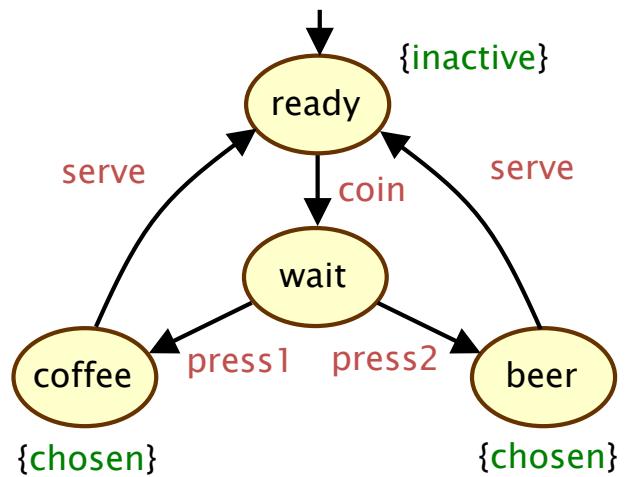
Concurrency: Synchronisation

- A formal definition of handshaking:
- $M_1 \parallel_H M_2$ is defined as for $M_1 \parallel M_2$
 - $M_1 \parallel_H M_2 = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$
- except that \rightarrow is defined as follows:

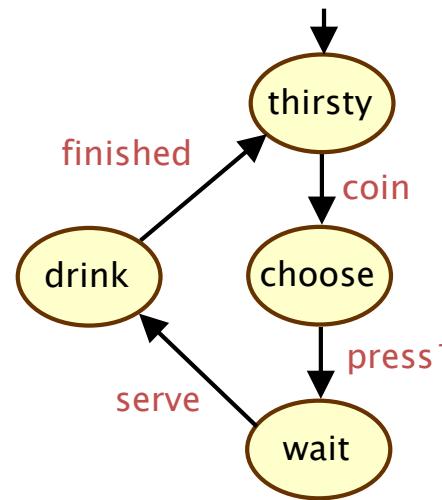
$$\frac{s_1 -\alpha \rightarrow_1 s_1' \wedge s_2 -\alpha \rightarrow_2 s_2'}{(s_1, s_2) -\alpha \rightarrow (s_1', s_2')} \quad \alpha \in H$$
$$\frac{s_1 -\alpha \rightarrow_1 s_1'}{(s_1, s_2) -\alpha \rightarrow (s_1', s_2)} \quad \alpha \notin H$$
$$\frac{s_2 -\alpha \rightarrow_2 s_2'}{(s_1, s_2) -\alpha \rightarrow (s_1, s_2')} \quad \alpha \notin H$$

Example

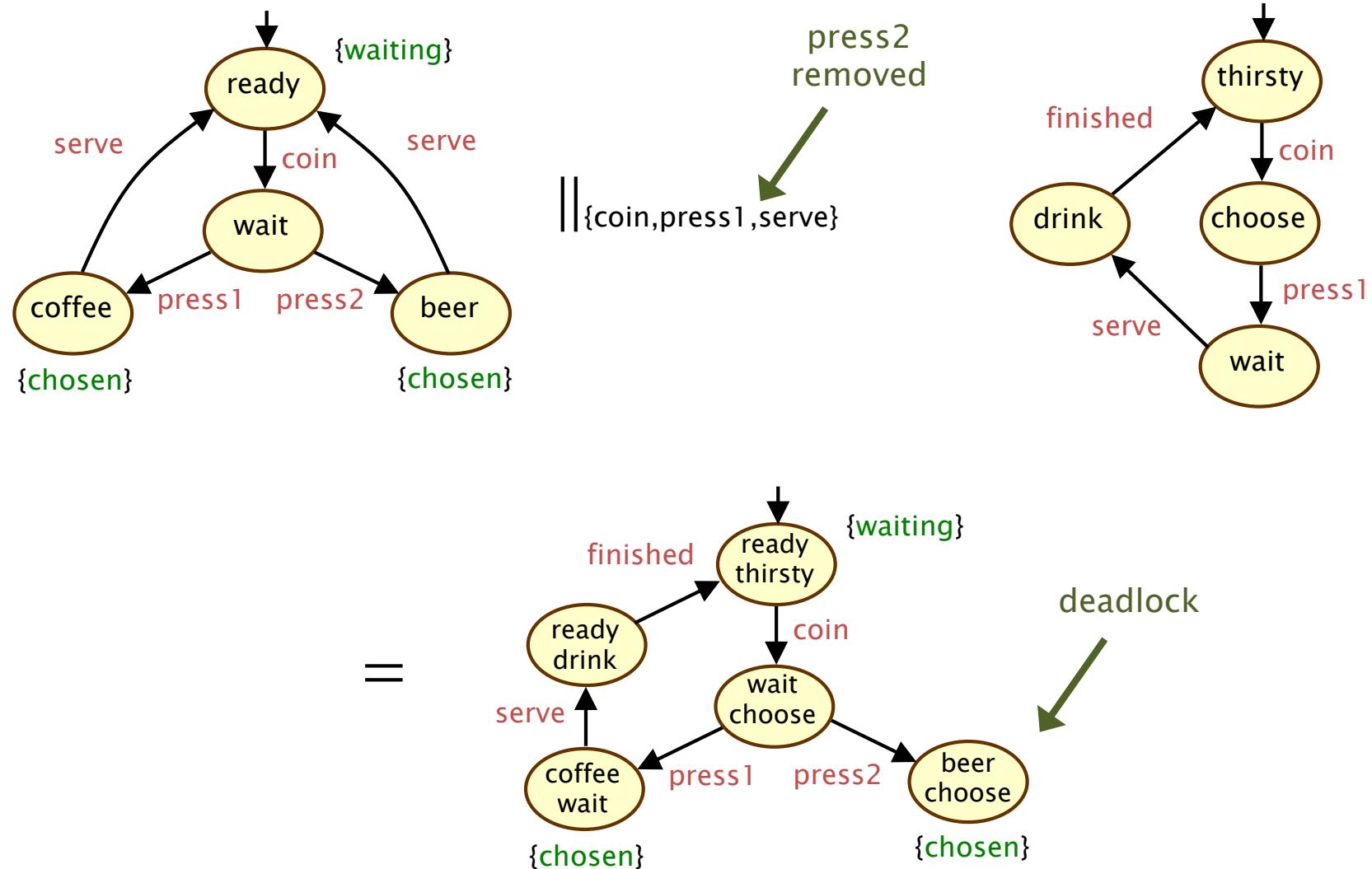
- What about this one?
 - (only the handshake set changed)



|| {coin,press1,serve}



Another example



Recall the earlier example (Lec 1)

- Does variable x always remain in the range $\{0, 1, \dots, 200\}$?

```
process Inc = while true do if x < 200 then x := x + 1 od
```

```
process Dec = while true do if x > 0 then x := x - 1 od
```

```
process Reset = while true do if x = 200 then x := 0 od
```

```
.....  
pan: assertion violated ((x >= 0) && (x <= 200)) (at depth 1802)  
pan: wrote pan_in.trail
```

```
.....  
State-vector 32 byte, depth reached 3598, errors: 1  
12609 states, stored
```

Parallel composition – Key ideas

- Nondeterminism models interleaving of parallel components
 - i.e. unknown execution order (or unknown scheduling)
- Parallel composition of two LTSs
 - resulting LTS has product state space $S_1 \times S_2$
 - i.e., each state has the state for both (or all) components

4. Linear-Time Properties



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

Announcements

- **Continuous assessment**
 - reminder: 4 assignments (5 for “extended” version)
 - due Thurs of weeks 3, 5, 8, 11 (and week 10 for extra one)
- **Assignment 1 (models and properties)**
 - formative; out now; due 12 noon Thur 25 Jan
 - submitted through Canvas
 - solutions worked through in tutorial sessions
- **Next week**
 - Thur lecture is moved to the tutorial slot:
 - Fri 10am (SportEx Lecture Theatre 1)

Recap: Modelling

- Nondeterminism
 - multiple possible behaviours of system being modelled
 - uses: unknown environments/inputs, abstraction, concurrency
- Parallel composition – key ideas:
- Nondeterminism models interleaving of parallel components
 - i.e., unknown execution order (or unknown scheduling)
- Parallel composition requires states of both components
 - i.e., resulting LTS has product state space $S_1 \times S_2$

Today

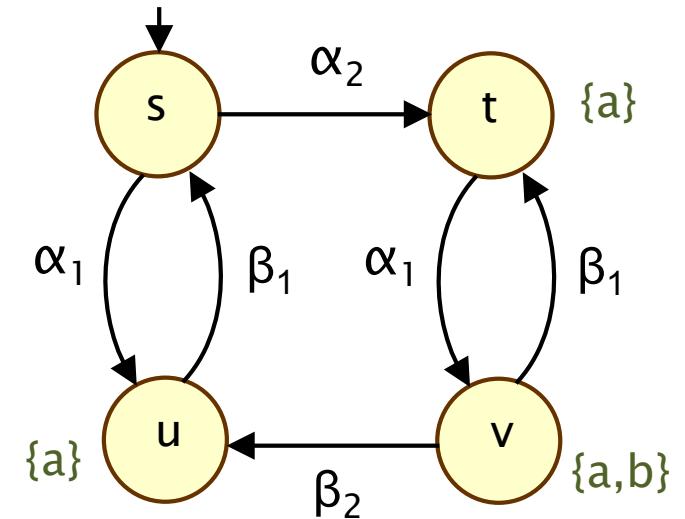
- Linear-time properties
 - formal definition
 - paths, traces, satisfaction
- Important classes of properties
 - invariants
 - safety
 - liveness
- See [BK08] chapter 3 (specifically: 3.2–3.4)

Some assumptions

- We will assume LTSs are finite
 - since we start to consider algorithms to check properties
- We assume no deadlocks
 - i.e. LTSs have no terminal states
 - and so all maximal paths are infinite
 - (we can easily check for deadlocks and "repair" them)

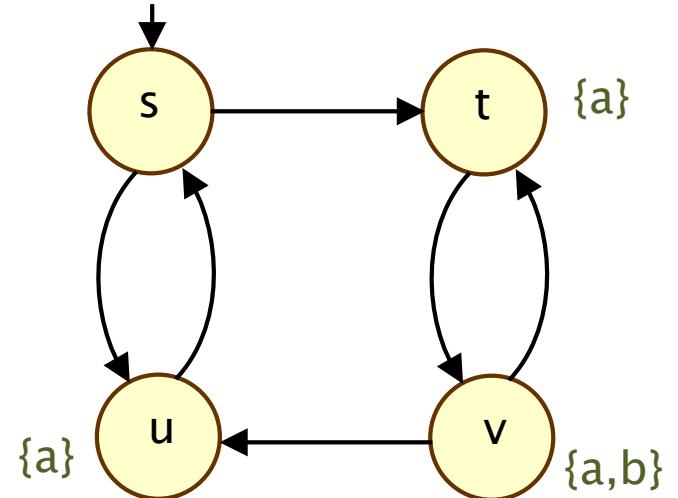
LTS labels

- Recall:
 - **state** labels (atomic prop.s) are used for facts/observations
 - **transition** labels (actions) primarily for interaction/composition



LTS labels

- Recall:
 - state labels (atomic prop.s) are used for facts/observations
 - transition labels (actions) primarily for interaction/composition



- So:
 1. properties are formally expressed using atomic propositions
 2. technically, can work on underlying graph of an LTS
- Paths
 - are now of the form $\pi = s \ t \ v \ t \ v \ u \dots$
 - i.e. we ignore actions

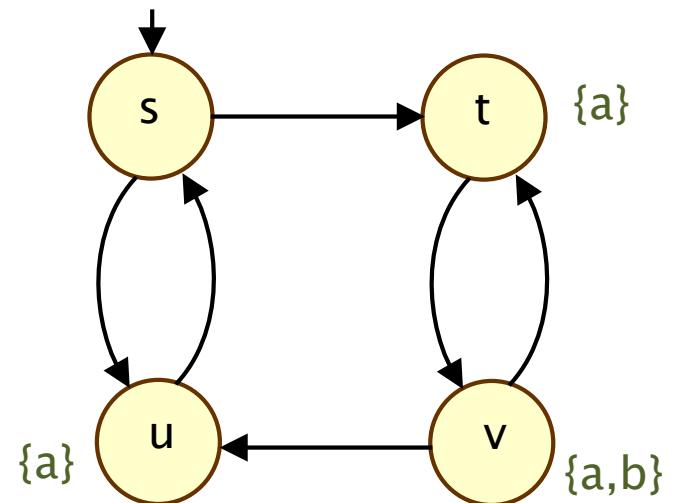
Traces

- Recall:

- an LTS is a tuple $M = (S, \text{Act}, \rightarrow, I, AP, L)$
- with a labelling function $L : S \rightarrow 2^{AP}$

- Example:

- $AP = \{a, b\}$
- $2^{AP} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
- e.g. $L(v) = \{a, b\}$



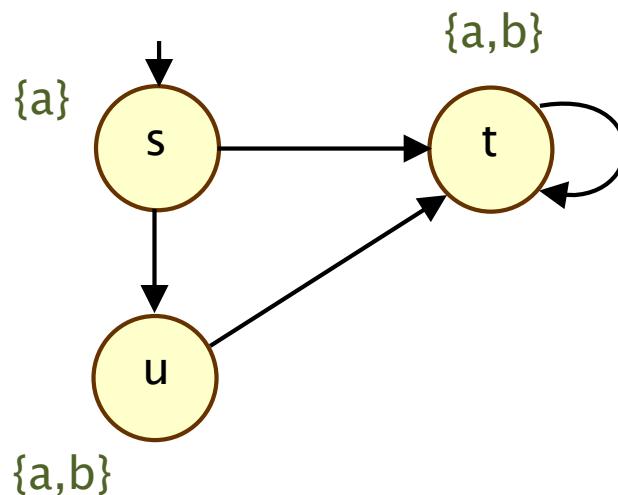
- Traces

- the sequences of (sets of) atomic propositions true in each state
- the trace of path $\pi = s_0 s_1 s_2 s_3 \dots$
- is $\text{trace}(\pi) = L(s_0) L(s_1) L(s_2) L(s_3) \dots$
- e.g. $\text{trace}(s \ t \ v \ t \ v \ u \dots) = \emptyset \ \{a\} \ \{a,b\} \ \{a\} \ \{a,b\} \ \{a\} \ \dots$

Notation: Paths and traces

- For LTS $M = (S, \text{Act}, \rightarrow, I, AP, L)$:
 - $\text{Paths}(M)$ is the set of all paths starting from an initial state in I
 - $\text{Traces}(M)$ is the set for all traces of those paths

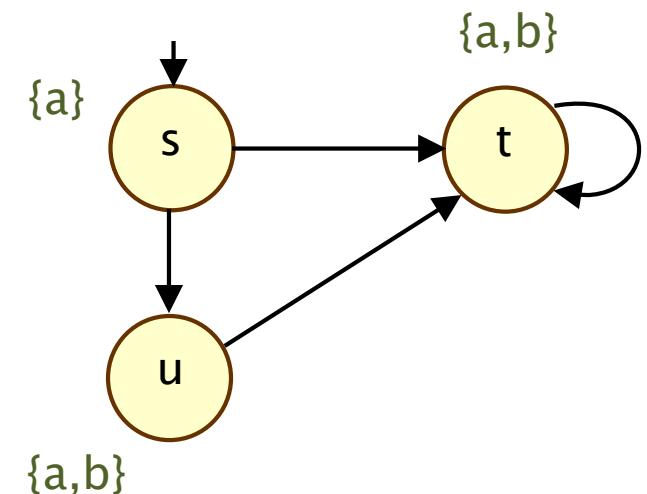
- Example



- $\text{Paths}(M) = \{ s u t^\omega, s t^\omega \}$
- $\text{Traces}(M) = \{ \{a\} \{a,b\}^\omega \}$

Linear-time properties

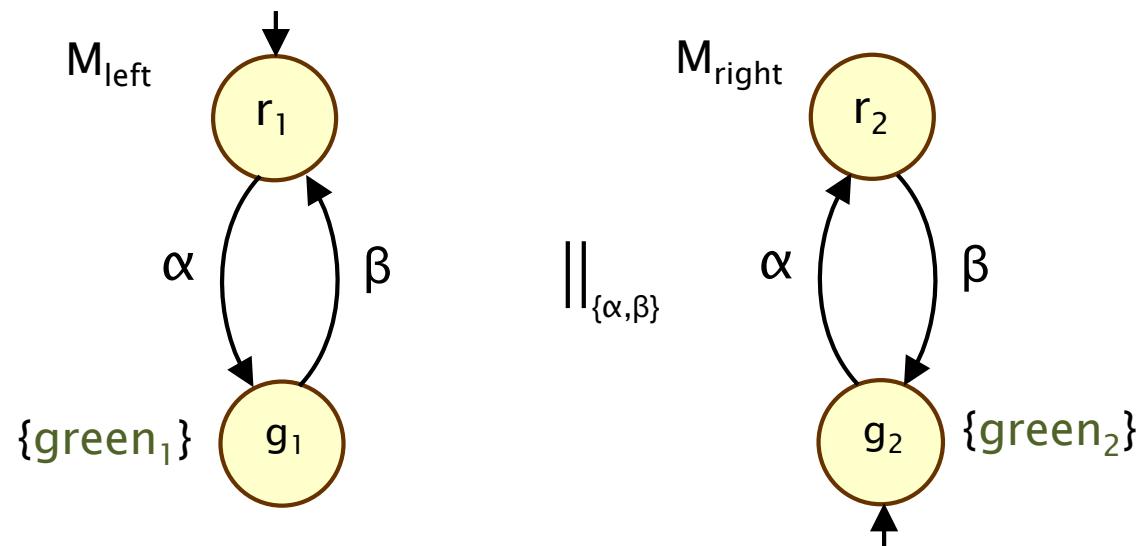
- A linear-time property is
 - (informally) a set of traces that an LTS is allowed to exhibit
 - e.g. "b appears at most once", "a and b never appear together"
 - (formally) a subset of $(2^{\text{AP}})^\omega$, i.e., a language of infinite words
- Satisfaction: $M \models P$
 - of a property $P \subseteq (2^{\text{AP}})^\omega$ by an LTS M
 - we say " M satisfies P ", or " P is true in M "
 - defined as: $M \models P \Leftrightarrow \text{Traces}(M) \subseteq P$
- Note:
 - properties are not tied to a particular model
 - we sometimes specify the complement of P ("good" vs. "bad")



Example

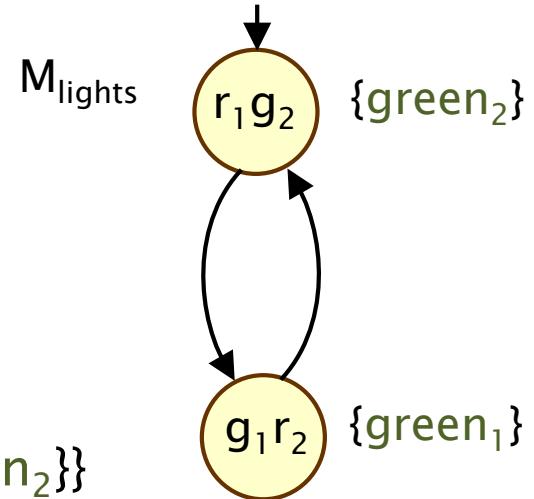
- Example: a pair of traffic lights

- $M_{\text{lights}} = M_{\text{left}} \parallel_{\{\alpha, \beta\}} M_{\text{right}}$



Example

- Example: a pair of traffic lights
 - $M_{\text{lights}} = M_{\text{left}} \sqcup_{\{\alpha, \beta\}} M_{\text{right}}$
- Labels
 - $AP = \{\text{green}_1, \text{green}_2\}$
 - $2^{AP} = \{\emptyset, \{\text{green}_1\}, \{\text{green}_2\}, \{\text{green}_1, \text{green}_2\}\}$
 - M_{lights} exhibits a single trace
- How do we define this property?
 - P: "the traffic lights never both show green simultaneously"
 - $P = \{ \{\text{green}_2\} \{\text{green}_1\} \{\text{green}_2\} \{\text{green}_1\} \dots \} ?$
 - no, because e.g. $\{\text{green}_2\} \{\text{green}_2\} \{\text{green}_2\} \dots$ is in P
 - properties are not tied to specific models
 - $P = \{ A_0 A_1 A_2 \dots \in (2^{AP})^\omega \mid A_j \neq \{\text{green}_1, \text{green}_2\} \text{ for all } j \geq 0 \}$

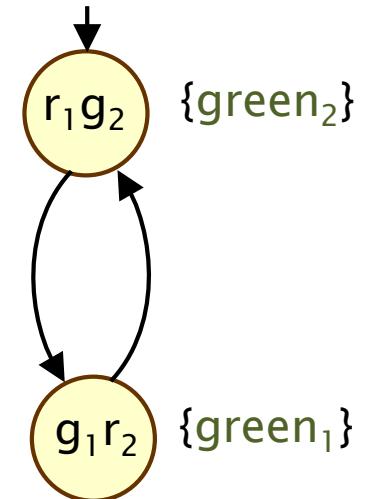


Classes of linear-time properties

- We identify several useful classes of property
 - important consequences for what properties we can express
 - and what algorithms/techniques are required to verify them
- Defined informally...
- Invariants
 - "something good is always true"
- Safety properties
 - "nothing bad happens"
- Liveness properties
 - "something good happens in the long run"

Invariants

- Informally:
 - a condition Φ about states must always be true
- Formally:
 - $P_{\text{inv}} \subseteq (2^{\text{AP}})^\omega$ is an **invariant** if there is a propositional logic formula Φ such that:
 - $P_{\text{inv}} = \{ A_0 A_1 A_2 \dots \in (2^{\text{AP}})^\omega \mid A_j \models \Phi \text{ for all } j \geq 0 \}$
- Examples:
 - P_1 = "one of the green lights is always on"
 - Φ_1 = $\text{green}_1 \vee \text{green}_2$
 - P_2 = "the traffic lights never both show green simultaneously"
 - Φ_2 = $\neg(\text{green}_1 \wedge \text{green}_2)$



Checking invariants

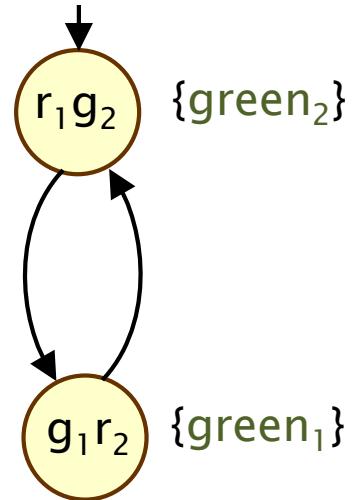
- Invariants:
 - checking invariants can done via reachability
 - $L(s) \models \Phi$ for all states s on all paths of the LTS
 - $L(s) \models \Phi$ for all reachable states s of the LTS
- Since we assume (for now) LTSs are finite
 - standard graph traversal, e.g. depth-first/breadth-first search
 - identify all reachable states s and check that $L(s) \models \Phi$
- Improvements
 - stop as soon as a violating state is found (i.e. $L(s) \not\models \Phi$)
 - use breadth-first search with a stack and return a path to the violating state

Safety properties

- Informally:
 - defined in terms of “bad” events, e.g. “a failure does not occur”
 - “bad” events happen in finite time, and cannot be recovered from
- More precisely
 - P_{safe} is a **safety property** if any (infinite) word where P_{safe} does not hold has a bad prefix
 - a **bad prefix** is a finite prefix σ' containing the bad event, such that no infinite path beginning with σ' satisfies P_{safe}
- Formally:
 - $P_{\text{safe}} \subseteq (2^{\text{AP}})^\omega$ is a **safety property** if, for all words $\sigma \in (2^{\text{AP}})^\omega \setminus P_{\text{safe}}$, there is a finite prefix σ' of σ such that:
 - $P_{\text{safe}} \cap \{ \sigma'' \in (2^{\text{AP}})^\omega \mid \sigma' \text{ is a prefix of } \sigma'' \} = \emptyset$

Examples

- All invariants are safety properties
 - e.g. P_2 = "the traffic lights never both show green simultaneously": $\Phi_2 = \neg(\text{green}_1 \wedge \text{green}_2)$
 - what are the bad prefixes?
 - e.g. $\{\text{green}_2\} \{\text{green}_1, \text{green}_2\}$
 - words of the form $A_0 A_1 \dots A_n$ with $A_i \models \Phi_2$ for all $0 \leq i < n$ and $A_n \not\models \Phi_2$



- But not all safety properties are invariants
 - e.g. P_3 = "green₁ is always preceded by green₂"
 - what are the bad prefixes?
 - e.g. $\emptyset \{\text{green}_1\}$
 - any word where green_1 appears before green_2
 - why is this not an invariant?

Question

- Are these safety properties? (assume $AP = \{\text{green}_1, \text{green}_2\}$)
- And why?
 - "at least one of the traffic lights always shows green"
yes, because it is an invariant, because...
 - " green_1 and green_2 occur in strict alternation"
yes, because...
 - green_2 is eventually true
no, because...
 - green_2 is true infinitely often
no because...
- The last two are *liveness* properties

Liveness properties

- Informally:
 - “something good happens eventually, or in the long run”
 - e.g. “the program always eventually terminates”
- More precisely
 - P_{live} is a **liveness property** if it does not rule out any prefixes
 - any finite word can be extended to an infinite word in P_{live}
- Formally:
 - $P_{\text{live}} \subseteq (2^{\text{AP}})^\omega$ is a **liveness property** if, for all finite words $\sigma \in (2^{\text{AP}})^*$, there exists an infinite word $\sigma' \in (2^{\text{AP}})^\omega$ such that $\sigma \sigma' \in P_{\text{live}}$

Summary

- Paths, traces
 - path: infinite sequence π of states from LTS M
 - trace: infinite word σ over 2^{AP}
- Properties
 - linear-time property = set P of infinite words over 2^{AP}
 - satisfaction: $M \models P$ if all traces of M are in P
- Classes of property
 - invariant: formula Φ is true in all (reachable) states
 - safety property: "nothing bad happens"
 - violating paths have a finite bad prefix
 - liveness: "something good happens in the long run"
 - any finite path can be extended to a satisfying one

Next lecture

- Linear temporal logic
 - see Chapter 5 of [BK08]

5. Temporal Logic



Computer-Aided Verification

Dave Parker

University of Birmingham

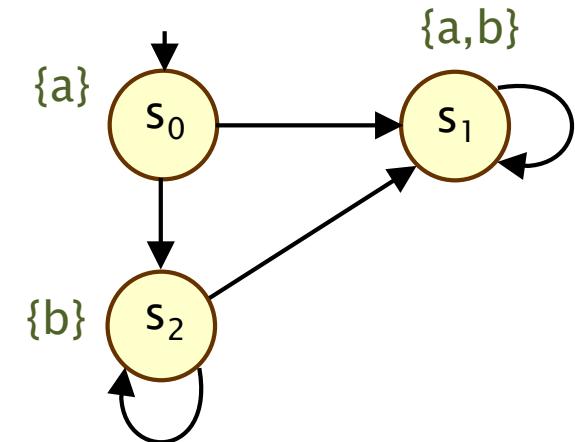
2017/18

This week

- **Next lecture**
 - is moved to the tutorial slot:
 - Fri 10am (SportEx Lecture Theatre 1)
- **Office hour**
 - I am away on Thurs
 - extra office hour today 4.30–5.30

Recap: Traces & properties

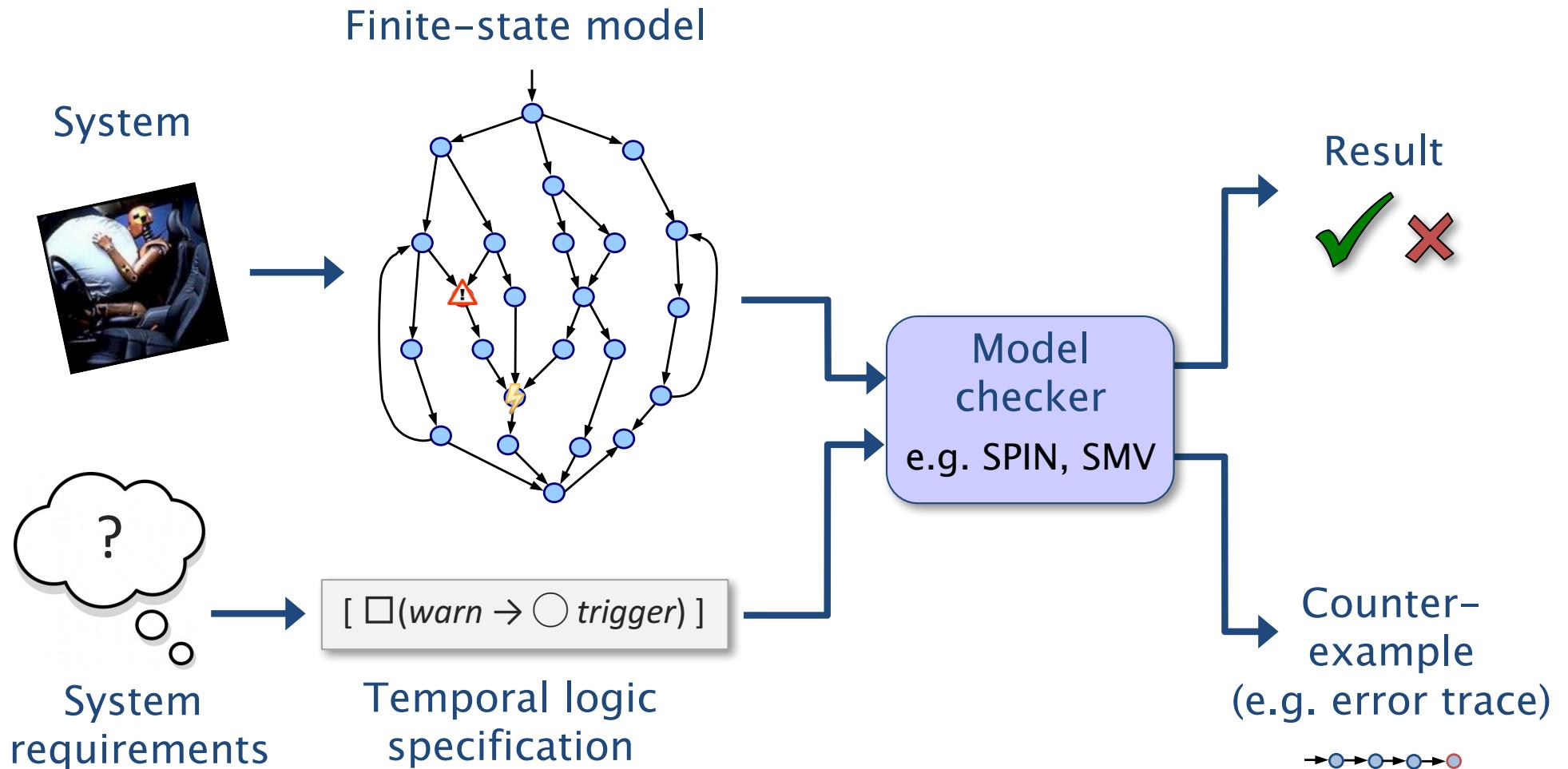
- Paths
 - infinite state sequence $\pi = s_0 s_2 s_2 s_1 s_1 s_1 \dots$
- Traces
 - infinite words over 2^{AP}
 - $\text{trace}(\pi) = \{a\} \{b\} \{b\} \{a,b\} \{a,b\} \{a,b\} \dots$
- Linear-time properties
 - set of allowable (“good”) traces/words $P \subseteq (2^{\text{AP}})^\omega$
 - satisfaction: $M \models P$ if all traces of M are in P
 - e.g. “ a is always eventually followed by b ”
 - $P = \{ A_0 A_1 A_2 \dots \in (2^{\text{AP}})^\omega \mid \text{for all } i \geq 0: a \in A_i \Rightarrow b \in A_j \text{ for some } j \geq i \}$
 - or: linear temporal logic: $\Box(a \rightarrow \Diamond b)$ (see later)



Recap: Properties

- Key classes of property:
- Invariant: formula Φ is true in all (reachable) states
 - can be checked on each state individually
- Safety property: "nothing bad happens"
 - violating paths have a finite bad prefix
- Liveness: "something good happens in the long run"
 - any finite path can be extended to a satisfying one

Model checking



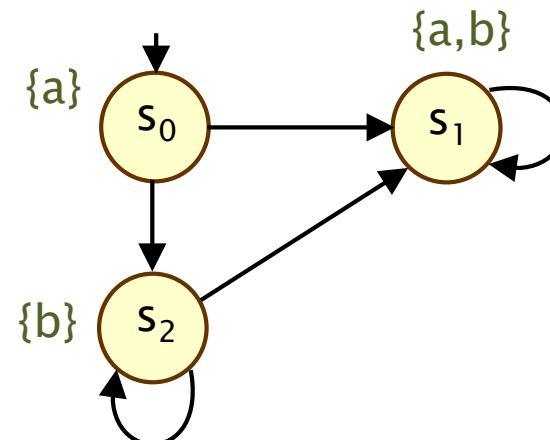
Next (today and next time)

- Propositional logic
- Temporal logic
- Linear temporal logic (LTL)
 - syntax, semantics, examples
- See [BK08] sections 5.1–5.1.4

Propositional logic

- Propositional logic formulas
 - for example: true, a , $\neg a$, $\neg(a \wedge b)$, $a \wedge (b \vee \neg c)$, $a \rightarrow c$
 - where a, b, c are atomic propositions
- Here: use for system observations (state properties)
 - $\text{green}_1 \vee \text{green}_2$, $\text{fail} \wedge \neg \text{alarm}$, $\neg(\text{critical}_1 \wedge \text{critical}_2)$

	a	b	$\neg(a \wedge b)$
\emptyset	F	F	T
$\{b\}$	F	T	T
$\{a\}$	T	F	T
$\{a,b\}$	T	T	F



$$\begin{aligned}
 s_0 &\models a \\
 s_1 &\models a \\
 s_0 &\models \neg(a \wedge b) \\
 s_1 &\not\models \neg(a \wedge b)
 \end{aligned}$$

Propositional logic: Syntax/semantics

- **Syntax** (which formulas are allowed)
- Formulas Φ in propositional logic are defined by the grammar:
 - $\Phi ::= \text{true} \mid \text{false} \mid a \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg \Phi$
 - where $a \in AP$ is an atomic proposition
- **Semantics** (what formulas mean)
- For a valuation $A \in 2^{AP}$ (a set of “true” propositions for a state), $A \models \Phi$ indicates A "satisfies" a propositional formula Φ :
 - $A \models \text{true}$ always
 - $A \models \text{false}$ never
 - $A \models a \iff a \in A$
 - $A \models \Phi_1 \wedge \Phi_2 \iff A \models \Phi_1 \text{ and } A \models \Phi_2$
 - $A \models \Phi_1 \vee \Phi_2 \iff A \models \Phi_1 \text{ or } A \models \Phi_2$
 - $A \models \neg \Phi \iff A \not\models \Phi$

Example:

$$\begin{aligned}\{a\} &\models a \vee b \\ \{a,b\} &\not\models \neg(a \wedge b)\end{aligned}$$

Logical equivalences

- We usually give more minimal grammars, e.g.:
 - $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi$
 - where $a \in AP$ is an atomic proposition
- Standard logical equivalences
 - $\text{false} \equiv \neg\text{true}$ (false)
 - $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$ (disjunction)
 - $\phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$ (implication)
 - $\phi_1 \leftrightarrow \phi_2 \equiv (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$ (equivalence)
 - $\phi_1 \oplus \phi_2 \equiv (\phi_1 \wedge \neg\phi_2) \vee (\neg\phi_1 \wedge \phi_2)$ (exclusive or)

Temporal logic

- Temporal logic
 - extends propositional logic with modal/temporal operators
 - which can refer to the (infinite) behaviour of a system
 - "temporal" – refers to relative ordering of events, not the precise times at which they happen (LTSs are time-abstract)
- Various applications
 - used, e.g. in philosophy, for many years
 - introduced to formal verification by Pnueli in the 70s
 - increased prominence thanks to model checking (early 80s)

Temporal logic

- Temporal logic for property specification in model checking
 - mathematically precise
 - intuitive (mostly!)
 - concise (usually!)
- LTL: Linear Temporal Logic
 - temporal logic for linear-time properties
 - there are alternatives: branching time (see CTL, later)
- Some key temporal operators
 - $\diamond a$ – "a is eventually true"
 - $\square a$ – "a is always true"

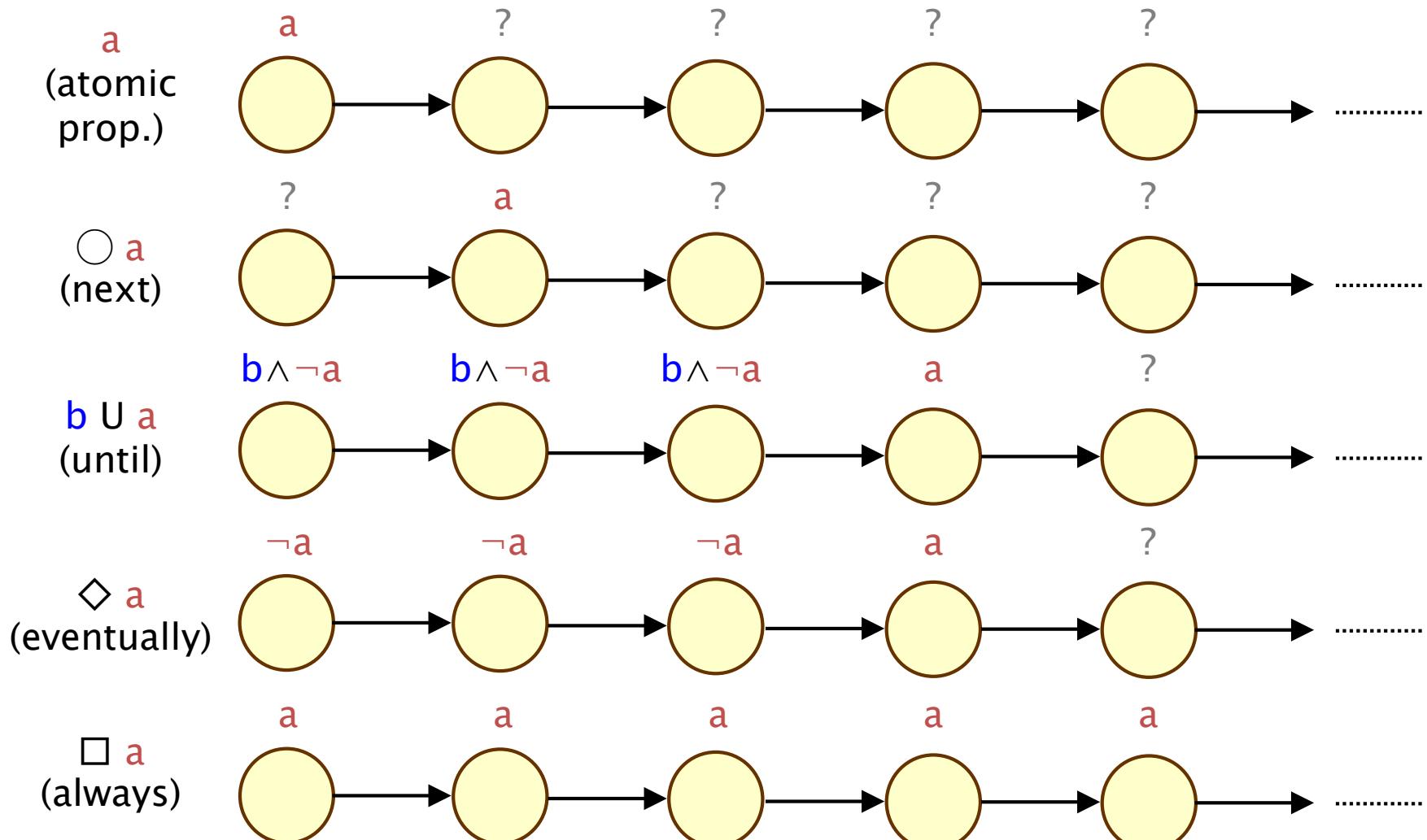
LTL – Syntax

- LTL formulas Ψ are defined by the grammar:
 - $\Psi ::= \text{true} \mid a \mid \Psi \wedge \Psi \mid \neg \Psi \mid \bigcirc \Psi \mid \Psi \cup \Psi$
 - where $a \in AP$ is an atomic proposition
- Temporal operators: "next" (\bigcirc) and "until" (\cup)
 - $\bigcirc \Psi$ means " Ψ is true in the next state"
 - $\Psi_1 \cup \Psi_2$ means " Ψ_2 is true eventually and Ψ_1 is true until then"
- Equivalences (in addition to false, \vee , \rightarrow , \leftrightarrow , \oplus)
 - "eventually Ψ ": $\lozenge \Psi \equiv \text{true} \cup \Psi$
 - "always Ψ ": $\square \Psi \equiv \neg \lozenge(\neg \Psi)$

LTL

- Some simple examples:
- $\square \neg(\text{critical}_1 \wedge \text{critical}_2)$
 - "the processes never enter the critical section simultaneously"
- $\diamond \text{end}$
 - "the program eventually terminates"
- $\neg \text{error} \cup \text{end}$
 - "the program terminates without any errors occurring"
- Alternative styles of syntax
 - $\bigcirc a \equiv \text{X } a$ ("next")
 - $\diamond a \equiv \text{F } a$ ("future", "finally")
 - $\square a \equiv \text{G } a$ ("globally")

LTL – Intuitive semantics



6. Linear Temporal Logic



Computer-Aided Verification

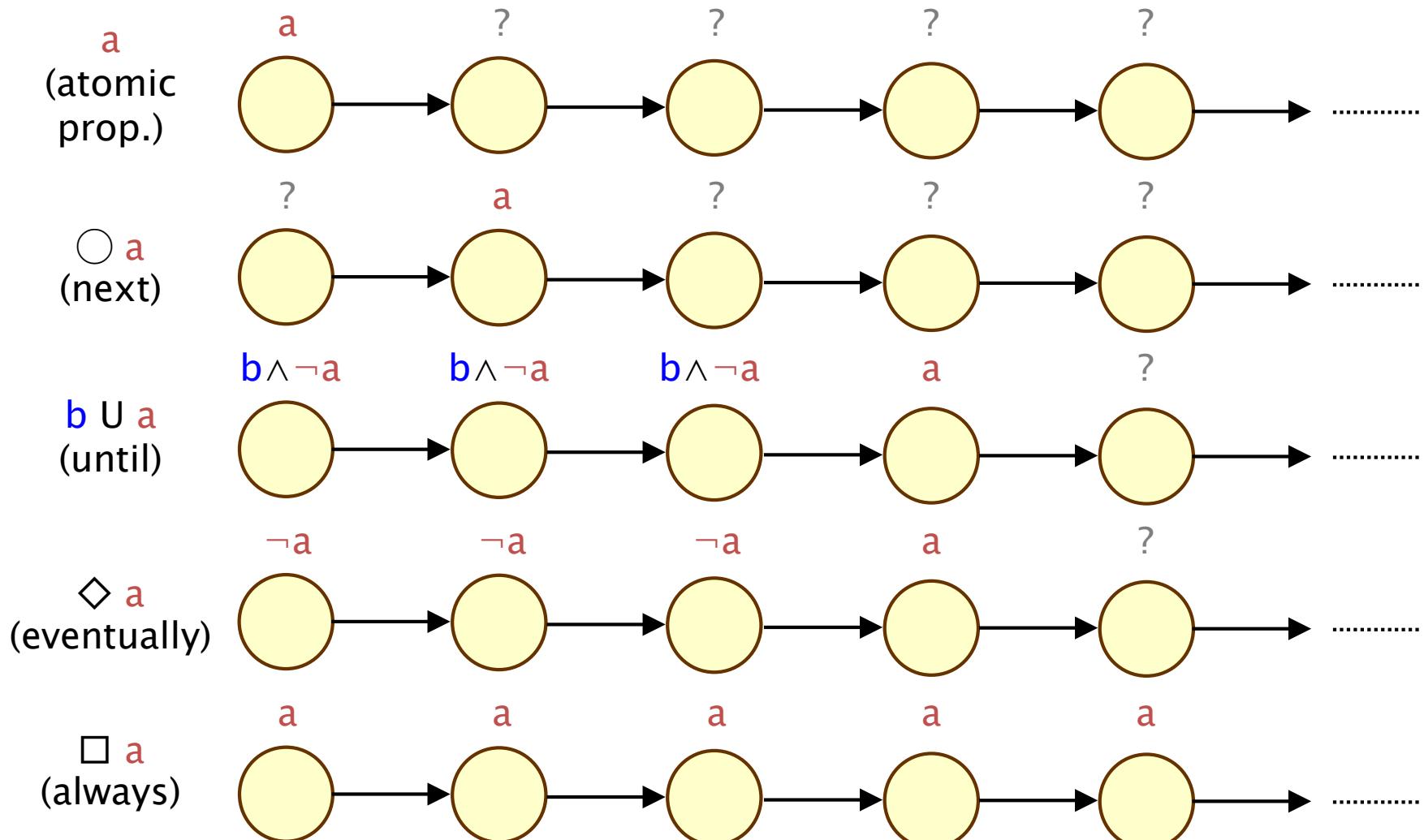
Dave Parker

University of Birmingham
2017/18

Recap: Temporal logic

- Propositional logic
 - syntax, semantics, equivalences (derived operators)
 - a, b (atomic propositions), \wedge (conjunction), \vee (disjunction),
 \neg (negation), \rightarrow (implication), etc.
- Temporal logic
 - precise, unambiguous specification of correctness properties
 - extends propositional logic with temporal operators
 - \bigcirc (next), \bigcup (until), \lozenge (eventually), \square (always)
- Linear temporal logic (LTL)

LTL – Intuitive semantics

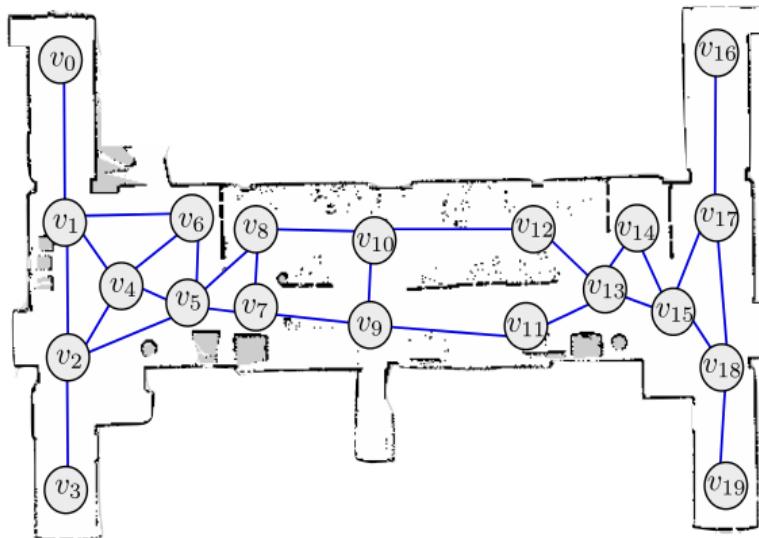


LTL – More properties

- LTL syntax:
 - $\psi ::= \text{true} \mid a \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc\psi \mid \psi \cup \psi \mid \lozenge\psi \mid \square\psi$
 - many more properties formed by combining temporal operators
 - simple examples: $(\lozenge a) \wedge (\lozenge b)$, $\bigcirc\bigcirc a$, $a \wedge \bigcirc\bigcirc a$
- $\square(a \rightarrow \lozenge b)$
 - "b always follows a"
- $\square(a \rightarrow \bigcirc b)$
 - "b always immediately follows a"
- $\square \lozenge a$
 - "a is true infinitely often"
- $\lozenge \square a$
 - "a becomes true and remains true forever"

Other uses of LTL

- Example: robot task specifications
 - $\neg \text{zone}_3 \mathbf{U} (\text{zone}_1 \wedge (\Diamond \text{zone}_4))$
 - visit zone 1 (without passing through zone 3), and then go to zone 4
 - $(\Box \neg \text{zone}_3) \wedge (\Box \Diamond \text{zone}_5)$
 - avoid zone 3 and patrol zone₅ infinitely often



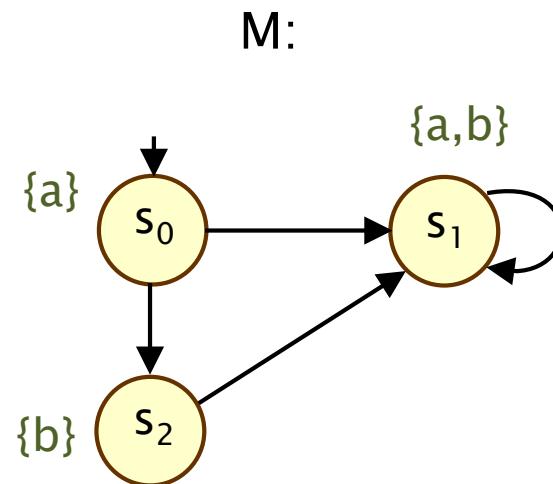
LTL semantics

LTL semantics

- When does an LTS M satisfy an LTL formula ψ ?
 - intuitively, if all paths of M satisfy ψ
- More precisely:
 - if all traces of all paths of M satisfy ψ :
 - $M \models \psi \Leftrightarrow \text{trace}(\pi) \models \psi$ for every $\pi \in \text{Paths}(M)$
- Alternatively (using a linear-time property):
 - $\text{Words}(\psi) = \{ \sigma \in (2^{\text{AP}})^\omega \mid \sigma \models \psi \}$
 - $M \models \psi \Leftrightarrow \text{Traces}(M) \subseteq \text{Words}(\psi)$

Examples

- $M \models \Box (a \vee b) ?$
- $M \models b ?$
- $M \models \bigcirc b ?$
- $M \models \Box \bigcirc b ?$
- $M \models \Box \lozenge \neg a ?$
- $M \models \Box((a \wedge \neg b) \rightarrow \lozenge \neg b) ?$



What can we express in LTL?

- Invariants?
 - yes: $\Box\Phi$, for some propositional formula Φ
 - in fact, *all* invariants can be represented
- Safety properties?
 - yes: e.g. $\Box(\text{receive} \rightarrow \bigcirc \text{ack})$
 - "ack always immediately follows receive"
- Liveness properties?
 - yes: e.g. $\Diamond \text{terminates}$
 - "the program eventually terminates"
 - yes: e.g. $\Box\Diamond \text{ready}$
 - "the server always gets back into a ready state"

Equivalence

- LTL formulae Ψ_1 and Ψ_2 are equivalent, written $\Psi_1 \equiv \Psi_2$ if:

- they are satisfied by exactly the same traces
 - $\sigma \models \Psi_1 \Leftrightarrow \sigma \models \Psi_2$ (for any trace σ)
 - i.e. $\text{Words}(\Psi_1) = \text{Words}(\Psi_2)$



With respect
to some set AP
of propositions

- Or, equivalently:

- if they are satisfied by exactly the same models
 - $M \models \Psi_1 \Leftrightarrow M \models \Psi_2$ (for any LTS M)

- This gives us a notion of expressiveness of LTL

- "expressiveness" = "expressivity" = "expressive power"
 - i.e. which models can LTL distinguish between?

LTL equivalences

- Equivalences
 - shorthand for common formulae, e.g.: $\diamond \psi \equiv \text{true} U \psi$
 - simplifications, e.g.: $\neg\neg p \equiv p$
 - syntax vs. semantics
- Equivalences for: propositional logic + temporal operators
- Temporal operator equivalences:
 - $\Box\psi \equiv \neg\diamond\neg\psi$ (duality)
 - $\Box\Box\psi \equiv \Box\psi$ (idempotency)
 - $\diamond\psi \equiv \psi \vee \bigcirc\diamond\psi$ (expansion law)
 - $\Box(\psi_1 \wedge \psi_2) \equiv \Box\psi_1 \wedge \Box\psi_2$ (distributive law)

Does not add
expressive power
to LTL

Example 1

- Prove (or disprove):

$$\diamond \psi \equiv \psi \vee \bigcirc \diamond \psi \quad ? \qquad \text{Yes}$$

- Can prove directly, using the relevant semantics for LTL:
 - $\sigma \models \psi_1 \vee \psi_2 \Leftrightarrow \sigma \models \psi_1 \text{ or } \sigma \models \psi_2$
 - $\sigma \models \bigcirc \psi \Leftrightarrow \sigma[1\dots] \models \psi$
 - $\sigma \models \psi_1 \cup \psi_2 \Leftrightarrow \exists k \geq 0 \text{ s.t. } \sigma[k\dots] \models \psi_2 \text{ and } \forall i < k \sigma[i\dots] \models \psi_1$

Example 2

- Prove (or disprove):

$$\neg(\Box a \rightarrow \Diamond b) \equiv \Box a \wedge \Box \neg b \quad ? \quad \text{Yes}$$

- Can prove by reusing simpler known equivalences

- $\psi_1 \rightarrow \psi_2 \equiv \neg\psi_1 \vee \psi_2$
- $\Box\psi \equiv \neg\Diamond\neg\psi$
- etc.

Example 3

- Prove (or disprove):

$$\Box\Diamond a \wedge \Box\Diamond b \equiv \Box\Diamond(a \wedge b) \quad ? \quad \text{No}$$

- Just need to provide a single trace as a counterexample
 - e.g. $\{a\} \{b\} \{a\} \{b\} \dots$
 - (which is satisfied by the left formula only)

LTL & Negation

- Are these statements equivalent? (for trace σ and LTL formula Ψ)
 - $\sigma \models \neg\Psi$
 - $\sigma \not\models \Psi$
- Yes
 - in fact, this is just the semantics of LTL
- Are these statements equivalent? (for LTS M and LTL formula Ψ)
 - $M \models \neg\Psi$
 - $M \not\models \Psi$
- No:
 - $M \models \neg\Psi$ means no trace satisfies Ψ
 - $M \not\models \Psi$ means it is not true that all traces satisfy Ψ
 - i.e. there exists some trace that does not satisfy Ψ

Existential properties

- Can we verify this, using LTL?
 - "there exists an execution that reaches program location l_2 "
- Yes: $M \models \Box \neg l_2$
- Can we verify this, using LTL?
 - "there exists an execution that visits l_2 infinitely often, and never passes through program location l_4 "
- Yes: $M \models \neg((\Box \lozenge l_2) \wedge (\Box \neg l_4))$
- Can we verify this, using LTL?
 - "for every execution, it is always possible to return to the initial state of the program"
- No...

7. Computational Tree Logic



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

Reminder

- Tutorials this week (Assignment 1 feedback)
 - Thur 4pm (surnames A-L, by default):
 - UG06, Murray Learning Centre
 - Fri 10am (surnames M-Z, by default):
 - Lecture Theatre 1, Sports and Exercise Sciences

Recap

- Linear Temporal Logic (LTL)
 - \bigcirc (next), $\textcolor{brown}{U}$ (until), \lozenge (eventually), \Box (always)
- Examples, common patterns
 - $\Box a$, $\Box(a \rightarrow \lozenge b)$, $\Box(a \rightarrow \bigcirc b)$, $\Box\lozenge a$, $\lozenge\Box a$
 - invariants, safety properties, liveness properties
- Semantics
 - LTL evaluated over infinite paths/traces (and LTSs)
 - $M \models \Psi \Leftrightarrow \text{all paths of LTL } M \text{ satisfy LTL formula } \Psi$
- Equivalence of LTL formulas: $\Psi_1 \equiv \Psi_2$
 - proof using common simpler equivalences/dualities

Overview

- Linear temporal logic (LTL)
 - (non)equivalences, negation
- Computation tree logic (CTL)
 - syntax, semantics
 - examples
 - CTL vs. LTL
- See [BK08] sections 5.1.4 and 6–6.3

Example 1

- Prove (or disprove):

$$\diamond \psi \equiv \psi \vee \bigcirc \diamond \psi \quad ? \qquad \text{Yes}$$

- Can prove directly, using the relevant semantics for LTL:
- For any trace $\sigma \in (2^{\text{AP}})^\omega$...

$$\begin{aligned}\sigma \models \diamond \psi &\Leftrightarrow \exists k \geq 0 \text{ s.t. } \sigma[k\dots] \models \psi \\&\Leftrightarrow \sigma[0\dots] \models \psi \text{ or } \exists k \geq 1 \text{ s.t. } \sigma[k\dots] \models \psi \\&\Leftrightarrow \sigma \models \psi \text{ or } \exists k \geq 1 \text{ s.t. } \sigma[k\dots] \models \psi \\&\Leftrightarrow \sigma \models \psi \text{ or } \exists j \geq 0 \sigma[1\dots][j\dots] \models \psi \\&\Leftrightarrow \sigma \models \psi \text{ or } \sigma[1\dots] \models \diamond \psi \\&\Leftrightarrow \sigma \models \psi \text{ or } \sigma \models \bigcirc \diamond \psi \\&\Leftrightarrow \sigma \models \psi \vee \bigcirc \diamond \psi\end{aligned}$$

Example 2

- Prove (or disprove):

$$\neg(\Box a \rightarrow \Diamond b) \equiv \Box a \wedge \Box \neg b \quad ? \quad \text{Yes}$$

- Can prove by reusing simpler known equivalences

$$\begin{aligned}\neg(\Box a \rightarrow \Diamond b) &\equiv \neg(\neg \Box a \vee \Diamond b) && \text{since } \Psi_1 \rightarrow \Psi_2 \equiv \neg \Psi_1 \vee \Psi_2 \\ &\equiv \neg \neg \Box a \wedge \neg \Diamond b && \text{since } \neg(\Psi_1 \vee \Psi_2) \equiv \neg \Psi_1 \wedge \neg \Psi_2 \\ &\equiv \Box a \wedge \neg \Diamond b && \text{since } \neg \neg \Psi \equiv \Psi \\ &\equiv \Box a \wedge \neg \Diamond \neg \neg b && \text{since } \Psi \equiv \neg \neg \Psi \\ &\equiv \Box a \wedge \Box \neg b && \text{since } \neg \Diamond \neg \Psi \equiv \Box \Psi\end{aligned}$$

Example 3

- Prove (or disprove):

$$\Box\Diamond a \wedge \Box\Diamond b \equiv \Box\Diamond(a \wedge b) \quad ? \quad \text{No}$$

- Just need to provide a single trace as a counterexample
 - e.g. $\{a\} \{b\} \{a\} \{b\} \dots$
 - (which is satisfied by the left formula only)

LTL & Negation

- Are these statements equivalent? (for trace σ and LTL formula Ψ)
 - $\sigma \models \neg\Psi$
 - $\sigma \not\models \Psi$
- Yes
 - in fact, this is just the semantics of LTL
- Are these statements equivalent? (for LTS M and LTL formula Ψ)
 - $M \models \neg\Psi$
 - $M \not\models \Psi$
- No:
 - $M \models \neg\Psi$ means no trace satisfies Ψ
 - $M \not\models \Psi$ means it is not true that all traces satisfy Ψ
 - i.e. there exists some trace that does not satisfy Ψ

Existential properties

- Can we verify this, using LTL?
 - "there exists an execution that reaches program location l_2 "
- Yes: $M \models \Box \neg l_2$
- Can we verify this, using LTL?
 - "there exists an execution that visits l_2 infinitely often, and never passes through program location l_4 "
- Yes: $M \models \neg((\Box \lozenge l_2) \wedge (\Box \neg l_4))$
- Can we verify this, using LTL?
 - "for every execution, it is always possible to return to the initial state of the program"
- No...

CTL

- CTL – Computation Tree Logic
 - branching notion of time (compared to linear time for LTL)
 - infinite trees of states, not infinite sequences of states
- Two path quantifiers: \forall (for all paths), \exists (there exists a path)
 - LTL implicitly uses \forall
- Example
 - $\exists \lozenge l_2$ – "does there exist an execution that reaches l_2 ?"
- CTL model checking
 - quite different to (and simpler than) LTL model checking

CTL syntax

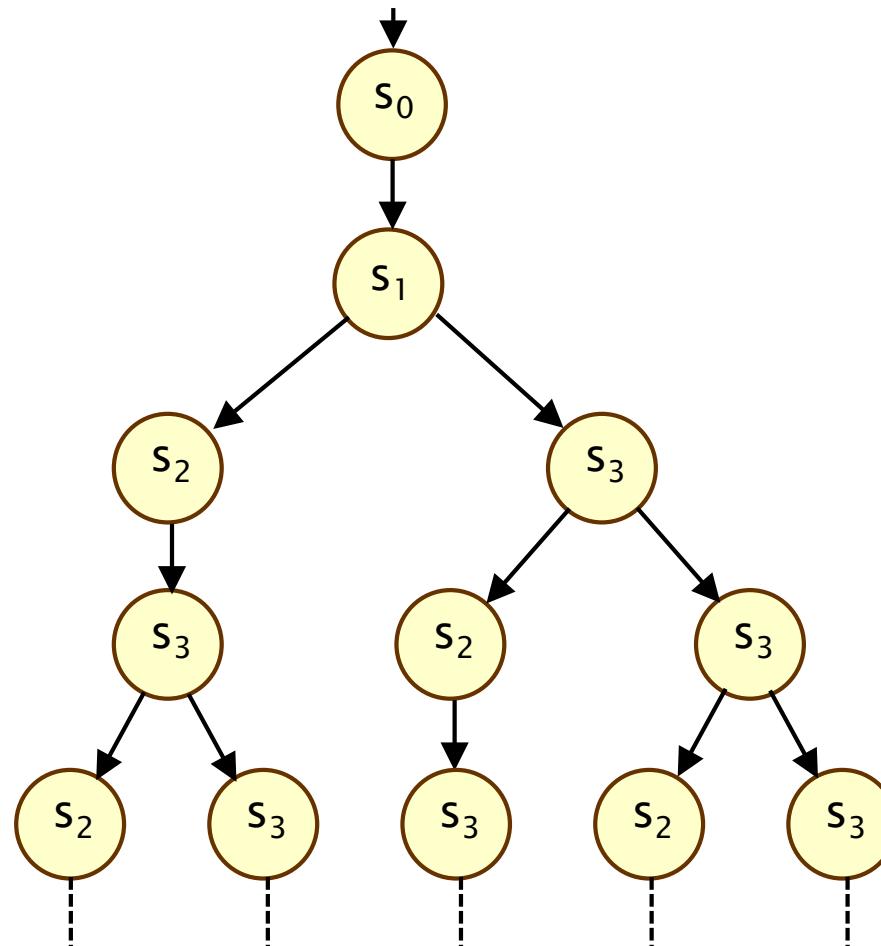
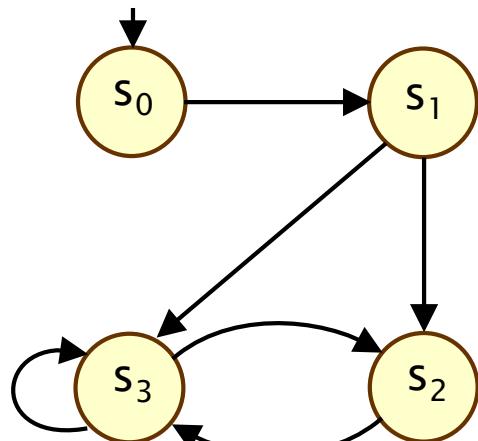
- Syntax split into state and path formulas
 - specify properties of states/paths, respectively
 - a CTL formula is a state formula ϕ
- State formulae:
 - $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \forall \psi \mid \exists \psi$
 - where $a \in AP$ and ψ is a path formula
- Path formulae
 - $\psi ::= \bigcirc \phi \mid \phi \cup \phi \mid \diamond \phi \mid \square \phi$
 - where ϕ is a state formula
- Examples (note the pairing of quantifiers/temporal operators)
 - $\exists \diamond I_2, \forall \bigcirc b, \forall \square \exists \diamond \text{initial}$

CTL – Alternative styles

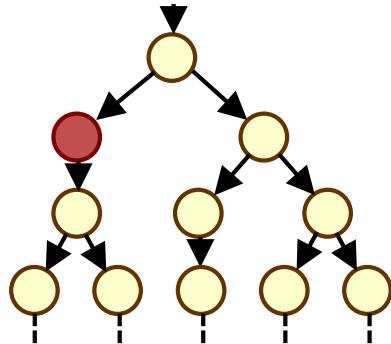
- Temporal operators:
 - $\bigcirc a \equiv \textcolor{brown}{X} a$ ("next")
 - $\lozenge a \equiv \textcolor{brown}{F} a$ ("future", "finally")
 - $\square a \equiv \textcolor{brown}{G} a$ ("globally")
- Path quantifiers:
 - $\forall \psi \equiv \textcolor{brown}{A} \psi$
 - $\exists \psi \equiv \textcolor{brown}{E} \psi$
- Brackets for quantifier scope: none/round/square
 - $\forall \lozenge \psi$
 - $\forall (\psi_1 \cup \psi_2)$
 - $\forall [\psi_1 \cup \psi_2]$

Computation trees

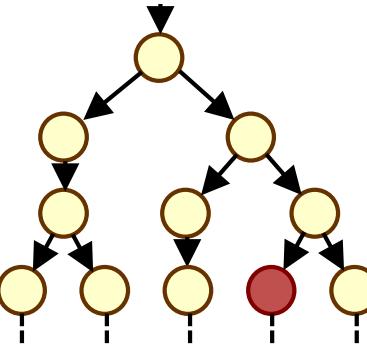
- LTS:
- (Prefix of) infinite computation tree
 - i.e. “unrolling of the LTS”



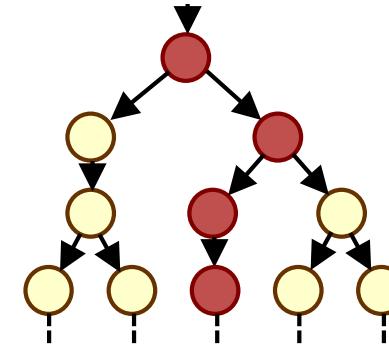
CTL – Intuitive semantics



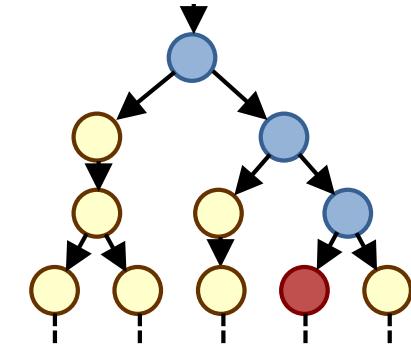
$\exists \circ \text{red}$



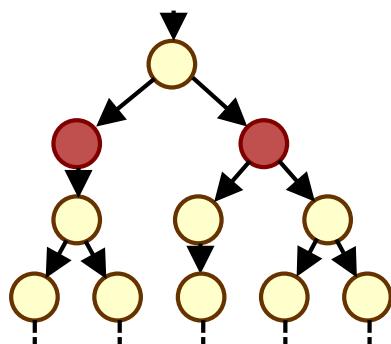
$\exists \diamond \text{red}$



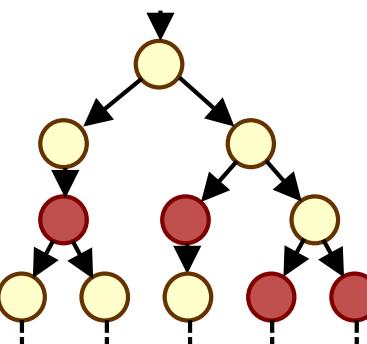
$\exists \square \text{red}$



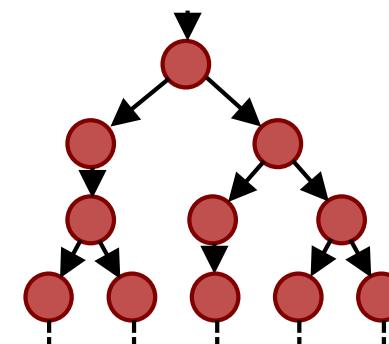
$\exists [\text{blue} \cup \text{red}]$



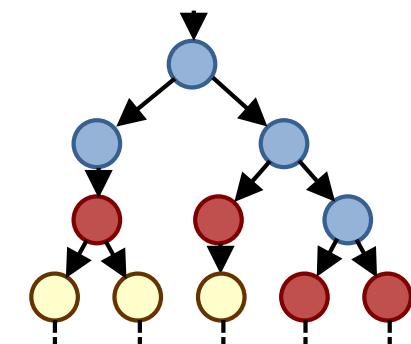
$\forall \circ \text{red}$



$\forall \diamond \text{red}$



$\forall \square \text{red}$



$\forall [\text{blue} \cup \text{red}]$

CTL examples

- $\forall \Box (\neg(\text{crit}_1 \wedge \text{crit}_2))$
 - mutual exclusion
- $\forall \Box \exists \Diamond \text{initial}$
 - for every computation, it is always possible to return to the initial state
- $\forall \Box (\text{request} \rightarrow \forall \Diamond \text{response})$
 - every request will eventually be granted
- $\forall \Box \forall \Diamond \text{crit}_1 \wedge \forall \Box \forall \Diamond \text{crit}_2$
 - each process has access to the critical section infinitely often

CTL semantics

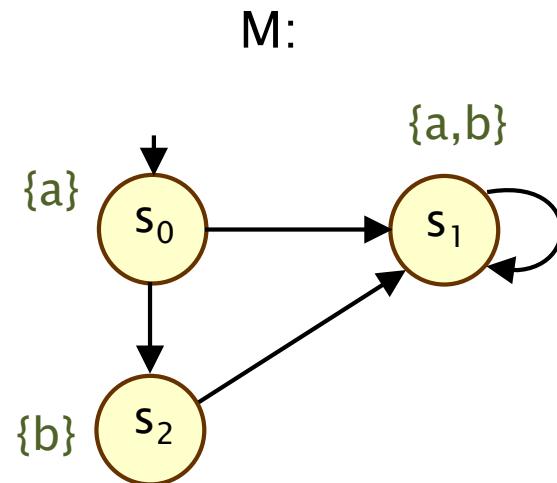
- Semantics of state formulae:
 - $s \models \phi$ denotes “ s satisfies ϕ ” or “ ϕ is true in s ”
- For a state s of an LTS $(S, \text{Act}, \rightarrow, I, AP, L)$:
 - $s \models \text{true}$ always
 - $s \models a$ $\Leftrightarrow a \in L(s)$
 - $s \models \phi_1 \wedge \phi_2$ $\Leftrightarrow s \models \phi_1$ and $s \models \phi_2$
 - $s \models \neg \phi$ $\Leftrightarrow s \not\models \phi$
 - $s \models \forall \psi$ $\Leftrightarrow \pi \models \psi$ for all $\pi \in \text{Path}(s)$
 - $s \models \exists \psi$ $\Leftrightarrow \pi \models \psi$ for some $\pi \in \text{Path}(s)$
- and for a path π :
 - $\pi \models \bigcirc \phi$ $\Leftrightarrow \pi[1] \models \phi$
 - $\pi \models \phi_1 \cup \phi_2$ $\Leftrightarrow \exists k \geq 0$ s.t. $\pi[k] \models \phi_2$ and $\forall i < k \pi[i] \models \phi_1$

($i+1$)th state
of path π



Examples

- $s_0 \models \forall \bigcirc b ?$
- $s_0 \models \exists \bigcirc \neg b ?$
- $s_0 \models \exists(a \cup a \wedge b) ?$
- $s_0 \models \exists \bigcirc \forall \Box(a \wedge b) ?$



8. CTL and LTL



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

Reminders

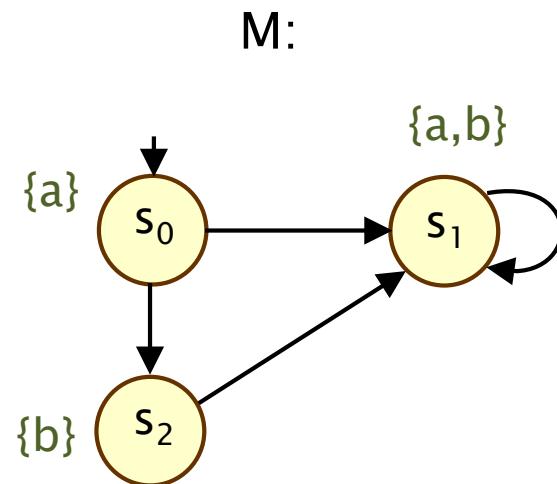
- Assignment 1
 - marks & individual feedback out today
 - also covered in this week's tutorials...
- Tutorials this week
 - Today (Thur) 4pm (surnames A-L, by default):
 - [UG06, Murray Learning Centre](#)
 - Tomorrow (Fri) 10am (surnames M-Z, by default):
 - [Lecture Theatre 1, Sports and Exercise Sciences](#)
- Assignment 2 (temporal logic)
 - out today, due in a week (12 noon, Thur 8 Feb)

Recap + Overview

- Temporal logic: Negation, existence of paths
- Computation Tree Logic (CTL)
 - usual temporal operators (\bigcirc , \mathbf{U} , \diamond , \square)
 - plus path quantifiers: \forall (for all paths), \exists (there exists a path)
 - evaluated over states, not paths
- Today
 - CTL equivalences and normal form
 - CTL vs. LTL (and CTL*)
 - fairness

Examples

- $s_0 \models \forall \bigcirc b ?$
- $s_0 \models \exists \bigcirc \neg b ?$
- $s_0 \models \exists(a \cup a \wedge b) ?$
- $s_0 \models \exists \bigcirc \forall \square (a \wedge b) ?$



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 $(S, \text{Act}, \rightarrow, I, AP, L)$:
 - $s \models \text{true}$ always
 - $s \models a$ $\Leftrightarrow a \in L(s)$
 - $s \models \phi_1 \wedge \phi_2$ $\Leftrightarrow s \models \phi_1$ and $s \models \phi_2$
 - $s \models \neg \phi$ $\Leftrightarrow s \not\models \phi$
 - $s \models \forall \psi$ $\Leftrightarrow \pi \models \psi$ for all $\pi \in \text{Path}(s)$
 - $s \models \exists \psi$ $\Leftrightarrow \pi \models \psi$ for some $\pi \in \text{Path}(s)$
- and for a path π :
 - $\pi \models \bigcirc \phi$ $\Leftrightarrow \pi[1] \models \phi$
 - $\pi \models \phi_1 \cup \phi_2$ $\Leftrightarrow \exists k \geq 0$ s.t. $\pi[k] \models \phi_2$ and $\forall i < k \pi[i] \models \phi_1$

($i+1$)th state
of path π



CTL equivalences

- Again various operators can be derived
 - propositional logic: \vee , \rightarrow , \leftrightarrow , \oplus
- Path quantifier duality:
 - $\forall \psi \equiv \neg \exists \neg \psi$
 - $\exists \psi \equiv \neg \forall \neg \psi$
- Temporal operators:
 - $\diamond \phi \equiv \text{true} \cup \phi$
 - $\square \phi \equiv ?$
- For example:
 - $\forall \square \phi \equiv \neg \exists \diamond (\neg \phi)$

Existential normal form (ENF)

- Often useful to consider **normal forms** for logics
 - e.g. checking equality, simplifying algorithms/proofs
- Recall: full syntax for CTL formula ϕ :
 - $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \forall \psi \mid \exists \psi$
 - $\psi ::= \bigcirc \phi \mid \phi \cup \phi \mid \lozenge \phi \mid \square \phi$
- **Existential normal form (ENF) for CTL**
 - no universal path quantifier (\forall) allowed, and no $\exists \lozenge$ formulae
 - $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \exists \bigcirc \phi \mid \exists (\phi \cup \phi) \mid \exists \square \phi$
- \forall can be removed using path quantifier duality:
 - $\forall \psi \equiv \neg \exists \neg \psi$

Conversion to ENF

- Allowed:
 - $\exists\bigcirc, \exists\mathsf{U}, \exists\Box$
- Not allowed:
 - $\exists\Diamond, \forall\bigcirc, \forall\mathsf{U}, \forall\Diamond, \forall\Box$
- Can always convert to ENF:
 - $\exists\Diamond\phi \equiv \exists(\text{true} \cup \phi)$
 - $\forall\bigcirc\phi \equiv \neg\exists\bigcirc\neg\phi$
 - $\forall\Diamond\phi \equiv \neg\exists\Box\neg\phi$
 - $\forall\Box\phi \equiv \neg\exists\Diamond\neg\phi \equiv \neg\exists(\text{true} \cup \neg\phi)$
 - $\forall(\phi_1 \cup \phi_2) \equiv \neg\exists((\neg\phi_2 \cup (\neg\phi_1 \wedge \neg\phi_2))) \wedge \neg\exists(\Box\neg\phi_2)$

ENF Conversion – Example

- CTL formula ϕ
 - $\phi = \forall \Diamond(\forall \bigcirc(b \vee \neg c) \vee \exists \Diamond(a \wedge b))$
- Convert to equivalent CTL formula ϕ' in ENF
 - $\phi' = \neg \exists \Box(\exists \bigcirc(\neg b \wedge c) \wedge \neg \exists(\text{true} \text{ U } (a \wedge b)))$
- (Start at the outside and work in)

CTL vs LTL

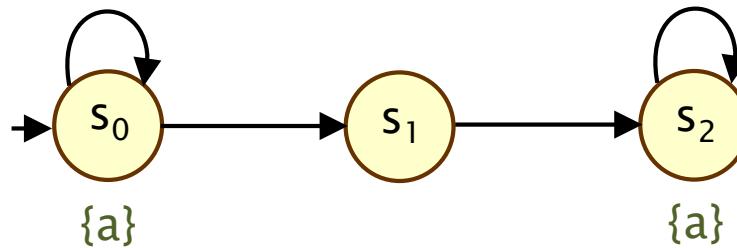
- How do we compare the expressiveness of CTL and LTL?
 - evaluated over states and paths, respectively
- Satisfaction of a CTL formula ϕ by an LTS M :
 - $M \models \phi$ if $s_0 \models \phi$ for all initial states s_0 of M
- CTL formulae ϕ_1 and ϕ_2 are equivalent ($\phi_1 \equiv \phi_2$) if
 - $M \models \phi_1 \Leftrightarrow M \models \phi_2$ (for any LTS M)
- CTL formula ϕ and LTL formula ψ are equivalent ($\phi \equiv \psi$) if
 - $M \models \phi \Leftrightarrow M \models \psi$ (for any LTS M)

Expressiveness of CTL and LTL

- Is CTL more expressive than LTL?
- What can we express in LTL that we cannot in CTL?
 - what about $\Box\Diamond a$?
 - no: $\forall \Box \forall \Diamond a \equiv \Box \Diamond a$
 - similarly: $\forall \Box(a \rightarrow \forall \bigcirc b) \equiv \Box(a \rightarrow \bigcirc b)$
 - what about $\Diamond \Box a$?
 - $\forall \Diamond \forall \Box a \equiv \Diamond \Box a$?

Expressiveness of CTL and LTL

- Counterexample showing: $\forall \Diamond \forall \Box a \not\equiv \Diamond \Box a$:
 - (note that a counterexample is now an LTS, not a trace)



- In fact, $\Diamond \Box a$ has no equivalent formula in CTL
- Similarly, $\forall \Box \exists \Diamond a$ has no equivalent formula in LTL
- The expressiveness of CTL and LTL are **incomparable**

CTL vs. LTL

- Key differences between CTL and LTL:
 - branching-time vs. linear-time
 - state-based vs. path-based
 - expressiveness: incomparable
 - model checking algorithms differ
 - CTL simpler and lower complexity than LTL
 - (linear in size of ϕ vs. exponential in size of ψ)
 - fairness dealt with more easily in LTL
- Both CTL and LTL are a subset of the logic CTL*
 - path quantifiers (\forall, \exists) arbitrarily nested with temporal operators

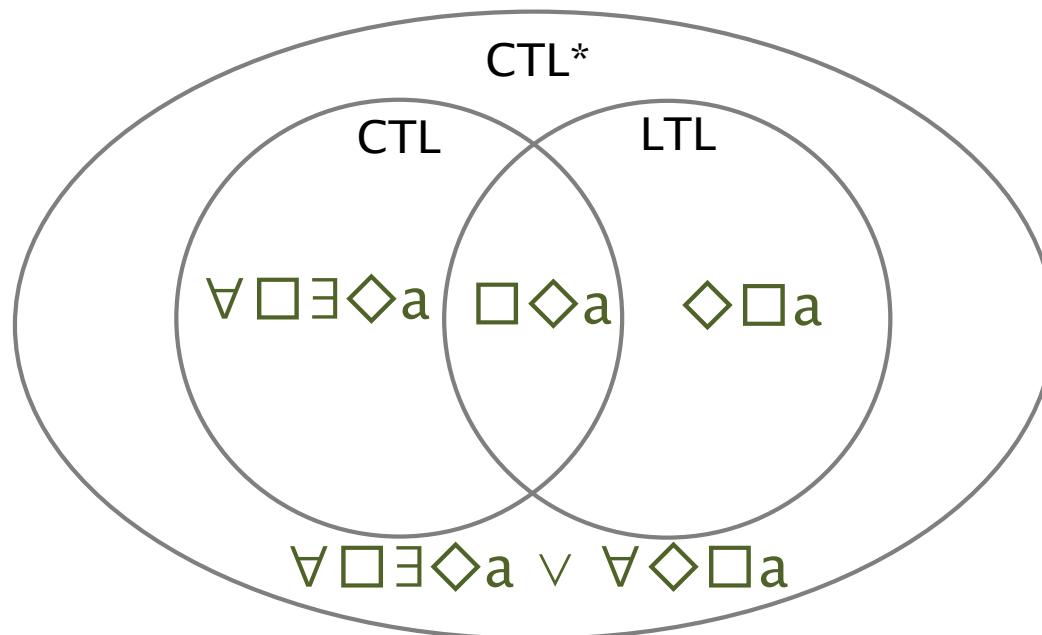
CTL*

- CTL* syntax

- $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \forall \psi \mid \exists \psi$
- $\psi ::= \phi \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc \psi \mid \psi \cup \psi \mid \diamond \psi \mid \square \psi$

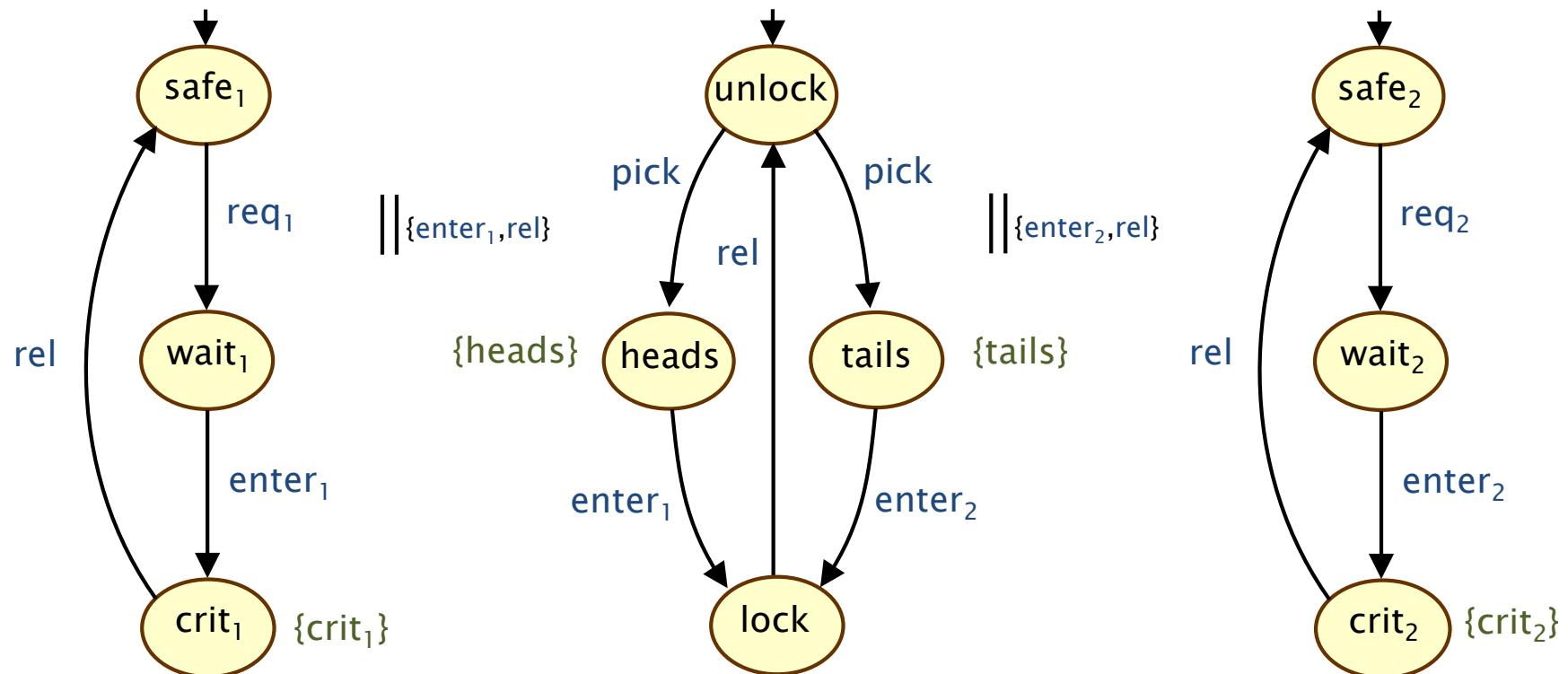
- Example

- $\forall \bigcirc \square a \wedge \exists \diamond \square b$



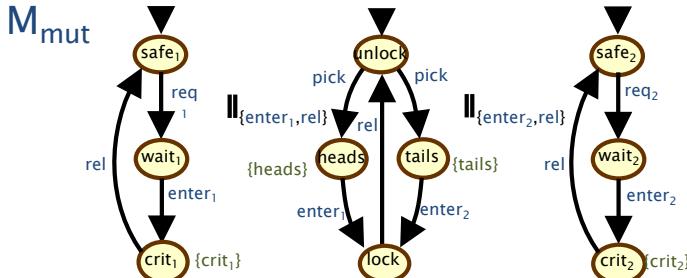
Fairness – motivation

- Rules out (infinite) behaviour considered to be unrealistic
 - often needed in order to verify liveness properties
- Example: two-process mutual exclusion + randomised arbiter
 - properties: $\square \neg(crit_1 \wedge crit_2)$ and $\square \lozenge crit_1 \wedge \square \lozenge crit_2$



Verification under fairness

- For the example M_{mut} :
 - $M_{\text{mut}} \not\models \square\lozenge\text{crit}_1 \wedge \square\lozenge\text{crit}_2$
- LTL semantics
 - $M \models \Psi \Leftrightarrow \text{trace}(\pi) \models \Psi$ for every path π of M
- LTL under fairness
 - $M \models_{\text{fair}} \Psi \Leftrightarrow \text{trace}(\pi) \models \Psi$ for every fair path π of M
- Many fairness conditions can be expressed in LTL
 - e.g. π is fair $\Leftrightarrow \pi \models \text{fair}$ where $\text{fair} = \square\lozenge\text{heads} \wedge \square\lozenge\text{tails}$
 - for the example: $M_{\text{mut}} \models_{\text{fair}} \square\lozenge\text{crit}_1 \wedge \square\lozenge\text{crit}_2$
- LTL verification under fairness
 - $M \models_{\text{fair}} \Psi \Leftrightarrow M \models (\text{fair} \rightarrow \Psi)$ (assuming M has no terminal states)



Summary

- Temporal logic
 - extends propositional logic with modal/temporal operators
- Linear temporal logic (LTL)
 - logic for linear time properties (over traces, LTSs)
 - syntax (\bigcirc , U , \diamond , \Box), semantics, equivalences
- Computation tree logic (CTL)
 - branching-time logic (over states, LTSs)
 - syntax ($\forall \psi, \exists \psi$), semantics (computation trees)
- Equivalences, expressiveness, negation, duality
- CTL vs LTL, CTL*, fairness

9. CTL Model Checking



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

Recap: CTL vs. LTL

- Key differences between CTL and LTL:
 - branching-time vs. linear-time
 - state-based vs. path-based
 - expressiveness: incomparable
 - model checking algorithms differ
 - CTL simpler and lower complexity than LTL
 - (linear in size of ϕ vs. exponential in size of ψ)
 - fairness dealt with more easily in LTL
- Both CTL and LTL are a subset of the logic CTL*
 - path quantifiers (\forall, \exists) arbitrarily nested with temporal operators

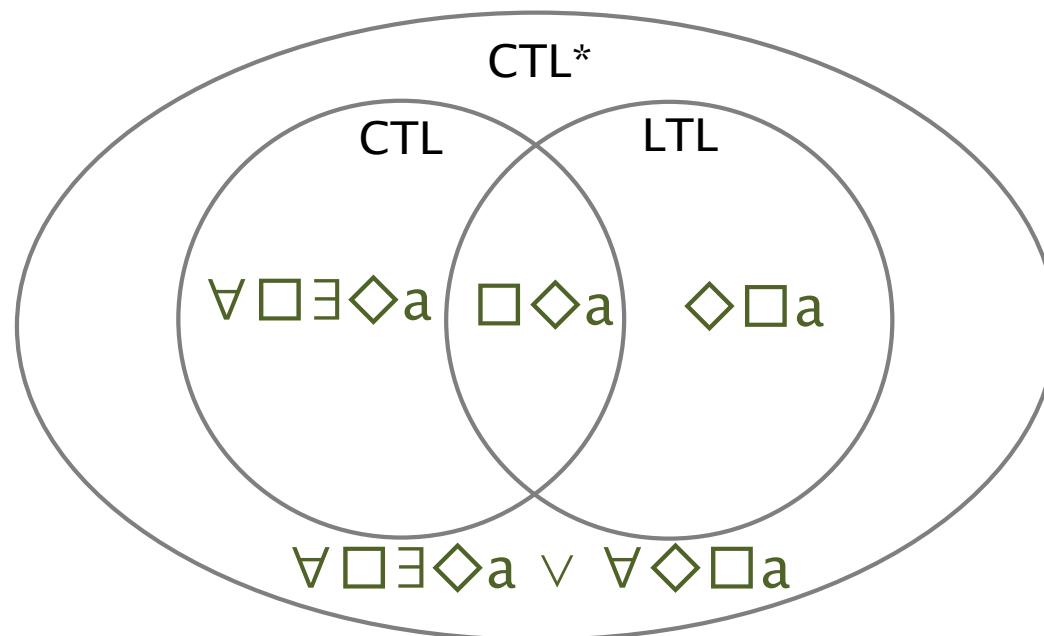
CTL*

- CTL* syntax

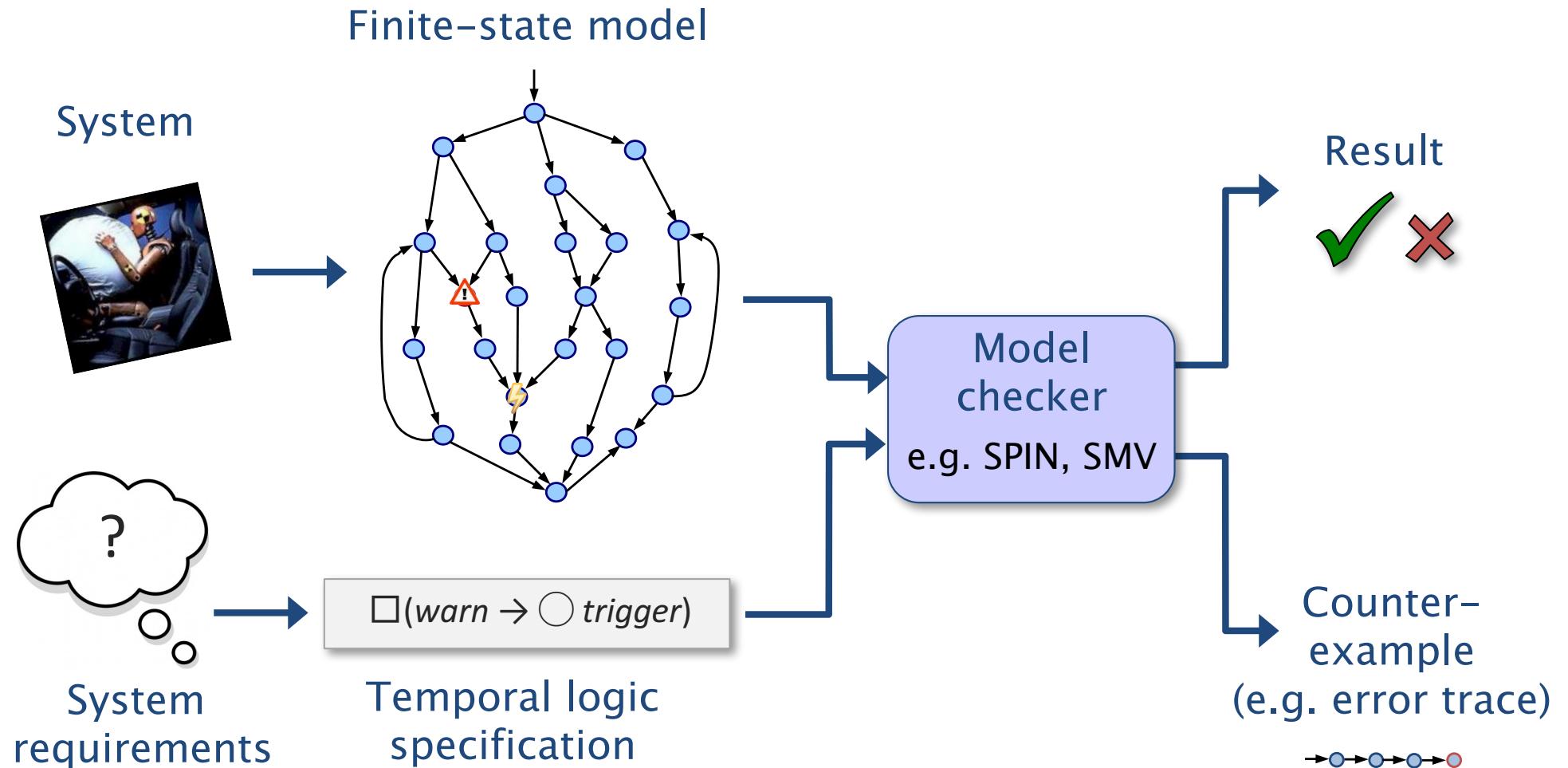
- $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \forall \psi \mid \exists \psi$
- $\psi ::= \phi \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc \psi \mid \psi \cup \psi \mid \diamond \psi \mid \square \psi$

- Example

- $\forall \bigcirc \square a \wedge \exists \diamond \square b$



Verification via model checking



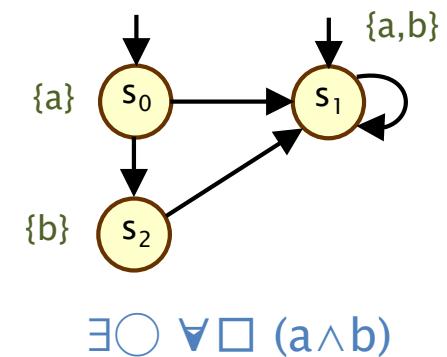
Overview

- CTL model checking
 - the model checking problem
 - basic algorithm
 - (existential normal form)
 - model checking $\exists U$
 - model checking $\exists \Box$
- See [BK08] Section 6.4

CTL model checking

- The CTL model checking problem is:
 - given an LTS $M = (S, \text{Act}, \rightarrow, I, AP, L)$ and a CTL formula ϕ ,
 - check whether $M \models \phi$
 - i.e. whether $s \models \phi$ for all initial states $s \in I$

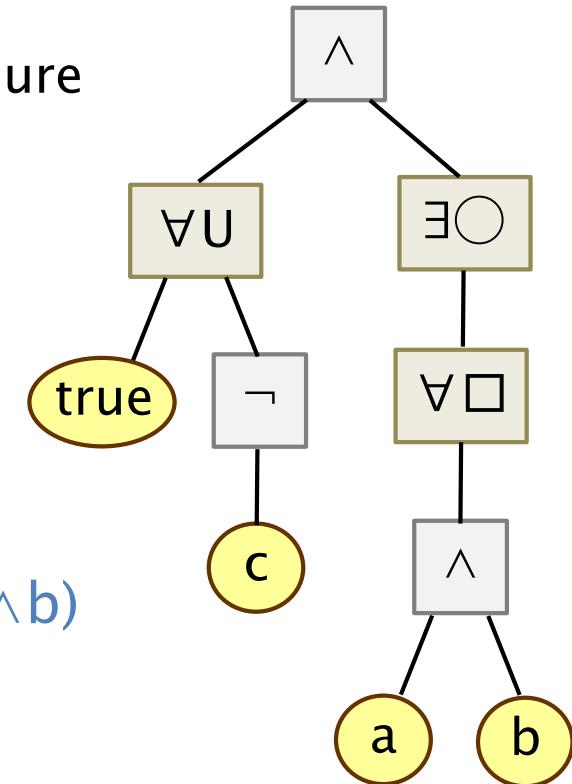
- Assumptions:
 - M is finite and has no terminal states



- $\text{Sat}(\phi)$ is the satisfaction set for CTL formula ϕ
 - i.e. the set of all states that satisfy ϕ
 - $\text{Sat}(\phi) = \{ s \in S \mid s \models \phi \}$
 - so model checking is determining whether $I \subseteq \text{Sat}(\phi)$

Basic algorithm

- Compute $\text{Sat}(\phi)$, then check if $I \subseteq \text{Sat}(\phi)$
 - so we in fact check if $s \models \phi$ for all states s
 - known as a **global** model checking procedure
- $\text{Sat}(\phi)$ is computed recursively
 - bottom-up traversal
 - of the **parse tree** of formula ϕ
- Example: $\phi = \forall(\text{true} \cup \neg c) \wedge \exists \square (a \wedge b)$



Computing $\text{Sat}(\phi)$

- Recursive computation of $\text{Sat}(\phi)$:

- $\text{Sat}(\text{true}) = S$
- $\text{Sat}(a) = \{ s \in S \mid a \in L(s) \}$
- $\text{Sat}(\phi_1 \wedge \phi_2) = \text{Sat}(\phi_1) \cap \text{Sat}(\phi_2)$
- $\text{Sat}(\neg\phi) = S \setminus \text{Sat}(\phi)$
- $\text{Sat}(\exists \bigcirc \phi) = \dots$
- $\text{Sat}(\forall \bigcirc \phi) = \dots$
- $\text{Sat}(\exists(\phi_1 \cup \phi_2)) = \dots$
- $\text{Sat}(\forall(\phi_1 \cup \phi_2)) = \dots$
- \dots

Existential Normal Form (ENF)

- We simplify model checking by first converting to ENF
 - no universal path quantifier (\forall) allowed, and no $\exists\Diamond$ formulae
 - $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \exists\bigcirc\phi \mid \exists(\phi \cup \phi) \mid \exists\Box\phi$
- Conversion to ENF:
 - $\exists\Diamond\phi \equiv \exists(\text{true} \cup \phi)$
 - $\forall\bigcirc\phi \equiv \neg\exists\bigcirc\neg\phi$
 - $\forall\Diamond\phi \equiv \neg\exists\Box\neg\phi$
 - $\forall\Box\phi \equiv \neg\exists\Diamond\neg\phi \equiv \neg\exists(\text{true} \cup \neg\phi)$
 - $\forall(\phi_1 \cup \phi_2) \equiv \neg\exists((\neg\phi_2 \cup (\neg\phi_1 \wedge \neg\phi_2))) \wedge \neg\exists(\Box\neg\phi_2)$

Computing $\text{Sat}(\phi)$

- Recursive computation of $\text{Sat}(\phi)$:

- $\text{Sat}(\text{true}) = S$
- $\text{Sat}(a) = \{ s \in S \mid a \in L(s) \}$
- $\text{Sat}(\phi_1 \wedge \phi_2) = \text{Sat}(\phi_1) \cap \text{Sat}(\phi_2)$
- $\text{Sat}(\neg\phi) = S \setminus \text{Sat}(\phi)$
- $\text{Sat}(\exists \bigcirc \phi) = \{ s \in S \mid \text{Post}(s) \cap \text{Sat}(\phi) \neq \emptyset \}$
- $\text{Sat}(\exists (\phi_1 \cup \phi_2)) = \text{CheckExistsUntil}(\text{Sat}(\phi_1), \text{Sat}(\phi_2))$
- $\text{Sat}(\exists \Box \phi) = \text{CheckExistsAlways}(\text{Sat}(\phi))$



Propositional logic

Immediate predecessors

Graph algorithms

Example

- Model the check CTL formula: $\phi = \forall \bigcirc a \wedge \exists \bigcirc \neg b$

- Convert to ENF:

- $\phi \equiv \neg \exists \bigcirc \neg a \wedge \exists \bigcirc \neg b$

- Evaluate $\text{Sat}(\phi)$ recursively

- $\text{Sat}(a) = \{s_0, s_1\}$

- $\text{Sat}(\neg a) = S \setminus \text{Sat}(a) = S \setminus \{s_0, s_1\} = \{s_2, s_3\}$

- $\text{Sat}(\exists \bigcirc \neg a) = \{s_0, s_1, s_3\}$

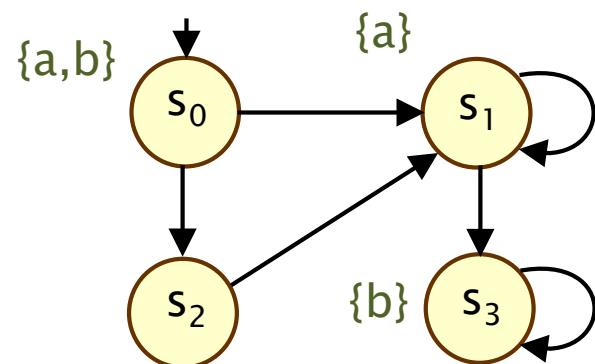
- $\text{Sat}(\neg \exists \bigcirc \neg a) = S \setminus \{s_0, s_1, s_3\} = \{s_2\}$

- $\text{Sat}(b) = \{s_0, s_3\}$

- $\text{Sat}(\neg b) = S \setminus \{s_0, s_3\} = \{s_1, s_2\}$

- $\text{Sat}(\exists \bigcirc \neg b) = \{s_0, s_1, s_2\}$

- $\text{Sat}(\phi) = \{s_2\} \cap \{s_0, s_1, s_2\} = \{s_2\} \Rightarrow M \not\models \phi$



Model checking $\exists U$

- Procedure to compute $\text{Sat}(\exists(\phi_1 \cup \phi_2))$
 - given $\text{Sat}(\phi_1)$ and $\text{Sat}(\phi_2)$
- Basic idea: backwards search of the LTS from ϕ_2 -states
 - $T_0 := \text{Sat}(\phi_2)$
 - $T_i := T_{i-1} \cup \{ s \in \text{Sat}(\phi_1) \mid \text{Post}(s) \cap T_{i-1} \neq \emptyset \}$
 - until $T_i = T_{i-1}$
 - $\text{Sat}(\exists(\phi_1 \cup \phi_2)) = T_i$
- (i.e. keep adding predecessors of states in T_{i-1})
- Based on expansion law
 - $\exists(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge \exists \bigcirc \exists(\phi_1 \cup \phi_2))$
 - (can be formulated as a fixed-point equation)

Example – $\exists U$

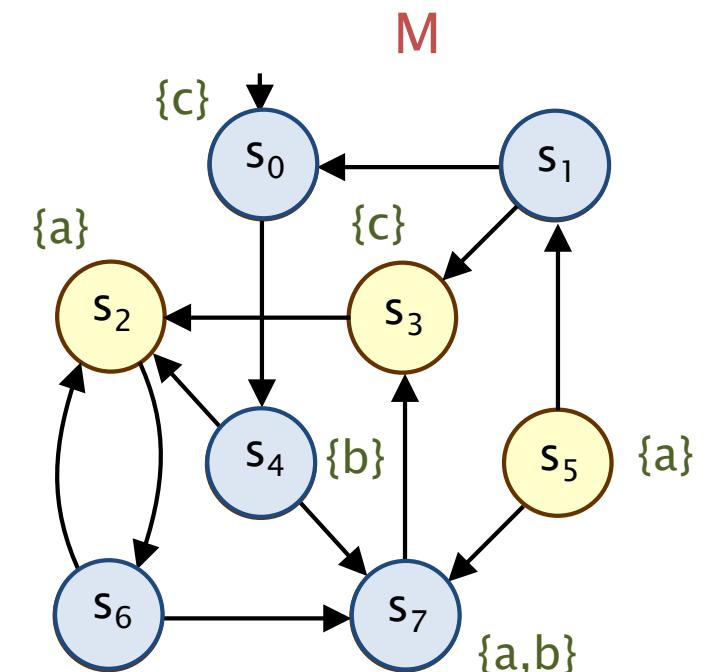
- Model the check CTL formula: $\phi = \exists (\neg a \cup b)$

- $\text{Sat}(\neg a) = S \setminus \{s_2, s_5, s_7\} = \{s_0, s_1, s_3, s_4, s_6\}$
- $\text{Sat}(b) = \{s_4, s_7\}$

- Backwards search

- $T_0 := \text{Sat}(b) = \{s_4, s_7\}$
- $T_1 := T_0 \cup \{s_0, s_4, s_6\} = \{s_0, s_4, s_6, s_7\}$
- $T_2 := T_1 \cup \{s_0, s_1, s_4, s_6\} = \{s_0, s_1, s_4, s_6, s_7\}$
- $T_3 := T_2 \cup \{s_0, s_1, s_4, s_6\} = \{s_0, s_1, s_4, s_6, s_7\}$
- $T_3 = T_2$
- $\text{Sat}(\phi) = \{s_0, s_1, s_4, s_6, s_7\}$

- So: $M \models \phi$



Model checking $\exists U$

- More detailed algorithm:

CheckExistsUntil(Sat(ϕ_1), Sat(ϕ_2)):

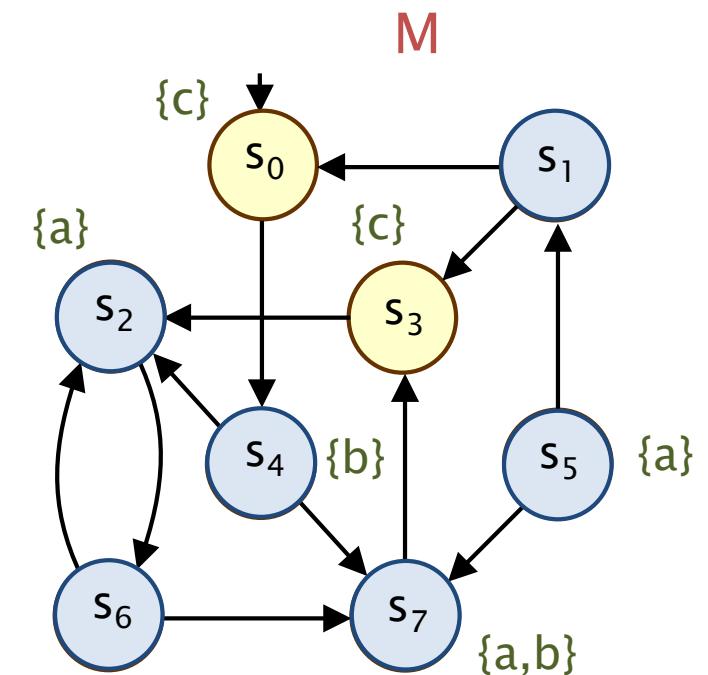
```
E := Sat( $\phi_2$ )
T := E
while (E  $\neq \emptyset$ ) do
    let s'  $\in$  E
    E := E  $\setminus$  {s'}
    for all s  $\in$  Pre(s') do
        if s  $\in$  Sat( $\phi_1$ )  $\setminus$  T then E := E  $\cup$  {s} ; T := T  $\cup$  {s} fi
    od
od
return T
```

Model checking $\exists \Box \phi$

- Procedure to compute $\text{Sat}(\exists \Box \phi)$
 - given $\text{Sat}(\phi)$
- It again helps to consider expansion laws:
 - $\exists(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge \exists \bigcirc \exists(\phi_1 \cup \phi_2))$
 - $\exists \Box \phi \equiv \phi \wedge \exists \bigcirc \exists \Box \phi$
- Basic idea: again, backwards search of the LTS
 - $T_0 := \text{Sat}(\phi)$
 - $T_i := T_{i-1} \cap \{ s \in \text{Sat}(\phi) \mid \text{Post}(s) \cap T_{i-1} \neq \emptyset \}$
 - until $T_i = T_{i-1}$
 - $\text{Sat}(\exists \Box \phi) = T_i$
- (i.e. keep removing states that are not predecessors of T_{i-1})

Example - $\exists \Box$

- Model the check CTL formula: $\phi = \forall \Diamond c$
 - convert to ENF: $\forall \Diamond c \equiv \neg \exists \Box \neg c$
 - $\text{Sat}(\neg c) = S \setminus \{s_0, s_3\} = \{s_1, s_2, s_4, s_5, s_6, s_7\}$
- Backwards search
 - $T_0 := \text{Sat}(\neg c) = \{s_1, s_2, s_4, s_5, s_6, s_7\}$
 - $T_1 := T_0 \cap \{s_2, s_4, s_5, s_6\} = \{s_2, s_4, s_5, s_6\}$
 - $T_2 := T_1 \cap \{s_2, s_4, s_6\} = \{s_2, s_4, s_6\}$
 - $T_3 := T_2 \cap \{s_2, s_4, s_6\} = \{s_2, s_4, s_6\}$
 - $T_3 = T_2$
 - $\text{Sat}(\exists \Box \neg c) = \{s_2, s_4, s_6\}$
 - $\text{Sat}(\phi) = S \setminus \{s_2, s_4, s_6\} = \{s_0, s_1, s_3, s_5, s_7\}$
- So: $M \models \phi$



Model checking $\exists \Box$

- More detailed algorithm:

CheckExistsAlways(Sat(ϕ)):

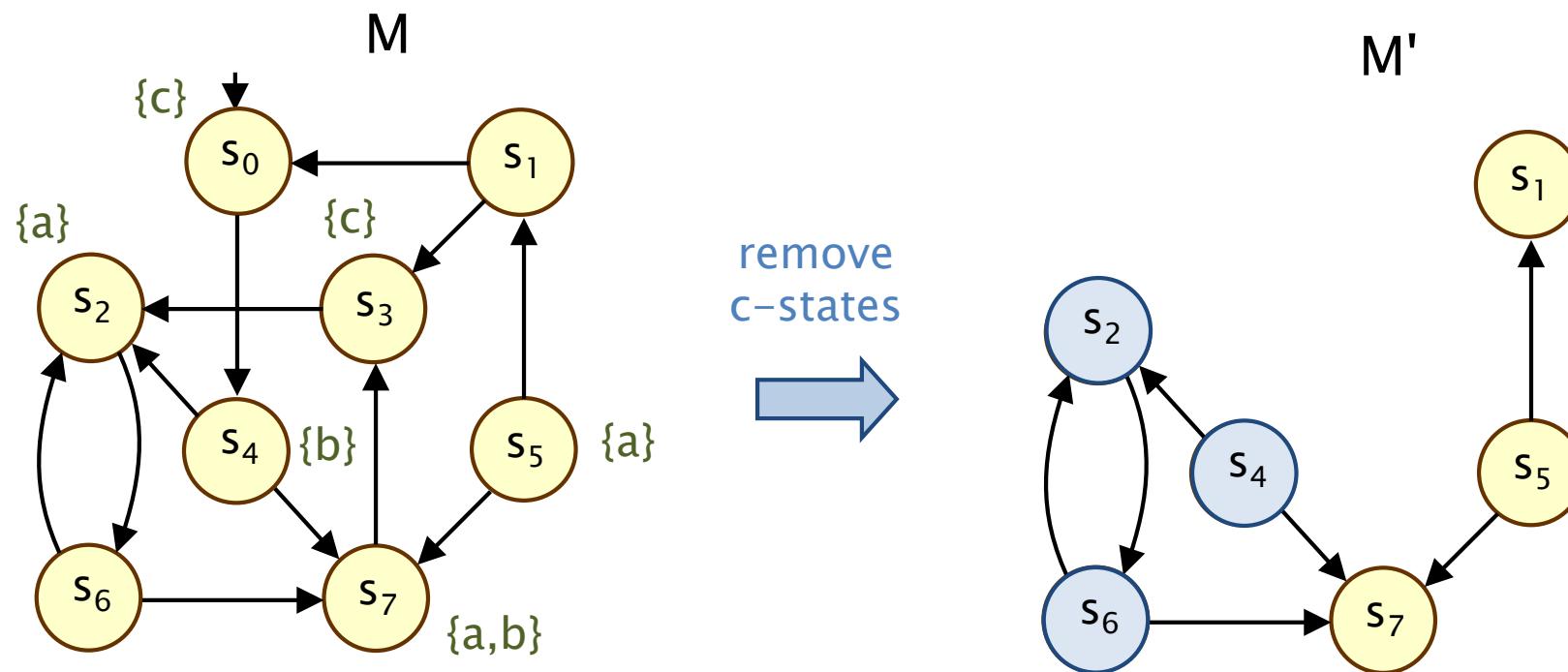
```
E := S \ Sat( $\phi$ )
T := Sat( $\phi$ )
for all s  $\in$  Sat( $\phi$ ) do count[s] := |Post(s)| od
while (E  $\neq$   $\emptyset$ ) do
    let s'  $\in$  E
    E := E \ {s'}
    for all s  $\in$  Pre(s') do
        if s  $\in$  T then
            count[s] := count[s] - 1
            if (count[s] = 0) then T := T \ {s}; E := E  $\cup$  {s} fi
        fi
    od
od
return T
```

Alternative algorithm for $\exists \Box$

- An alternative algorithm to model check $\exists \Box \phi$ on LTS M
 - based on **strongly connected components**
- Strongly connected components (SCCs)
 - **SCC** = maximal, connected sub-graph
 - **non-trivial SCC** = SCC with at least one transition
- Model checking $\exists \Box$
 1. construct a modified LTS M' by
 - removing all states not satisfying ϕ , i.e. those in $S \setminus \text{Sat}(\phi)$
 - and removing all transitions to/from those states
 2. find the non-trivial strongly connected components (SCCs) in M'
 3. $\text{Sat}(\phi)$ is the set of states that can reach an SCC in M'

Example revisited - $\exists \Box$

- Model the check CTL formula: $\phi' = \exists \Box \neg c$
 - convert M to produce M'
 - identify non-trivial SCCs in M': $\{s_2, s_6\}$
 - identify states than can reach the SCCs: $\text{Sat}(\phi') = \{s_2, s_4, s_6\}$



Complexity

- The time complexity of CTL model checking
 - for LTS M and CTL formula ϕ
- is: $O(|M| \cdot |\phi|)$
 - i.e. linear in both model and formula size
 - where $|M|$ = number of states + number of transitions in M
 - and $|\phi|$ = number of operators in ϕ
- Worst-case execution:
 - all operators are temporal operators
 - each one performs single traversal of whole model

Summary

- CTL model checking
 - global model checking algorithm
 - recursive computation of $\text{Sat}(\phi)$
 - based on parse tree of ϕ
- Conversion to existential normal form (ENF)
 - $\exists\bigcirc, \exists U, \exists\Box$ only
- Graph-based algorithms
 - $\exists\bigcirc$ – check predecessors
 - $\exists U, \exists\Box$ – backwards graph traversal
 - $\exists\Box$ – also via strongly connected components

Next lecture

- Automata-based model checking
 - see Sections 4–4.2 of [BK08]

10. Automata-based Model Checking



Computer-Aided Verification

Dave Parker

University of Birmingham

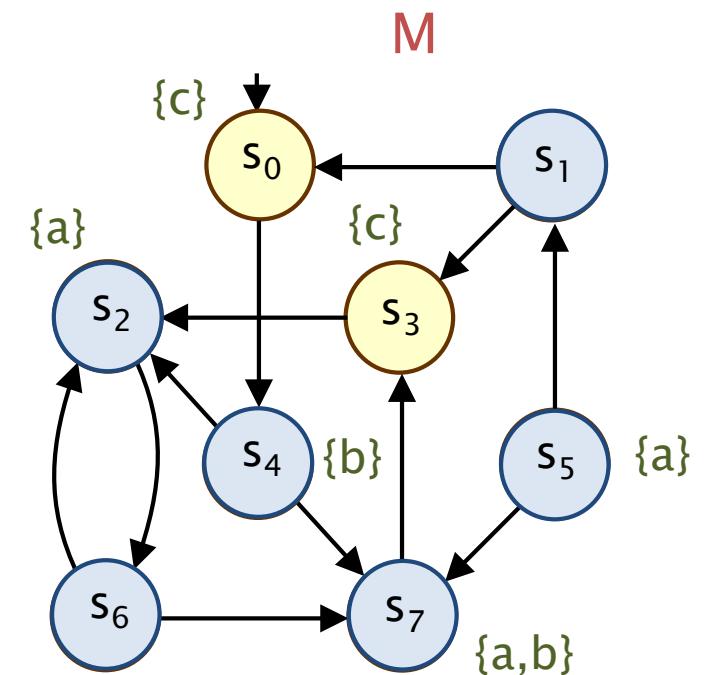
2017/18

Model checking $\exists \Box$

- Procedure to compute $\text{Sat}(\exists \Box \phi)$
 - given $\text{Sat}(\phi)$
- It again helps to consider expansion laws:
 - $\exists(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge \exists \bigcirc \exists(\phi_1 \cup \phi_2))$
 - $\exists \Box \phi \equiv \phi \wedge \exists \bigcirc \exists \Box \phi$
- Basic idea: again, backwards search of the LTS
 - $T_0 := \text{Sat}(\phi)$
 - $T_i := T_{i-1} \cap \{ s \in \text{Sat}(\phi) \mid \text{Post}(s) \cap T_{i-1} \neq \emptyset \}$
 - until $T_i = T_{i-1}$
 - $\text{Sat}(\exists \Box \phi) = T_i$
- (i.e. keep removing states that are not predecessors of T_{i-1})

Example - $\exists \Box$

- Model the check CTL formula: $\phi = \forall \Diamond c$
 - convert to ENF: $\forall \Diamond c \equiv \neg \exists \Box \neg c$
 - $\text{Sat}(\neg c) = S \setminus \{s_0, s_3\} = \{s_1, s_2, s_4, s_5, s_6, s_7\}$
- Backwards search
 - $T_0 := \text{Sat}(\neg c) = \{s_1, s_2, s_4, s_5, s_6, s_7\}$
 - $T_1 := T_0 \cap \{s_2, s_4, s_5, s_6\} = \{s_2, s_4, s_5, s_6\}$
 - $T_2 := T_1 \cap \{s_2, s_4, s_6\} = \{s_2, s_4, s_6\}$
 - $T_3 := T_2 \cap \{s_2, s_4, s_6\} = \{s_2, s_4, s_6\}$
 - $T_3 = T_2$
 - $\text{Sat}(\exists \Box \neg c) = \{s_2, s_4, s_6\}$
 - $\text{Sat}(\phi) = S \setminus \{s_2, s_4, s_6\} = \{s_0, s_1, s_3, s_5, s_7\}$
- So: $M \models \phi$



Model checking $\exists \Box$

- More detailed algorithm:

CheckExistsAlways(Sat(ϕ)):

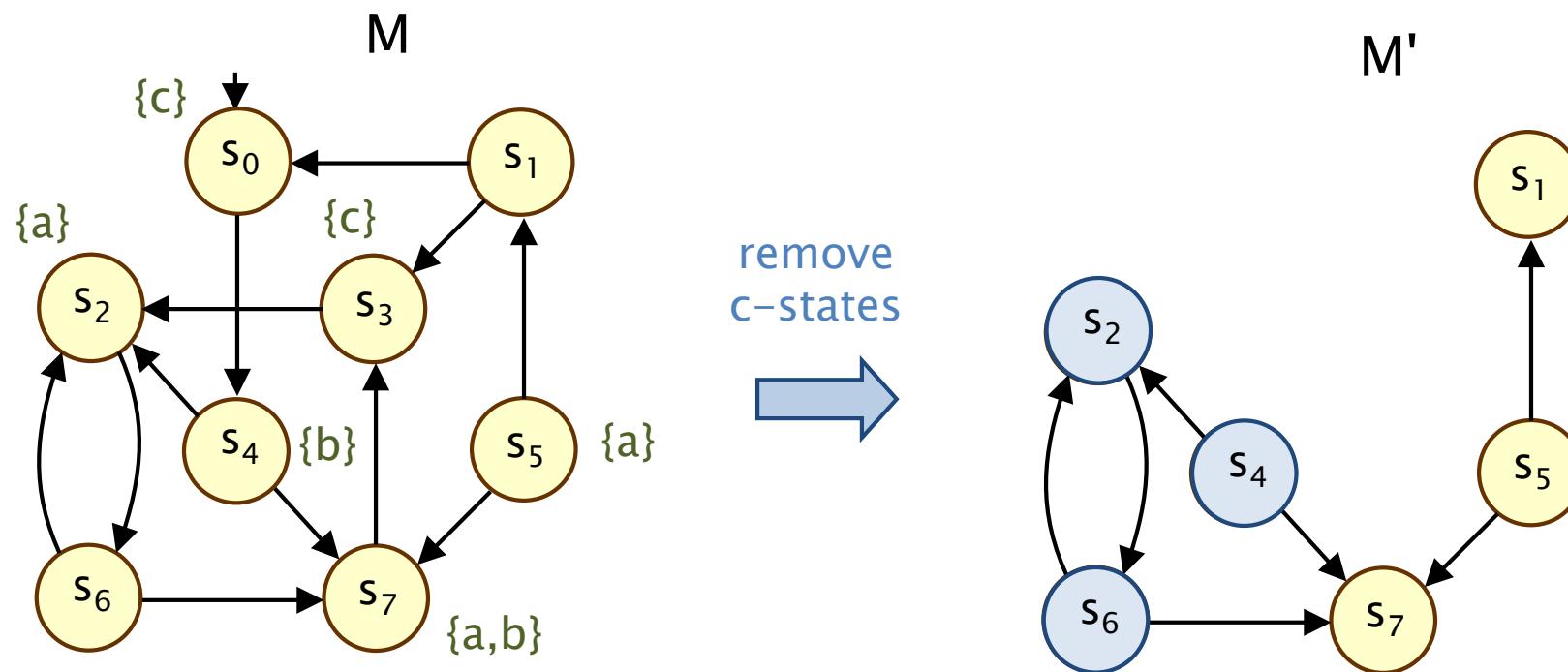
```
E := S \ Sat( $\phi$ )
T := Sat( $\phi$ )
for all s  $\in$  Sat( $\phi$ ) do count[s] := |Post(s)| od
while (E  $\neq$   $\emptyset$ ) do
    let s'  $\in$  E
    E := E \ {s'}
    for all s  $\in$  Pre(s') do
        if s  $\in$  T then
            count[s] := count[s] - 1
            if (count[s] = 0) then T := T \ {s}; E := E  $\cup$  {s} fi
        fi
    od
od
return T
```

Alternative algorithm for $\exists \Box$

- An alternative algorithm to model check $\exists \Box \phi$ on LTS M
 - based on **strongly connected components**
- Strongly connected components (SCCs)
 - **SCC** = maximal, connected sub-graph
 - **non-trivial SCC** = SCC with at least one transition
- Model checking $\exists \Box$
 1. construct a modified LTS M' by
 - removing all states not satisfying ϕ , i.e. those in $S \setminus \text{Sat}(\phi)$
 - and removing all transitions to/from those states
 2. find the non-trivial strongly connected components (SCCs) in M'
 3. $\text{Sat}(\phi)$ is the set of states that can reach an SCC in M'

Example revisited - $\exists \Box$

- Model the check CTL formula: $\phi' = \exists \Box \neg c$
 - convert M to produce M'
 - identify non-trivial SCCs in M': $\{s_2, s_6\}$
 - identify states than can reach the SCCs: $\text{Sat}(\phi') = \{s_2, s_4, s_6\}$



Complexity

- The time complexity of CTL model checking
 - for LTS M and CTL formula ϕ
- is: $O(|M| \cdot |\phi|)$
 - i.e. linear in both model and formula size
 - where $|M|$ = number of states + number of transitions in M
 - and $|\phi|$ = number of operators in ϕ
- Worst-case execution:
 - all operators are temporal operators
 - each one performs single traversal of whole model

CTL model checking: Wrapping up

- CTL model checking
 - global model checking algorithm
 - recursive computation of $\text{Sat}(\phi)$
 - based on parse tree of ϕ
- Conversion to existential normal form (ENF)
 - $\exists\bigcirc, \exists U, \exists \Box$ only
 - i.e. reduces to looking for existence of paths
- Graph-based algorithms on LTS
 - backwards graph traversal or SCCs

9. Automata-based Model Checking



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

Branching-time vs. linear-time

- So far:
 - model checking for branching-time properties (CTL)
 - e.g. $\phi = (\forall \Box \exists \Diamond a) \wedge (\exists \Box b)$
- Now: linear-time properties, e.g. as specified in LTL
 - e.g. $\Diamond \Box c \wedge \Box(d \rightarrow \bigcirc \neg c)$
- Next lectures: automata-based properties
 - connections between automata and logic
 - first: finite automata and safety properties

Overview

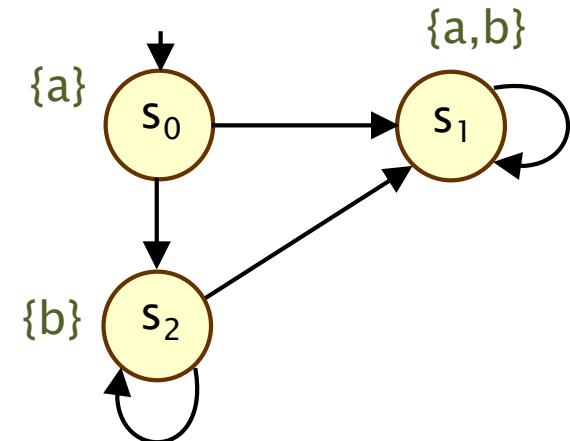
- Recap
 - linear-time properties
 - safety properties
- Nondeterministic finite automata (NFAs)
 - regular languages
 - regular expressions
- Regular safety properties
 - LTS–NFA products
 - model checking
- See [BK08] Sections 4–4.2

Reminder: Notation

- A (finite or infinite) word over a finite alphabet Σ is
 - a finite sequence $w = A_0A_1\dots A_n$ where $A_i \in \Sigma$ for all $0 \leq i < n$
 - an infinite sequence $\sigma = A_0A_1\dots$ where $A_i \in \Sigma$ for all $i \geq 0$
- A prefix w of word $\sigma = A_0A_1\dots$ is
 - a finite word $B_0B_1\dots B_n$ with $B_i = A_i$ for all $0 \leq i \leq n$
- A suffix σ' of word $\sigma = A_0A_1\dots$ is
 - an infinite word $B_0B_1\dots$ with $B_i = A_{i+j}$ for some $j \geq 0$ and all $0 \leq i \leq n$
- Σ^* denotes the set of finite words over Σ
- Σ^ω denotes the set of infinite words over Σ

Recap – Linear-time properties

- Paths: sequences of connected states
 - e.g. $\pi = s_0 s_2 s_2 s_1 s_1 \dots$
- Traces: infinite words over 2^{AP}
 - $\text{trace}(\pi) = \{a\} \{b\} \{b\} \{a,b\} \{a,b\} \{a,b\} \dots$
 - $\text{Traces}(M) = \text{traces of all paths}$
- Linear-time properties
 - set of allowable traces $P \subseteq (2^{\text{AP}})^\omega$
 - e.g. $\square(a \rightarrow \Diamond b)$ – "a is always eventually followed by b"
 - $M \models P \Leftrightarrow \text{Traces}(M) \subseteq P \Leftrightarrow \text{trace}(\pi) \in P \text{ for all paths } \pi \text{ of } M$
- Classes of (linear-time) property:
 - invariant, safety property, liveness property...
 - independent of any particular model...



Recap – Safety properties

- Informally:
 - "nothing bad happens", e.g. "a failure does not occur"
 - defined in terms of the "bad" events, which happen in finite time
- More precisely
 - P_{safe} is a **safety property** if any (infinite) word where P_{safe} does not hold has a bad prefix
 - a **bad prefix** is a finite prefix σ' containing the bad event, such that no infinite path beginning with σ' satisfies P_{safe}
 - the bad prefixes define the safety property
- Formally:
 - $P_{\text{safe}} = (2^{\text{AP}})^\omega \setminus \{ w.\sigma' \in (2^{\text{AP}})^* \mid \text{for some bad prefix } w, \text{ suffix } \sigma' \}$

Example safety properties

- Example safety properties:
 - over AP = $\{\text{red}_1, \text{green}_1, \text{red}_2, \text{green}_2\}$
- "the traffic lights never both show green simultaneously":
 - what are the bad prefixes?
 - any finite word ending in $\{\text{green}_1, \text{green}_2\}$
- "green₁ is always preceded (immediately) by red₁"
 - what are the bad prefixes?
 - any finite word where green₁ appears and red₁ did not appear immediately before it

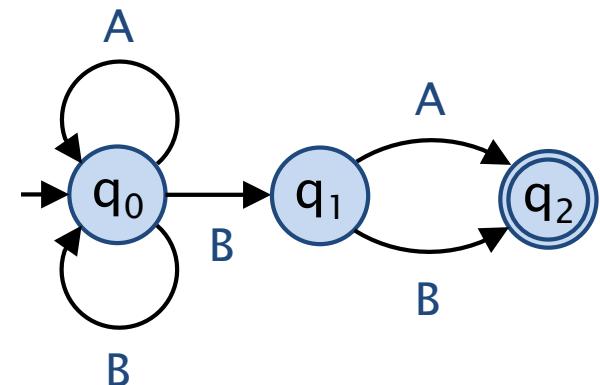
Nondeterministic finite automata

- A **nondeterministic finite automaton (NFA)** is:

- a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$

- where:

- Q is a finite set of states
 - Σ is an alphabet
 - $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function
 - $Q_0 \subseteq Q$ is a set of initial states
 - $F \subseteq Q$ is a set of “accept” states

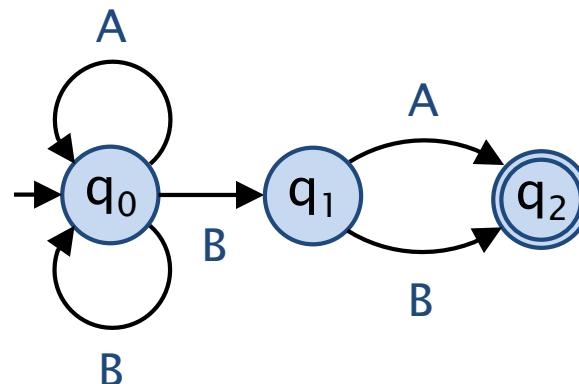


- Example

- $Q = \{q_0, q_1, q_2\}$, $Q_0 = \{q_0\}$, $F = \{q_2\}$
 - $\Sigma = \{A, B\}$, $\delta(q_0, A) = \{q_0\}$, $\delta(q_0, B) = \{q_0, q_1\}$, ...

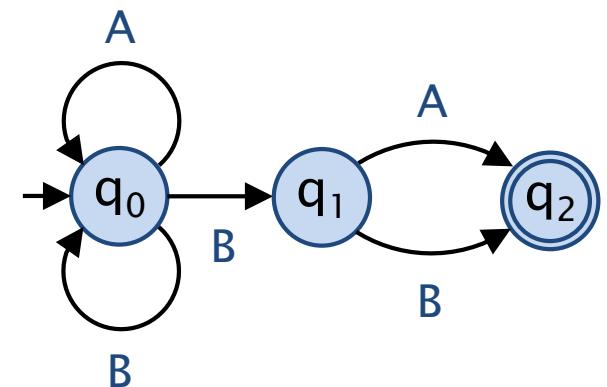
Runs of an NFA

- For an NFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F) \dots$
- There is an **A transition** from q to q' (written $q - A \rightarrow q'$)
 - if $q' \in \delta(q, A)$
- A **run** of \mathcal{A} on a finite word $w = A_0 A_1 \dots A_{n-1}$ is:
 - a sequence of automata states $q_0 q_1 \dots q_n$ such that:
 - $q_0 \in Q_0$ and $q_i - A_i \rightarrow q_{i+1}$ for all $0 \leq i < n$
- **Example**
 - a word: **BBA**
 - a run: **$q_0 q_0 q_1 q_2$**



Language of an NFA

- An **accepting run** is a run ending in an accept state
 - i.e. a run $q_0 q_1 \dots q_n$ with $q_n \in F$
- Word **w** is accepted by \mathcal{A} iff:
 - there exists an accepting run of \mathcal{A} on w
- Example
 - **BBA** (accepted)
 - **BAA** (not accepted)
- The **language of \mathcal{A}** , denoted $\mathcal{L}(\mathcal{A})$ is:
 - the set of all words accepted by \mathcal{A}
- Automata \mathcal{A} and \mathcal{A}' are **equivalent** if $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$



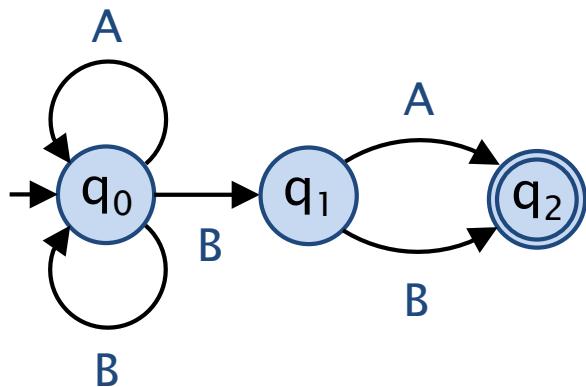
language:
“penultimate
symbol is B”

Regular expressions

- Regular expressions E over a finite alphabet Σ
 - are given by the following grammar:
 - $E ::= \emptyset \mid \varepsilon \mid A \mid E + E \mid E \cdot E \mid E^*$
 - where $A \in \Sigma$
- Language $\mathcal{L}(E) \subseteq \Sigma^*$ of a regular expression:
 - $\mathcal{L}(\emptyset) = \emptyset$ (empty language)
 - $\mathcal{L}(\varepsilon) = \{ \varepsilon \}$ (empty word)
 - $\mathcal{L}(A) = \{ A \}$ (symbol)
 - $\mathcal{L}(E_1 + E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$ (union)
 - $\mathcal{L}(E_1 \cdot E_2) = \{ w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(E_1) \text{ and } w_2 \in \mathcal{L}(E_2) \}$ (concatenation)
 - $\mathcal{L}(E^*) = \{ w^i \mid w \in \mathcal{L}(E) \text{ and } i \in \mathbb{N} \}$ (finite repetition)

Regular languages

- A set of finite words $\mathcal{L} \subseteq \Sigma^*$ is a regular language...
 - iff $\mathcal{L} = \mathcal{L}(E)$ for some regular expression E
 - iff $\mathcal{L} = \mathcal{L}(A)$ for some finite automaton A



$(A+B)^*B(A+B)$

(i.e. penultimate symbol is B)

Operations on NFAs

- **Intersection of two NFAs**
 - build (synchronised) product automaton
 - cross product of $\mathcal{A}_1 \otimes \mathcal{A}_2$ accepts $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$
- **Language emptiness of an NFA**
 - reduces to reachability
 - $L(\mathcal{A}) \neq \emptyset$ iff can reach a state in F from an initial state in Q_0
- **Other important operations**
 - construction of an NFA from a regular expression, inductively
 - determinisation (convert to deterministic finite automaton (DFA))
 - complementation of an NFA (via conversion to a DFA)

11. Automata-based Model Checking



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

Mid-term questionnaires

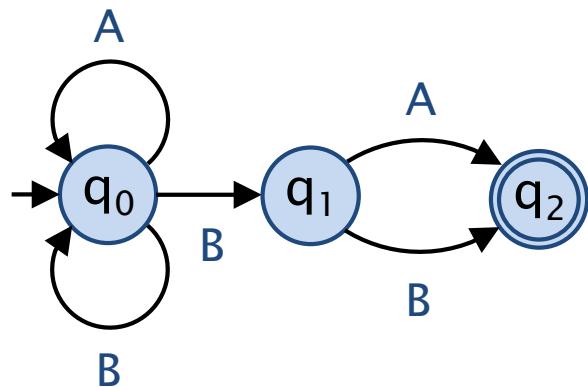
- Generally, all happy
 - lectures, tutorials, feedback, etc.
- Comments/suggestions
 - don't split tutorial sessions
 - more unassessed exercises
 - assignment timing for extended version
 - additional (practical) context for material

Overview

- Regular safety properties
 - nondeterministic finite automata (NFAs)
- Model checking regular safety properties
 - LTS-NFA products
- See [BK08] Sections 4–4.2

Recap: Regular languages

- A set of finite words $\mathcal{L} \subseteq \Sigma^*$ is a regular language...
 - iff $\mathcal{L} = \mathcal{L}(E)$ for some regular expression E
 - iff $\mathcal{L} = \mathcal{L}(A)$ for some finite automaton A

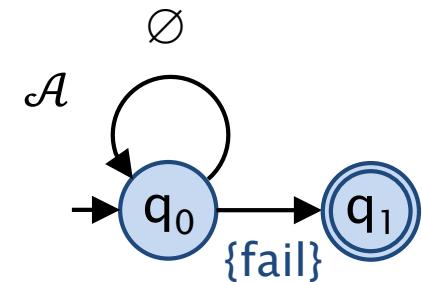


$(A+B)^*B(A+B)$

(i.e. penultimate symbol is B)

Languages/automata as properties

- Recall:
 - a linear-time **property** is a set of infinite words $P \subseteq (2^{\text{AP}})^\omega$
 - (we assume non-terminating systems with infinite paths/traces)
- But we could represent a set of **finite** traces/words
 - as a regular language over alphabet 2^{AP}
 - or an NFA over alphabet 2^{AP}
- Simple example
 - finite traces where a failure eventually occurs
 - $\text{AP} = \{\text{fail}\}$
 - $2^{\text{AP}} = \{\emptyset, \{\text{fail}\}\}$
 - $\mathcal{L}(\mathcal{A}) = \{ \{\text{fail}\}, \emptyset \{\text{fail}\}, \emptyset \emptyset \{\text{fail}\}, \emptyset \emptyset \emptyset \{\text{fail}\}, \dots \}$



Regular safety properties

- A regular safety property is
 - a safety property for which the set of “bad prefixes” (finite violations) forms a regular language
 - Example:
 - "a failure never occurs" – $\square \neg \text{fail}$
 - $\text{AP} = \{\text{fail}\}$
 - $2^{\text{AP}} = \{\emptyset, \{\text{fail}\}\}$
 - The bad prefixes are represented by an NFA over 2^{AP}
 - $\mathcal{L}(\mathcal{A}) = \{ \{\text{fail}\}, \emptyset \{\text{fail}\}, \emptyset \emptyset \{\text{fail}\}, \emptyset \emptyset \emptyset \{\text{fail}\}, \dots \}$
 - Note: we actually represent just minimal bad prefixes here
 - we could also represent all bad prefixes
- NFA \mathcal{A} :
-
- ```
graph LR; start(()) --> q0((q0)); q0 -- {fail} --> q0; q0 -- {fail} --> q1(((q1)));
```
- Regexp:
- $\emptyset^* \{\text{fail}\}$

# Example 2

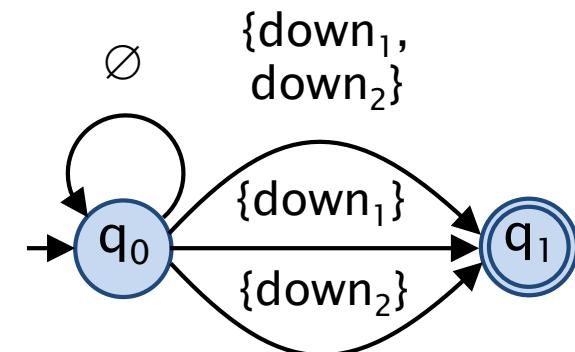
- Regular safety property
  - "both servers are always up" –  $\square(\neg \text{down}_1 \wedge \neg \text{down}_2)$
  - AP = {down<sub>1</sub>, down<sub>2</sub>}
  - $2^{\text{AP}} = \{\emptyset, \{\text{down}_1\}, \{\text{down}_2\}, \{\text{down}_1, \text{down}_2\}\}$
- Bad prefixes
  - any finite word ending in {down<sub>1</sub>}, {down<sub>2</sub>} or {down<sub>1</sub>, down<sub>2</sub>}

- NFA:

$\neg \text{down}_1 \wedge \neg \text{down}_2$



shorthand  
for

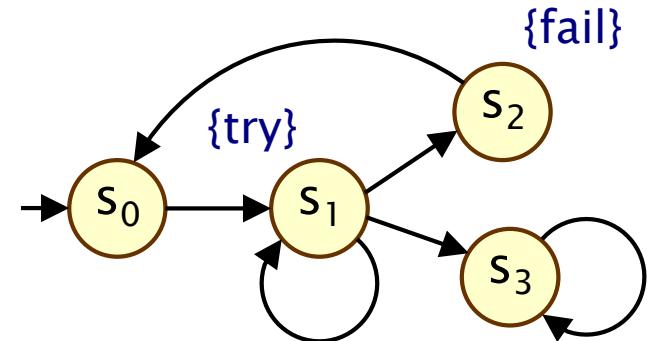


Propositional  
formula

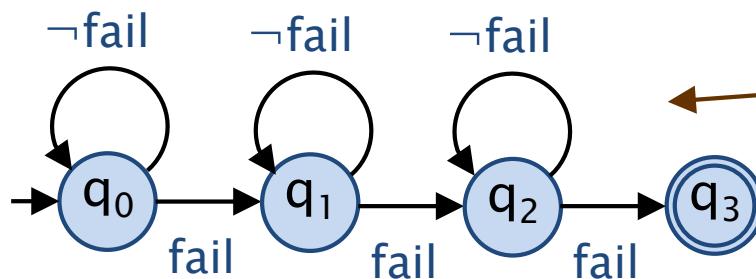
Sets of atomic  
propositions

# Example 3

- Regular safety property:
  - “at most 2 failures occur”
  - $\text{AP} = \{\text{try}, \text{fail}\}$
  - $2^{\text{AP}} = \{ \emptyset, \{\text{fail}\}, \{\text{try}\}, \{\text{fail,try}\} \}$



- NFA for bad prefixes:



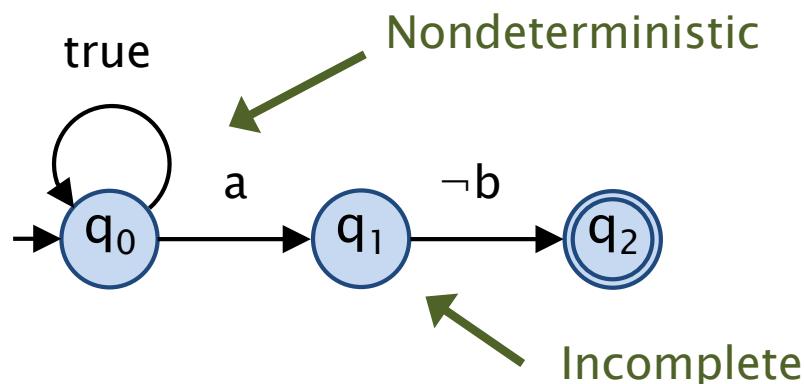
fail denotes:  
 $\{\text{fail}\}, \{\text{fail,try}\}$   
¬fail denotes:  
 $\emptyset, \{\text{try}\}$

- Regular expression for bad prefixes:  
 $(\neg\text{fail})^*.\text{fail}.(\neg\text{fail})^*.\text{fail}.(\neg\text{fail})^*.\text{fail}$

fail denotes:  
 $(\{\text{fail}\} + \{\text{fail,try}\})$   
¬fail denotes:  
 $(\emptyset + \{\text{try}\})$

# Example 4

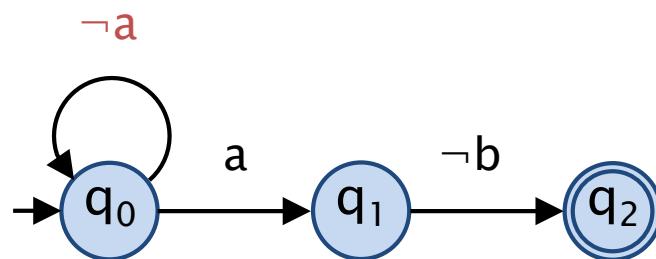
- NFA for regular safety property
  - $AP = \{a, b\}$



- Bad prefixes
  - any finite word where  $a$  appears and then  $b$  does not appear immediately afterwards
- Regular safety property
  - "b always immediately follows a" –  $\square(a \rightarrow \bigcirc b)$

# Example 5

- NFA for regular safety property
  - $AP = \{a, b\}$



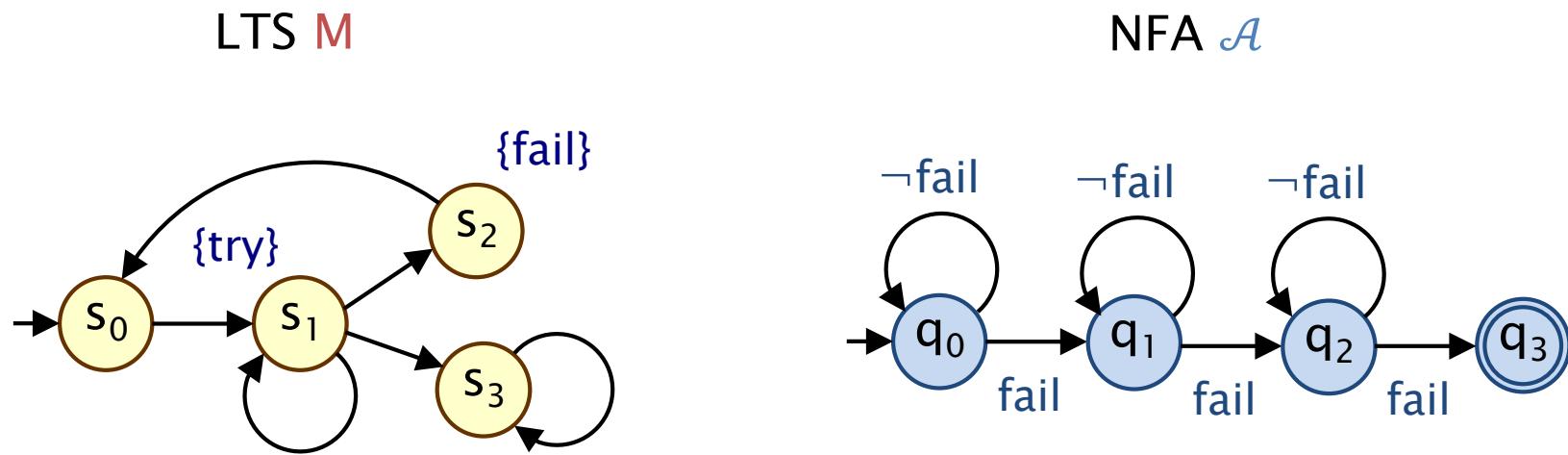
- Bad prefixes
  - any finite word where  $b$  does not appear immediately after the first  $a$  (if there is an  $a$ )
- Regular safety property
  - "b always immediately follows the first occurrence of a"

# Model checking

- Given an LTS  $M$  and regular safety property  $P_{\text{safe}}$ 
  - $M \models P_{\text{safe}} \Leftrightarrow \text{Traces}(M) \subseteq P_{\text{safe}}$   
 $\Leftrightarrow \text{Traces}_{\text{fin}}(M) \cap \text{BadPref}(P_{\text{safe}}) = \emptyset$
- Given also an NFA  $\mathcal{A}$  representing the bad prefixes of  $P_{\text{safe}}$ 
  - $M \models P_{\text{safe}} \Leftrightarrow \text{Traces}_{\text{fin}}(M) \cap \mathcal{L}(\mathcal{A}) = \emptyset$
- Model checking  $M$  against regular safety property  $P_{\text{safe}}$ 
  - check if any finite behaviour of  $M$  intersects with  $P_{\text{safe}}$
  - which we do by constructing a product of  $M$  and  $\mathcal{A}$



# Example 3 revisited



“at most 2 failures occur”

# Product of an LTS and an NFA

- For an LTS  $M$  and an NFA  $\mathcal{A}$ 
  - we construct the product LTS of  $M$  and  $\mathcal{A}$ , denoted  $M \otimes \mathcal{A}$
- Synchronous parallel composition
  - transitions of NFA  $\mathcal{A}$  synchronise with state labels of LTS  $M$
  - allows finite traces/words that are in both  $M$  and  $\mathcal{A}$
- Product definition (informal)
  - states are pairs  $(s, q)$  of states from  $M$  and  $\mathcal{A}$
  - transitions go from  $(s, q)$  to  $(s', q')$  if  
 $s - \alpha \rightarrow s'$  in  $M$  and  $q - L(s) \rightarrow q'$  in  $\mathcal{A}$
  - initial states are  $(s_0, q)$  where  $s_0$  is some initial state of  $M$  and  $q_0 - L(s_0) \rightarrow q$  for some initial state  $q_0$  of  $\mathcal{A}$
  - states  $(s, q)$  are labelled with "accept" if  $q$  is accepting in  $\mathcal{A}$

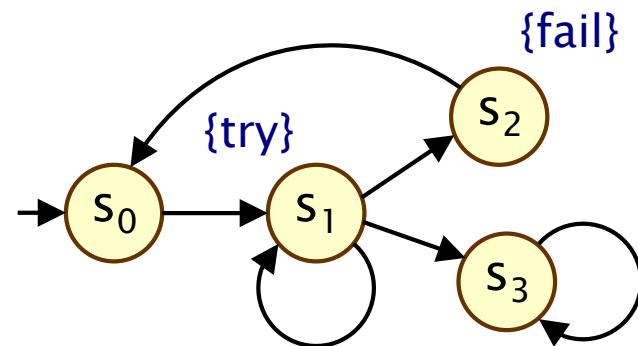
# Product of an LTS and an NFA

- Formally:
  - for LTS  $M = (S, \text{Act}, \rightarrow, I, AP, L)$  and NFA  $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$
- The product  $D \otimes A$  is:
  - the LTS  $(S \times Q, \text{Act}, \rightarrow', I', \{\text{accept}\}, L')$
- where:
  - $I' = \{(s_0, q) \mid s_0 \in I \text{ and } q_0 - L(s_0) \rightarrow q \text{ for some } q_0 \in Q_0\}$
  - $L'((s, q)) = \{\text{accept}\} \text{ if } q \in F \text{ and } L'((s, q)) = \emptyset \text{ otherwise}$
  - $\rightarrow'$  is defined as follows:

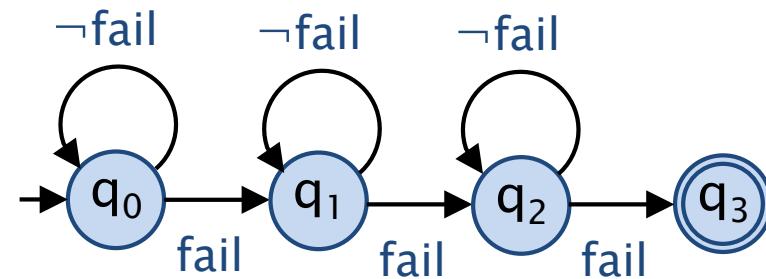
$$\frac{s - \alpha \rightarrow s' \wedge q - L(s') \rightarrow q'}{(s, q) - \alpha \rightarrow (s', q')}$$

# Example 3 revisited

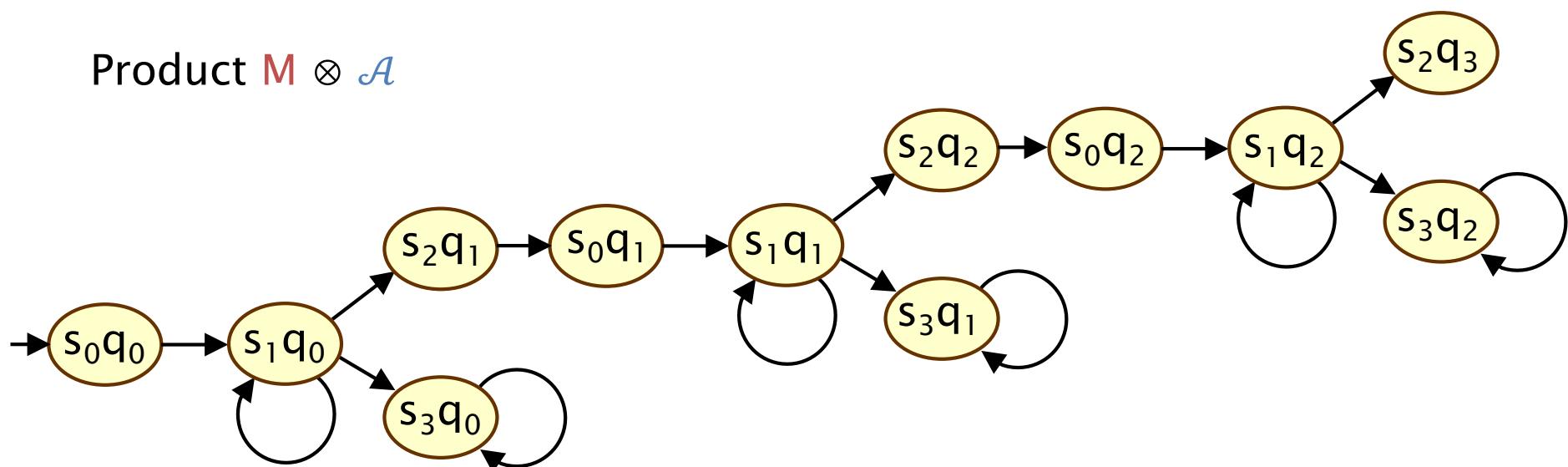
LTS  $M$



NFA  $\mathcal{A}$



Product  $M \otimes \mathcal{A}$

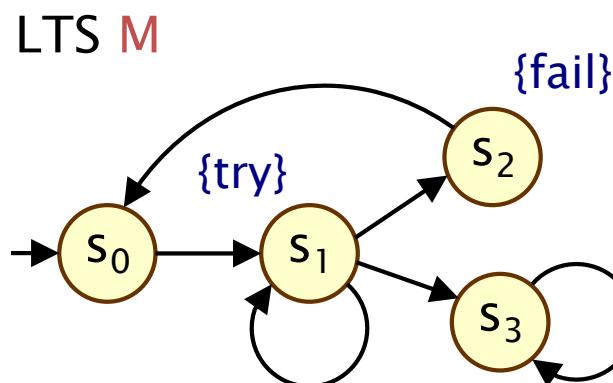
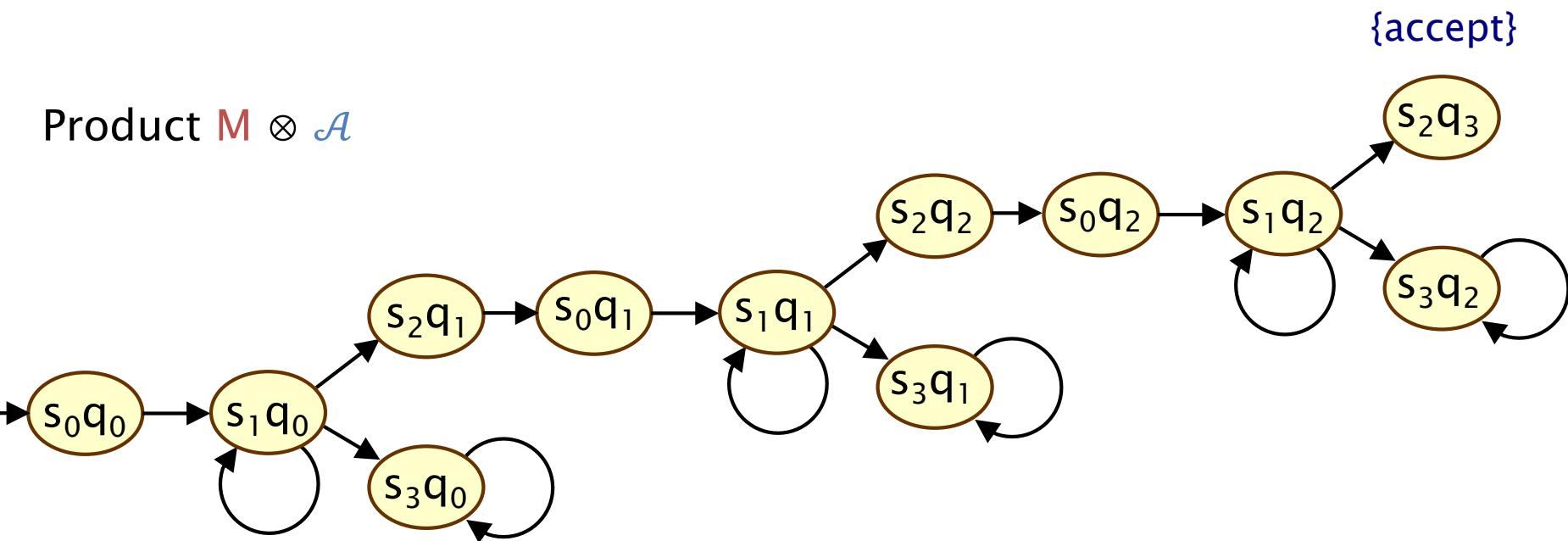


# Model checking

- Given an LTS  $M$  and regular safety property  $P_{\text{safe}}$ 
  - and NFA  $\mathcal{A}$  representing the bad prefixes of  $P_{\text{safe}}$
  - $M \models P_{\text{safe}} \Leftrightarrow \text{Traces}_{\text{fin}}(M) \cap \mathcal{L}(\mathcal{A}) = \emptyset$
- In other words
  - $M \models P_{\text{safe}}$  iff no "accept" state is reachable in  $M \otimes \mathcal{A}$
  - so model checking  $P_{\text{safe}}$  reduces to checking an invariant
- i.e., can be checked through reachability
  - see earlier discussion of invariants
  - also equivalent to checking CTL  $M \otimes \mathcal{A} \models \neg \exists (\text{true} U \text{accept})$

$$M \models P_{\text{safe}} \Leftrightarrow M \otimes \mathcal{A} \models \Box \neg \text{accept}$$

# Example 3 revisited



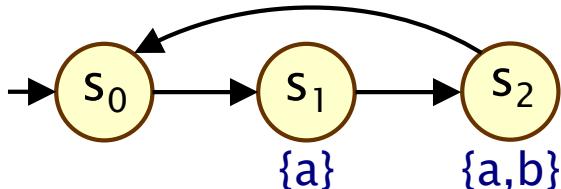
$P_{\text{safe}} = \text{"at most 2 failures occur"}$

$M \not\models P_{\text{safe}}$  (since  $M \otimes \mathcal{A} \not\models \Box \neg \text{accept}$ )

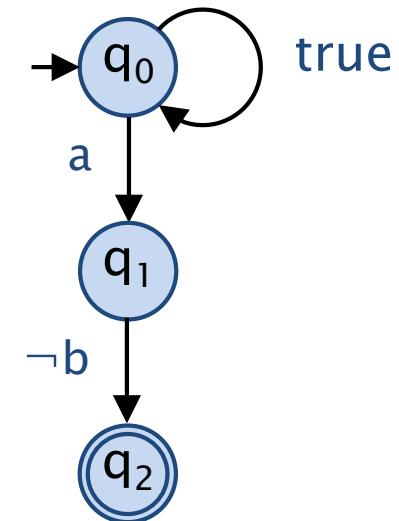
# Example

- Model check  $\square(a \rightarrow \bigcirc b)$  on LTS  $M$

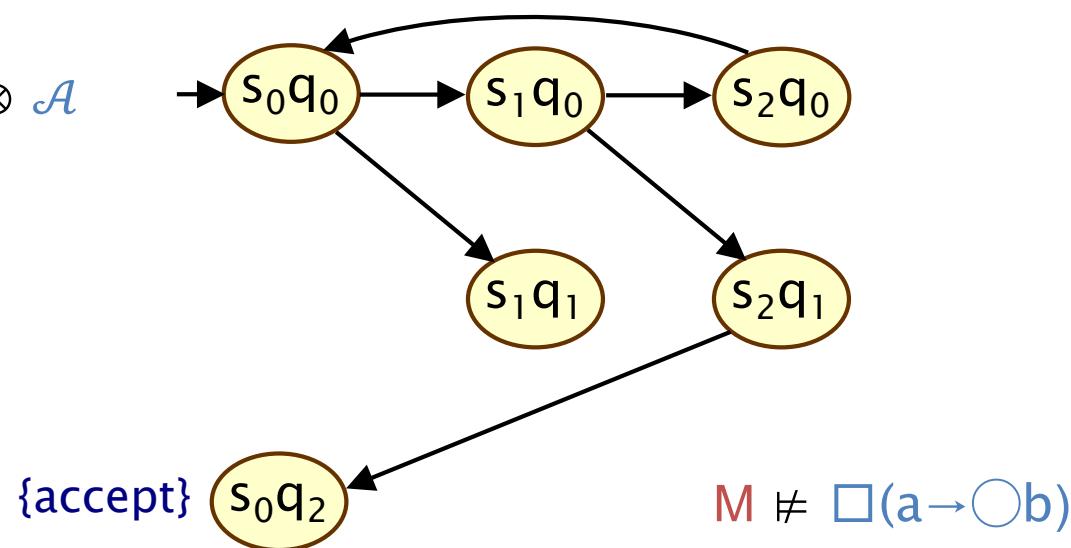
LTS  $M$



NFA  $\mathcal{A}$



Product  $M \otimes \mathcal{A}$

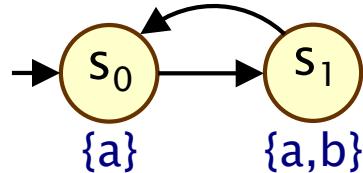


$M \not\models \square(a \rightarrow \bigcirc b)$

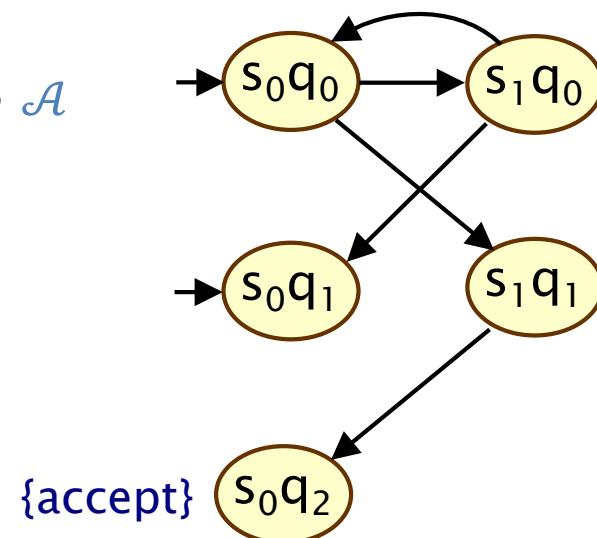
# Example

- Model check  $\square(a \rightarrow \bigcirc b)$  on LTS  $M$

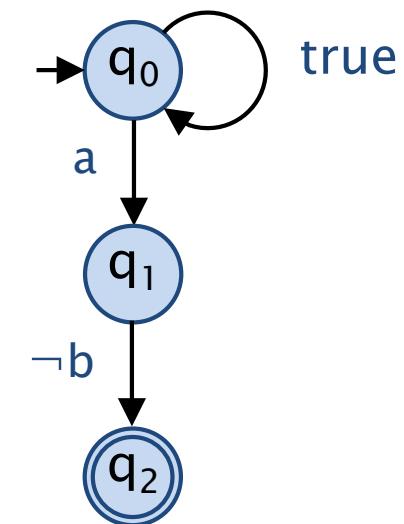
LTS  $M$



Product  $M \otimes \mathcal{A}$



NFA  $\mathcal{A}$



$M \not\models \square(a \rightarrow \bigcirc b)$

# Summary

- Regular safety properties as finite automata
  - bad prefixes represented by an NFA
  - transitions labelled with propositional formulae
- Product of LTS and NFA
  - synchronise state labels of LTS with transitions of NFA
  - represents intersection of possible paths/runs
- Model checking regular safety properties
  - construct product LTS
  - reduces to checking an invariant, i.e. reachability

# Next lecture

- Automata on infinite words
  - see Chapter 4.3,4.4 of [BK08]

# 12. $\omega$ -regular languages



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

# Overview

- Recap (safety properties & NFAs) + examples
- $\omega$ -regular languages/properties
- Nondeterministic Büchi automata (NBAs)
- See [BK08] Sections 4.3–4.4, 5.2

# Recap

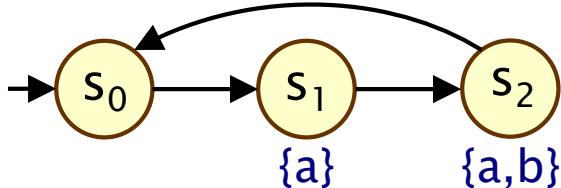
- Model checking regular safety property  $P_{\text{safe}}$  on LTS  $M$ 
  - 1. find NFA  $\mathcal{A}$  representing the bad prefixes of  $P_{\text{safe}}$
  - 2. build LTS-NFA product  $M \otimes \mathcal{A}$
  - 3. check no "accept" state is reachable in  $M \otimes \mathcal{A}$

$$M \models P_{\text{safe}} \iff M \otimes \mathcal{A} \models \Box \neg \text{accept}$$

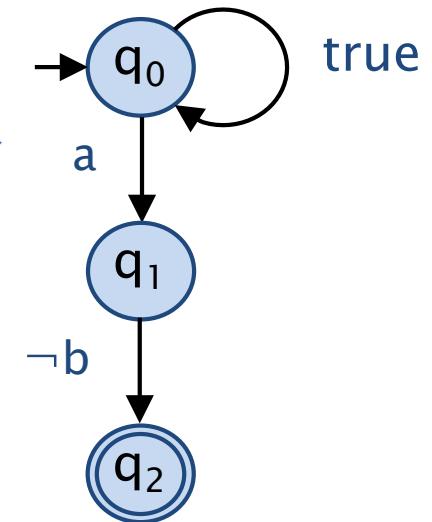
# Example

- Model check  $\square(a \rightarrow \bigcirc b)$  on LTS  $M$

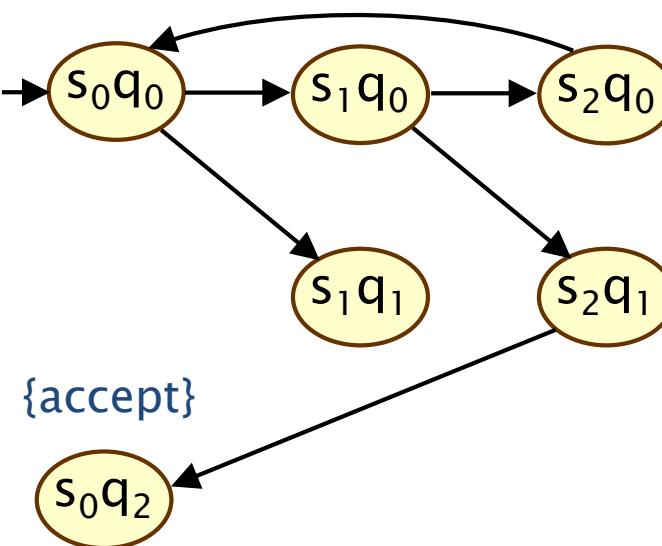
LTS  $M$



NFA  $\mathcal{A}$



Product  $M \otimes \mathcal{A}$

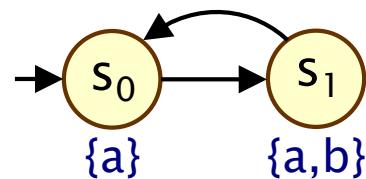


$M \not\models \square(a \rightarrow \bigcirc b)$

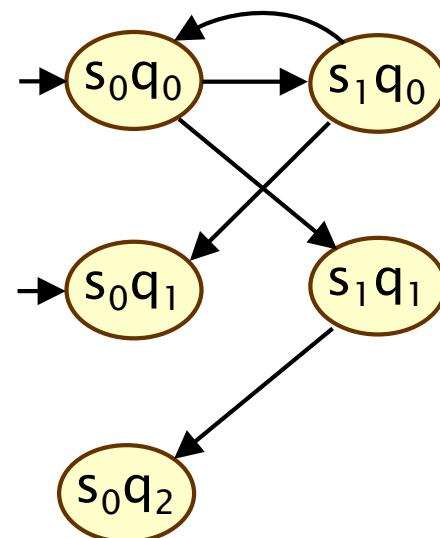
# Example

- Model check  $\square(a \rightarrow \bigcirc b)$  on LTS  $M$

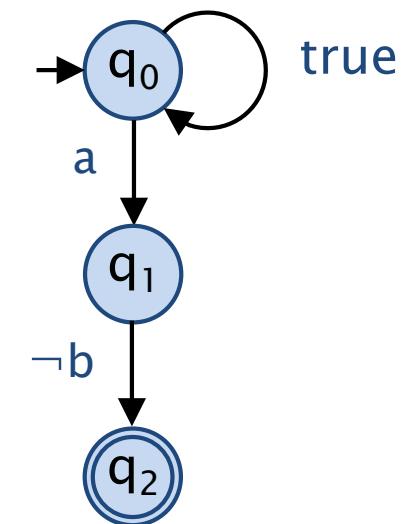
LTS  $M$



Product  $M \otimes \mathcal{A}$



NFA  $\mathcal{A}$

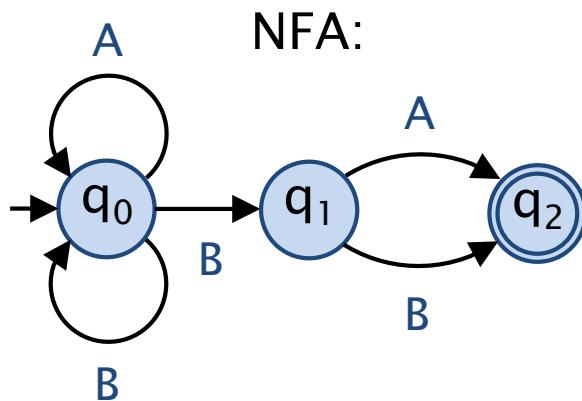


$M \not\models \square(a \rightarrow \bigcirc b)$

# Beyond regular languages

- So far: regular safety properties (e.g. in LTL)
  - ("bad behaviour happens in finite time")
- What about other properties (e.g. in LTL)?
  - liveness, e.g. "for every request, an ack eventually follows"
  - fairness, e.g. "every enabled process is scheduled infinitely often"
- Regular languages:
  - e.g. "penultimate symbol is B"
- This lecture:
  - $\omega$ -regular languages/expressions
  - nondeterministic Büchi automata

Regexp:  $(A+B)^*B(A+B)$



# $\omega$ -regular expressions

- A regular expression  $E$  over alphabet  $\Sigma$  takes the form:
  - $E ::= \emptyset \mid \varepsilon \mid A \mid E + E \mid E \cdot E \mid E^*$  (where  $A \in \Sigma$ )
- An  $\omega$ -regular expression over  $\Sigma$  takes the form:
  - $G = E_1.(F_1)^\omega + E_2.(F_2)^\omega + \dots + E_n.(F_n)^\omega$
  - where  $E_i$  and  $F_i$  are regular expressions with  $\varepsilon \notin \mathcal{L}(F_i)$
- Example:  $(A+B+C)^*(B+C)^\omega$  for  $\Sigma = \{ A, B, C \}$
- $\mathcal{L}_\omega(G) \subseteq \Sigma^\omega$  is the language of an  $\omega$ -regular expression  $G$ 
  - $\mathcal{L}_\omega(G) = \mathcal{L}(E_1).\mathcal{L}(F_1)^\omega \cup \mathcal{L}(E_2).\mathcal{L}(F_2)^\omega + \dots + \mathcal{L}(E_n).\mathcal{L}(F_n)^\omega$
  - where  $\mathcal{L}(E)$  is the language of regular expression  $E$
  - and  $\mathcal{L}(E)^\omega = \{ w^\omega \mid w \in \mathcal{L}(E) \}$

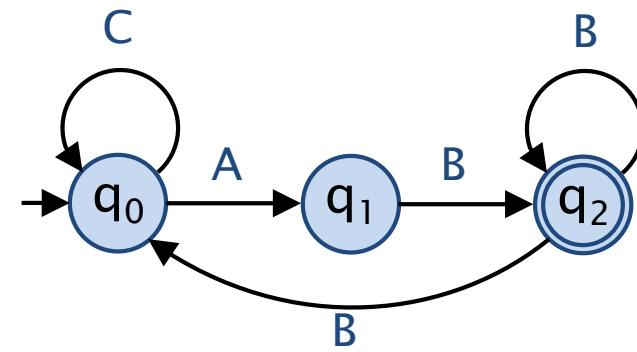
# $\omega$ -regular languages/properties

- Language  $\mathcal{L} \subseteq \Sigma^\omega$  is an  $\omega$ -regular language if
  - $\mathcal{L} = \mathcal{L}_\omega(G)$  for some  $\omega$ -regular expression G
- $P \subseteq (2^{\text{AP}})^\omega$  is an  $\omega$ -regular property
  - if P is an  $\omega$ -regular language over  $2^{\text{AP}}$
- Example (for AP = {wait,crit})
  - e.g.  $((\neg\text{crit})^*\text{crit})^\omega$
  - "crit is true infinitely often"

$\text{crit}$  – shorthand for  $\{\{\text{crit}\}, \{\text{wait}, \text{crit}\}\}$   
 $\neg\text{crit}$  – shorthand for  $\{\emptyset, \{\text{wait}\}\}$
- Note:
  - any regular safety property is an  $\omega$ -regular property
  - all linear-time properties seen so far are  $\omega$ -regular
  - any LTL formula corresponds to an  $\omega$ -regular property

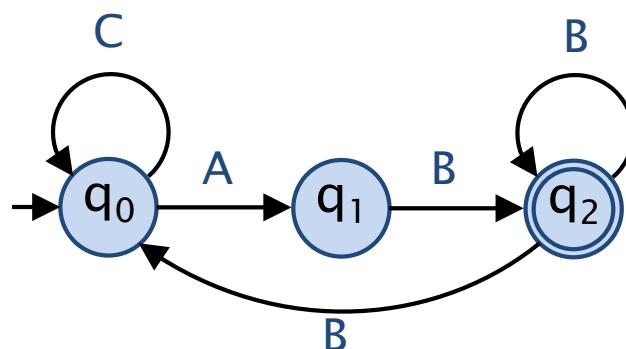
# Nondeterministic Büchi automata

- A nondeterministic Büchi automaton (NBA) is:
  - a tuple  $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$
- where:
  - $Q$  is a finite set of states
  - $\Sigma$  is an alphabet
  - $\delta : Q \times \Sigma \rightarrow 2^Q$  is a transition function
  - $Q_0 \subseteq Q$  is a set of initial states
  - $F \subseteq Q$  is a set of “accept” states
- i.e. NBAs are identical, syntactically, to NFAs
  - the difference is the acceptance condition...
  - “accept” states need to be visited infinitely often



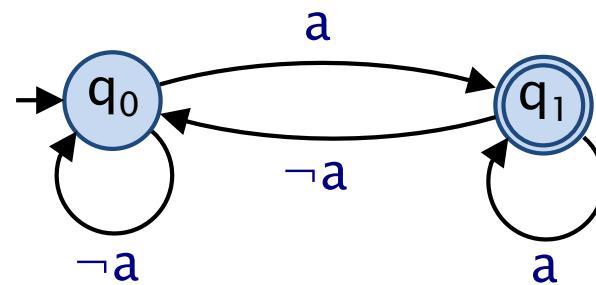
# Language of an NBA

- A run of NBA  $\mathcal{A}$  on an infinite word  $w = A_0A_1A_2\dots$  is:
  - a sequence of automata states  $q_0q_1q_2\dots$  such that:
  - $q_0 \in Q_0$  and  $q_i - A_i \rightarrow q_{i+1}$  for all  $i \geq 0$
- An accepting run is a run with  $q_i \in F$  for infinitely many  $i$
- The language of  $\mathcal{A}$ , denoted  $\mathcal{L}_\omega(\mathcal{A})$  is:
  - the set of all (infinite) words accepted by  $\mathcal{A}$



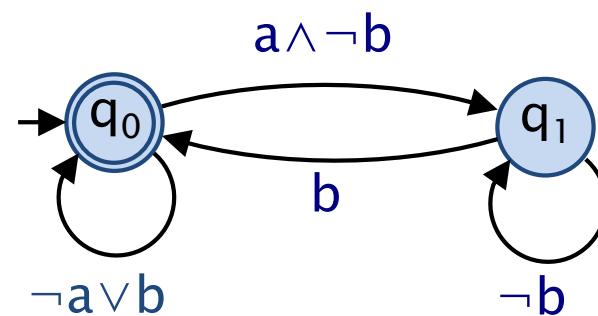
# Example

- "infinitely often a" -  $\Box\Diamond a$



# Example

- "b always follows a" –  $\Box(a \rightarrow \Diamond b)$

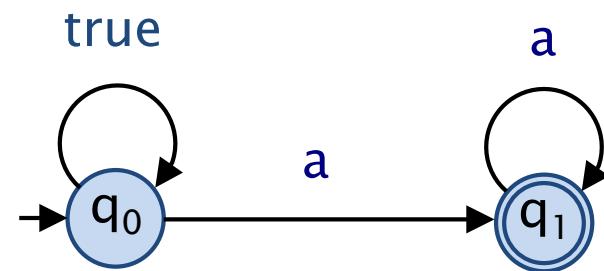


# Nondeterministic Büchi automata

- Represent  $\omega$ -regular languages
  - same expressivity as  $\omega$ -regular expressions
- Can be built systematically from  $\omega$ -regular expressions
- Are an example of  $\omega$ -automata
  - there are many others: Rabin, Streett, Muller, ...
- Are closed under intersection
- Are closed under complementation
- Are more expressive than deterministic Büchi automata
  - unlike for finite automata

# Example

- "eventually always a" –  $\diamond\Box a$



# 13. LTL Model Checking



Computer-Aided Verification

Dave Parker

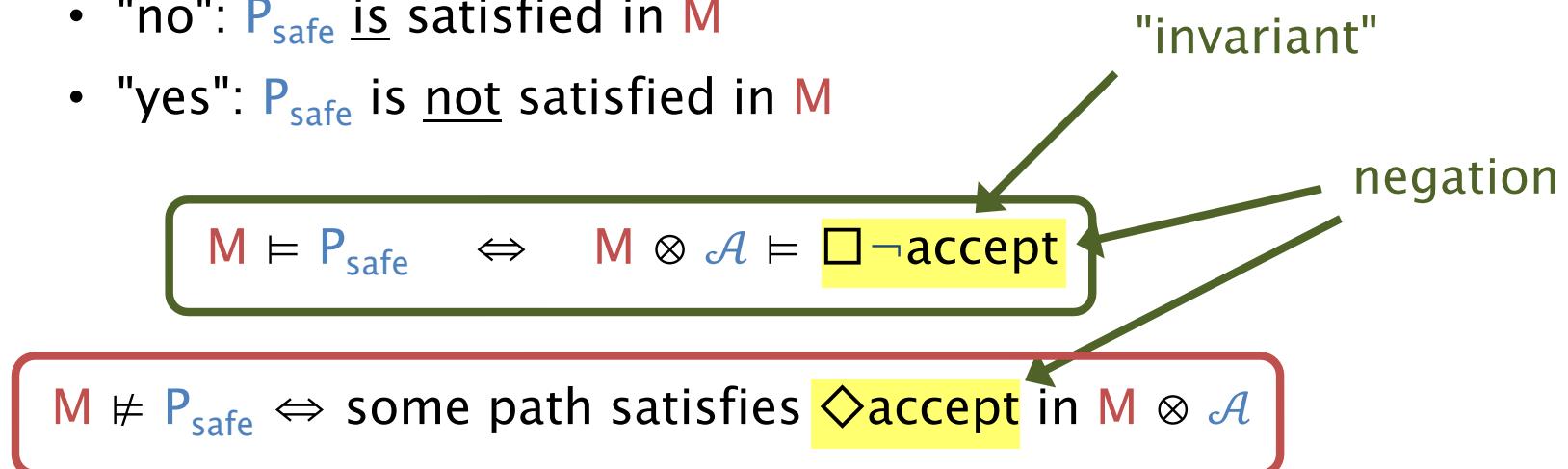
University of Birmingham

2017/18

# Recap: Regular safety properties

- Model checking regular safety property  $P_{\text{safe}}$  on LTS  $M$

- 1. find NFA  $\mathcal{A}$  representing the bad prefixes of  $P_{\text{safe}}$
- 2. build LTS-NFA product  $M \otimes \mathcal{A}$
- 3. is an "accept" state reachable in  $M \otimes \mathcal{A}$ ?
  - "no":  $P_{\text{safe}}$  is satisfied in  $M$
  - "yes":  $P_{\text{safe}}$  is not satisfied in  $M$



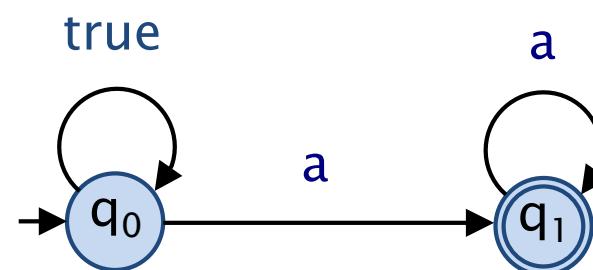
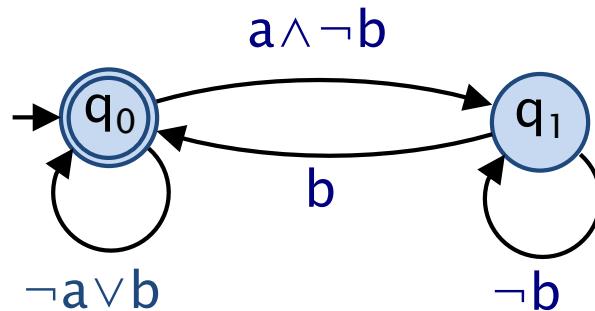
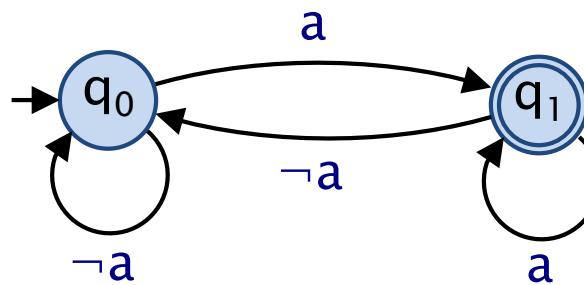
- In this lecture, we generalise this approach
  - to model checking for LTL over LTSs

# Overview

- Nondeterministic Büchi automata (NBAs)
  - to represent  $\omega$ -regular languages, LTL formulae
  - non-blocking NBAs
- LTS–NBA product
- LTL model checking
- See [BK08] Sections 4.3–4.4, 5.2

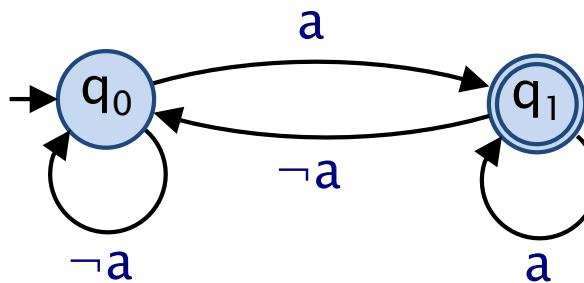
# Nondeterministic Büchi automata

- Nondeterministic Büchi automata (NBAs)
  - represent  $\omega$ -regular languages (e.g. LTL formulae)
- Examples:

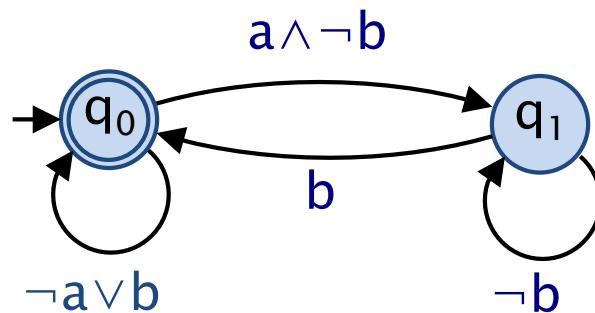


# Nondeterministic Büchi automata

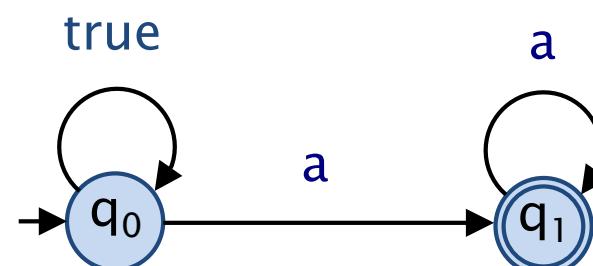
- Nondeterministic Büchi automata (NBAs)
  - represent  $\omega$ -regular languages (e.g. LTL formulae)
- Examples:



"infinitely often a" –  $\square\lozenge a$



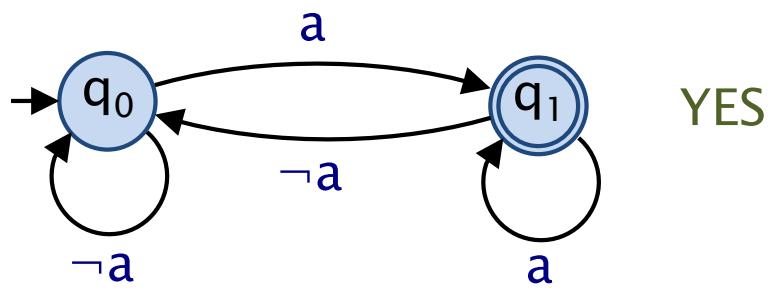
"b always follows a" –  $\square(a \rightarrow \lozenge b)$



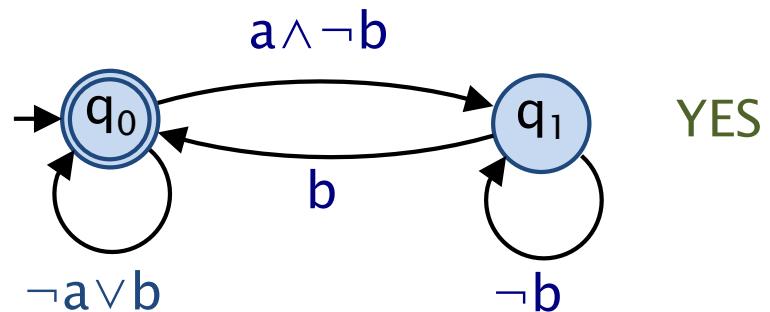
"eventually always a" –  $\lozenge\square a$

# Non-blocking NBAs

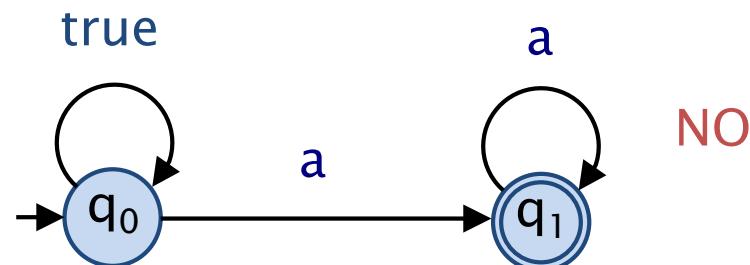
- An NBA  $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$  is non-blocking if:
  - every symbol is available in every state
  - i.e.  $\delta(q, A) \neq \emptyset$  for all states  $q \in Q$  and symbols  $A \in \Sigma$
  - so every infinite word has a run through  $\mathcal{A}$



YES



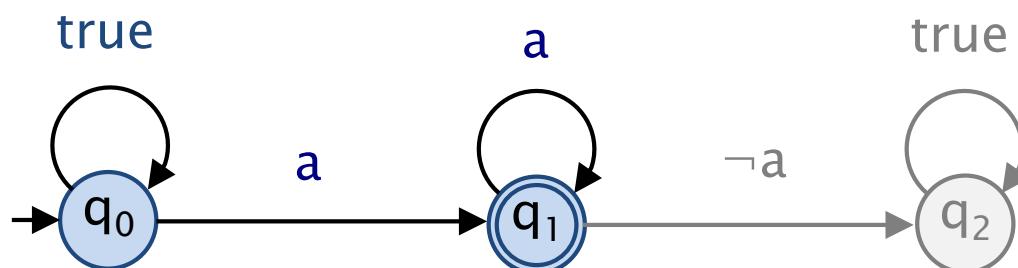
YES



NO

# Non-blocking NBAs

- An NBA  $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$  is non-blocking if:
  - every symbol is available in every state
  - i.e.  $\delta(q, A) = \emptyset$  for all states  $q \in Q$  and symbols  $A \in \Sigma$
  - so every infinite word has a run through  $\mathcal{A}$
- We can always convert to a non-blocking NBA
  - by adding a "trap" state

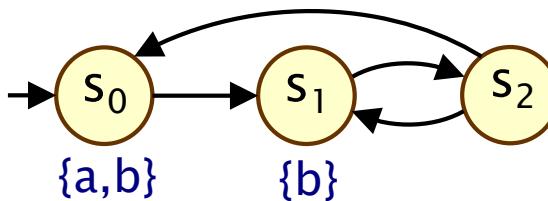


# Product of an LTS and an NBA

- For an LTS  $M$  and a non-blocking NBA  $\mathcal{A}$ 
  - we construct the product of  $M$  and  $\mathcal{A}$ , denoted  $M \otimes \mathcal{A}$
- Identical construction to the case of an NFA
  - synchronous parallel composition
  - transitions of NBA  $\mathcal{A}$  synchronise with state labels of LTS  $M$
  - allows infinite traces/words that are in both  $M$  and  $\mathcal{A}$
- Forms the basis of a model checking procedure
  - of  $\omega$ -regular languages (and thus LTL)

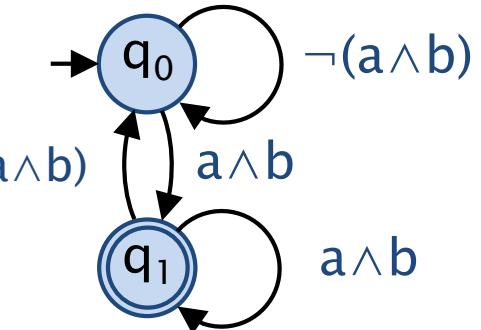
# Example 1 - LTS–NBA product

LTS  $M$

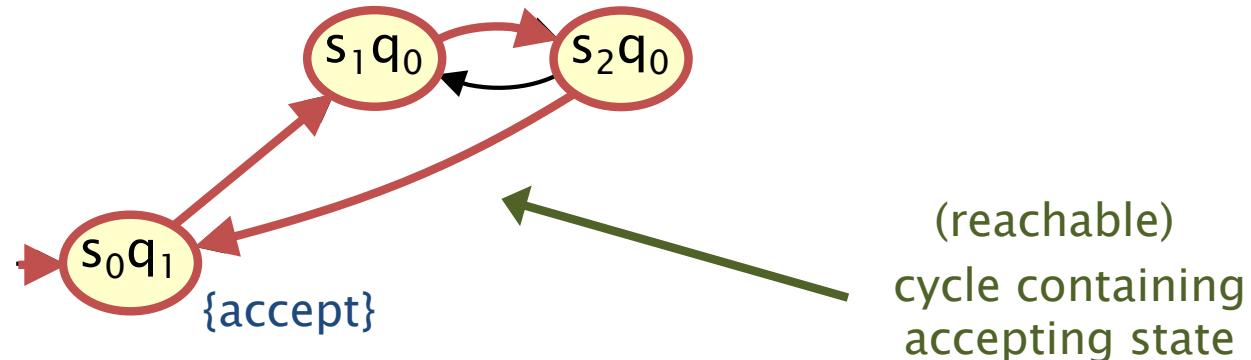


$$\psi = \square \diamond (a \wedge b)$$

NBA  $\mathcal{A}_\psi$



Product  $M \otimes \mathcal{A}$



So: there is a path in  $M$  which satisfies  $\psi$

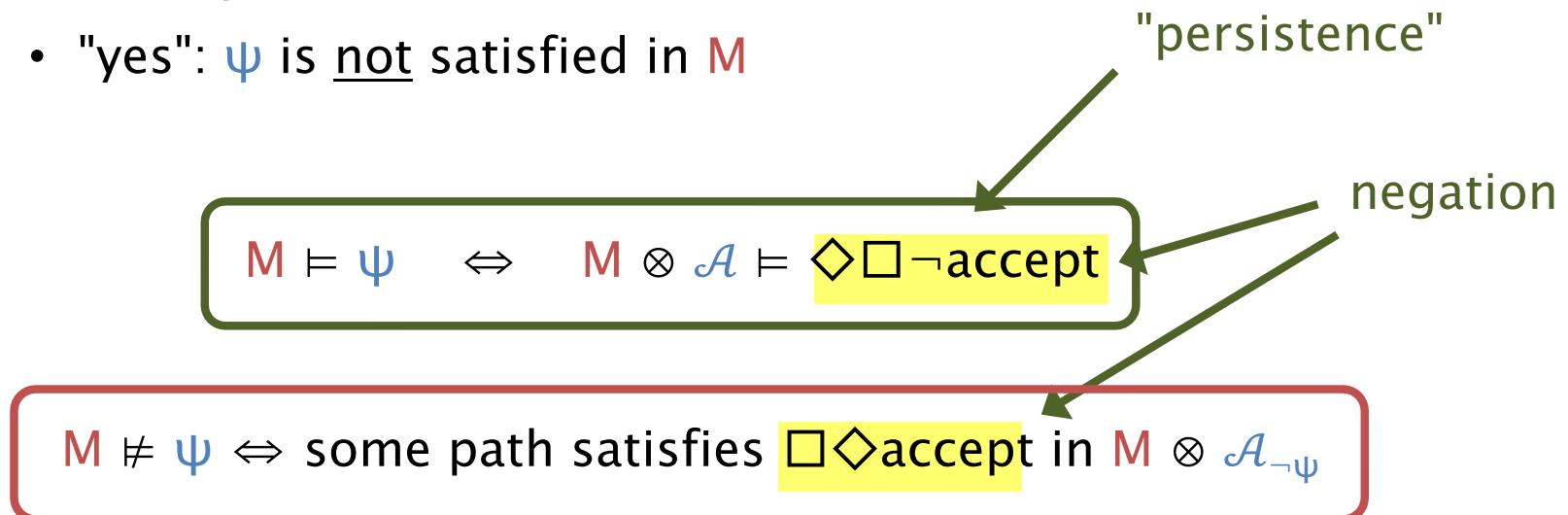
Note: this is not the procedure for model checking  $\psi$  on  $M$

# Model checking LTL

- Given an LTS  $M$  and an LTL formula  $\psi$ 
  - $M \models \psi \Leftrightarrow \text{Traces}(M) \subseteq \text{Words}(\psi)$
- Negating the LTL formula (i.e.,  $\neg\psi$ ), we have:
  - $M \models \psi \Leftrightarrow \text{Traces}(M) \cap \text{Words}(\neg\psi) = \emptyset$
- Given also an NBA  $\mathcal{A}_{\neg\psi}$  representing the formula  $\neg\psi$ :
  - $M \models \psi \Leftrightarrow \text{Traces}(M) \cap \mathcal{L}_\omega(\mathcal{A}_{\neg\psi}) = \emptyset$
- Constructing the product  $M \otimes \mathcal{A}_{\neg\psi}$ :
  - $M \models \psi \Leftrightarrow$  there is no accepting path (cycle) in  $M \otimes \mathcal{A}_{\neg\psi}$

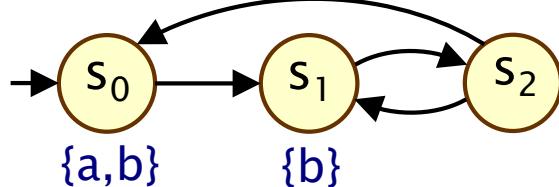
# LTL model checking

- Model checking LTL formula  $\psi$  on LTS  $M$ 
  - 1. find NBA  $\mathcal{A}_{\neg\psi}$  representing the negation  $\neg\psi$  of  $\psi$
  - 2. build LTS-NBA product  $M \otimes \mathcal{A}_{\neg\psi}$
  - 3. is a cycle containing an "accept" state reachable in  $M \otimes \mathcal{A}_{\neg\psi}$ ?
    - "no":  $\psi$  is satisfied in  $M$
    - "yes":  $\psi$  is not satisfied in  $M$

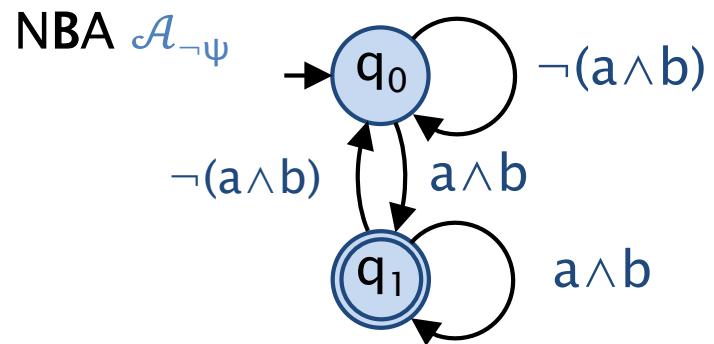


# Example 1 - LTL model checking

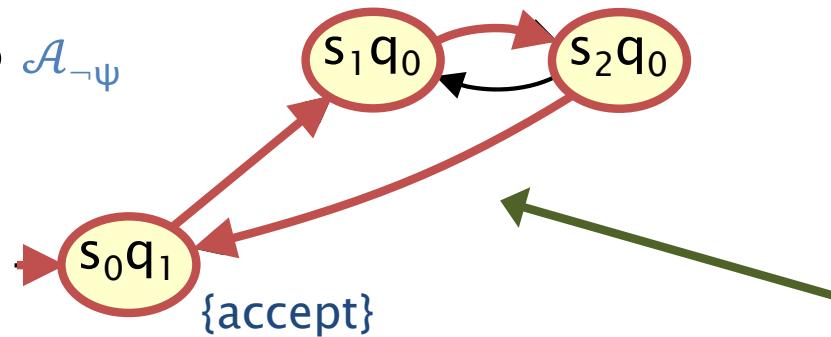
Model check  $\psi = \Diamond \Box \neg(a \wedge b)$  on LTS  $M$



$\neg\psi = \Box \Diamond(a \wedge b)$



Product  $M \otimes \mathcal{A}_{\neg\psi}$

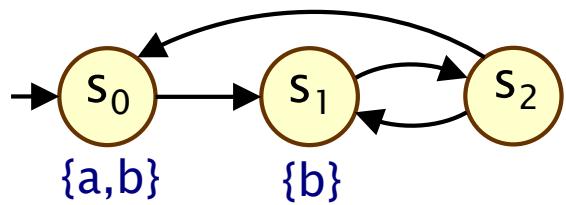


(reachable)  
cycle containing  
accepting state

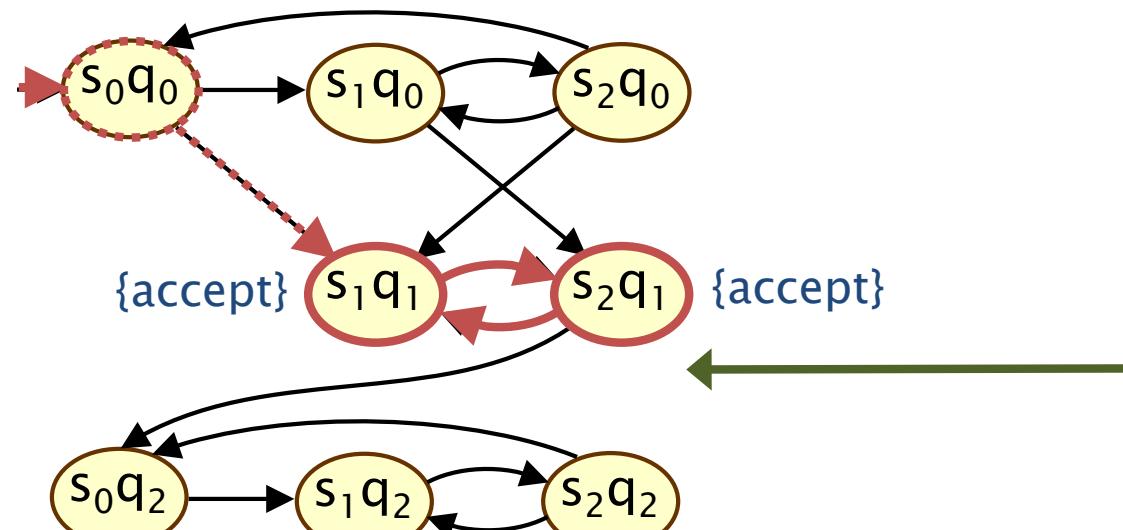
$M \not\models \psi$   
i.e.  $\psi$  not satisfied

# Example 2 - LTL model checking

Model check  $\psi = \square \diamond a$  on LTS  $M$

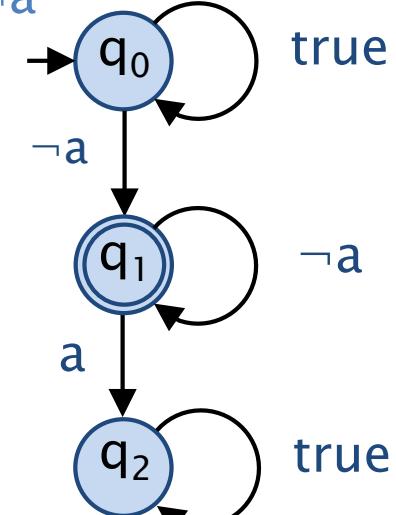


Product  $M \otimes \mathcal{A}_{\neg\psi}$



$\neg\psi = \diamond \square \neg a$

NBA  $\mathcal{A}_{\neg\psi}$



(reachable)  
cycle containing  
accepting state

$M \not\models \psi$

i.e.  $\psi$  not satisfied

# Comparison: Linear-time model checking

Model checking regular safety property  $P_{\text{safe}}$  on LTS  $M$

- 1. NFA  $\mathcal{A}$  for bad prefixes of  $P_{\text{safe}}$
- 2. build product  $M \otimes \mathcal{A}$
- 3. is an "accept" state reachable in  $M \otimes \mathcal{A}$ ?
  - "no":  $P_{\text{safe}}$  is satisfied in  $M$
  - "yes":  $P_{\text{safe}}$  not satisfied in  $M$

$$M \models P_{\text{safe}} \Leftrightarrow M \otimes \mathcal{A} \models \Box \neg \text{accept}$$

$M \not\models P_{\text{safe}} \Leftrightarrow$  some path satisfies  $\Diamond \text{accept}$  in  $M \otimes \mathcal{A}$

Model checking LTL formula  $\psi$  on LTS  $M$

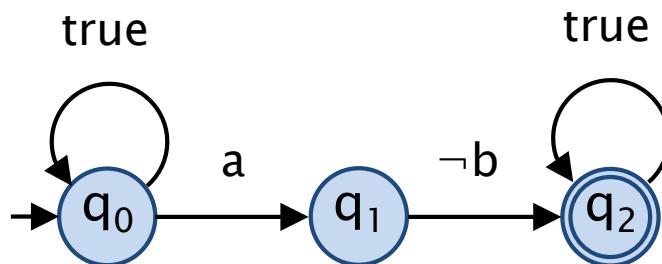
- 1. NBA  $\mathcal{A}_{\neg \psi}$  for  $\neg \psi$
- 2. build product  $M \otimes \mathcal{A}_{\neg \psi}$
- 3. is an "accept" cycle reachable in  $M \otimes \mathcal{A}_{\neg \psi}$ ?
  - "no":  $\psi$  is satisfied in  $M$
  - "yes":  $\psi$  not satisfied in  $M$

$$M \models \psi \Leftrightarrow M \otimes \mathcal{A} \models \Diamond \Box \neg \text{accept}$$

$M \not\models \psi \Leftrightarrow$  some path satisfies  $\Box \Diamond \text{accept}$  in  $M \otimes \mathcal{A}_{\neg \psi}$

# Regular safety properties

- Regular safety properties
  - are a subclass of  $\omega$ -regular properties
  - and many can be represented as LTL
  - so LTL model checking also works there (but is more costly)
- Recall the example:  $\square(a \rightarrow \bigcirc b)$ 
  - an NBA for its negation is...



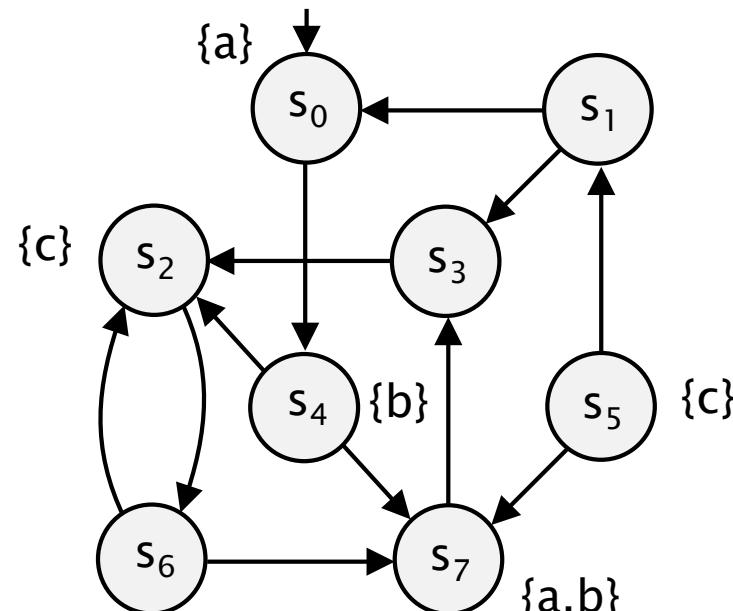
# LTL model checking: Algorithms

- LTL-to-automaton translation
  - various algorithms, tools exist (not covered on this module)
  - (See [BK08] Section 5.2)
- Cycle detection – various options
  - 1. search for reachable non-trivial SCCs containing "accept"
  - 2. find all "accept" states, perform DFS to find back edges
  - 3. nested depth-first search
  - (See [BK08] Section 4.4.2)

# Assm 2 Qu 2

- Recall this example from Assm 2...

- LTS  $M$



- $M \models \Box \Diamond ((a \wedge \neg b) \vee \neg c)$
- $M \not\models \Box \Diamond (a \wedge b)$

# Complexity of LTL model checking

- The time complexity of LTL model checking
  - for LTS  $M$  and LTL formula  $\Psi$
- is:  $O(|M| \cdot 2^{|\Psi|})$ 
  - i.e. linear in model and exponential in formula size
  - where  $|M| =$  number of states + number of transitions in  $M$
  - and  $|\Psi| =$  number of operators in  $\Psi$
- Worst-case execution:
  - there are LTL formulas  $\Psi$  whose NBA  $\mathcal{A}_{\neg\Psi}$  is of size  $O(2^{|\Psi|})$
  - the product to be analysed is  $|M| \cdot |\mathcal{A}_{\neg\Psi}|$
  - checking for cycles can be done in linear time (nested DFS)

# Summary

- LTL model checking procedure
  - convert negation of formula to an equivalent NBA
  - construct LTS-NBA product
  - look for cycles containing accept state in the product
- Same basic idea as for regular safety properties
  - (negation + automaton + product + search)
- Next time
  - counterexamples, fairness, complexity, state space explosion, ...

# 14. Model Checking



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

# Assignments

- Assignment 3 (model checking and automata)
  - out now; due 12 noon next Thur (1 Mar)
  - tutorials in week 9 (8/9 Mar)
- Assignment 5 (“extended” version only)
  - due Thur of week 10 (15 March)
  - released early; out now

# Module syllabus

- Modelling sequential and parallel systems
  - labelled transitions systems, parallel composition
- Temporal logic
  - LTL, CTL and CTL\*, etc.
- Model checking
  - CTL model checking algorithms
  - automata-theoretic model checking (LTL)
- Verification tools: SPIN
- Advanced verification techniques
  - bounded model checking via propositional satisfiability
  - (symbolic execution), (symbolic model checking)
- Quantitative verification
  - (real-time systems), probabilistic systems

# Overview

- Model checking
  - strengths & weaknesses
  - example applications
- Counterexamples
  - evidence of property refutation
  - or witness to desired behaviour
- Complexity & scalability
  - model size is crucial, many approaches to tackle this

# LTL model checking: Summary

- Model checking algorithm
  - construct NBA  $\mathcal{A}_{\neg\Psi}$  for negation of formula  $\Psi$  to be verified
  - check for reachable "accept" cycles in product  $M \otimes \mathcal{A}_{\neg\Psi}$
- LTL-to-automaton translation
  - various algorithms, tools exist (not covered on this module)
- Cycle detection – various options
  - 1. search for reachable non-trivial SCCs containing "accept"
  - 2. find all "accept" states, perform DFS to find back edges
  - 3. nested depth-first search (DFS)

# Complexity of LTL model checking

- The time complexity of LTL model checking
  - for LTS  $M$  and LTL formula  $\Psi$
- is:  $O(|M| \cdot 2^{|\Psi|})$ 
  - i.e. linear in model and exponential in formula size
  - where  $|M| =$  number of states + number of transitions in  $M$
  - and  $|\Psi| =$  number of operators in  $\Psi$
- Worst-case execution:
  - there are LTL formulas  $\Psi$  whose NBA  $\mathcal{A}_{\neg\Psi}$  is of size  $O(2^{|\Psi|})$
  - the product to be analysed is  $|M| \cdot |\mathcal{A}_{\neg\Psi}|$
  - checking for cycles can be done in linear time (nested DFS)

# CTL & LTL model checking

- Model checking – key ideas
  - LTSs to model nondeterministic/concurrent systems
  - temporal logics to formally define behaviour
  - relationships between logic & automata
- CTL/LTL model checking – differences
  - CTL: recursive descent + backwards model search
  - LTL: automata-based + cycle detection
  - CTL model checking simple and lower complexity
- CTL/LTL model checking – common themes
  - reduce a hard problem to an instance of a simpler one
  - reduce checking of “good” executions to a search for a “bad” one
  - reduction to basic graph algorithms

# Model checking: Pros & cons

- **Strengths**
  - exhaustive analysis, sound mathematical underpinning
  - fully automated, tool support, limited expertise required
  - general verification approach, broadly applicable
  - partial system verification (property-based)
  - diagnostic information ([counterexamples](#)) in case of errors
- **Weaknesses**
  - [scalability](#) (state-space explosion)
  - verifies only the stated requirements, not total correctness
  - verifies a [model](#) of the system, not the actual system
  - developing appropriately abstract models may require expertise

# Verification in practice

- (see Canvas page for links to papers)
- SLAM: model checking for device drivers in Windows
  - more generally: checking for client violation of APIs
  - simple examples: spinlock must be locked/unlocked in strict alternation; a file can be read only after it is opened.
  - uses “counterexample-guided abstraction refinement”
- Example application domains
  - NASA Martian Rover control software
  - safety and dependability of satellite control software
  - breaking/fixing the Needham–Schroeder public–key protocol
  - Facebook: static analysis of code errors in a rapid release cycle

# Counterexamples

# Example

- Recall this simple concurrent program:

```
process Inc = while true do if x < 200 then x := x + 1 od

process Dec = while true do if x > 0 then x := x - 1 od

process Reset = while true do if x = 200 then x := 0 od
```

- Property specification:
  - variable  $x$  always remains in the range  $\{0, 1, \dots, 200\}$
  - i.e., the invariant  $\Box \text{safe}$  where  $\text{safe}$  means  $0 \leq x \leq 200$
- Property is false (not satisfied)
  - evidenced by a path which reaches a state where  $x = -1$

# Example – counterexample

- Counterexample trace produced by the model checker:

```
.....
605: proc 1 (Inc) [((x<200))]
606: proc 1 (Inc) [x = (x+1)]
607: proc 2 (Dec) [((x > 0))]
608: proc 1 (Inc) [(1)]
609: proc 3 (Reset) line 13 "pan_in" (state 2) [((x==200))]
610: proc 3 (Reset) line 13 "pan_in" (state 3) [x=0]
611: proc 3 (Reset) line 13 "pan_in" (state 1) [(1)]
612: proc 2 (Dec) line 5 "pan_in" (state 3) [x = (x-1)]
613: proc 2 (Dec) line 5 "pan_in" (state 1) [(1)]
spin: line 17 "pan_in", Error: assertion violated spin: text of failed
assertion: assert(((x>=0)&&(x<=200)))
```

# Counterexamples

- Counterexample for  $M \not\models \Psi$  where  $\Psi$  is an LTL formula
  - path  $\pi$  of  $M$  that refutes  $\Psi$  (i.e. indicates why  $\Psi$  is false)
  - in practice: sufficiently long prefix of  $\pi$  showing why  $\pi$  refutes  $\Psi$
- Examples
  - counterexample for  $\Box a$  ?     • finite path ending in  $\neg a$
  - counterexample for  $\Diamond a$  ?     • 2-state path ending in  $\neg a$
  - counterexample for  $\lozenge a$  ?
    - finite prefix of  $\neg a$  states followed by single cycle of  $\neg a$  states
  - counterexample for  $a \mathbf{U} b$  ?
    - finite path of  $a \wedge \neg b$  states ending in  $\neg a \wedge \neg b$
    - or finite prefix of  $a \wedge \neg b$  states followed by cycle of  $a \wedge \neg b$  states

# Diversion: LTL model checking of $a \cup b$

- Formula to be verified

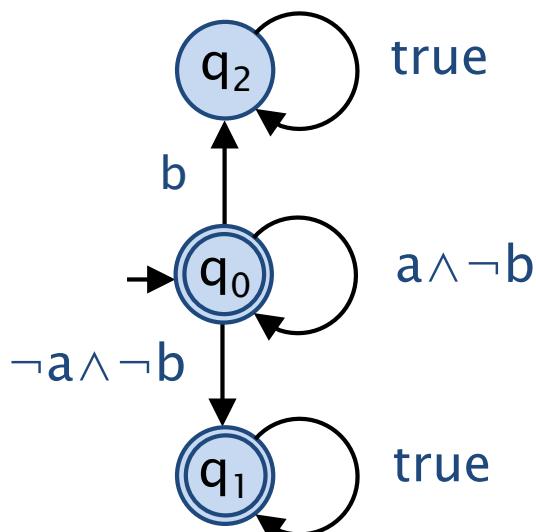
- $\psi = a \cup b$

- Negation

- $\neg\psi = \neg(a \cup b)$

- Automaton

- NBA  $\mathcal{A}_{\neg\psi}$



# Counterexamples

- Counterexample for  $M \not\models \Psi$  where  $\Psi$  is an LTL formula
  - path  $\pi$  of  $M$  that refutes  $\Psi$  (i.e. indicates why  $\Psi$  is false)
  - in practice: sufficiently long prefix of  $\pi$  showing why  $\pi$  refutes  $\Psi$
- Examples
  - counterexample for  $\Box a$  ?     • finite path ending in  $\neg a$
  - counterexample for  $\Diamond a$  ?     • 2-state path ending in  $\neg a$
  - counterexample for  $\lozenge a$  ?
    - finite prefix of  $\neg a$  states followed by single cycle of  $\neg a$  states
  - counterexample for  $a \cup b$  ?
    - finite path of  $a \wedge \neg b$  states ending in  $\neg a \wedge \neg b$
    - or finite prefix of  $a \wedge \neg b$  states followed by cycle of  $a \wedge \neg b$  states
  - counterexample for arbitrary LTL formula?
    - finite prefix plus cycle, extracted from LTS–NBA product

# Counterexamples (and witnesses)

- Counterexample for  $M \not\models \phi$  where  $\phi$  is an CTL formula
  - depends on path quantifier  $\forall / \exists$
- For formulae of the form  $\forall \psi$  (e.g.,  $\phi = \forall \Box a$ )
  - same as for LTL, just discussed
  - (ignoring nested operators)
- For formulae of the form  $\exists \psi$  (e.g.,  $\phi = \exists \Box a$ )
  - there may be no convenient form of counterexamples
  - but it is often natural to provide witnesses when  $M \models \phi$
  - a witness is a path giving an example of how  $\psi$  can be true
  - a witness for  $\psi$  is a counterexample for  $\neg \psi$

# 15. The SPIN Model Checker



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

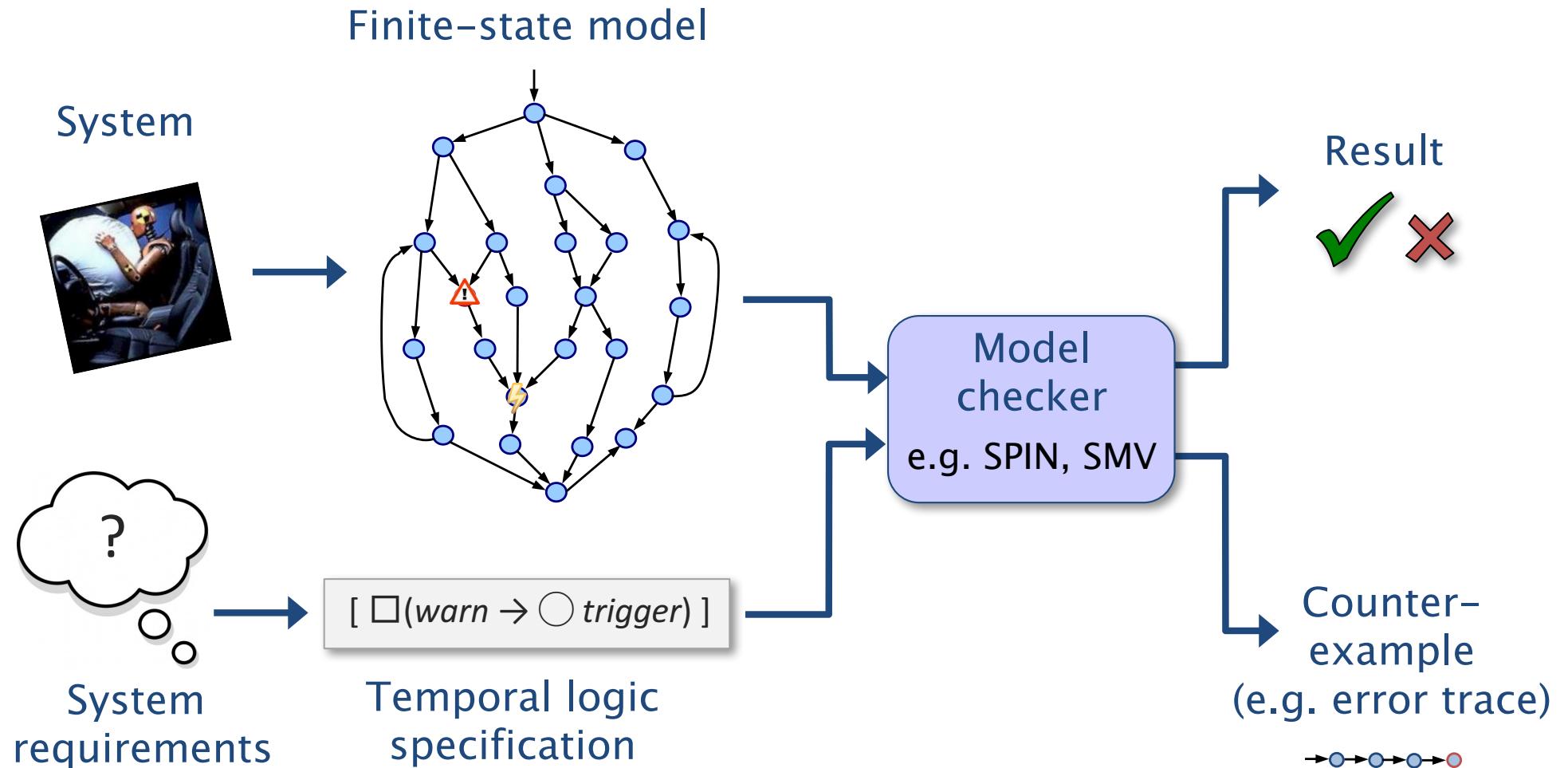
# Module syllabus

- Modelling sequential and parallel systems
  - labelled transitions systems, parallel composition
- Temporal logic
  - LTL, CTL and CTL\*, etc.
- Model checking
  - CTL model checking algorithms
  - automata-theoretic model checking (LTL)
- Verification tools: SPIN
- Advanced verification techniques
  - bounded model checking via propositional satisfiability
  - (symbolic execution), (symbolic model checking)
- Quantitative verification
  - (real-time systems), probabilistic systems

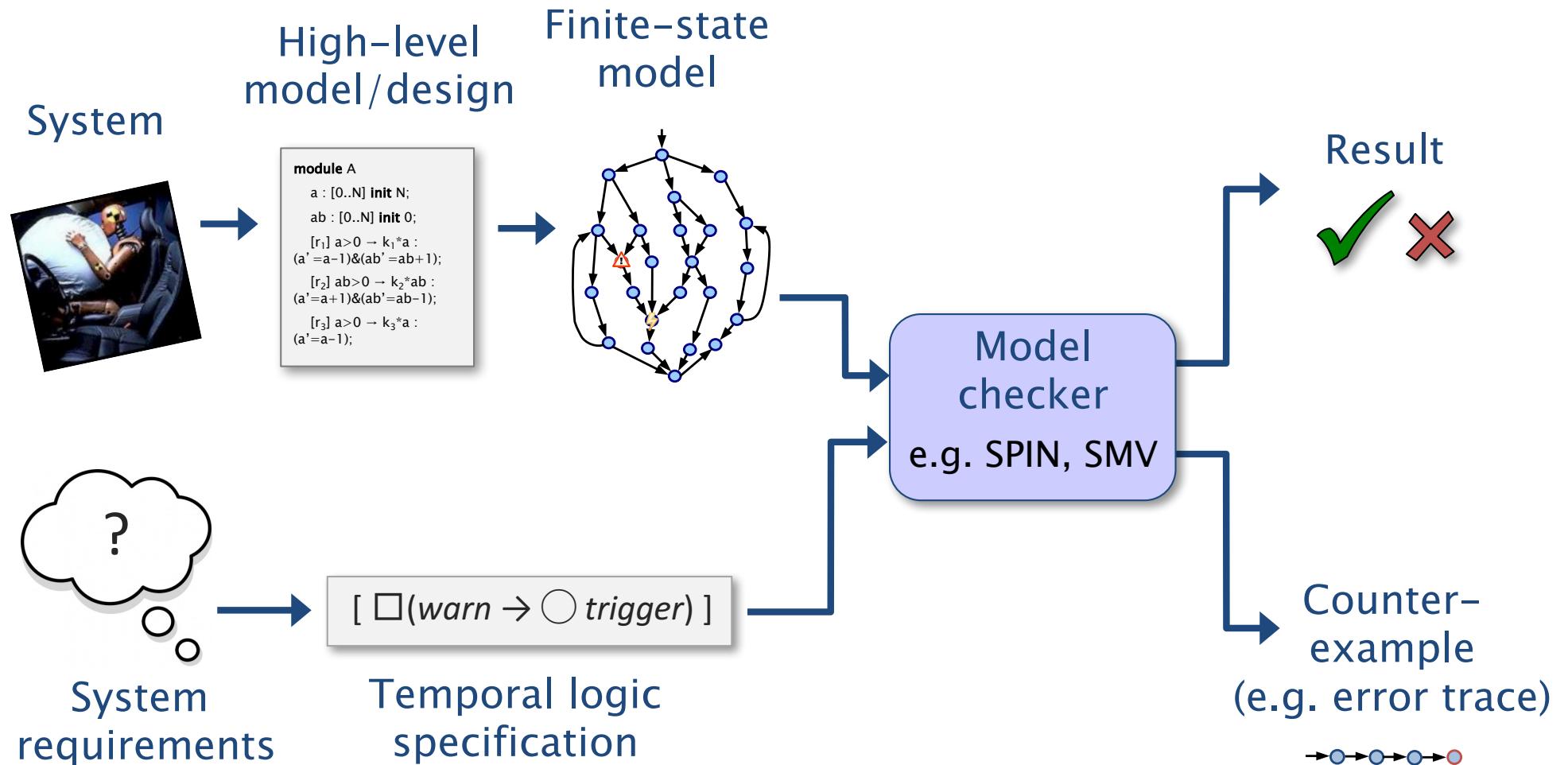
# Overview

- The SPIN model checker
  - overview
  - ProMeLa
  - demo
  - examples
- Background reading & reference:
  - basic manual: <http://spinroot.com/spin/Man/Manual.html>
  - tool options: <http://spinroot.com/spin/Man/Spin.html>

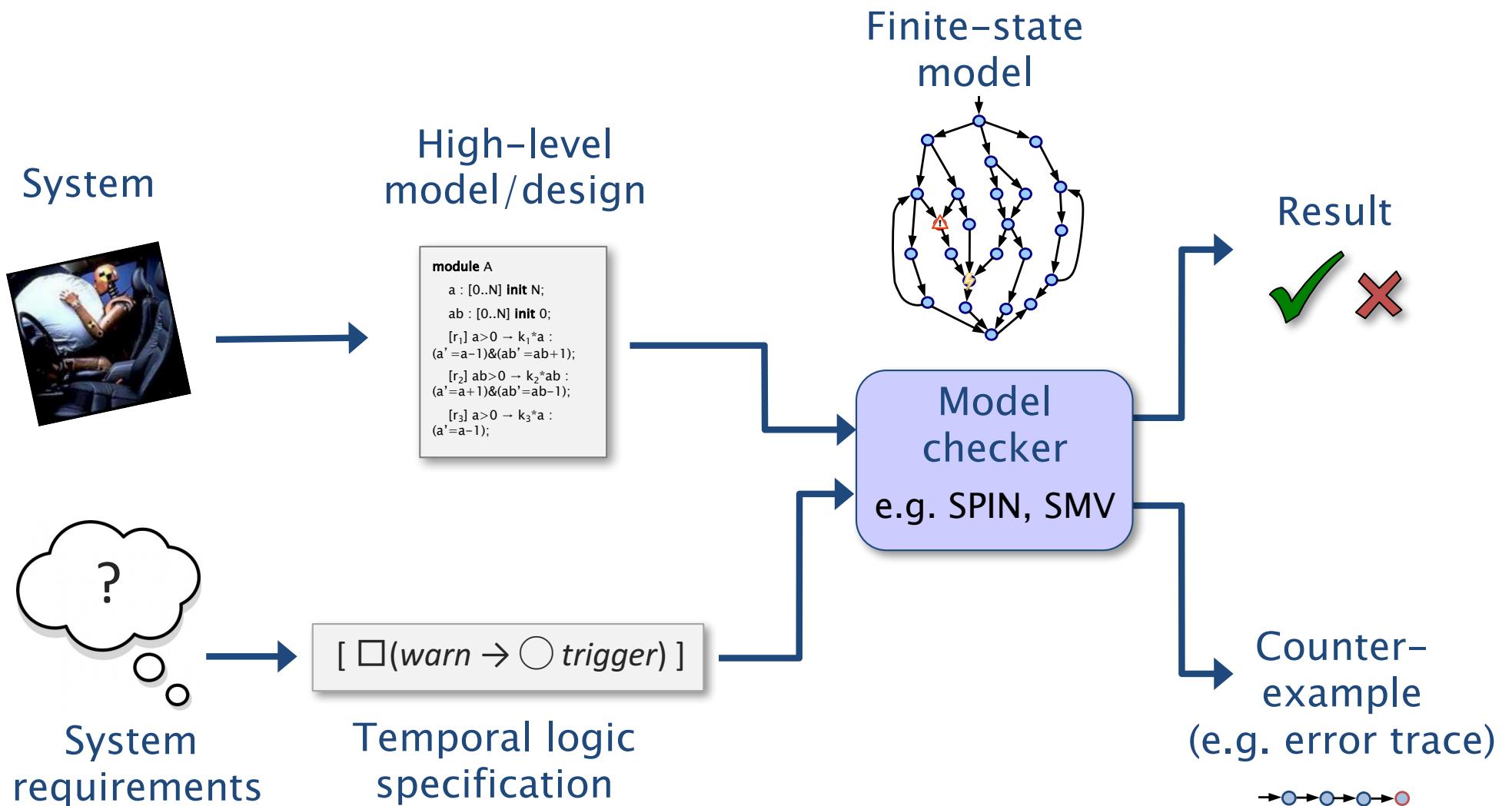
# Verification via model checking



# Verification via model checking



# Verification via model checking



# High-level model description

- Example concurrent program:

```
process Inc = while true do if x < 200 then x := x + 1 fi od

process Dec = while true do if x > 0 then x := x - 1 fi od

process Reset = while true do if x = 200 then x := 0 fi od
```

- ProMeLa/SPIN model (fragment):

```
int x=0;
proctype Inc () {
 do :: true -> if :: (x < 200) -> x = x + 1 fi od
}
proctype Dec() {
 do :: true -> if :: (x > 0) -> x = x - 1 fi od
}
...
```

# SPIN

- SPIN model checker
  - prominent, widely used verification tool
  - open source, freely available (<http://spinroot.com/>)
  - developed by Gerald Holzmann at Bell Labs
  - many success stories: NASA mission software (Deep Space 1, Mars Exploration Rovers), Toyota control software investigation
- Key features
  - custom modelling language: ProMeLa
  - on-the-fly model checking for safety, liveness, LTL
    - verification vs falsification (bug hunting)
    - state storage using hash table of lists of states
  - simulator (random, interactive, guided)
  - separate user interface (iSpin)

# ProMeLa

- ProMeLa: Process Meta Language
  - modelling language for the SPIN tool
- Key ingredients
  - **processes** (one or more, in parallel)
  - typed **data** variables (can be shared, for communication)
  - **channels** (synchronous/asynchronous communication)
- Language notation
  - **guarded commands** (nondeterministic choice)
  - plus **imperative**-style control flow (and embedded C)
- Semantics
  - labelled transition systems (LTSs), via program graphs

# ProMeLa – Guarded commands

- Guarded commands

```
:: guard -> statement
```

- execute statement if guard is true

- Conditionals

```
if :: guard -> statement1
 :: !guard -> statement2
fi
```

- if (guard) then statement<sub>1</sub> else statement<sub>2</sub>

Nondeterminism

- Loops

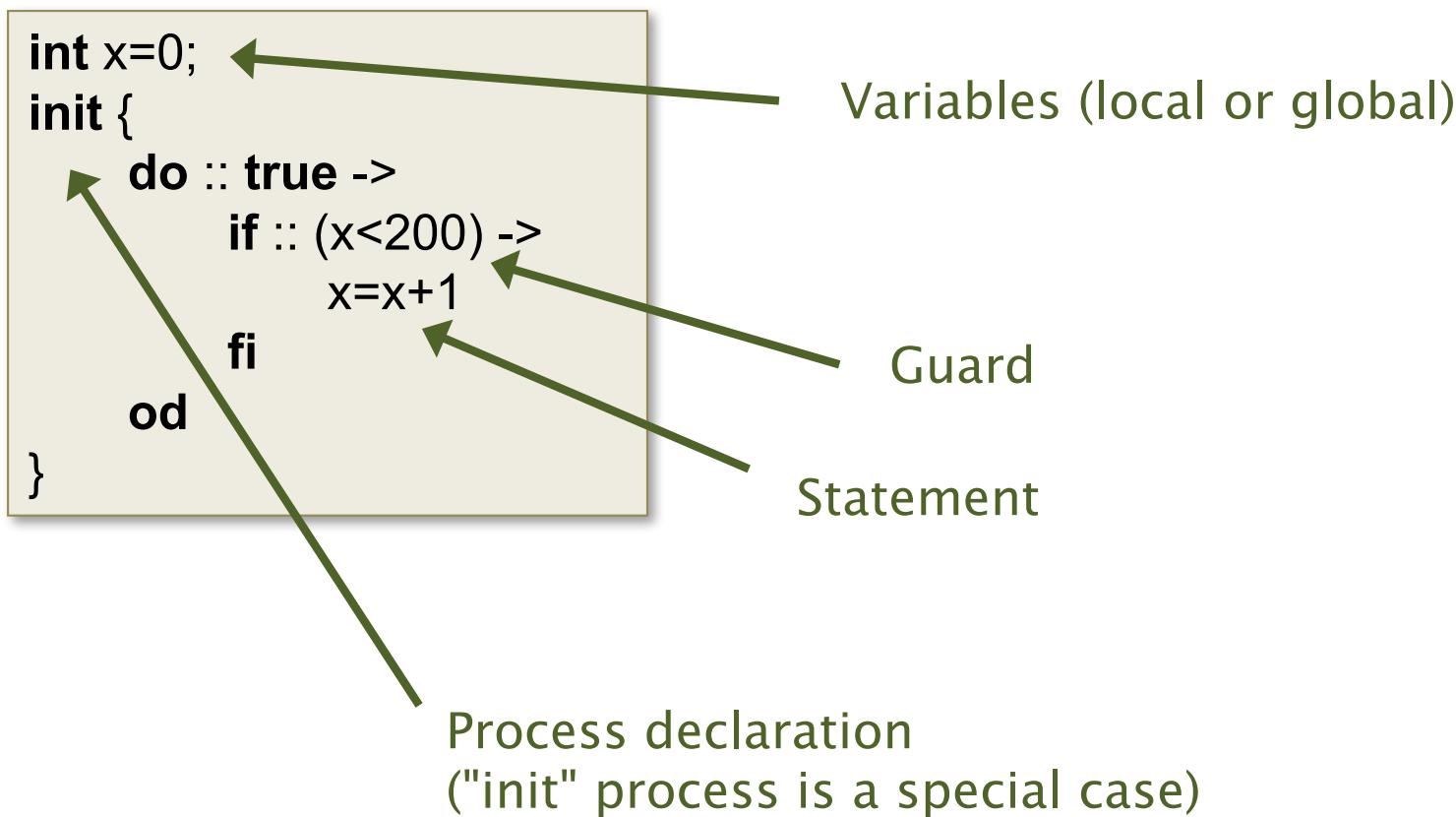
```
do :: guard -> statement od
```

- while (guard) do statement

```
do :: guard1 -> statement1
 :: guard2 -> statement2
od
```

# ProMeLa – Example

- One process, one variable (integer x)



# Demo

# ProMeLa – Concurrent program

- Earlier example:

```
int x = 0;

proctype Inc() {
 do :: true -> if :: (x < 200) -> x = x + 1 fi od
}

proctype Dec() {
 do :: true -> if :: (x > 0) -> x = x - 1 fi od
}

proctype Reset() {
 do :: true -> if :: (x==200) -> x=0 fi od
}

init {
 run Inc() ; run Dec() ; run Reset()
}
```

# ProMeLa – Concurrent program

- Add a "monitor" process that checks whether  $0 \leq x \leq 200$

```
int x = 0;
proctype Inc() {
 do :: true -> if :: (x < 200) -> x = x + 1 fi od
}
proctype Dec() {
 do :: true -> if :: (x > 0) -> x = x - 1 fi od
}
proctype Reset() {
 do :: true -> if :: (x==200) -> x=0 fi od
}
proctype Check () {
 assert (x >= 0 && x <= 200)
}
init {
 run Inc() ; run Dec() ; run Reset() ; run Check()
}
```

# 17. Bounded Model Checking



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

# Module syllabus

- Modelling sequential and parallel systems
  - labelled transitions systems, parallel composition
- Temporal logic
  - LTL, CTL and CTL\*, etc.
- Model checking
  - CTL model checking algorithms
  - automata-theoretic model checking (LTL)
- Verification tools: SPIN
- Advanced verification techniques
  - bounded model checking via propositional satisfiability
  - (symbolic execution), symbolic model checking
- Quantitative verification
  - (real-time systems), probabilistic systems

# Overview (next 2 lectures)

- Motivation & overview
  - model checking & scalability
  - bounded model checking via satisfiability
- Propositional logic and satisfiability (SAT)
- Encoding simple (Boolean variable) models with prop. logic
  - model checking invariants via SAT solving
- Software model checking via bounded model checking & SAT
  - loop unwinding
  - single static assignment
  - predicate logic

# Scalability of model checking

- Model checking complexity
  - $O(|M| \cdot |\Phi|)$  for CTL and  $O(|M| \cdot 2^{|\Psi|})$  for LTL
- Key issue: "state space explosion problem"
  - the size of the model  $M$  for real systems (e.g. software)
  - model size is exponential in the size of the model description
- Many model checking problems reduce to reachability
  - standard graph traversal,
  - e.g. depth-first/breadth-first search
- Efficiency & scalability
  - storage/look-up of visited states crucial
  - SPIN uses hash table of lists of states

# Scalability of model checking

- Many solutions for scalability (and efficiency) proposed
  - abstraction (and automated generation of)
  - symmetry reduction
  - partial order reduction
  - symbolic model checking (binary decision diagrams)
  - on-the fly model checking
  - bounded model checking via propositional satisfiability
- Issues to consider:
  - symbolic vs. explicit-state model checking
  - verification vs falsification (bug hunting)
  - soundness, completeness

# Bounded model checking

- Key idea
  - unroll model (e.g. control flow graph) for fixed number of steps  $k$
  - construct a propositional formula which is satisfiable if and only if there is an error within  $k$  steps
  - reduce model checking problem to satisfiability (SAT) problem
  - check (efficiently) using SAT solver
- Bounded model checking (BMC) via SAT
  - originally proposed for hardware model checking
  - subsequently adapted to software verification
  - example software: CBMC (BMC for C and C++ programs)
  - many industrial applications, e.g. hardware, embedded software

# Propositional logic and satisfiability

- Propositional logic formulae  $\Phi$ :
  - $\Phi ::= \text{true} \mid \text{false} \mid b \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg \Phi \mid \Phi \rightarrow \Phi \mid \Phi \leftrightarrow \Phi \mid \dots$
  - where  $b$  is a Boolean variable
  - e.g.  $\text{true}$ ,  $b$ ,  $\neg b$ ,  $\neg(b_1 \wedge b_2)$ ,  $b_1 \wedge (b_2 \vee \neg b_3)$
- Satisfiability
  - given propositional formula  $\Phi$  over variables  $b_1, \dots, b_n$
  - $\Phi$  is **satisfiable** if there exists a valuation of  $b_1, \dots, b_n$  such that  $\Phi(b_1, \dots, b_n)$  evaluates to true
- Example
  - $\Phi(b_1, b_2, b_3) = (b_1 \leftrightarrow \neg b_2) \wedge (b_2 \rightarrow b_3) \wedge (b_3 \vee b_1)$
  - is satisfiable:  $b_1=\text{true}$ ,  $b_2=\text{false}$ ,  $b_3=\text{true}$

# Propositional satisfiability (SAT)

- Propositional satisfiability problem (SAT)
  - (or Boolean satisfiability problem)
  - “is propositional formula  $\Phi$  satisfiable?”
- Theoretically important
  - one of the first problems to be proved to be NP-complete
  - (i.e. good example of a “hard” problem)
- Practically important
  - many practical (search) problems can be reduced to SAT
  - many efficient algorithms, tools (SAT solvers) exist
  - huge progress in recent years (big research field in own right)

# Example: Z3 solver

- Example:  $(b_1 \leftrightarrow \neg b_2) \wedge (b_2 \rightarrow b_3) \wedge (b_3 \vee b_1)$
- Z3 solver: <http://rise4fun.com/Z3/>

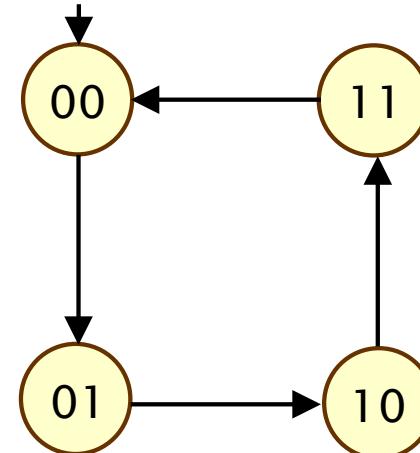
```
(declare-const b1 Bool)
(declare-const b2 Bool)
(declare-const b3 Bool)
(define-fun conjecture () Bool
 (and
 (= b1 (not b2))
 (and
 (=> b2 b3)
 (or b3 b1)
)
)
)
(assert conjecture)
(check-sat)
(get-model)
```

# Conjunctive normal form

- Conjunctive normal form (CNF)
  - $\Phi$  is a conjunction of disjunctions, e.g.  $(\neg b_1 \vee b_2) \wedge (b_2 \vee b_3)$
  - i.e.  $\Phi = \wedge_{i=1 \dots n} \vee_{j=1 \dots m_i} \text{lit}_{ij}$
  - where  $\text{lit}_{ij}$  is a literal  $b_k$  or  $\neg b_k$
- We will assume the use of propositional formula in CNF
  - in practice, solvers require inputs to be in CNF
- Can always convert to CNF
  - (de Morgan, double negation, distributive laws)
  - e.g.  $\neg((\neg b_1 \rightarrow \neg b_2) \wedge b_3) \Rightarrow (\neg b_1 \vee b_2) \wedge (b_2 \vee b_3)$

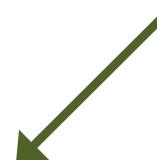
# Encoding a model

- Simple example: 2-bit counter
- Encode states in propositional logic:
  - using two Boolean variables  $l, r$
  - e.g. state 10 (representing binary encoding of 2) has  $l=1, r=0$
  - (use true=1, false=0)



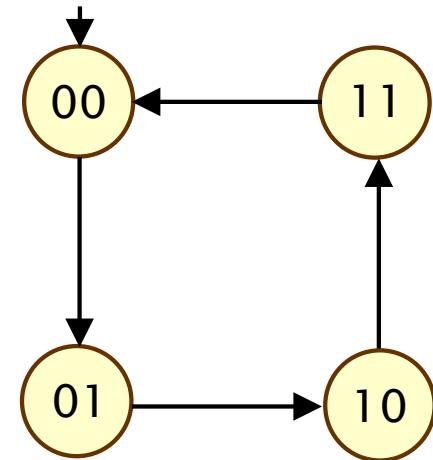
- Encode model in propositional logic:
  - initial state(s):  $\text{Init}(l_0, r_0) = \neg l_0 \wedge \neg r_0$
  - transition relation:  $T(l_i, r_i, l_{i+1}, r_{i+1}) = (l_{i+1} = (l_i \neq r_i)) \wedge (r_{i+1} = \neg r_i)$

“symbolic”  
encoding



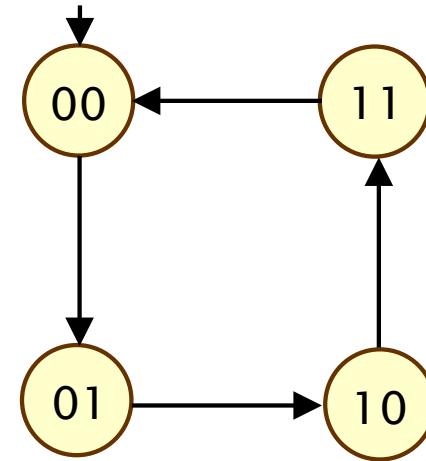
# Encoding a path

- Using same example...
- Path of length  $k$ :
  - $(l_0, r_0), (l_1, r_1), \dots, (l_k, r_k)$
- Encoding paths of length  $k$  (in propositional logic)
  - $\text{Init} \wedge T_1 \wedge \dots \wedge T_k$
  - $\Phi_k = \text{Init}(l_0, r_0) \wedge T(l_0, r_0, l_1, r_1) \wedge T(l_1, r_1, l_2, r_2) \wedge \dots \wedge T(l_{k-1}, r_{k-1}, l_k, r_k)$
  - e.g.  $\Phi_2 = (\neg l_0 \wedge \neg r_0) \wedge (l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0) \wedge (l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1)$
- There is a path of length  $k$ ...
  - if and only if  $\Phi_k$  is satisfiable
  - e.g. (for  $k=2$ ):  $l_0=0, r_0=0, l_1=0, r_1=1, l_2=1, r_2=0$



# Encoding model checking

- Example:
  - invariant: “the counter is always less than 3”
  - i.e. “always  $P_i$ ”, where  $P_i = \neg(l_i \wedge r_i)$  (“not 11”)
- Encoding in propositional logic
  - property is false if there exists a counterexample
  - $\text{Init} \wedge (T_1 \wedge \dots \wedge T_k) \wedge (\neg P_0 \vee \neg P_1 \vee \dots \vee \neg P_k)$
- Example ( $k=2$ ):
  - $(\neg l_0 \wedge \neg r_0) \wedge (l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0) \wedge (l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1) \wedge ((l_0 \wedge r_0) \vee (l_1 \wedge r_1) \vee (l_2 \wedge r_2))$
  - not satisfiable      ← incomplete!
  - (but is satisfiable for  $k=3$ )



# In Z3

```
(declare-const l0 Bool)
(declare-const r0 Bool)
(declare-const l1 Bool)
(declare-const r1 Bool)
(declare-const l2 Bool)
(declare-const r2 Bool)
(define-fun init ((l Bool) (r Bool)) Bool
 (and (not l) (not r))
)
(define-fun trans ((li Bool) (ri Bool) (lj Bool) (rj Bool)) Bool
 (and
 (= lj (not (= li ri)))
 (= rj (not ri))
)
)
(assert (and (init l0 r0) and (trans l0 r0 l1 r1) (trans l1 r1 l2 r2)))
(check-sat)
(get-model)
```

# Software model checking

- Simple example program
  - x and y are integer variables

```
x = x + y;
if (x ≠ 1) {
 x := 2;
} else {
 x++;
}
assert (x≤3);
```



$$\begin{aligned}& (x_1 = x_0 + y_0) \wedge \\& (x_2 = 2) \wedge \\& (x_3 = x_1 + 1) \wedge \\& (x_4 = (x_1 \neq 1) ? x_2 : x_3) \wedge \\& \neg(x_4 \leq 3)\end{aligned}$$

- Notice
  - simple imperative language (close to Java, C++)
  - variables can be uninitialised
  - properties specified as assertions (invariants?)

# 18. Bounded Model Checking of Software



Computer-Aided Verification

Dave Parker

University of Birmingham

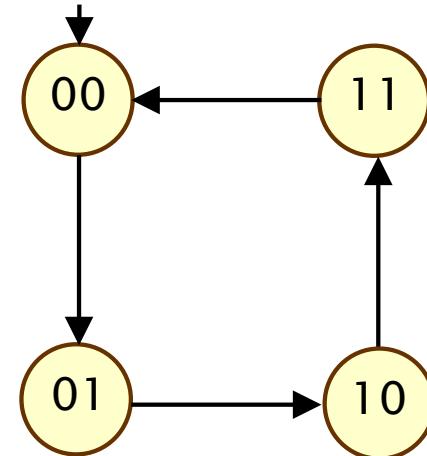
2017/18

# Module syllabus

- Modelling sequential and parallel systems
  - labelled transition systems, linear-time properties
- Temporal logic
  - LTL, CTL and CTL\*, etc.
- Model checking
  - CTL model checking
  - automata-theoretic model checking (LTL)
- Quantitative verification
  - probabilistic (and timed) systems
- Verification tools
- Advanced verification techniques
  - bounded model checking via propositional satisfiability

# Bounded model checking via SAT

- Main steps
  - bounded unfolding (depth  $k$ )
  - encode in propositional logic (CNF)
  - reduction to (efficient) SAT problem
- Key ideas
  - verification/falsification
  - bug hunting
  - negate property to check
  - symbolic approach



invariant: “the counter  
is always less than 3”

$$P_i = \neg(l_i \wedge r_i)$$

$$\text{Init} \wedge (T_1 \wedge \dots \wedge T_k) \wedge \\ (\neg P_0 \vee \neg P_1 \vee \dots \vee \neg P_k)$$

E.g. ( $k=2$ ):  $(\neg l_0 \wedge \neg r_0) \wedge (l_1 = (l_0 \neq r_0) \wedge r_1 = \neg r_0) \wedge \\ (l_2 = (l_1 \neq r_1) \wedge r_2 = \neg r_1) \wedge ((l_0 \wedge r_0) \vee (l_1 \wedge r_1) \vee (l_2 \wedge r_2))$

# Software model checking

- Simple example program
  - x and y are integer variables

```
x := x + y;
if (x ≠ 1) {
 x := 2;
} else {
 x++;
}
assert (x≤3);
```

- Notice
  - simple imperative language (close to Java, C++)
  - variables can be uninitialised
  - properties specified as assertions

# Overview

- Bounded model checking of software
- Main steps:
  - control flow simplification
  - loop unwinding
  - conversion to single static assignment form
  - conversion to conjunctive normal form (CNF)
  - solution using SAT (or SMT) solvers

# Control flow simplification

- Simplify structure of code for easier analysis
  - convert to programs comprising `while` loops, `ifs` and `gotos`
  - ensure expressions are side-effect free

```
count := 0;
for (i := 1; i ≤ n; i++) {
 if (a[i] = x) {
 j := count++;
 if (count ≥ 10) {
 break;
 }
 }
}
```

# Control flow simplification

- Simplify structure of code for easier analysis
  - convert to programs comprising **while** loops, **ifs** and **gotos**
  - ensure expressions are side-effect free

```
count := 0;
for (i := 1; i ≤ n; i++) {
 if (a[i] = x) {
 j := count++;
 if (count ≥ 10) {
 break;
 }
 }
}
```



```
count := 0;
i := 1;
while (i ≤ n) {
 if (a[i] = x) {
 j := count;
 count := count+1;
 if (count ≥ 10) {
 goto loop_exit;
 }
 }
 i := i+1;
}
loop_exit:
```

# Loop unwinding

- Convert to loop-free program
  - unwinding loops to a fixed depth k
  - (recall: only need to consider while loops + gotos)
- 1 unwinding:

```
while (condition) {
 body
}
statements
```



```
if (condition)
 body
 while (condition) {
 body
 }
}
statements
```

# Loop unwinding

- 2 unwindings:

```
while (condition) {
 body
}
statements
```



```
if (condition) {
 body
 if (condition)
 body
 while (condition) {
 body
 }
}
statements
```

# Loop unwinding

- 2 unwindings (assume  $k=2$ ):

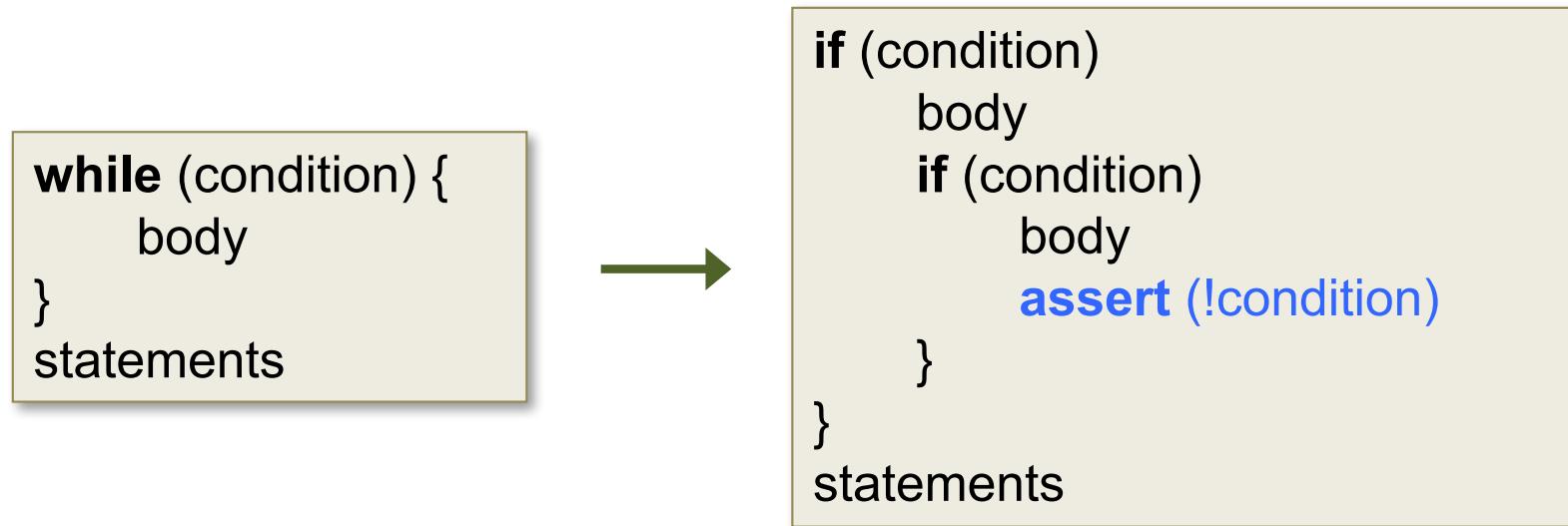
```
while (condition) {
 body
}
statements
```



```
if (condition) {
 body
 if (condition)
 body
 }
} statements
```

# Loop unwinding

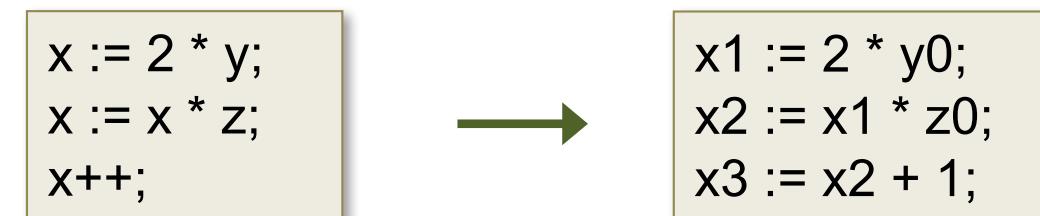
- 2 unwindings ( $k=2$ , with unwinding assertion):



- Bounded model checking:  $k$  unwindings of all loops
  - note: not just depth-bounded search of program state space

# Single static assignments

- Single static assignment (SSA) form
  - used as an intermediate representation in program analysis
  - every variable seen exactly once
  - multiple versions of each variable created
- Conversion to SSA:
  - each assignment of variable uses a new version
  - each access of a variable uses the latest version



# Single static assignments

- SSA form conversion for conditionals:
  - (recall: while loops unwound into if statements)

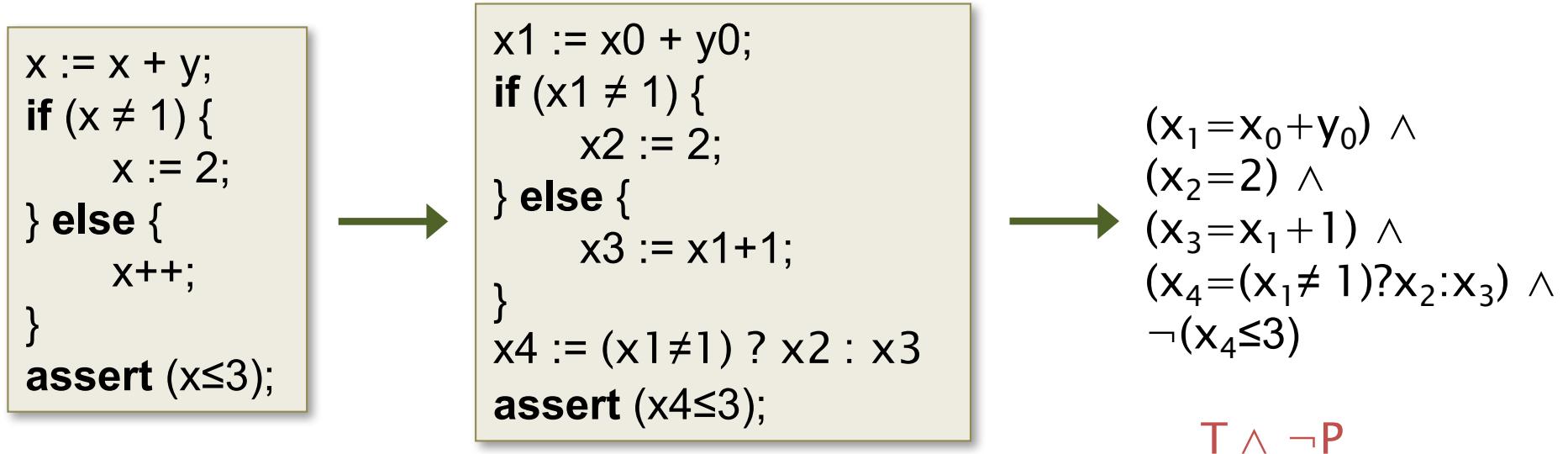
```
if (x > z) {
 y := x;
} else {
 y := z + 1;
}
w := 2 * y;
```



```
if (x0 > z0) {
 y1 := x0;
} else {
 y2 := z0 + 1;
}
y3 := (x0 > z0) ? y1 : y2;
w1 := 2 * y3;
```

# Conversion to CNF

- Conversion to conjunctive normal form (CNF)
- Simple example program from earlier



- Formula over predicates, not Boolean variables

# Checking satisfiability

- Checking satisfiability of formulae over predicates
- Bit blasting
  - convert integers to usual binary encoding
  - map integer variables from predicates to Boolean variables
  - solve using standard SAT solver
  - precise, faithful encoding of program
- SMT (satisfiability modulo theories) solvers
  - as for SAT, many efficient algorithms/tools developed
  - infinite-ranging variables, richer data types
  - but may not match e.g. execution of real C code

# Example: Z3 (SMT)

```
(declare-const x0 Int)
(declare-const y0 Int)
(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)
(declare-const x4 Int)
(define-fun conjecture () Bool
 (and
 (= x1 (+ x0 y0))
 and
 (= x2 2)
 and
 (= x3 (+ x1 1))
 and
 (= x4 (ite (not (= x1 1)) x2 x3))
 (not (<= x4 3))
)
)
(assert conjecture)
(check-sat)
(get-model)
```

# Example

- Apply bounded model check, unrolling loops to depth 2

```
for (i := 1; i ≤ 10; i++) {
 if (a[i] > j) {
 j := a[i];
 }
}
assert j>0;
```

# Example

- Apply bounded model check, unrolling loops to depth 2

```
for (i := 1; i ≤ 10; i++) {
 if (a[i] > j) {
 j := a[i];
 }
}
assert j>0;
```



```
i := 1;
while (i ≤ 10) {
 if (a[i] > j) {
 j := a[i];
 }
 i := i + 1;
}
assert j>0
```

```

for (i := 1; i ≤ 10; i++) {
 if (a[i] > j) {
 j := a[i];
 }
}
assert j>0;

```



```

i := 1;
while (i ≤ 10) {
 if (a[i] > j) {
 j := a[i];
 }
 i := i + 1;
}
assert j>0

```



```

i1 := 1;
if (i1 ≤ 10) {
 if (a0[i1] > j0) {
 j1 := a0[i0];
 }
 j2 := (a0[i1] > j0) ? j1 : j0
 i2 := i1 + 1;
 if (i2 ≤ 10) {
 if (a[i2] > j2) {
 j3 := a0[i2];
 }
 j4 := (a0[i2] > j2) ? j3 : j2
 i3 := i2 + 1;
 }
 j5 := (i2 ≤ 10) ? j4 : j2
 i4 := (i2 ≤ 10) ? i3 : i2
}
j6 := (i1 ≤ 10) ? j5 : j0
i5 := (i1 ≤ 10) ? i4 : i1
assert j6>0;

```



$$(i_1 = 1) \wedge (j_1 = a_0[i_0]) \wedge (j_2 = a_0[i_0] > j_0 ? j_1 : j_0) \wedge \dots \wedge \neg(j_6 > 0)$$

# Summary

- **Bounded model checking (for software)**
  - bounded search for bugs by unwind program loops
  - incomplete (without extensions) but effective in practice
  - relies on translation to efficiently solvable problem (SAT/SMT)
- **Main steps:**
  - control flow simplification
  - loop unwinding
  - conversion to single static assignment form
  - conversion to conjunctive normal form (CNF)
  - solution using SAT (or SMT) solvers

# 19. Symbolic Model Checking



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

# Rest of the module

- **Lectures**
  - today: symbolic model checking
  - Thursday: probabilistic model checking
  - no lectures next week
- **Assignments & exercises**
  - Assignment 3 – solutions on Canvas (and Q2ii remarked)
  - Assignment 4 (SPIN) – out now, due Thursday of week 11
  - Assignment 5 (extended only) – due Thursday
  - non-assessed exercise (bounded model checking) online

# Module syllabus

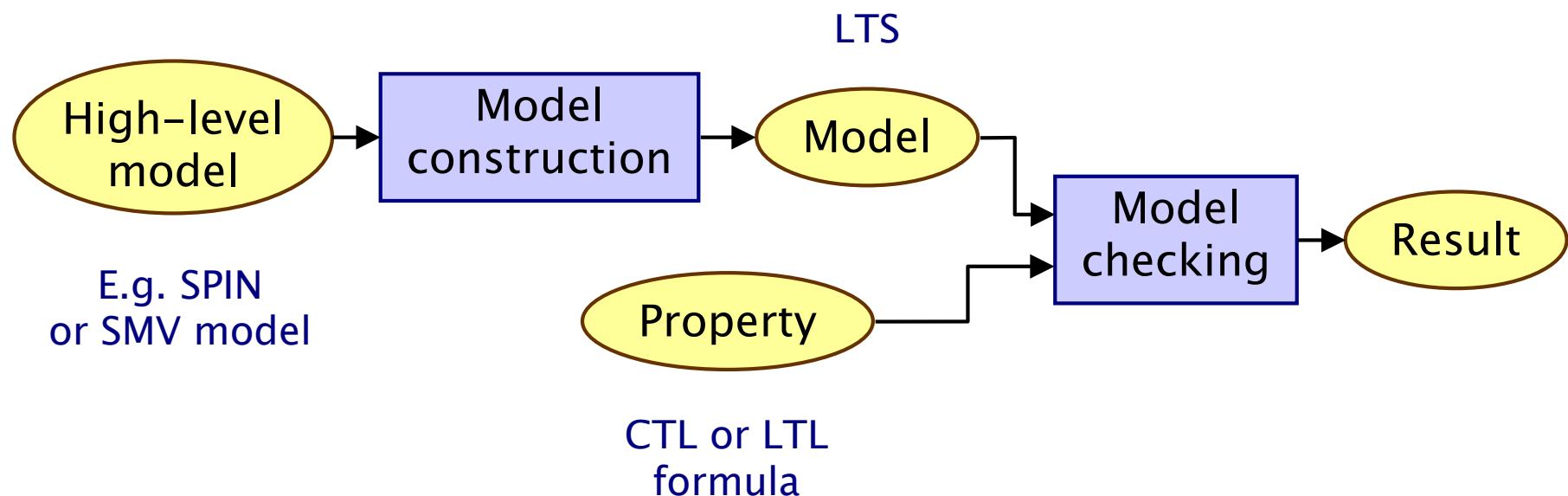
- Modelling sequential and parallel systems
  - labelled transitions systems, parallel composition
- Temporal logic
  - LTL, CTL and CTL\*, etc.
- Model checking
  - CTL model checking algorithms
  - automata-theoretic model checking (LTL)
- Verification tools: SPIN
- Advanced verification techniques
  - bounded model checking via propositional satisfiability
  - symbolic model checking
  - probabilistic model checking

# Overview

- **Last time**
  - bounded model checking via SAT (or SMT)
  - “symbolic” encoding of model checking problem
  - targets falsification up to a finite number of unwindings
  - can be made complete, e.g. with  $k$ -induction
- **This lecture**
  - symbolic model checking
  - binary decision diagrams (BDDs)
  - exploits regularity to improve scalability of model checking
  - i.e. targets state space explosion problem
  - well suited to verification (as opposed to falsification)
  - applicable much more widely

# Model checking implementation

- Overview of the model checking process
  - two phases: **model construction**, **model checking**
  - several different logics, multiple algorithms
  - but... they have various aspects/operations in common
    - basic set operations, reachability, strongly connected components, ...
    - manipulation of **transition relation** and **state sets**



# Explicit vs. symbolic data structures

- Symbolic data structures
  - usually based on **binary decision diagrams** (BDDs) or variants
  - avoid explicit enumeration of data by **exploiting regularity**
  - potentially **very compact/efficient storage** (but not always)
- Sets of states:
  - **explicit**: bit vectors, hashing
  - **symbolic**: BDDs
- Transition relations:
  - **explicit**: sparse adjacency matrix
  - **symbolic**: BDDs

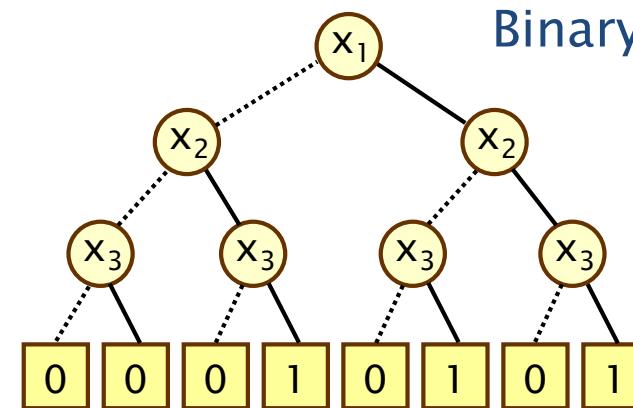
# Representations of Boolean formulas

- Propositional formula:  $f = (x_1 \vee x_2) \wedge x_3$

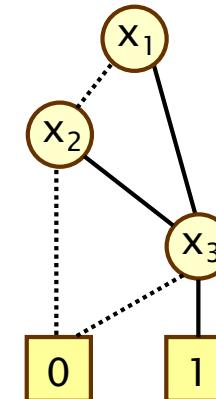
Truth table

| $x_1$ | $x_2$ | $x_3$ | $f$ |
|-------|-------|-------|-----|
| 0     | 0     | 0     | 0   |
| 0     | 0     | 1     | 0   |
| 0     | 1     | 0     | 0   |
| 0     | 1     | 1     | 1   |
| 1     | 0     | 0     | 0   |
| 1     | 0     | 1     | 1   |
| 1     | 1     | 0     | 0   |
| 1     | 1     | 1     | 1   |

Binary decision tree

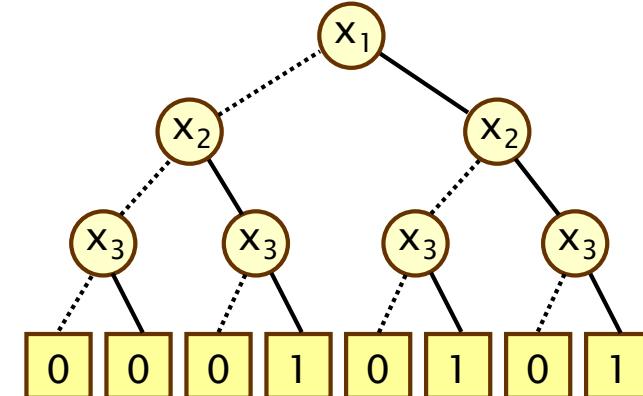


Binary decision diagram



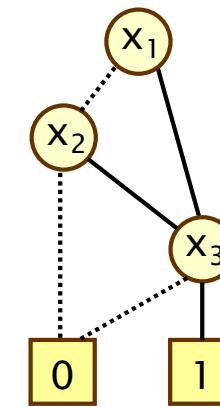
# Binary decision trees

- Graphical representation of Boolean functions
  - $f(x_1, \dots, x_n) : \{0,1\}^n \rightarrow \{0,1\}$
- Binary tree with two types of nodes
- Non-terminal nodes
  - labelled with a Boolean variable  $x_i$
  - two children: 1 (“then”, solid line) and 0 (“else”, dotted line)
- Terminal nodes (or “leaf” nodes)
  - labelled with 0 or 1
- To read the value of  $f(x_1, \dots, x_n)$ 
  - start at root (top) node
  - take “then” edge if  $x_i=1$
  - take “else” edge if  $x_i=0$
  - result given by leaf node



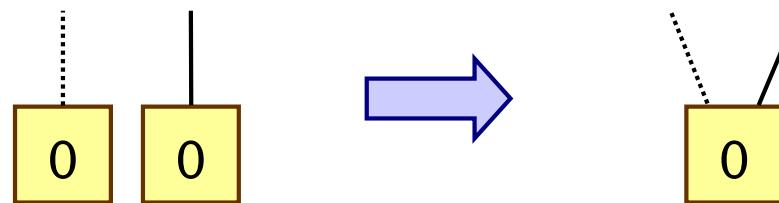
# Binary decision diagrams

- Binary decision diagrams (BDDs)
  - based on binary decision trees, but **reduced** and **ordered**
  - sometimes called reduced ordered BDDs (ROBDDs)
  - actually directed acyclic graphs (DAGs), not trees
  - **compact, canonical** representation for Boolean functions
- Variable ordering
  - a BDD assumes a fixed total ordering over its set of Boolean variables
  - e.g.  $x_1 < x_2 < x_3$
  - along any path through the BDD, variables appear at most once each and always in the correct order

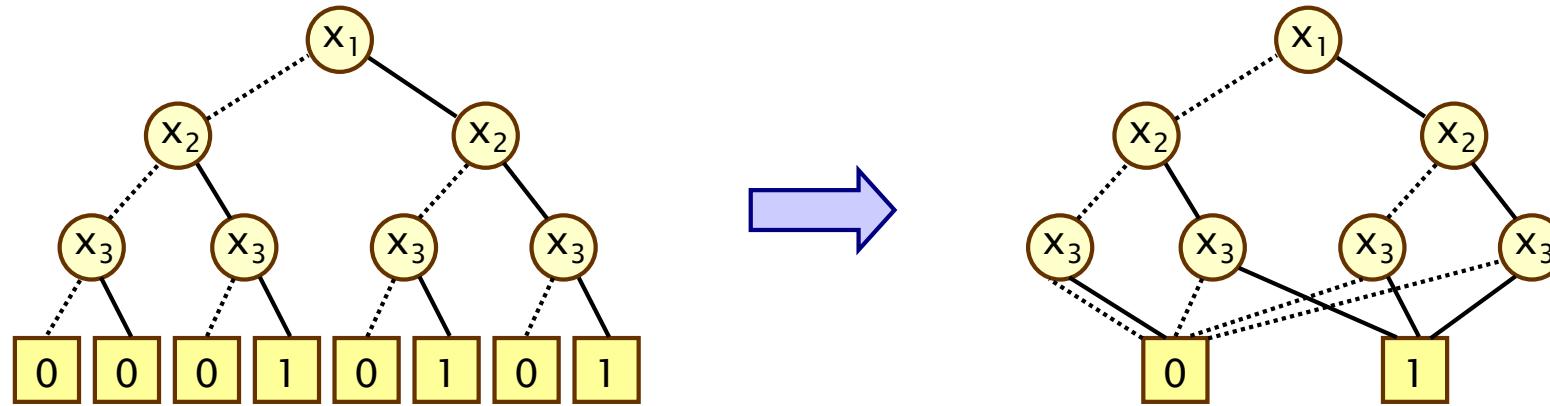


# BDD reduction rule 1

- Rule 1: Merge identical terminal nodes

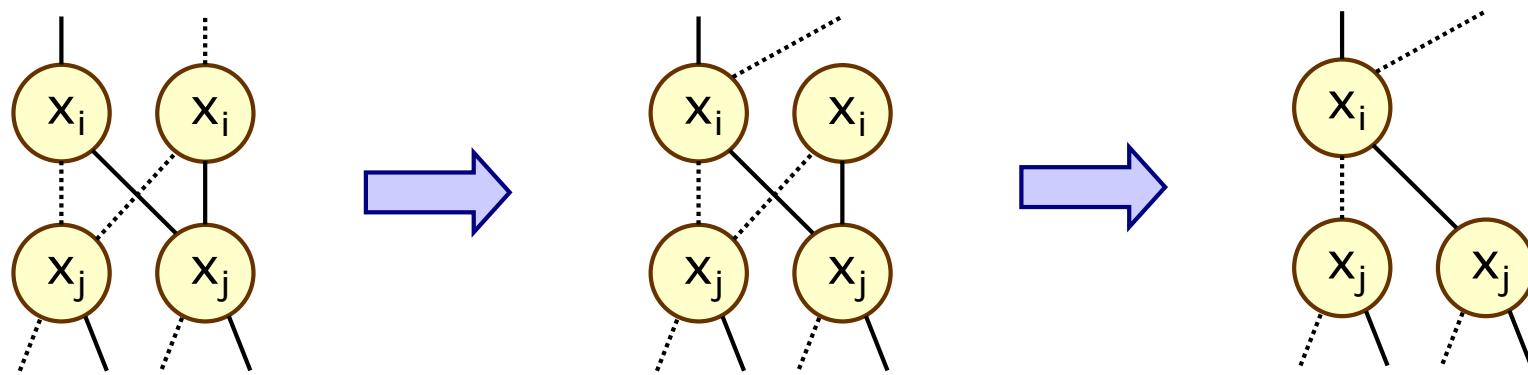


- Example:

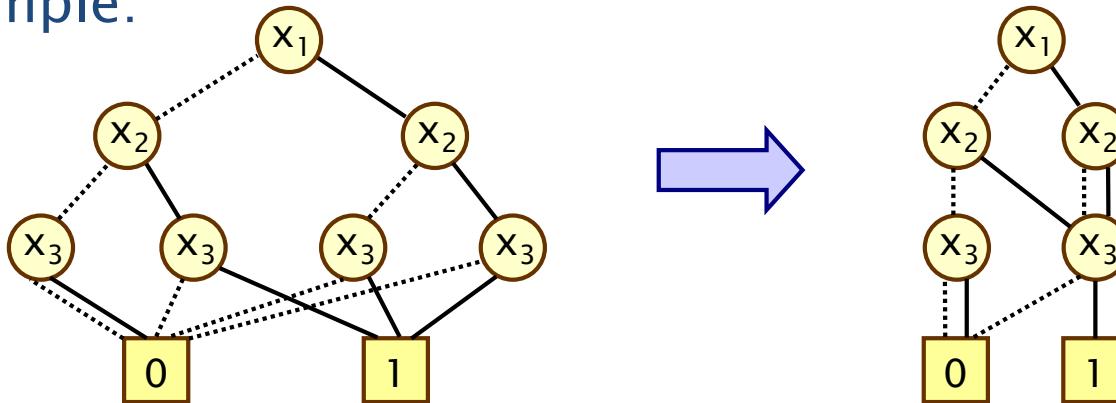


# BDD reduction rule 2

- Rule 2: Merge isomorphic nodes, redirect incoming nodes

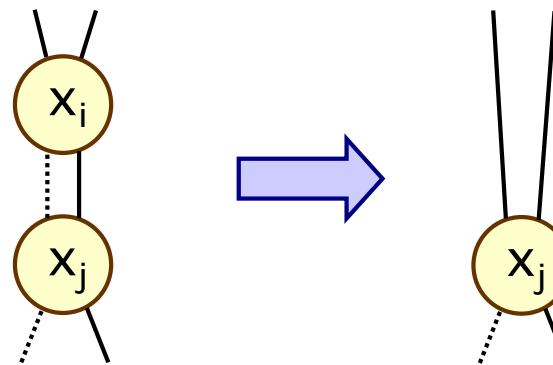


- Example:

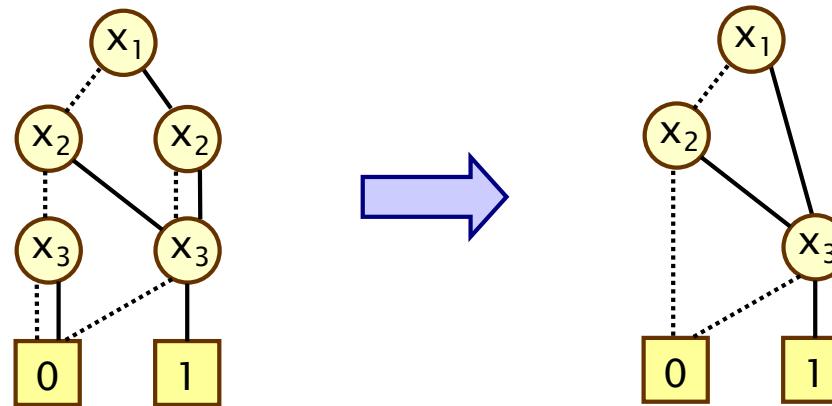


# BDD reduction rule 3

- Rule 3: Remove redundant nodes (with identical children)

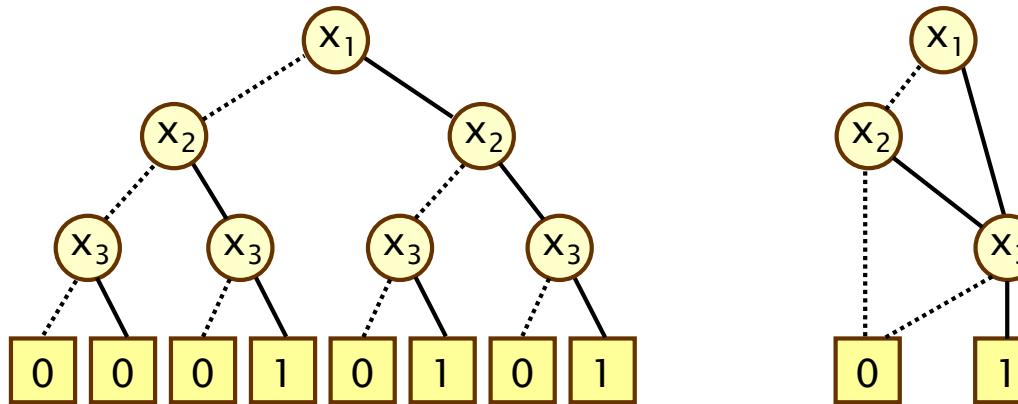


- Example:



# Canonicity

- BDDs are a canonical representation for Boolean functions
  - two Boolean functions are **equivalent** if and only if the BDDs which represent them are **isomorphic**
  - uniqueness relies on: **reduced BDDs, fixed variable ordered**



- Important implications for implementation efficiency
  - can be tested in linear (or even constant) time

# BDD variable ordering

- BDD size can be very sensitive to the variable ordering

- example:  $f = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$

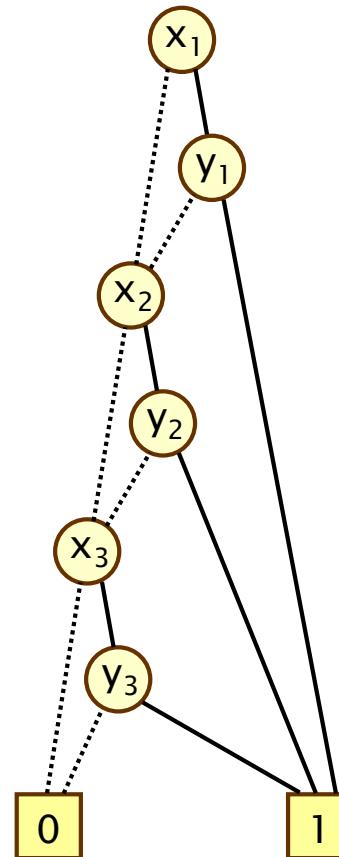
- two orderings:

- $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$
    - $x_1 < x_2 < x_3 < y_1 < y_2 < y_3$

- which is better?

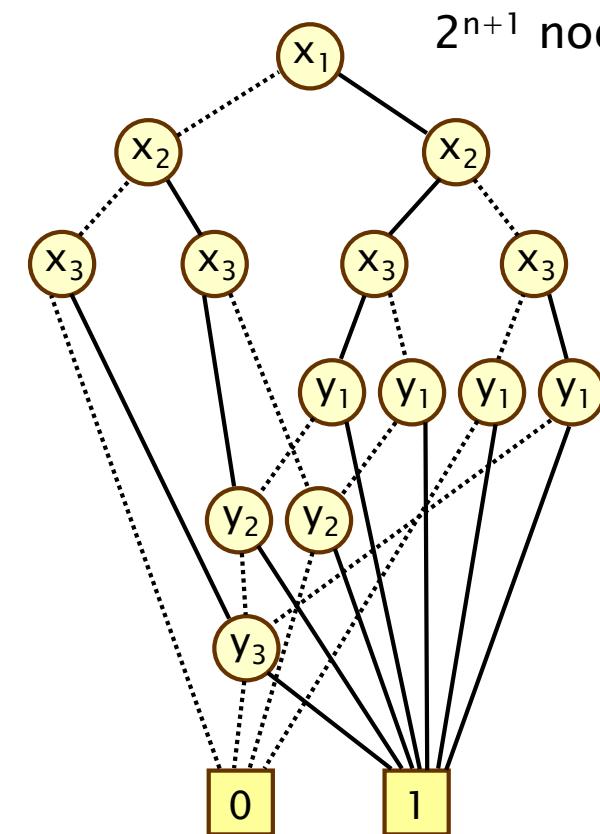
$x_1 < y_1 < x_2 < y_2 < x_3 < y_3$

$2n+2$  nodes



$x_1 < x_2 < x_3 < y_1 < y_2 < y_3$

$2^{n+1}$  nodes



# BDDs to represent sets of states

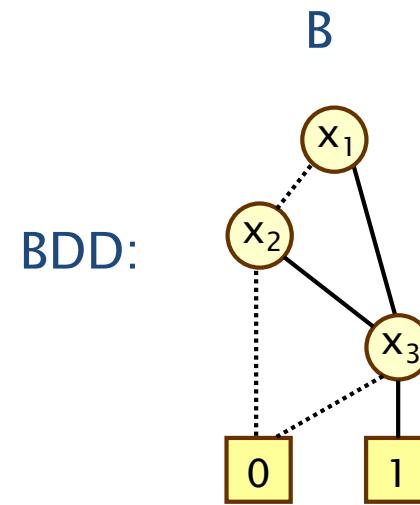
- Consider a state space  $S$  and some subset  $S' \subseteq S$
- We can represent  $S'$  by its **characteristic function**  $\chi_{S'}$ 
  - $\chi_{S'} : S \rightarrow \{0,1\}$  where  $\chi_{S'}(s) = 1$  if and only if  $s \in S'$
- Assume we have an encoding of  $S$  into  $n$  Boolean variables
  - this is always possible for a finite set  $S$
  - e.g. enumerate the elements of  $S$  and use a **binary encoding**
  - (note: there may be more efficient encodings though)
- So  $\chi_{S'}$  can be seen as a function  $\chi_{S'}(x_1, \dots, x_n) : \{0,1\}^n \rightarrow \{0,1\}$ 
  - which is simply a Boolean function
  - which can therefore be represented as a BDD

# BDD and sets of states – Example

- State space  $S: \{0, 1, 2, 3, 4, 5, 6, 7\}$
- Encoding of  $S: \{000, 001, 010, 011, 100, 101, 110, 111\}$
- Subset  $S' \subseteq S: \{3, 5, 7\} \rightarrow \{011, 101, 111\}$

Truth table:

| $x_1$ | $x_2$ | $x_3$ | $f_B$ |
|-------|-------|-------|-------|
| 0     | 0     | 0     | 0     |
| 0     | 0     | 1     | 0     |
| 0     | 1     | 0     | 0     |
| 0     | 1     | 1     | 1     |
| 1     | 0     | 0     | 0     |
| 1     | 0     | 1     | 1     |
| 1     | 1     | 0     | 0     |
| 1     | 1     | 1     | 1     |

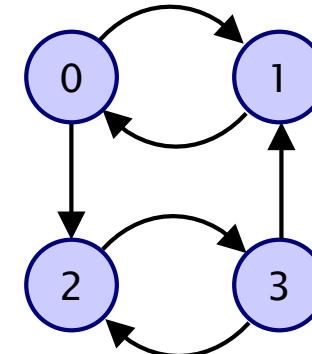


# BDDs and transition relations

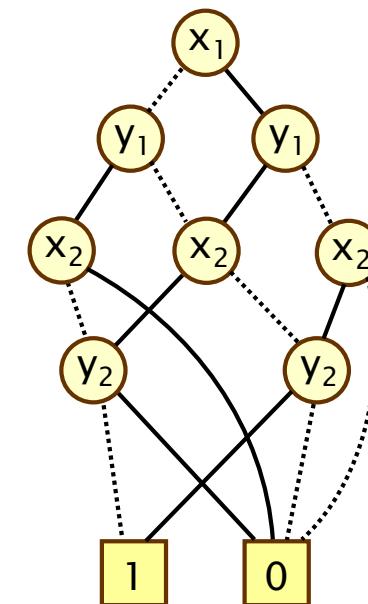
- Transition relations can also be represented by their characteristic function, but over pairs of states
  - relation:  $R \subseteq S \times S$
  - characteristic function:  $\chi_R : S \times S \rightarrow \{0,1\}$
- For an encoding of state space  $S$  into  $n$  Boolean variables
  - we have Boolean function  $f_R(x_1, \dots, x_n, y_1, \dots, y_n) : \{0,1\}^{2n} \rightarrow \{0,1\}$
  - which can be represented by a BDD
- Row and column variables
  - for efficiency reasons, we **interleave** the **row variables**  $x_1, \dots, x_n$  and **column variables**  $y_1, \dots, y_n$
  - i.e. we use function  $f_R(x_1, y_1, \dots, x_n, y_n) : \{0,1\}^{2n} \rightarrow \{0,1\}$

# BDDs and transition relations

- Example:
  - 4 states: 0, 1, 2, 3
  - Encoding:  $0 \mapsto 00$ ,  $1 \mapsto 01$ ,  $2 \mapsto 10$ ,  $3 \mapsto 11$



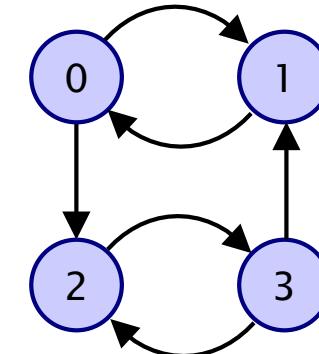
| Transition | $x_1$ | $x_2$ | $y_1$ | $y_2$ | $x_1y_1x_2y_2$ |
|------------|-------|-------|-------|-------|----------------|
| (0,1)      | 0     | 0     | 0     | 1     | 0001           |
| (0,2)      | 0     | 0     | 1     | 0     | 0100           |
| (1,0)      | 0     | 1     | 0     | 0     | 0010           |
| (2,3)      | 1     | 0     | 1     | 1     | 1101           |
| (3,1)      | 1     | 1     | 0     | 1     | 1011           |
| (3,2)      | 1     | 1     | 1     | 0     | 1110           |



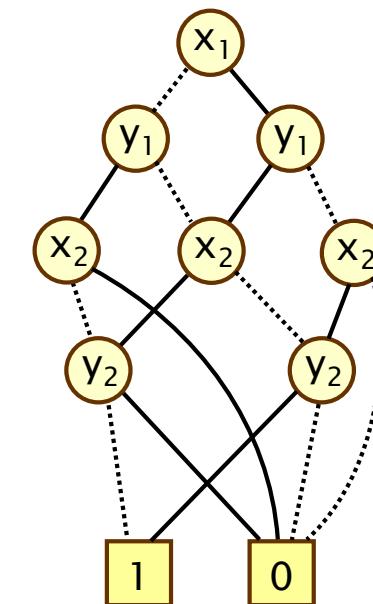
# BDDs and transition relations

- We can also think of the transition relation as an adjacency matrix

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |

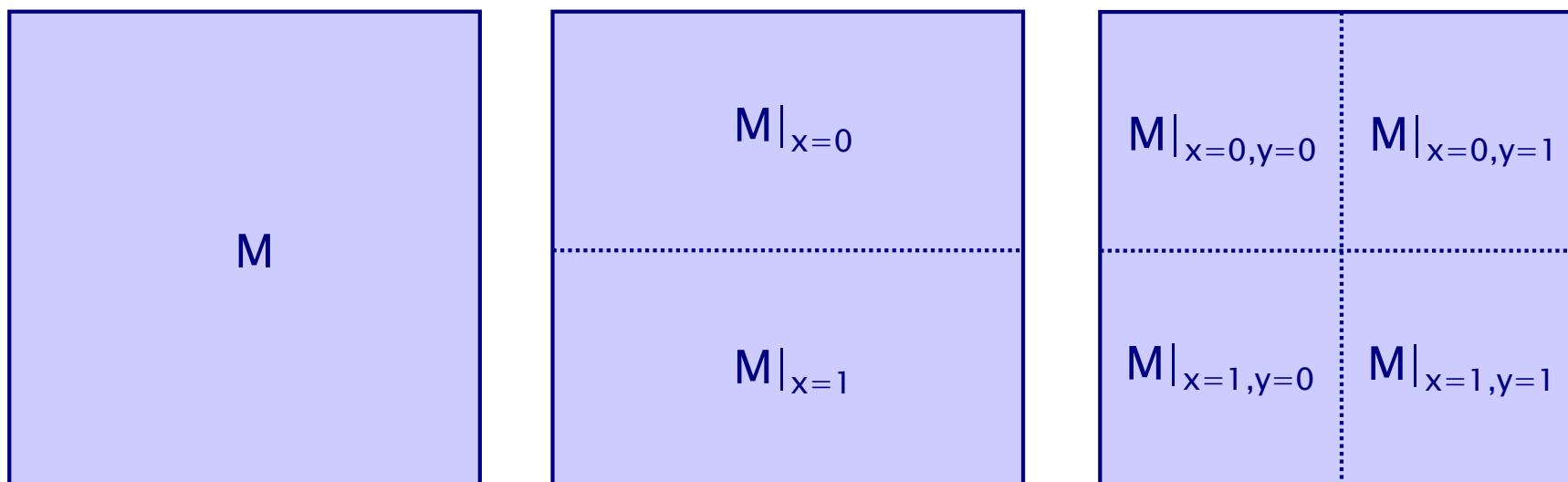


| Transition | $x_1$ | $x_2$ | $y_1$ | $y_2$ | $x_1y_1x_2y_2$ |
|------------|-------|-------|-------|-------|----------------|
| (0,1)      | 0     | 0     | 0     | 1     | 0001           |
| (0,2)      | 0     | 0     | 1     | 0     | 0100           |
| (1,0)      | 0     | 1     | 0     | 0     | 0010           |
| (2,3)      | 1     | 0     | 1     | 1     | 1101           |
| (3,1)      | 1     | 1     | 0     | 1     | 1011           |
| (3,2)      | 1     | 1     | 1     | 0     | 1110           |



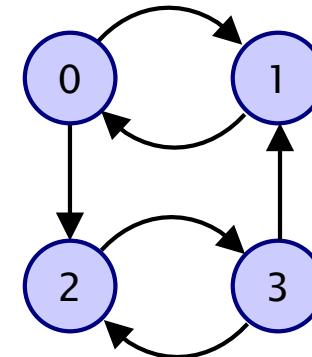
# Matrices and BDDs – Recursion

- Descending one level in the BDD (i.e. setting  $x_i = b$ )
  - splits the matrix represented by the BDD in half
  - row variables ( $x_i$ ) give horizontal split
  - column variables ( $y_i$ ) give vertical split

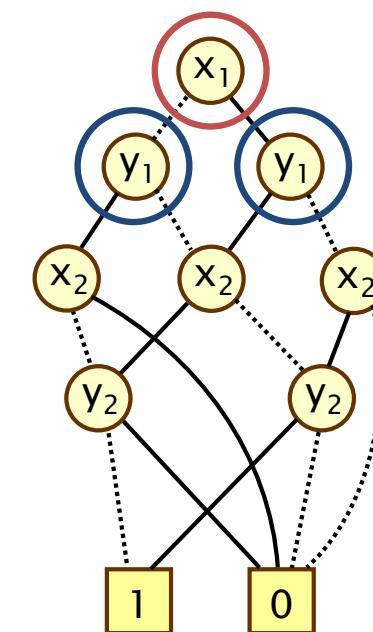


# Matrices and BDDs – Recursion

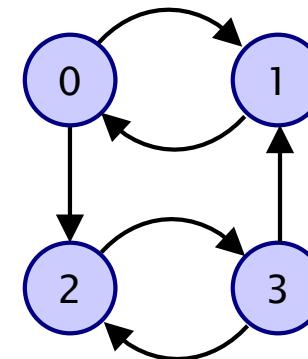
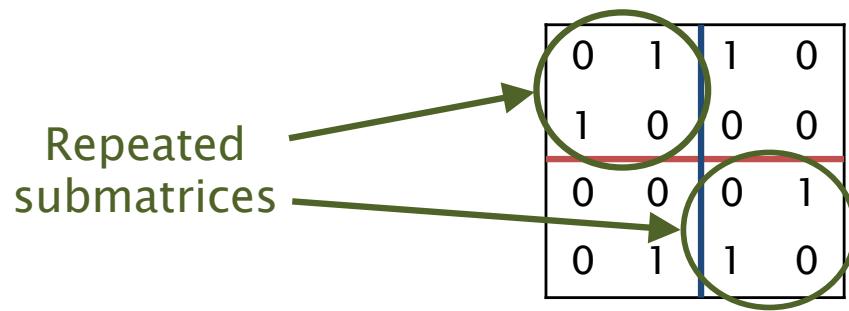
|       |   |   |   |
|-------|---|---|---|
| 0     | 1 | 1 | 0 |
| 1     | 0 | 0 | 0 |
| <hr/> |   |   |   |
| 0     | 0 | 0 | 1 |
| 0     | 1 | 1 | 0 |



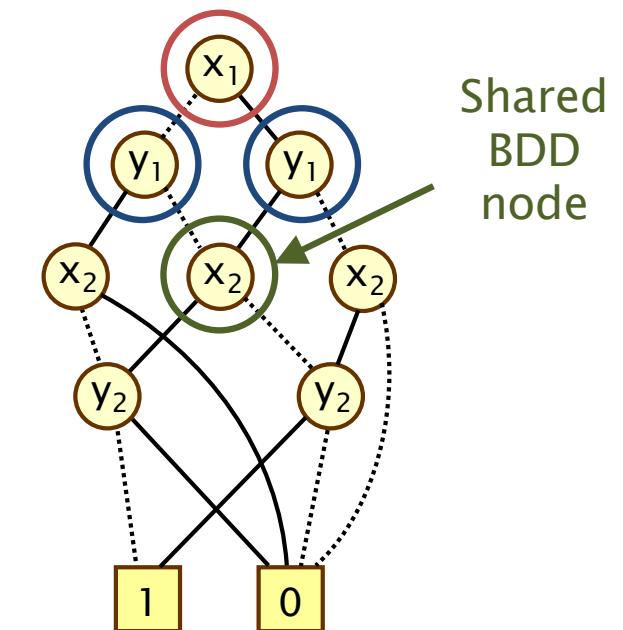
| Transition | $x_1$ | $x_2$ | $y_1$ | $y_2$ | $x_1y_1x_2y_2$ |
|------------|-------|-------|-------|-------|----------------|
| (0,1)      | 0     | 0     | 0     | 1     | 0001           |
| (0,2)      | 0     | 0     | 1     | 0     | 0100           |
| (1,0)      | 0     | 1     | 0     | 0     | 0010           |
| (2,3)      | 1     | 0     | 1     | 1     | 1101           |
| (3,1)      | 1     | 1     | 0     | 1     | 1011           |
| (3,2)      | 1     | 1     | 1     | 0     | 1110           |



# Matrices and BDDs – Regularity

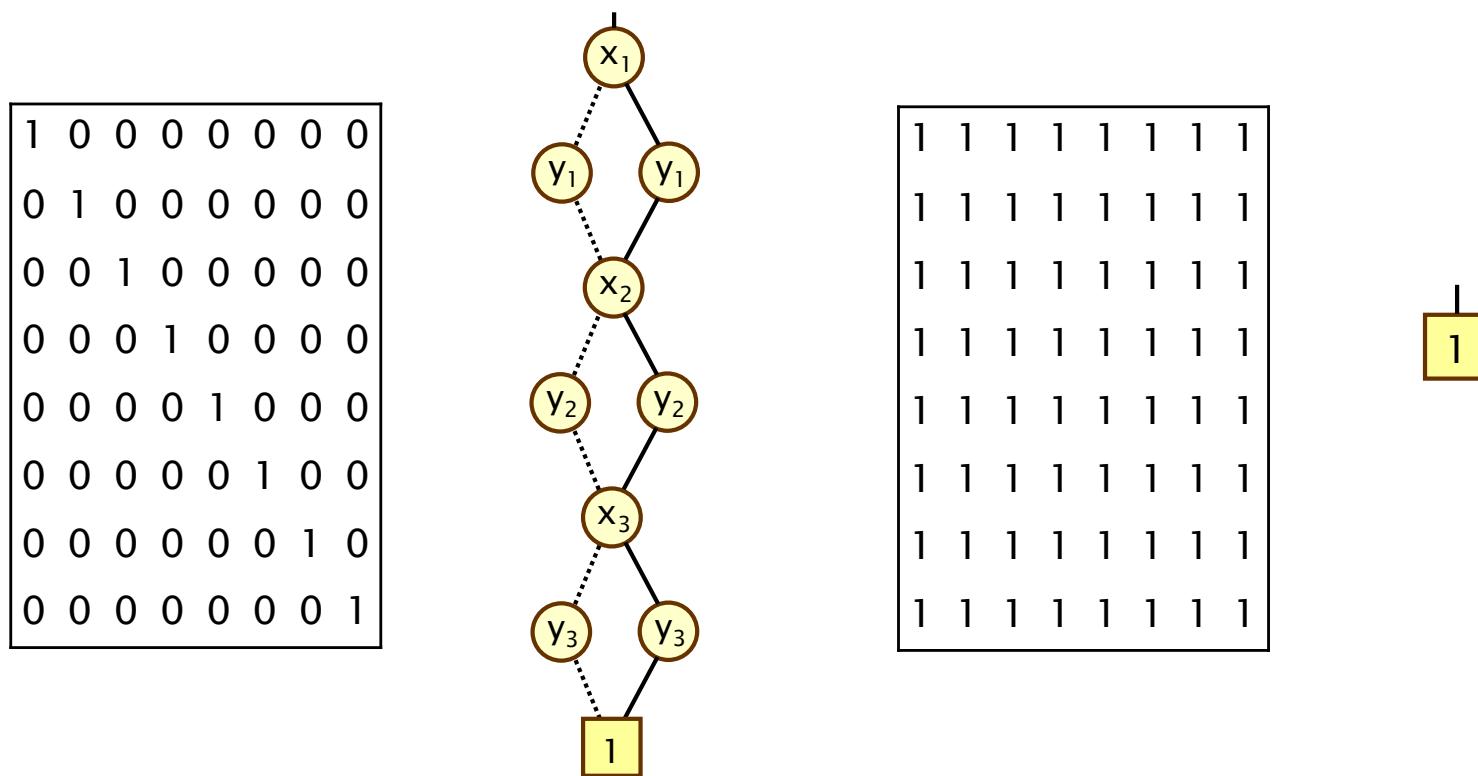


| Transition | $x_1$ | $x_2$ | $y_1$ | $y_2$ | $x_1y_1x_2y_2$ |
|------------|-------|-------|-------|-------|----------------|
| (0,1)      | 0     | 0     | 0     | 1     | 0001           |
| (0,2)      | 0     | 0     | 1     | 0     | 0100           |
| (1,0)      | 0     | 1     | 0     | 0     | 0010           |
| (2,3)      | 1     | 0     | 1     | 1     | 1101           |
| (3,1)      | 1     | 1     | 0     | 1     | 1011           |
| (3,2)      | 1     | 1     | 1     | 0     | 1110           |



# Matrices and BDDs – Compactness

- Some simple matrices (or relations) have extremely compact representations as BDDs
  - e.g. the identity matrix or a constant matrix



# Flashback: CTL model checking $\exists U$

- Procedure to compute  $\text{Sat}(\exists(\phi_1 \cup \phi_2))$ 
  - given  $\text{Sat}(\phi_1)$  and  $\text{Sat}(\phi_2)$
- Basic idea: backwards search of the LTS from  $\phi_2$ -states
  - $T_0 := \text{Sat}(\phi_2)$
  - $T_i := T_{i-1} \cup \{ s \in \text{Sat}(\phi_1) \mid \text{Post}(s) \cap T_{i-1} \neq \emptyset \}$
  - until  $T_i = T_{i-1}$
  - $\text{Sat}(\exists(\phi_1 \cup \phi_2)) = T_i$
- (i.e. keep adding predecessors of states in  $T_{i-1}$ )
- Based on expansion law
  - $\exists(\phi_1 \cup \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge \exists \bigcirc \exists(\phi_1 \cup \phi_2))$
  - (can be formulated as a fixed-point equation)

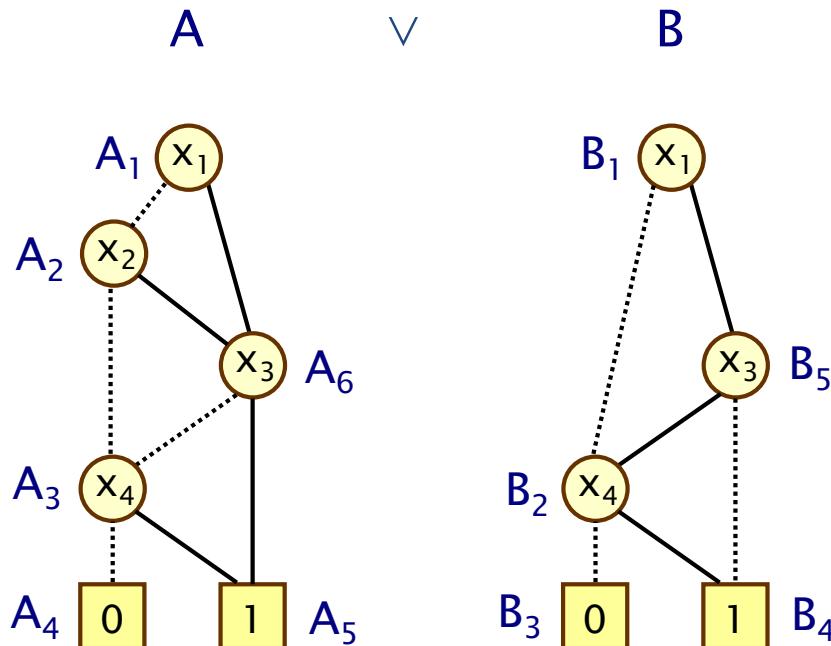
# Manipulating BDDs

- Need efficient ways to manipulate Boolean functions
  - while they are represented as BDDs
  - i.e. algorithms which are applied directly to the BDDs
- Basic operations on Boolean functions:
  - negation ( $\neg$ ), conjunction ( $\wedge$ ), disjunction ( $\vee$ ), etc.
  - can all be applied directly to BDDs
- Key operation on BDDs:  $\text{Apply}(\text{op}, A, B)$ 
  - where  $A$  and  $B$  are BDDs and  $\text{op}$  is a binary operator over Boolean values, e.g.  $\wedge$ ,  $\vee$ , etc.
  - $\text{Apply}(\text{op}, A, B)$  returns the BDD representing function  $f_A \text{ op } f_B$
  - often just use infix notation, e.g.  $\text{Apply}(\wedge, A, B) = A \wedge B$
  - efficient algorithm: recursive depth-first traversal of  $A$  and  $B$
  - complexity (and size of result) is  $O(|A| \cdot |B|)$ 
    - where  $|C|$  denotes size of BDD  $C$

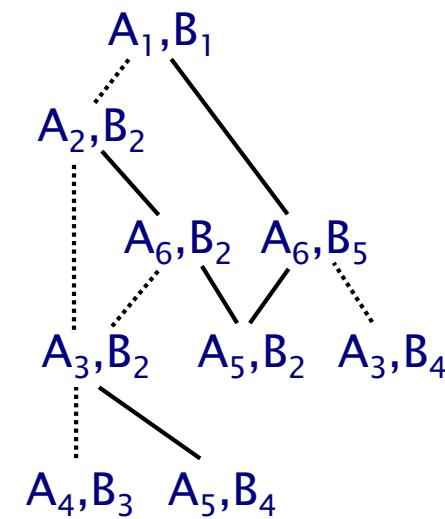
# Apply – Example

- Example:  $\text{Apply}(\vee, A, B)$

Argument BDDs, with node labels:

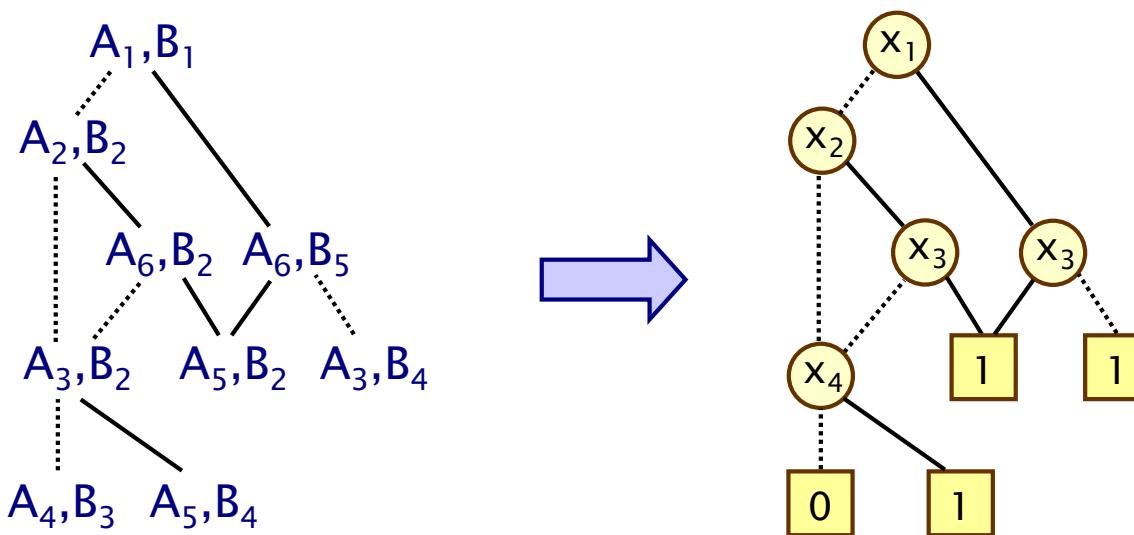


Recursive calls to Apply:



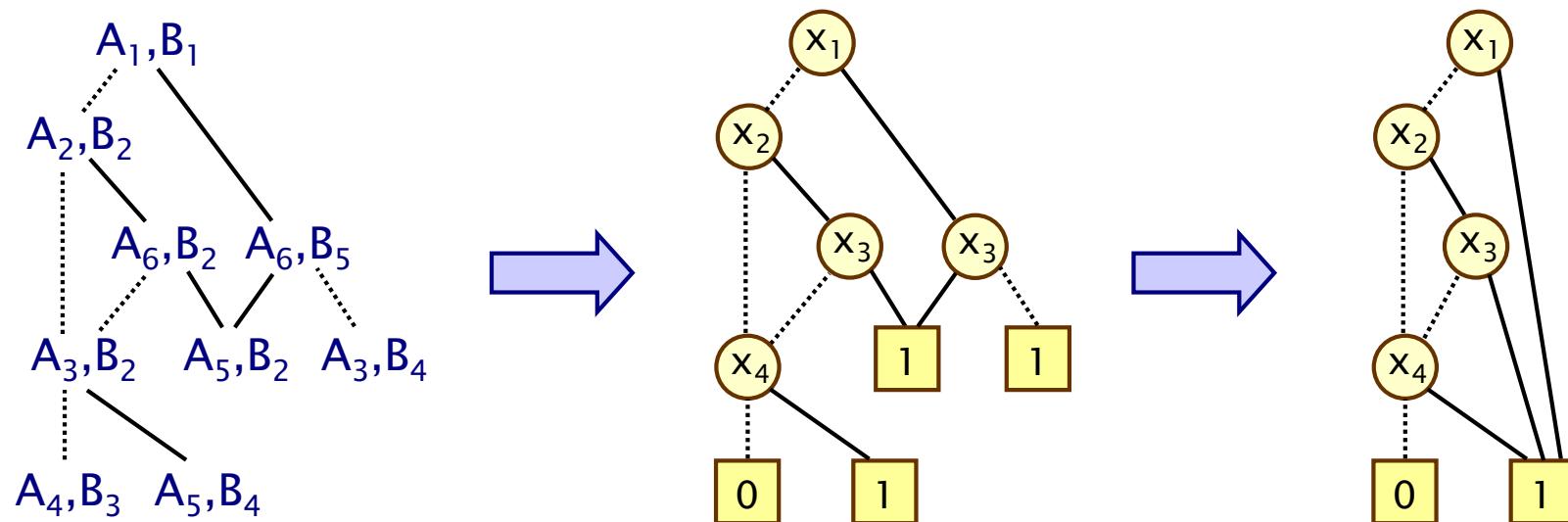
# Apply – Example

- Example:  $\text{Apply}(\vee, A, B)$ 
  - recursive call structure implicitly defines resulting BDD



# Apply – Example

- Example:  $\text{Apply}(\vee, A, B)$ 
  - but the resulting BDD needs to be reduced
  - in fact, we can do this as part of the recursive Apply operation, implementing reduction rules bottom-up



# Implementation of BDDs

- Store all BDDs currently in use as one multi-rooted BDD
  - no duplicate BDD subtrees, even across multiple BDDs
  - every time a new node is created, check for existence first
  - sometimes called the “unique table”
  - implemented as set of hash tables, one per Boolean variable
  - need: node referencing/dereferencing, garbage collection
- Efficiency implications
  - very significant memory savings
  - trivial checking of BDD equality (pointer comparison)
- Caching of BDD operation results for reuse
  - store result of every BDD operation (memory dependent)
  - applied at every step of recursive BDD operations
  - relies on fast check for BDD equality

# Operations with BDDs

- Operations on sets of states easy with BDDs
  - set union:  $A \cup B$ , in BDDs:  $A \vee B$
  - set intersection:  $A \cap B$ , in BDDs:  $A \wedge B$
  - set complement:  $S \setminus A$ , in BDDs:  $\neg A$
- Graph-based algorithms (e.g. reachability)
  - need forwards or backwards image operator
    - i.e. computation of all successors/predecessors of a state
    - again, easy with BDD operations (conjunction, quantification)
  - other ingredients
    - set operations (see above)
    - equality of state sets (fixpoint termination) – equality of BDDs

# Summing up...

- Implementation of model checking
  - graph-based algorithms, e.g. reachability, SCC detection
  - manipulation of sets of states, transition relations
- Binary decision diagrams (BDDs)
  - representation for Boolean functions
  - efficient storage/manipulation of sets, transition relations
  - suits symbolic of (e.g.) CTL model checking

# 20. Probabilistic Model Checking



Computer-Aided Verification

Dave Parker

University of Birmingham

2017/18

# Reminders & updates

- No lectures next week
- Assessment 4 (SPIN)
  - due 12 noon Thur 22 Mar
  - help: Facebook, email, office hours, ...
- Exam & revision
  - revision lecture at start of summer term
  - see message next week about content/resources

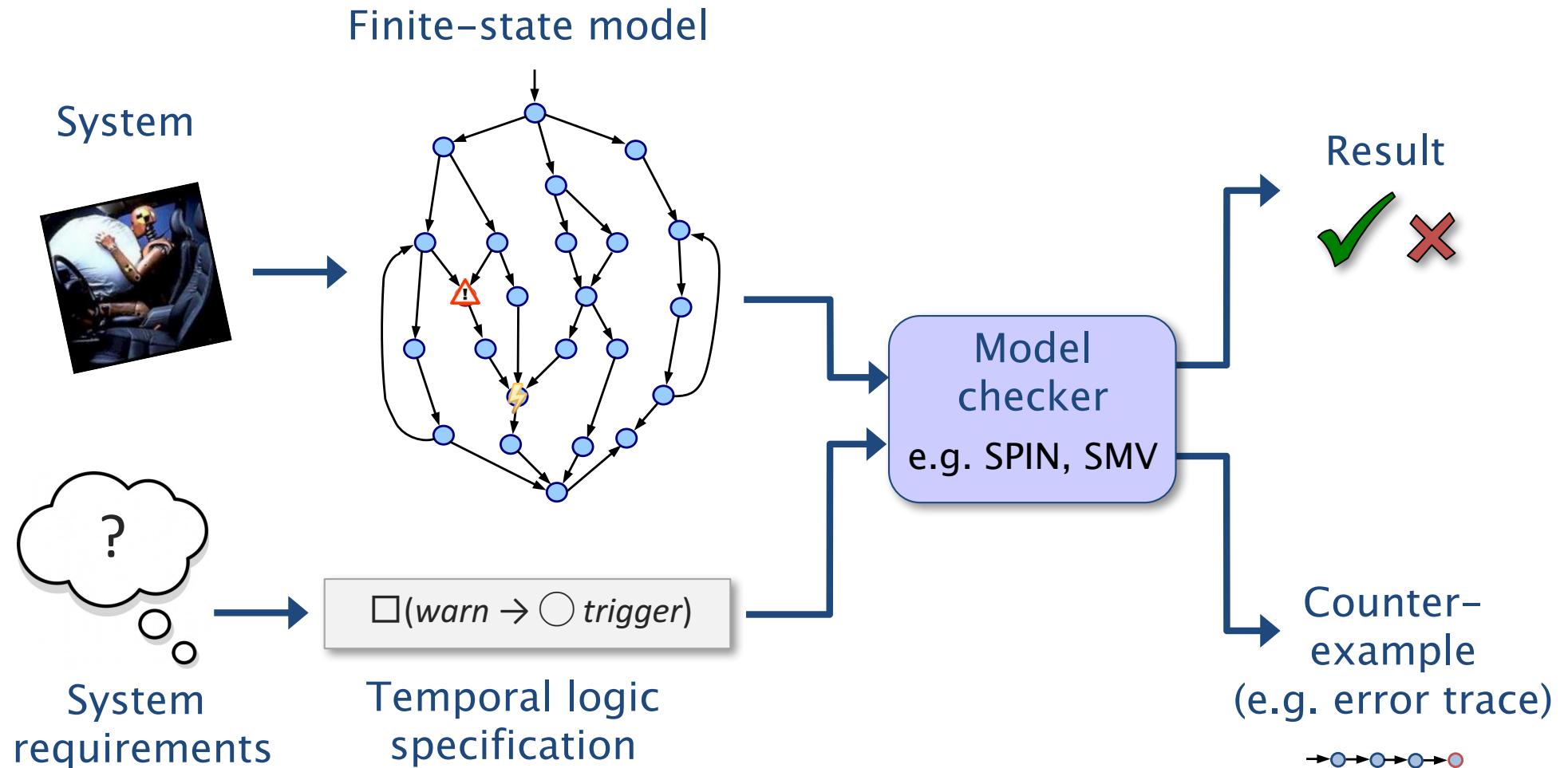
# Module syllabus

- Modelling sequential and parallel systems
  - labelled transitions systems, parallel composition
- Temporal logic
  - LTL, CTL and CTL\*, etc.
- Model checking
  - CTL model checking algorithms
  - automata-theoretic model checking (LTL)
- Verification tools: SPIN
- Advanced verification techniques
  - bounded model checking via propositional satisfiability
  - symbolic model checking
  - probabilistic model checking

# Overview

- Quantitative verification
  - motivation
  - application areas
- Probabilistic model checking
  - discrete-time Markov chains (DTMCs)
  - probabilistic temporal logic (PCTL)
- Background reading:
  - "[Quantitative Verification: Formal Guarantees for Timeliness, Reliability and Performance](#)"
  - PRISM: <http://www.prismmodelchecker.org/>
  - [BK08] Chapter 10

# Verification via model checking

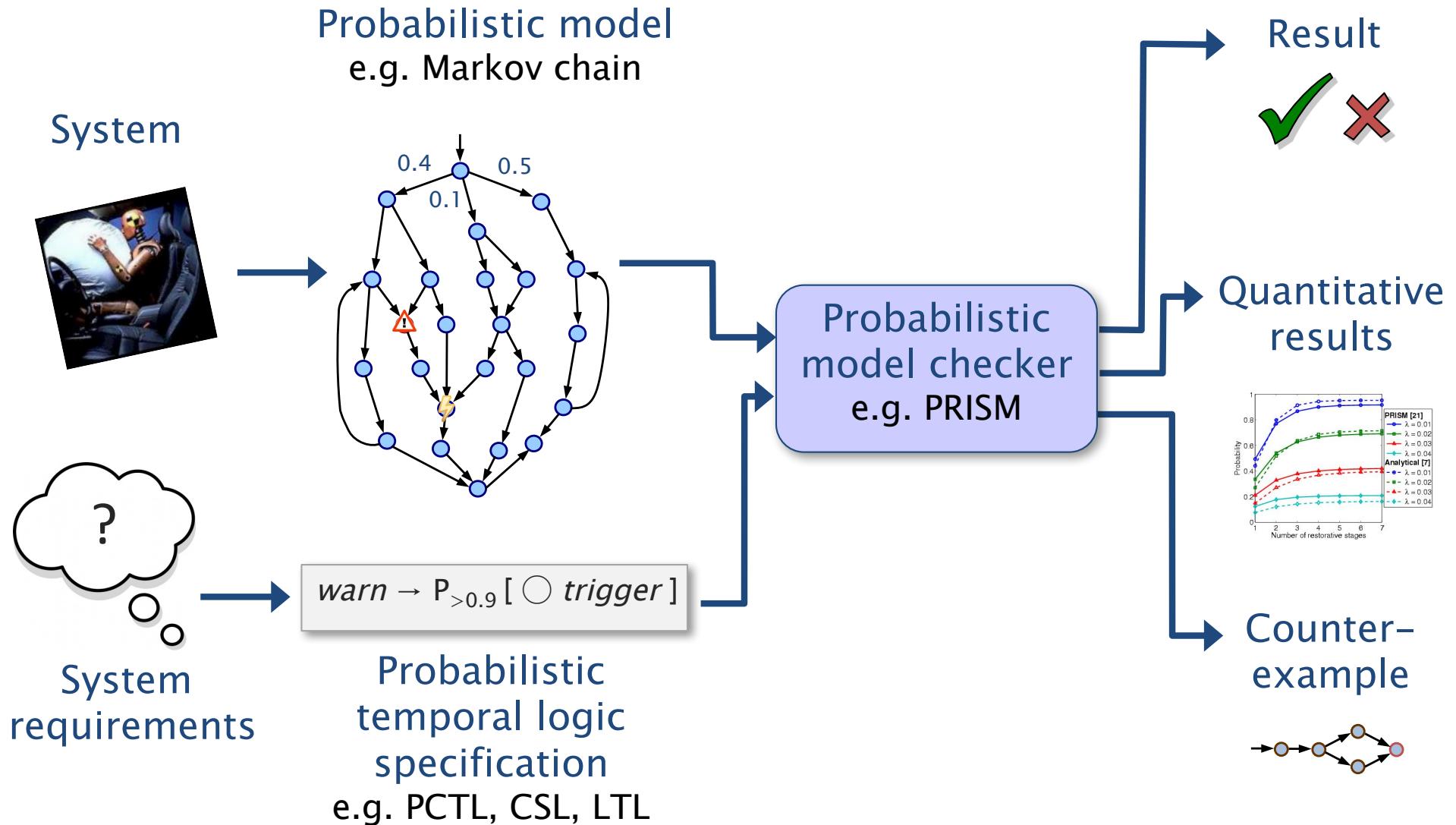


# Motivation

- Verifying probabilistic systems...
  - unreliable or unpredictable behaviour
    - failures of physical components
    - unreliable sensors/actuators
    - message loss in wireless communication
  - randomisation in algorithms/protocols
    - random back-off in communication protocols
    - random routing to reduce flooding or provide anonymity
- We need to verify quantitative system properties
  - “the probability of the airbag failing to deploy within 0.02 seconds of being triggered is at most 0.001”
  - “with probability 0.99, the packet arrives within 10 ms”



# Probabilistic model checking



# Probabilistic model checking

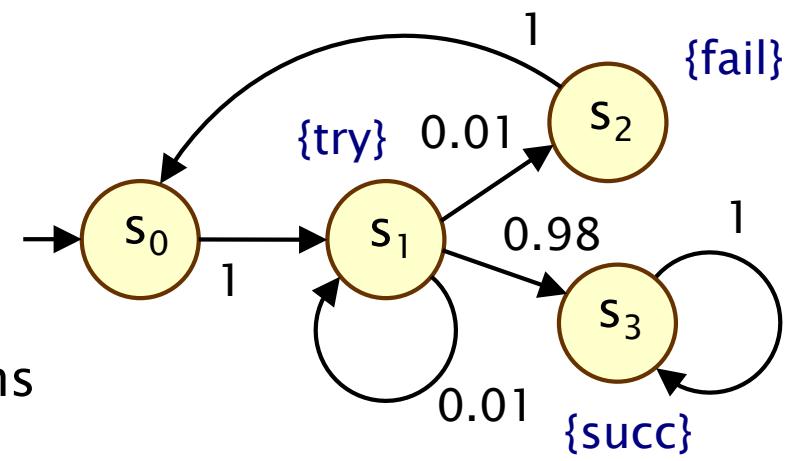
- Construction and analysis of finite probabilistic models
  - e.g. Markov chains, Markov decision processes, ...
  - specified in high-level modelling formalisms
  - **exhaustive** model exploration (all possible states/executions)
- Automated analysis of wide range of quantitative properties
  - properties specified using temporal logic
  - “**exact**” results obtained via numerical computation
  - linear equation systems, iterative methods, uniformisation, ...
  - as opposed to, for example, Monte Carlo simulations
  - efficient techniques from verification + performance analysis
  - mature tool support available, e.g. PRISM

# Case studies

- Randomised communication protocols
  - Bluetooth, FireWire, Zeroconf, 802.11, Zigbee, gossiping, ...
- Security protocols/systems
  - pin cracking, anonymity, quantum crypto, contract signing, ...
- Performance & reliability
  - airbag controller, nanotechnology, cloud computing, ...
- Planning & controller synthesis
  - robotics, autonomous driving, dynamic power management, ...
- And many more
  - cell signalling pathways, DNA computing, randomised algorithms
  - see: [www.prismmodelchecker.org/casestudies](http://www.prismmodelchecker.org/casestudies)

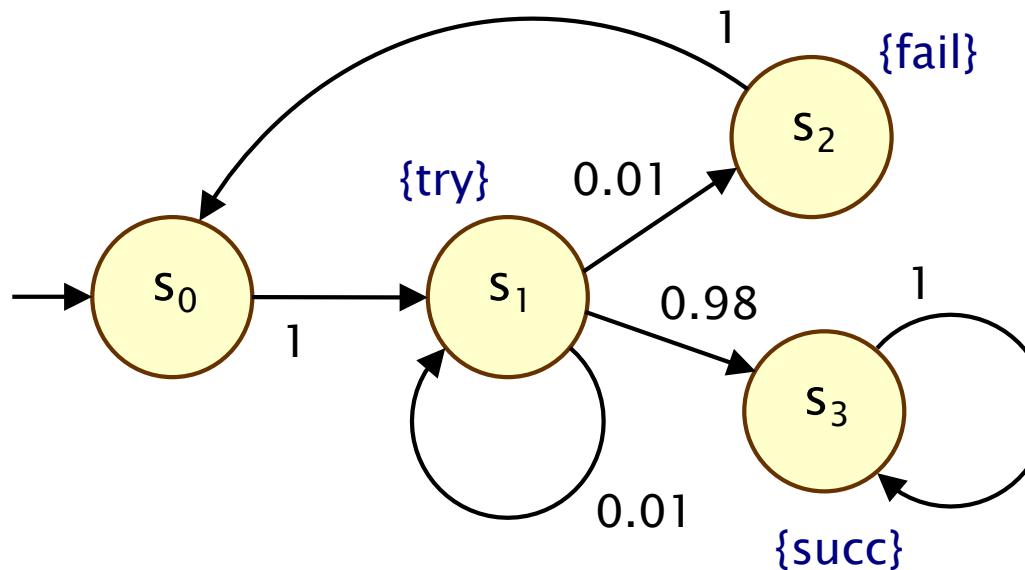
# Discrete-time Markov chains

- Discrete-time Markov chains (DTMCs)
  - labelled transition systems augmented with probabilities
- States
  - set of states representing possible configurations of the system being modelled
- Transitions
  - transitions between states model evolution of systems state; occur in discrete time-steps
- Probabilities
  - probabilities of making transitions between states are given by discrete probability distributions



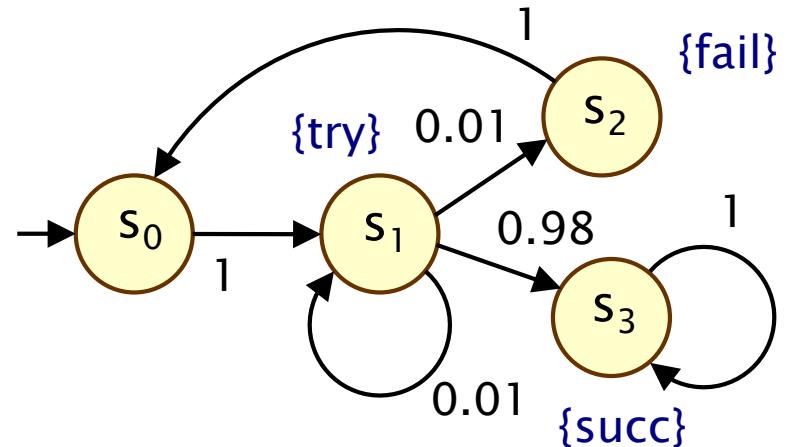
# Simple DTMC example

- Modelling a very simple communication protocol
  - after one step, process starts trying to send a message
  - with probability 0.01, channel unready so wait a step
  - with probability 0.98, send message successfully and stop
  - with probability 0.01, message sending fails, restart



# Discrete-time Markov chains

- Formally, a DTMC  $D$  is
  - a tuple  $(S, s_{\text{init}}, P, L)$
- where:
  - $S$  is a set of **states** (“state space”)
  - $s_{\text{init}} \in S$  is the **initial state**
  - $P : S \times S \rightarrow [0,1]$  is the **transition probability matrix**
    - where  $\sum_{s' \in S} P(s, s') = 1$  for all  $s \in S$
  - $AP$  is a set of **atomic propositions**
  - $L : S \rightarrow 2^{AP}$  is a **labelling function**
- Transition probabilities
  - $P(s, s')$  gives the probability of moving from  $s$  to  $s'$

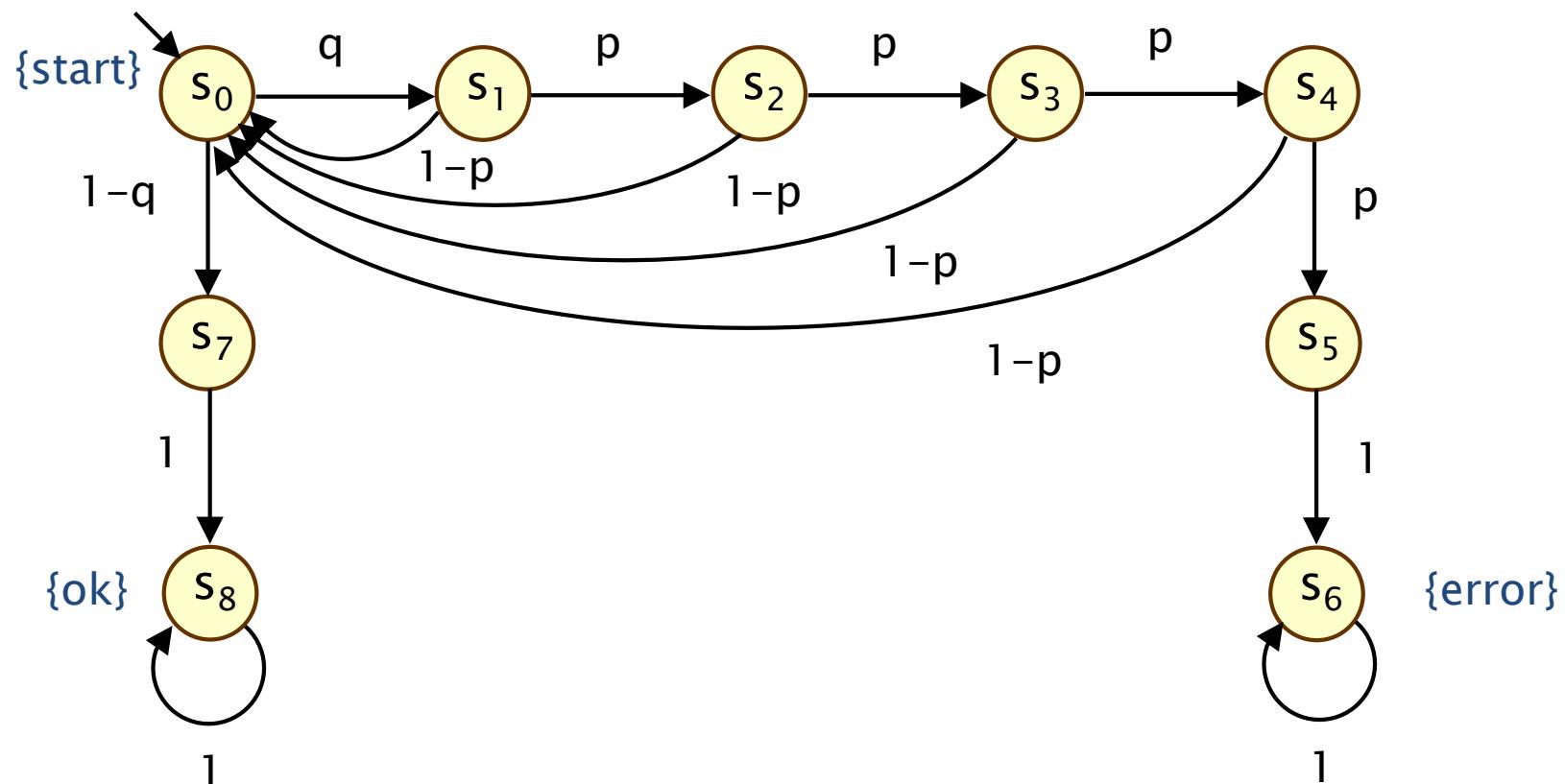


# DTMC example – Zeroconf

- Zeroconf = “Zero configuration networking”
  - self-configuration for local, ad-hoc networks
  - automatic configuration of unique IP for new devices
  - simple; no DHCP, DNS, ...
- Basic idea:
  - 65,024 available IP addresses (IANA-specified range)
  - new node picks address U at random
  - broadcasts “probe” messages: “Who is using U?”
  - any node already using U replies; protocol restarts
  - messages may not get sent (transmission fails, host busy, ...)
  - so: nodes send multiple (n) probes, waiting after each one

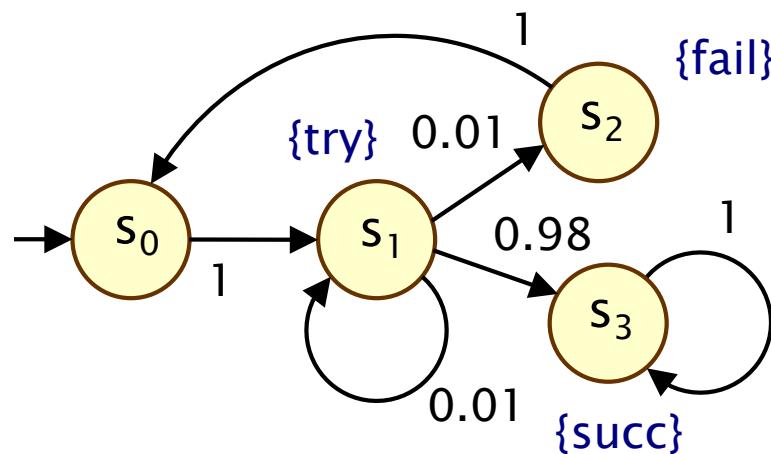
# DTMC for Zeroconf

- $n=4$  probes,  $m$  existing nodes in network
- probability of message loss:  $p$
- probability that new address is in use:  $q = m/65024$



# Paths in DTMCs

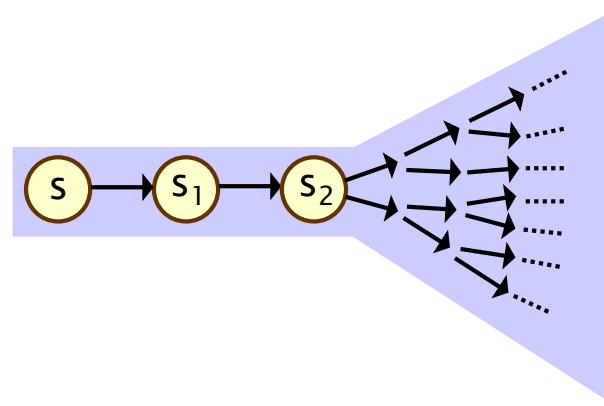
- A (finite or infinite) path through a DTMC
  - is a sequence of states  $s_0 s_1 s_2 s_3 \dots$  such that  $P(s_i, s_{i+1}) > 0 \forall i$
  - represents an execution (i.e. one possible behaviour) of the system which the DTMC is modelling
  - Paths( $s$ ) is the set of all (infinite) paths starting in  $s$



- Examples:
  - never succeeds:  $(s_0 s_1 s_2)^\omega$
  - tries, waits, fails, retries, succeeds:  $s_0 s_1 s_1 s_2 s_0 s_1 (s_3)^\omega$

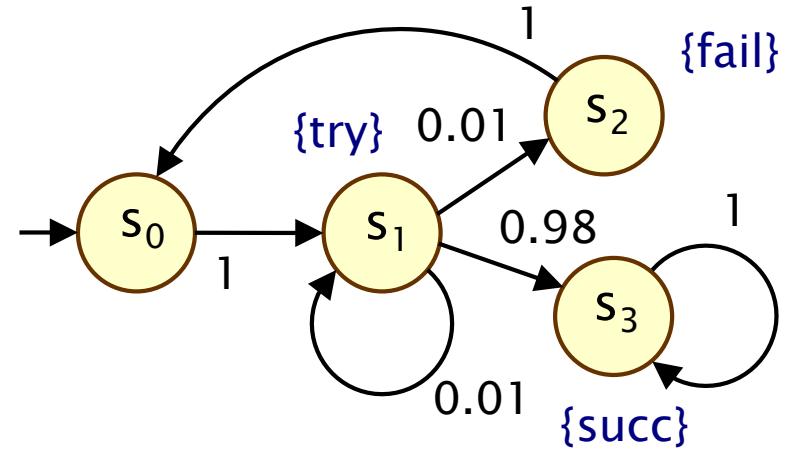
# Paths and probabilities

- To reason (quantitatively) about this system
  - need to define a **probability measure** over paths
- More precisely:
  - probability measure  $\text{Pr}_s$  over  $\text{Paths}(s)$
  - basic idea: defined on finite paths, extended to infinite paths
  - $\mathbf{P}(ss_1s_2) = \mathbf{P}(s,s_1)\mathbf{P}(s_1,s_2)$



# Paths and probabilities

- Examples
- “try and fail immediately”
  - paths starting with prefix  $s_0s_1s_2$
  - probability :  $P(s_0s_1s_2)$   
 $= P(s_0,s_1)P(s_1,s_2) = 1 \cdot 0.01 = 0.01$



- “eventually successful and with no failures”

– paths  $s_0s_1s_3\dots$ ,  $s_0s_1s_1s_3\dots$ ,  $s_0s_1s_1s_1s_3\dots$ , ...

– probability:

$$= P_{s0}(s_0s_1s_3) + P_{s0}(s_0s_1s_1s_3) + P_{s0}(s_0s_1s_1s_1s_3) + \dots$$

$$= 1 \cdot 0.98 + 1 \cdot 0.01 \cdot 0.98 + 1 \cdot 0.01 \cdot 0.01 \cdot 0.98 + \dots$$

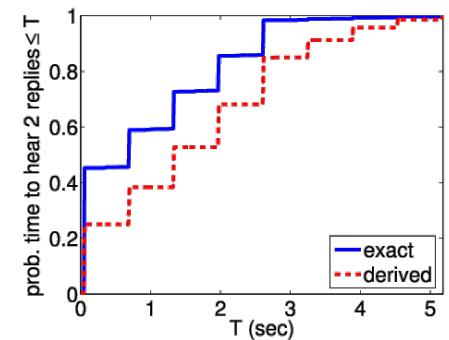
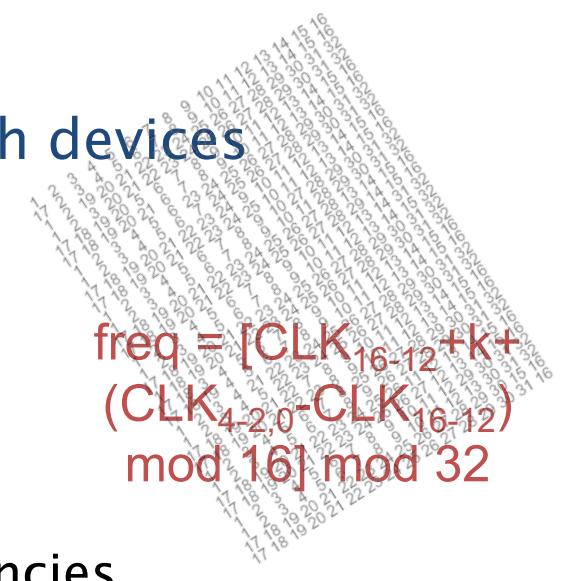
$$= 0.9898989898\dots$$

$$= \frac{98}{99}$$

In practice,  
computed by  
solving linear  
equation systems

# Case study: Bluetooth

- Device discovery between a pair of Bluetooth devices
  - performance essential for this phase
- Complex discovery process
  - two asynchronous 28-bit clocks
  - pseudo-random hopping between 32 frequencies
  - random waiting scheme to avoid collisions
  - 17,179,869,184 initial configurations
- Probabilistic model checking (PRISM)
  - “probability discovery time exceeds 6s is always < 0.001”
  - “worst-case expected discovery time is at most 5.17s”



# PCTL

- Temporal logic for describing properties of DTMCs
  - PCTL = Probabilistic Computation Tree Logic
- Extension of (non-probabilistic) temporal logic CTL
  - key addition is **probabilistic operator P**
  - quantitative extension of CTL's  $\forall$  and  $\exists$  operators
- Example
  - $\text{send} \rightarrow P_{\geq 0.95} [\lozenge^{\leq 10} \text{deliver}]$
  - “if a message is sent, then the probability of it being delivered within 10 steps is at least 0.95”

# CTL syntax

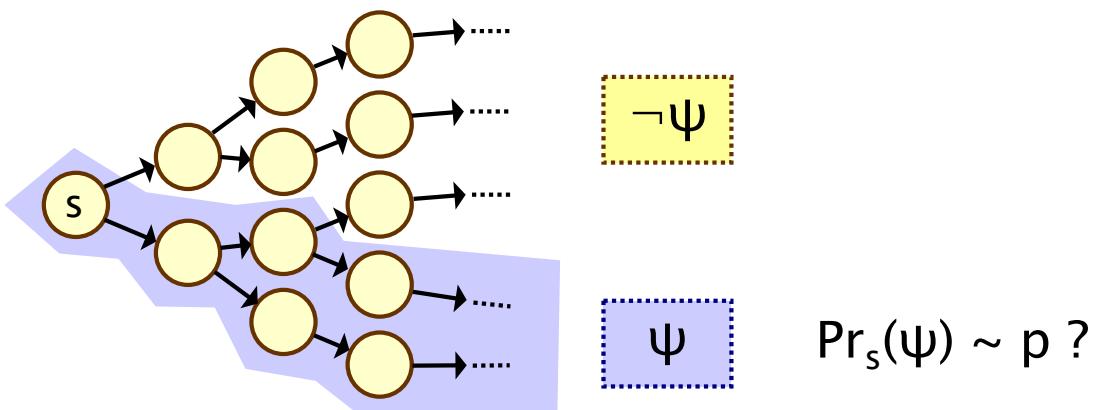
- Syntax split into state and path formulae
  - specify properties of states/paths, respectively
  - a CTL formula is a state formula  $\phi$
- State formulae:
  - $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \forall \psi \mid \exists \psi$
  - where  $a \in AP$  and  $\psi$  is a path formula
- Path formulae
  - $\psi ::= \bigcirc \phi \mid \phi \cup \phi \mid \dots$
  - where  $\phi$  is a state formula

# PCTL syntax

- Syntax split into state and path formulae
  - specify properties of states/paths, respectively
  - a PCTL formula is a state formula  $\phi$
- State formulae:
  - $\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid P_{\sim p}[\psi]$
  - where  $a \in AP$  and  $\psi$  is a path formula,  
 $p \in [0,1]$  is a probability bound,  $\sim \in \{<, >, \leq, \geq\}$
- Path formulae
  - $\psi ::= \bigcirc \phi \mid \phi \cup \phi \mid \phi \cup^{\leq k} \phi \mid \dots$
  - where  $\phi$  is a state formula,  $k \in \mathbb{N}$

# PCTL semantics for DTMCs

- Semantics of the probabilistic operator  $P$ 
  - example:  $s \models P_{<0.25} [\Diamond \text{fail}] \Leftrightarrow$  “the probability of atomic proposition fail being true in the next state of outgoing paths from  $s$  is less than 0.25”
  - informal definition:  $s \models P_{\sim p} [\psi]$  means that “**the probability, from state  $s$ , that  $\psi$  is true for an outgoing path satisfies  $\sim p$** ”
  - formally:  $s \models P_{\sim p} [\psi] \Leftrightarrow \Pr_s \{\pi \in \text{Path}(s) \mid \pi \models \psi\} \sim p$



# PCTL examples

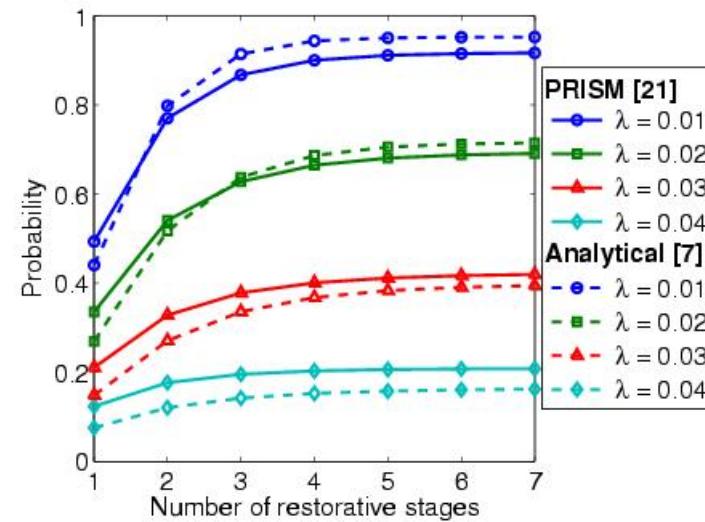
- $P_{\leq 0.05} [ \lozenge \text{err/total} > 0.1 ]$ 
  - “with probability at most 0.05, more than 10% of the NAND gate outputs are erroneous”
- $P_{\geq 0.8} [ \lozenge^{\leq k} \text{reply\_count} = n ]$ 
  - “the probability that the sender has received n acknowledgements within k clock-ticks is at least 0.8”
- $P_{< 0.4} [ \neg \text{fail}_A \cup \text{fail}_B ]$ 
  - “the probability that component B fails before component A is less than 0.4”
- $\neg \text{oper} \rightarrow P_{\geq 1} [ \lozenge ( P_{> 0.99} [ \square^{\leq 100} \text{oper} ] ) ]$ 
  - “if the system is not operational, it almost surely reaches a state from which it has a greater than 0.99 chance of staying operational for 100 time units”

# Qualitative vs. quantitative properties

- P operator of PCTL can be seen as a **quantitative** analogue of the CTL operators  $\forall$  (for all) and  $\exists$  (there exists)
- **Qualitative** PCTL properties
  - $P_{\sim p} [\psi]$  where p is either 0 or 1
- **Quantitative** PCTL properties
  - $P_{\sim p} [\psi]$  where p is in the range (0,1)
- $P_{>0} [\Diamond \phi]$  is identical to  $\exists \Diamond \phi$ 
  - there exists a finite path to a  $\phi$ -state
- $P_{\geq 1} [\Diamond \phi]$  is (similar to but) weaker than  $\forall \Diamond \phi$ 
  - a  $\phi$ -state is reached “almost surely”

# Numerical properties

- Consider a PCTL formula  $P_{\sim p} [\psi]$ 
  - if the probability is **unknown**, how to choose the bound  $p$ ?
- When the outermost operator of a PTCL formula is  $P$ 
  - PRISM allows formulae of the form  $P_{=?} [\psi]$
  - “**what is the probability that path formula  $\psi$  is true?**”
- Model checking is no harder: compute the values anyway
- Useful to spot patterns, trends
- Example
  - $P_{=?} [\Diamond \text{err/total} > 0.1]$
  - “**what is the probability that 10% of the NAND gate outputs are erroneous?**”

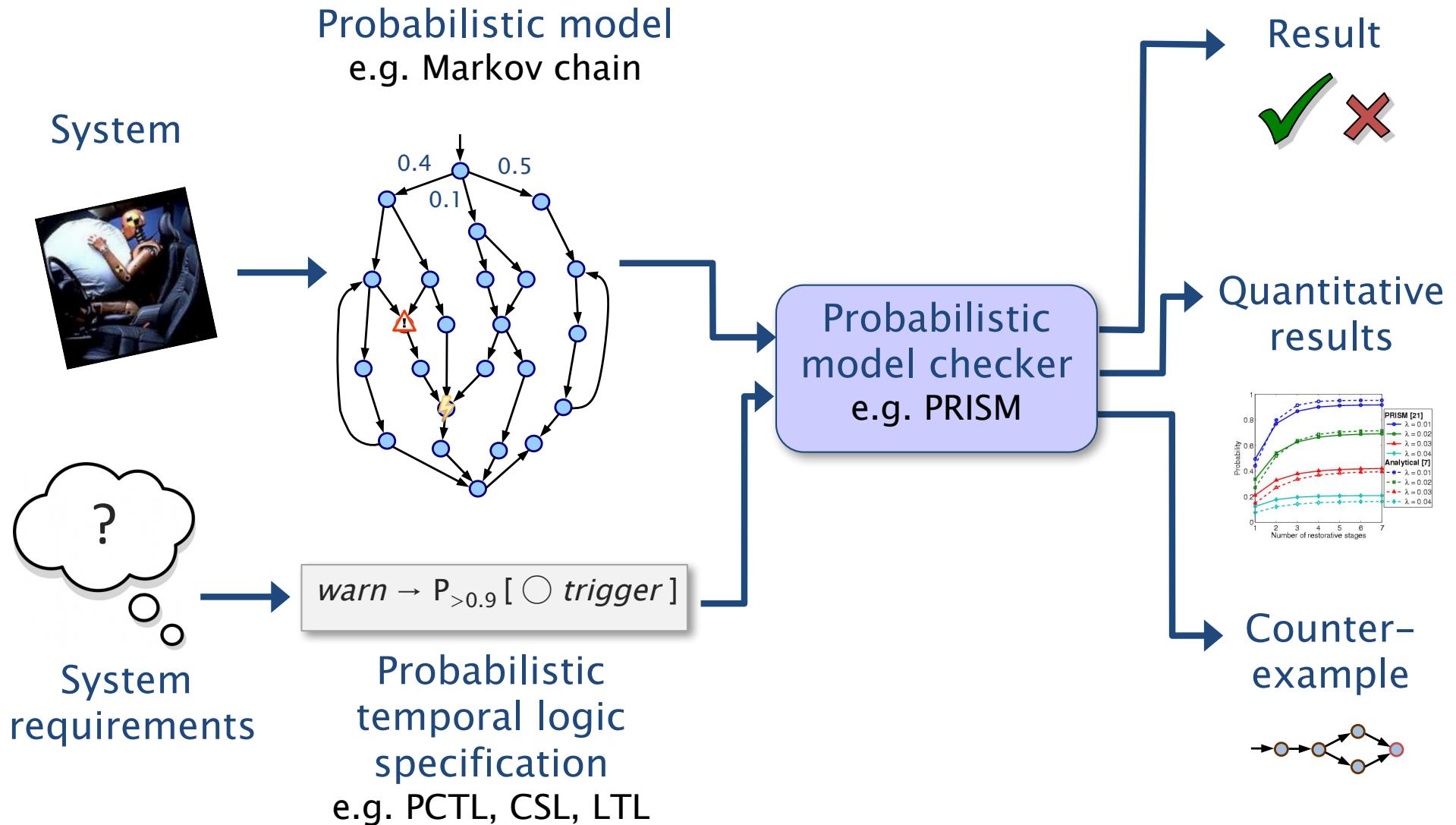


# Probabilistic model checking

- More specification formalisms
  - probabilistic LTL
  - e.g.  $P_{=?} (\Box \Diamond \text{ send})$ : “what is the probability that the protocol successfully sends a message infinitely often?”
  - e.g.  $P_{=?} (\neg \text{zone}_3 \cup (\text{zone}_1 \wedge (\Diamond \text{ zone}_4)))$ : “ what is the probability of visiting zone 1, without passing through zone 3, and then going to zone 4?”
  - PCTL\* (subsumes PCTL and probabilistic LTL)
  - costs, rewards, ...
- More probabilistic models
  - continuous-time Markov chains
    - adds a notion of real (not discrete) time
  - Markov decision processes...
    - adds nondeterminism

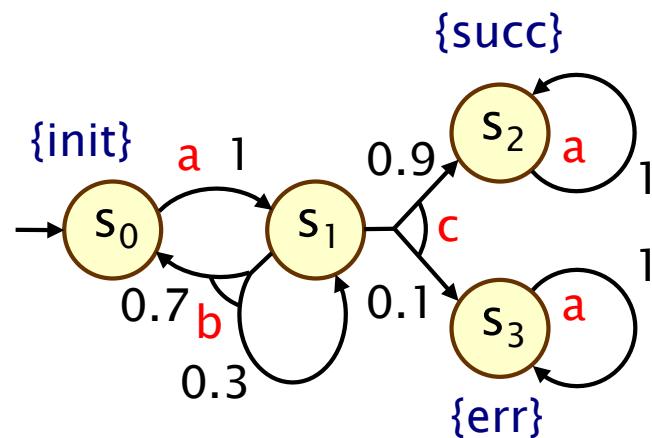


# Probabilistic model checking



# Markov decision processes (MDPs)

- Markov decision processes (MDPs)
  - model **nondeterministic** as well as **probabilistic** behaviour
  - widely used also in: AI, planning, optimal control, ...



- Nondeterminism for:
  - **control**: decisions made by a controller or scheduler
  - **adversarial** behaviour of the environment
  - **concurrency/scheduling**: interleavings of parallel components
  - **abstraction**, or under-specification, of unknown behaviour

# Summary

- Quantitative verification
  - reasoning about probability, time, ...
  - unreliable or unpredictable behaviour, randomisation
  - quantitative "correctness": reliability, timeliness, performance, ...
- Probabilistic model checking
  - discrete-time Markov chains (DTMCs)
  - paths, probability measures
  - probabilistic temporal logic (PCTL)
- PRISM
  - <http://www.prismmodelchecker.org/>

# Revision Lecture



Computer-Aided Verification

Dave Parker

University of Birmingham

2016/17

# Today

- Exam
  - details; revision; syllabus
- Module content summary
  - main topics covered
  - key points/ideas
  - feedback & tips from assignments
  - examples (e.g. from 2016 past paper)
- Questions, interruptions welcome...

# Exam

- Final module mark:
  - 20% continuous assessment (see Canvas) + 80% exam
- Exam
  - 1.5 hours (9.30am Thur 11 May)
  - 4 questions (answer all; equal weighting)
  - no appendix/supplementary material
- Revision resources
  - Assignments 1–3 (look carefully at model solutions)
  - also non-assessed exercises with solutions (CTL, BMC)
  - 2016 exam paper (my.bham, Canvas)
  - see also the Baier/Katoen book

# Module contents (examinable)

- Modelling sequential and parallel systems
  - labelled transitions systems, parallel composition
- Temporal logic
  - LTL, CTL and CTL\*, etc.
- Model checking
  - CTL model checking algorithms
  - automata-theoretic model checking (LTL)
- ~~Verification tools: SPIN~~
- Advanced verification techniques
  - bounded model checking via propositional satisfiability
  - ~~symbolic model checking~~
  - ~~probabilistic model checking~~

# Modelling systems

- Modelling sequential systems as LTSs
  - e.g. imperative programs, reactive systems
  - nondeterminism
- Modelling parallel systems: concurrency through interleaving
  - either asynchronous ( $M_1 \parallel M_2$ ) or synchronous ( $M_1 \parallel_H M_2$ )
  - also parallel programs/processes with shared variables
- Feedback & tips (Assignment 1)
  - always make clear what the states are
  - states: everything needed to capture dynamic (nothing more)
  - for a product, states should have both component parts of state
  - never duplicate states
  - lay states out logically when drawing LTSs

# 2016 past paper

- Qu 1a (drawing an LTS)

1. Below is a simple concurrent program, comprising two independent parallel processes accessing a shared integer variable  $x$ , which is initially set to 0.

```
l0 : while ($x \neq 1$) {
 l1 : $x := 2 - x;$
 }
 l2 : end
```

```
l0 : $x := 1;$
l1 : end
```

For convenience, lines of the program are labelled with program locations (of the form  $l_i$ ).

- (a) Draw a labelled transition system representing the system described above, where, apart from the mutual dependency on the variable  $x$ , the two processes operate asynchronously.

# Properties (of LTSs)

- Linear-time properties
  - infinite paths & traces (sequences of states & labellings)
  - properties are just sets of (“good”) traces
- Property classes, informally:
  - invariants: “something good is always true”
  - safety properties: “nothing bad happens” (in finite time)
  - liveness properties: “something good happens in the long run”
- Feedback & tips (Assignment 2)
  - property class (e.g. safety) does not relate to a specific model
  - to justify what class a property is, look carefully at definitions: e.g. invariants (proposition), safety properties (bad prefix), liveness (all finite words extendable)

# Temporal logic

- Temporal logics: CTL, LTL, etc.
  - propositional logic + temporal operators ( $\bigcirc$ ,  $U$ ,  $\diamond$ ,  $\square$ )
  - CTL adds existential/universal quantification over paths
  - syntax, semantics, translation, satisfaction in an LTS
  - equivalences: proving, disproving
- Feedback & tips (English to logic)
  - always make clear what the atomic propositions are
  - remember common templates
    - e.g.  $\diamond a$ ,  $\square \neg b$ ,  $\neg a U b$ ,  $\square(a \rightarrow \diamond b)$ ,  $\square(a \rightarrow \bigcirc b)$ ,  $\square \diamond a$ ,  $\diamond \square b$ ,  $\diamond \square \neg a$
- Feedback & tips (equivalences)
  - equivalences: again not specific to a model
  - just remember common equivalences/dualities
    - esp. for propositional logic and individual temporal operators

# 2016 past paper

- Qu 1b (English to temporal logic)
  - (b) For each of the following properties, explain how it can be expressed in the specified temporal logic, with respect to the labelled transition system (LTS) above, state whether it is satisfied in the LTS and, if it is not, give a counterexample.
    - (i)  $x$  is always greater than or equal to 0 (in LTL);
    - (ii)  $x$  eventually becomes equal to 1 (in LTL);
    - (iii) it is always possible to reach a state where  $x > 0$  (in CTL);
    - (iv)  $x$  takes the value 2 infinitely often (in LTL);
    - (v) in any execution where  $x$  eventually equals 1,  $x$  is equal to 2 only finitely often (in LTL).

# 2016 past paper

- Qu 3c (expressive power)

- (c) Property (i) above requires  $b$  to be true immediately after  $a$ ; whereas property (ii) requires  $b$  to be true at some point in the future. An alternative property might be  $\square(a \rightarrow \Diamond^{\leq k} b)$ , which requires  $b$  to be true within  $k$  steps.

Formally, this uses a new temporal operator  $\Diamond^{\leq k}$ , which can be added to the existing semantics for LTL as follows. For integer  $k \geq 0$ , LTL formula  $\psi$  and any infinite trace  $\sigma$ , we have:

$$\sigma \models \Diamond^{\leq k} \psi \iff \exists j \leq k \text{ such that } \sigma[j \dots] \models \psi$$

Does this new operator  $\Diamond^{\leq k}$  add expressive power to LTL? Justify your answer.

# Model checking

- CTL model checking
  - conversion to existential normal form (ENF)
  - recursive computation of satisfying states  $\text{Sat}(\phi)$
  - basic set operations for propositional logic
  - look at transitions for  $\exists\bigcirc$ , graph algorithms for  $\exists U$ ,  $\exists \Box$
- Feedback & tips (model checking in general)
  - don't forget to validate your answers
  - show your working for all parts of the algorithms
  - if asked, relate back to original model
- CTL:
  - see unassessed exercises (Canvas) for CTL model checking

# Automata-based model checking

- Finite automata (NFAs) & regular languages
- Regular safety properties
  - bad prefixes represented by an NFA
  - model checking: reachable accept states in LTS–NFA product
  - $M \not\models P_{\text{safe}} \Leftrightarrow$  some path satisfies  $\Diamond \text{accept}$  in  $M \otimes \mathcal{A}$
- Büchi automata (NBAs) &  $\omega$ -regular languages (e.g. LTL)
  - model checking: accepting cycles in LTS–NFA product
  - $M \not\models \Psi \Leftrightarrow$  some path satisfies  $\Box \Diamond \text{accept}$  in  $M \otimes \mathcal{A}_{\neg\Psi}$
- Counterexamples/witnesses

# Automata-based model checking

- Feedback & tips
  - again: don't forget to validate your answers
  - and, if asked, relate back to original LTS (as well as product)
  - remember automata for common LTL formulae
  - also patterns/recipes in common automata
  - products: layout the LTS logically
  - initial state is not always  $(s_0, q_0)$

# Questions

- Questions
  - quickest/easiest on Facebook/email
  - office hour today (3–4pm)
  - further office hours next week (TBA)