Model Checking

(Summer 2022)

Jan Kretinsky

Chair for Foundations of Software Reliability and Theoretical Computer Science
Technical University of Munich

Organization

Language: English

Lecture: Tu 12:00–14:00 in 03.13.010
We 12:00–14:00 in 03.09.014
Lecturer: Jan Kretinsky, `jan.kretinsky@tum.de`
Office hours: by appointment

Tutorials: Mo 16:00–18:00 in 03.13.010
Tutors: Kush Grover, Muqsit Azeem

Misc: 4V+2Ü
Exam: date TBA
(please do take part in the exercises to prepare)
Bonus: implement your model checker

Web site: slides, exercises, announcements

`https://www.in.tum.de/i07/lehre/ss22/model-checking/`

Other

Preliminaries:

basic knowledge of logics, discrete structures, graph theory, . . .

Literature:

Baier, Katoen: Principles of Model Checking, MIT Press, 2008

Clarke, Grumberg, Peled: Model Checking, MIT Press, 1999

Clarke, Henzinger, Veith, Bloem: Handbook of model Checking, Springer, 2018

Emerson: Temporal and Modal Logic, chapter 16 in Handbook of Theoretical Computer Science, vol. B, Elsevier, 1991

Vardi: An Automata-Theoretic Approach to Linear Temporal Logic, LNCS 1043, 1996

Holzmann: The SPIN Model Checker, Addison-Wesley, 2003

Part 1: Introduction

What does “Model Checking” mean?

What does “Model Checking” mean?

A technical term from logic

temporal logic: extension of predicate logic

in German: “Modellprüfung” (rarely used)

```
***STOP: 0x000000D1 (0x00000000, 0xF73120AE, 0xC0000008, 0xC0000000)

A problem has been detected and Windows has been shut down to prevent damage
to your computer

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen, restart your
computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a
new installation, ask your hardware or software manufacturer for any windows updates
you might need.

If problems continue, disable or remove any newly installed hardware or software.
Disable BIOS memory options such as caching or shadowing. If you need to use Safe
Mode to remove or disable components, restart your computer, press F8 to select
Advanced Startup Options, and then select Safe Mode.

**** ABCD.SYS - Address F73120AE base at C0000000, DateStamp 36B072A3

Kernel1 Debugger Using: COM2 (Port 0x2F8, Baud Rate 19200)
Beginning dump of physical memory
Physical memory dump complete. Contact your system administrator or
technical support group.
```

Motivation

Computer systems permeate more and more areas of our lives:

PCs, mobile phone, GPS, ...

control systems in cars, planes, ...

in banks (ATMs, credit risk assessment)

Correspondingly, we require more and more dependable hardware and software systems;

but the more complex a system grows, the more difficult it becomes to protect it against mistakes or attacks.

Bugs can have substantial economic impact or even endanger lives.
Estimated cost of bugs in the US: 60 bn dollars per year (Source: Der Spiegel).

Pentium bug (1994)

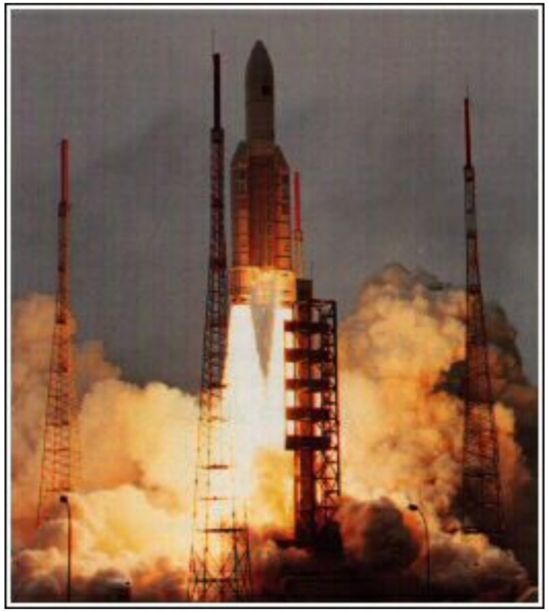
The Pentium CPU computed wrong results for certain floating point operations, e.g.

$$4195835 - (4195835/3145727) \times 3145727 = 256$$

Cause: for efficiency reasons, the division operation used a table with 1066 pre-computed entries of which five were wrong.

Estimated cost for exchanging the CPUs: 500 million dollars

Ariane 5 crash (1996)



Ariane 5 began to disintegrate 39 seconds after launch because of aerodynamic loads resulting from an angle of attack of more than 20 degrees.

Cause of the Ariane crash

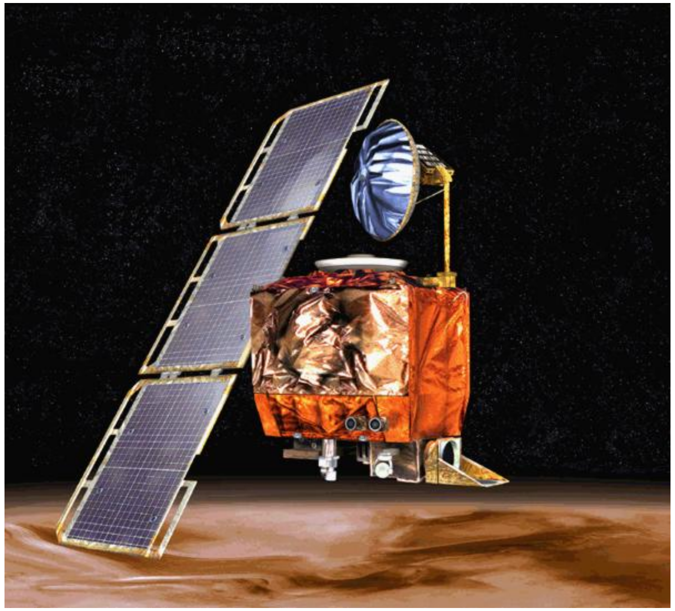
The angle of attack was caused by incorrect altitude data following a software exception.

The software exception was raised by an overflow in the conversion of a 64-bit floating-point number to a 16-bit signed integer value. The result was an operand error.

The operand error occurred because of an unexpected high value of the horizontal velocity sensed by the platform. The value was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in higher horizontal velocity values.

Direct cost 500.000.000 EUR, indirect cost 2.000.000.000 EUR

Loss of Mars Climate Orbiter (1999)



Cause: unchecked type mismatch of metric and imperial units.

Power shutdown on USS Yorktown (1998)

Cause: A sailor mistakenly typed 0 in a field of the kitchen inventory application. Subsequent divison by this field cause an arithmetic exception, which propagated through the system, crashed all LAN consoles and remote terminal units, and led to power shutdown for about 3 hours.

Unintended acceleration of Toyota cars (2009-2014)

Electronic Throttle Control System (ETCS) suspected .

NASA team investigated it in 2010-11, results inconclusive.

Further investigation in 2012-13 found unprotected critical variables, stack overflows, memory corruption.

Jury found that ETCS defects caused a death, experts testified ETCS is unsafe.

Toyota fined 1.2 billion dollars for concealing safety defects in 2014, although not directly for software bugs.

Apple’s SSL bug (2014)

Security check not performed due to a simple code error

```
...
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
...
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
err = sslRawVerify(...);
...
```

Approaches to solving the problem

Avoid bugs:

- Appropriate programming languages
- Software engineering methods

Detect bugs:

- Simulation, testing

Prove their absence:

- Deductive methods (Hoare)
- Program analysis

Detect bugs and prove their absence:

- Model checking

Simulation and testing

Can be used to find bugs in the design phase (simulation) or in the final product (testing).

Methods: Blackbox/whitebox testing, coverage metrics, etc.

Advantage: can find (obvious) bugs quickly and cost-efficiently

Disadvantage: incompleteness

> No coverage metric guarantees the absence of bugs, even at 100%, nor gives any estimate of the number of remaining bugs.
>
> Achieving complete coverage becomes more difficult with growing complexity.
>
> Concurrent systems very difficult to test.

Program analysis

Analyzes an overapproximation of the program

Symbolic execution on an abstract domain (like intervals)

Advantages:

- Can prove absence of standard bugs (e.g. division by 0, array out of bounds).
- Often quite efficient, applicable to large systems.

Disadvantages:

- Incompleteness, can produce large number of false alarms.
- Analyzes standard errors, not very specification oriented.

Deductive verification

Proofs using formal program semantics (Dijkstra, Hoare et al.)

Example: Hoare logic:

$$\{P\}\ S\ \{Q\}$$

Meaning: Whenever P holds before the execution of S, then Q holds afterwards.

Proof rules, e.g.

$$\{P\}\ \mathbf{skip}\ \{P\} \qquad \{P[x/e]\}\ x := e\ \{P\} \qquad \frac{\{P\}\ S_1\ \{Q\} \wedge \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Example: proof rule for loops

$$\{P\}\ \mathbf{while}\ \beta\ \mathbf{do}\ C\ \{Q\}$$

Show that there exists an invariant I with the following properties:

$$P \Rightarrow I \qquad \{I \wedge \beta\}\ C\ \{I\} \qquad I \wedge \neg\beta \Rightarrow Q$$

Termination: find a function $f(x, y, \ldots)$ of the program variables such that

$$\{\beta \wedge f(x, y, \ldots) = k\}\ C\ \{f(x, y, \ldots) < k\} \qquad f(x, y, \ldots) \leq 0 \Rightarrow \neg\beta$$

The program C is deemed correct if $\{true\}\ C\ \{P\}$ holds, where P is the function of interest.

Advantages and disadvantages

Advantages:

Complete; its power is limited only by the (human) prover.

Disadvantages:

see above

Onerous proofs “by hand” (help from theorem provers).

Very difficult for concurrent systems.

Summary

Simulation and testing can detect bugs but not prove their absence.
(They consider a subset of the possible executions.)

Deductive methods and program analysis can prove the absence of bugs, but can yield false positive.
(They consider a superset of the possible executions).

Model checking considers all and only the possible executions of a system

$\rightarrow$ detection of bugs *and* proof of their absence is both possible (in principle).
$\rightarrow$ computationaly costly (but can be merged with program analysis)
$\rightarrow$ particularly attractive for concurrent systems

Reactive systems

Examples: operating system, server, ATM, telephone switching system, . . .

Concurrent systems; no "function" is being computed, termination usually undesirable.

We are interested in certain properties of their executions, e.g.

No deadlocks.

No two processes can be in some "critical section" concurrently.

Whenever a process wants to enter a critical section it will eventually be able to do so.

All requests are eventually served.

$\Rightarrow$ formalization using temporal logics

Propositional logic (Syntax)

Formulae of propositional logic consist of atomic propositions, e.g.

$A \;\widehat{=}\;$ "Anna is an architect"

$B \;\widehat{=}\;$ "Bruno is a bear"

and connectives, e.g. $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not"), $\rightarrow$ ("implies").

Examples

Example formulae of propositional logic:

$A \wedge B$ (“Anna is an architect and Bruno is a bear”)

$\neg B$ (“Bruno is not a bear”)

Are such formulae true?

Answer: It depends.

Some formulae are always true ($A \vee \neg A$) or always false ($B \wedge \neg B$).

But in general, formulae are evaluated w.r.t. some valuation (or: “world”).

Semantics of propositional logic

A valuation $\mathcal{B}$ is a function assigning a truth value (1 or 0) to each atomic proposition.

The semantics of a formula (defined inductively) is the set of valuations making the formula "true" and denoted $[\![F]\!]$. E.g.,

if $F = A$ then $[\![F]\!] = \{\, \mathcal{B} \mid \mathcal{B}(A) = 1 \,\}$;

if $F = F_1 \wedge F_2$ then $[\![F]\!] = [\![F_1]\!] \cap [\![F_2]\!]$; ...

Other notations: $\mathcal{B} \models F$ iff $\mathcal{B} \in [\![F]\!]$.
We say: "$\mathcal{B}$ fulfils F" or "$\mathcal{B}$ is a model of F".

The model-checking problem in propositional logic

Problem: Given a valuation $\mathcal{B}$ and a formula F of propositional logic;
check whether $\mathcal{B}$ is a model of F.

Solution:

Replace the atomic propositions by their truth values in $\mathcal{B}$, then use a truth table to evaluate to 1 or 0.

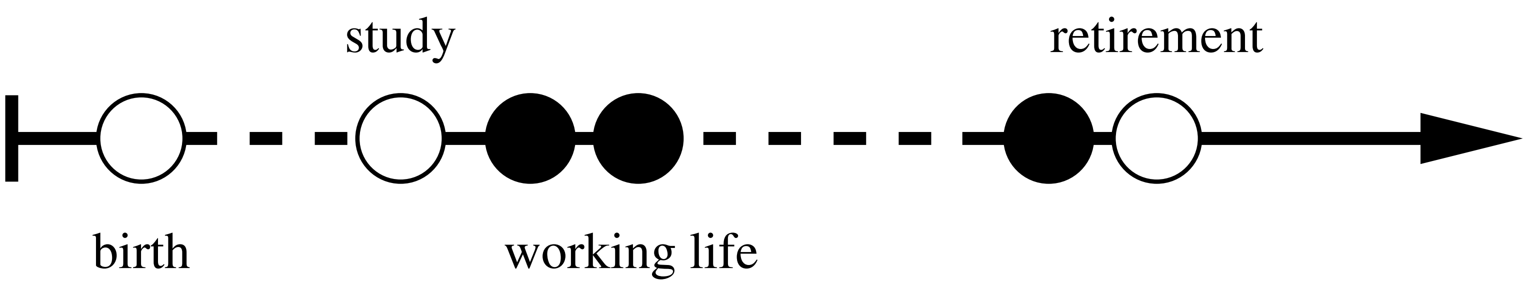
Examples: Let $\mathcal{B}_1(A) = 1$ and $\mathcal{B}_1(B) = 0$. Then $\mathcal{B}_1 \not\models A \wedge B$ and $\mathcal{B}_1 \models \neg B$.

Let $\mathcal{B}_2(A) = 1$ and $\mathcal{B}_2(B) = 1$. Then $\mathcal{B}_2 \models A \wedge B$ and $\mathcal{B}_2 \not\models \neg B$.

Temporal logic

Takes into account that truth values of atomic proposition may change with time (the “world” transforms).

Example: Truth values of *A* in the course of Anna’s life:



Possible statements:

Anna will eventually be an architect (at some point in the future).

Anna is an architect until she retires.

$\Longrightarrow$ Extension of propositional logic with temporal connectives (eventually, until)

Preview

Linear-time temporal logics (example: LTL)

formulae with temporal operators

evaluated w.r.t. (infinite) sequences of valuations

Model-checking problem for LTL: Given an LTL formula and a sequence of valuations, check whether the sequence is a model of the formula.

Computation-tree logic (CTL, CTL*)

Considers (infinite) *trees* of valuations.

Interpretation: non-determinism; multiple possible developments.

Connection with program verification

State space of a program:

value of program counter

contents of variables

contents of stack, heap, ...

Possible atomic propositions:

"Variable x has a positive value."

"The program counter is at label ℓ."

Given a set of atomic propositions, each program state gives rise to a valuation!

Programs and temporal logics

Linear-time temporal logic:

Each program execution yields a sequence of valuations.

Interpretation of the program: the set of possible sequences

Question of interest: Do all sequences satisfy a given LTL formula?

Computation-tree logic:

The program may branch at certain points, its possible executions yield a tree of valuations.

Interpretation of the program: tree with the (valuation of the) initial state as its root

Question of interest: Does this tree satisfy a given CTL formula?

Thus: verification problem $\widehat{=}$ model-checking problem

Model checking

Apart from its definition in terms of logic, the term model checking is generally understood to mean methods that

verify whether a given system satisfies a given specification;

work automatically;

either prove correctness of the system w.r.t. to the specification;

or exhibit a counterexample, i.e. an execution violating the specification (at least in the linear-time framework).

Pros and cons of model checking

Advantages:

works automatically(!)

suitable for reactive, concurrent, distributed systems

can check temporal-logic properties, not just reachability

Disadvantages:

Programs are generally Turing-powerful $\rightarrow$ undecidability

Approach: concentrate on decidable subclasses, here: finite automata; interesting connections to automata theory!

State space often very large $\rightarrow$ computationally expensive

approach: efficient algorithms and data structures

Problems with model checking

For the aforementioned problems we cannot hope to verify arbitrary properties of arbitrary programs!

Possibly we must consider a simplified mathematical model of the system of interest that ignores its “unimportant” aspects.

Construction of such models and the specification as well as the actual verification require effort and (possibly high) cost.

$\Rightarrow$ useful in early design phases

$\Rightarrow$ economic gain for critical systems where failure is costly (CPUs, communication protocols, aircraft, . . .)

Success stories of model checking

Since end of the 1970s: research on theoretical foundations

Since the 1990s: industrial applications

First hardware verification, later software verification:

verification of the cache protocol in the IEEE Futurebus+ (1992)

The tool SMV was able to find several bugs after four years of trying to validate the protocol with other means.

verification of the floating-point unit of the Pentium4 (2001)

Static Driver Verifier (Microsoft, 2000–2004) (Windows device drivers)

Research groups in big companies: IBM, Intel, Microsoft, OneSpin Solutions, ...

Turing award 2007 for its “inventors”: Clarke, Emerson, Sifakis

Goal of this course

The course teaches the fundamentals of model checking, its theory and applications, especially modelling systems, formulating specifications, and verifying them.

Modelling: transition systems, Kripke structures; tools: Spin, SMV

Specification: temporal logics (LTL, CTL)

Verification: fundamental techniques and extensions (partial-order reduction, BDDs, abstraction, bounded model checking)

Part 2: Kripke structures

System model

We shall use a very generic (and unspecific) model, i.e. transition systems, essentially directed graphs:

$$\mathcal{T} = (S, \rightarrow, r)$$

S	$\widehat{=}$	state space; states that the system may attain (finite or infinite set)
$\rightarrow \subseteq S \times S$	$\widehat{=}$	transition relation; describes which actions or "steps" are possible
$r \in S$	$\widehat{=}$	initial state ("root")

Example 1: Producer/Consumer

(Pseudocode) program with variables and concurrency:

var *turn* $\{0,1\}$ **init** 0;
cobegin $\{P \parallel K\}$ **coend**

$P =$

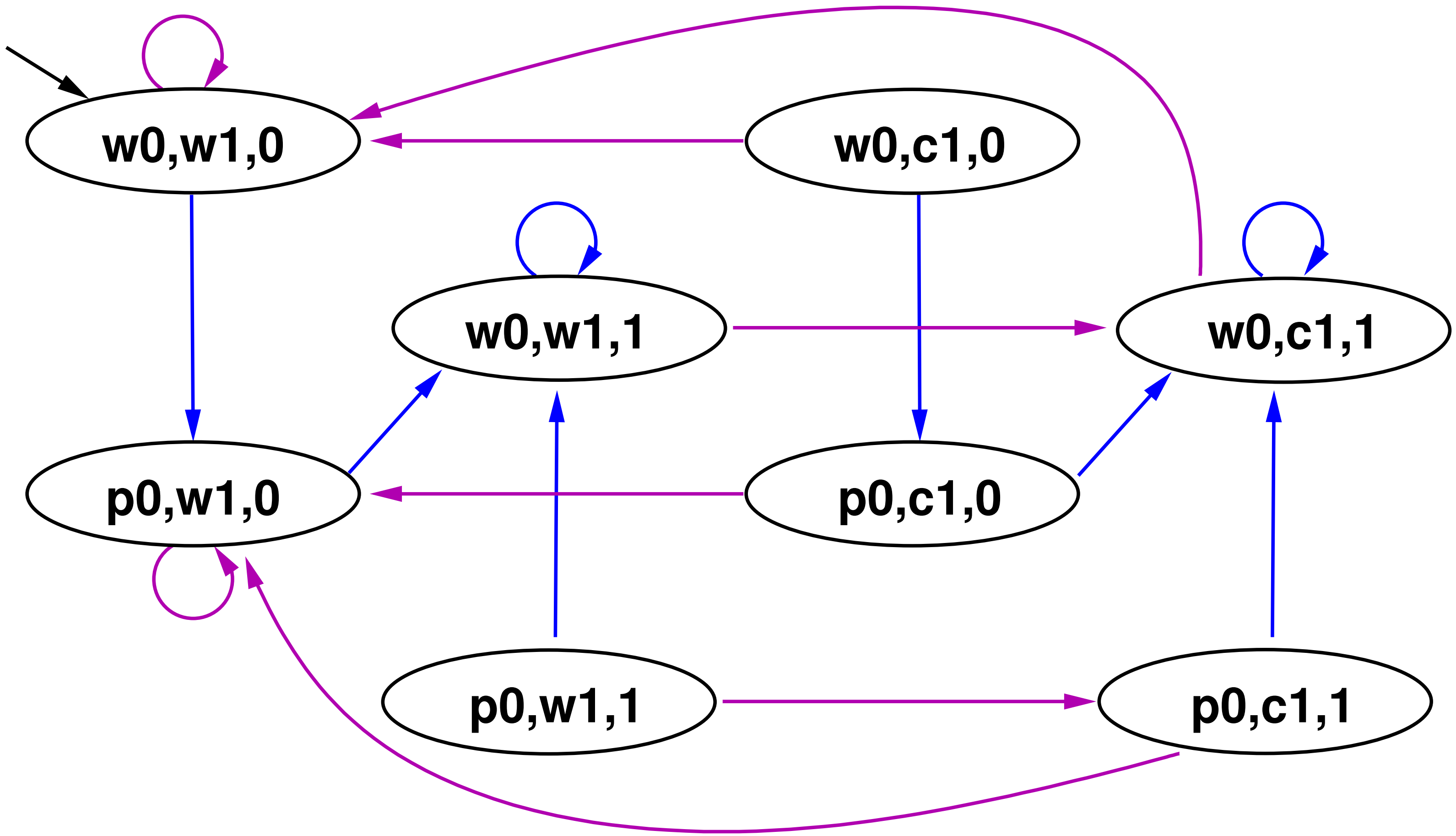
```
start;
while true do
    w0: wait (turn = 0);
    p0: /* produce */
        turn := 1;
od;
end
```

$K =$

```
start;
while true do
    w1: wait (turn = 1);
    c1: /* consume */
        turn := 0;
od;
end
```

Example 1: Corresponding transition system

$S = \{w_0, p_0\} \times \{w_1, c_1\} \times \{0, 1\}$; initial state $(w_0, w_1, 0)$



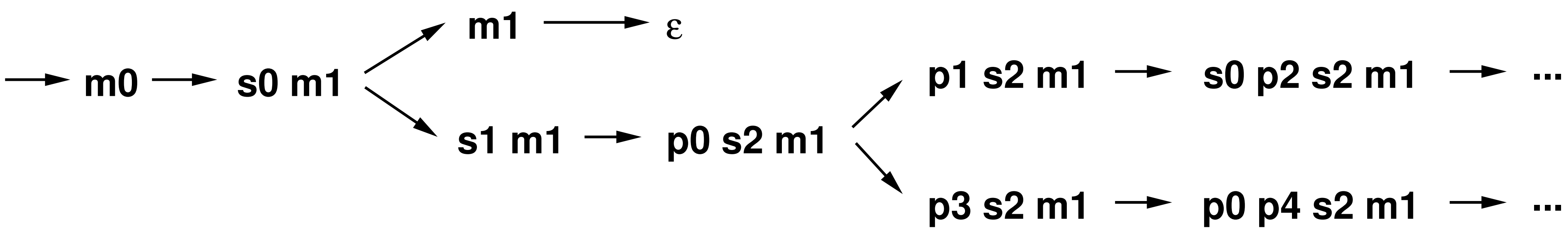
Example 2: Recursive Program

procedure p;
p_0: **if** ? **then**
p_1:   **call** s;
p_2:   **if** ? **then call** p; **end if**;
    else
p_3:   **call** p;
    end if
p_4: **return**

procedure s;
s_0: **if** ? **then return**; **end if**;
s_1: **call** p;
s_2: **return**;

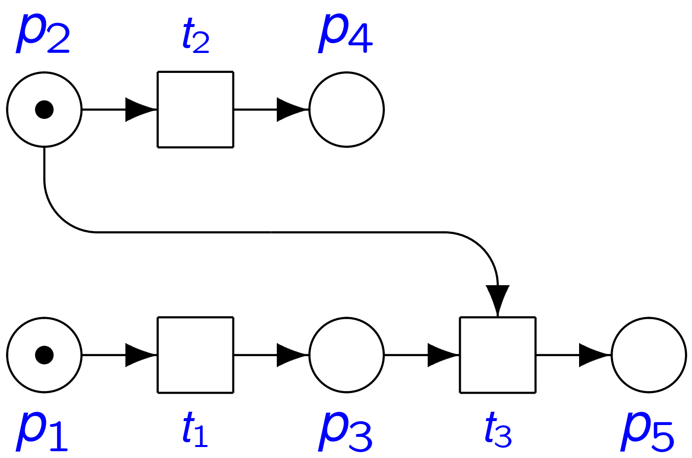
procedure $main$;
m_0: **call** s;
m_1: **return**;

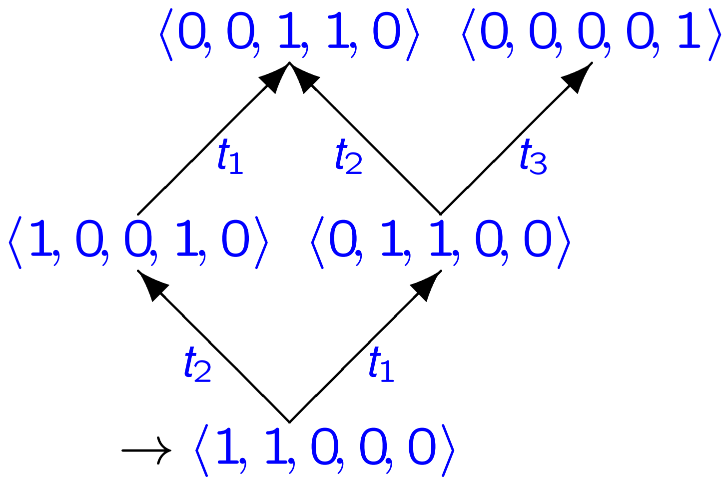
$S = \{p_0, \ldots, p_4, s_0, \ldots, s_2, m_0, m_1\}^*$, initial state m_0



Example 3: Petri net

State space = set of markings





Implicitly given transition systems

Quite often, a transition system is given to us “implicitly”, i.e. in the form of a program, from which we extract the transition system.

In such a setting, we are given an initial state and a function for computing the direct successor states of a given state, such that transitions are computed only on demand.

Some of our analysis methods will be suitable for such a setting.

Notation for transition systems

We write $s \to t$ if $(s, t) \in \to$.

If $s \to t$ then s is called a direct predecessor of t and t a direct successor of s.

S^* denotes the *finite*, S^ω the *infinite* sequences (words) over S.

$w = s_0 \ldots s_n$ is a path of length n if $s_i \to s_{i+1}$ for all $0 \leq i < n$.

$\rho = s_0 s_1 \ldots$ is an infinite path if $s_i \to s_{i+1}$ for all $i \geq 0$.

Notation for transition systems II

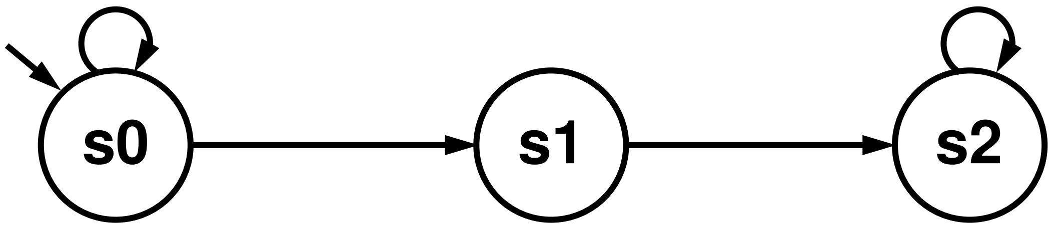
$\rho(i)$ denotes the i-th element of ρ and ρ^i the suffix starting at $\rho(i)$.

$s \rightarrow^* t$ if there is a path from s to t.

$s \rightarrow^+ t$ if there is a path from s to t with at least one transition.

If $s \rightarrow^* t$ then s is a predecessor of t and t a successor of s.

Example



$S = \{s_0, s_1, s_2\}$; initial state s_0

$s_0 \to s_0 \quad s_0 \to s_1 \quad s_1 \to s_2 \quad s_2 \to s_2$

$s_0 s_1 s_2$ is a path of length 2, i.e. $s_0 \to^* s_2$ and $s_0 \to^+ s_2$

$s_1 \to^* s_1$ but $s_1 \not\to^+ s_1$

$\rho = s_0 s_0 s_1 s_2 s_2 s_2 \ldots$ is an infinite path.

$\rho(2) = s_1 \qquad \rho^1 = s_0 s_1 s_2 s_2 s_2 \ldots$

Finite and infinite transition systems

In principle, a transition system many have infinitely many states. Some of the possible reasons are:

Data: integers, reals, lists, trees, pointer structures, . . .

Control: (recursive) procedures, dynamic thread creation, . . .

Communication: unbounded FIFO channels, . . .

Unknown parameters: number of participants in a protocol, . . .

Real time: discrete or continuous time

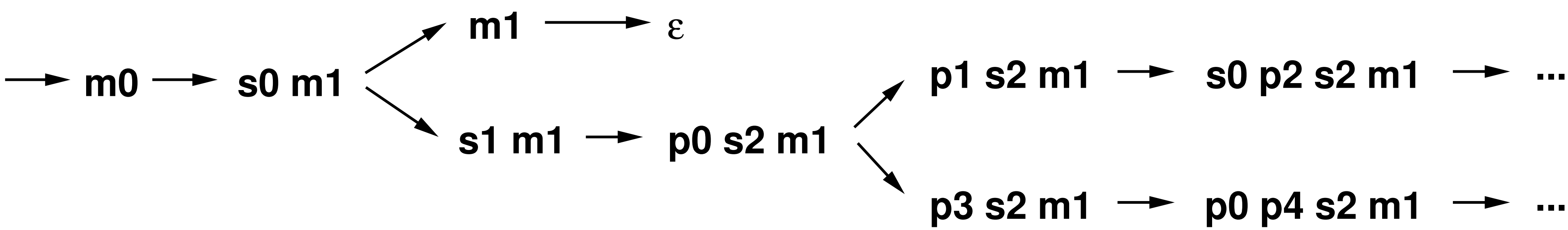
Some (not all!) of these features lead to Turing-powerful computation models (and thus undecidable verification problems).

Example: Recursive program

```
procedure p;
p0:  if ? then
p1:          call s;
p2:          if ? then call p; end if;
     else
p3:          call p;
     end if
p4:  return
```

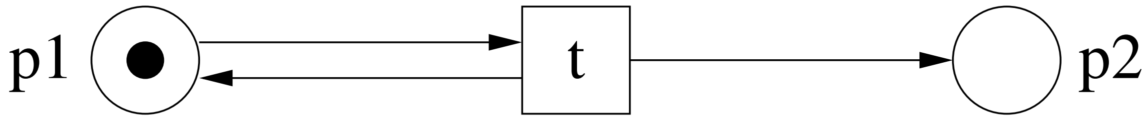
```
procedure s;
s0:  if ? then return; end if;
s1:  call p;
s2:  return;

procedure main;
m0:  call s;
m1:  return;
```

The state space of this example is infinite (stack!), however, LTL and CTL model checking remain decidable.



Example: Petri net

Petri nets may have an infinite state space, too:



Reachable states are: $\langle 1, 0\rangle,\ \langle 1, 1\rangle,\ \langle 1, 2\rangle,\ \ldots$

Reachability and LTL decidable, CTL undecidable.

Finite state spaces

For now, we restrict ourselves to finite state spaces.

Finite systems: e.g. hardware systems, programs with finite data typs (Boolean programs), certain communication protocols, . . .

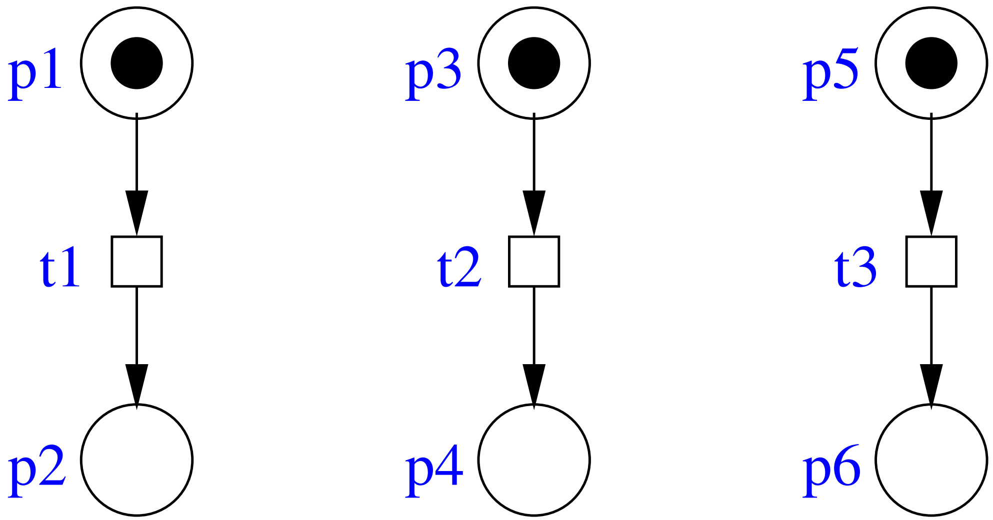
Finite systems may also be obtained by abstracting an infinite system.

Remaining problem: state-space explosion, systems may be finite but VERY large.

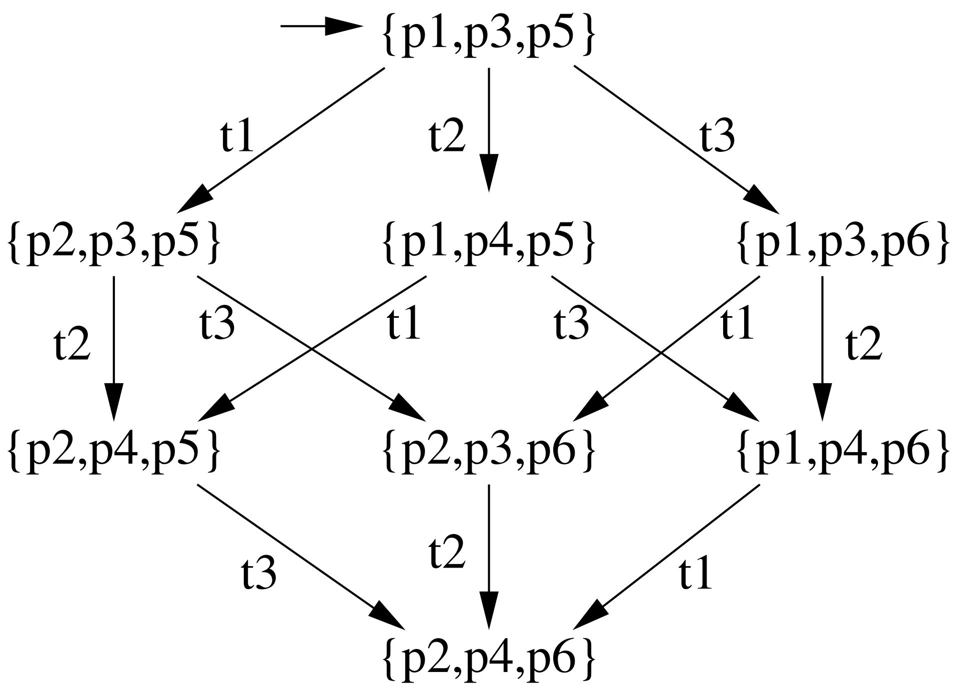
Reasons for state-space explosion (1)

A common reason is concurrency.

Example: Consider the following Petri net:



The reachability graph has got $8 = 2^3$ states and $6 = 3!$ paths.



With n components we have 2^n states and $n!$ paths.

Reasons for state-space explosion (2)

A second common reason is data.

e.g. programs with a few large or many small variables

size of state space: 2 to the number of bits

Counteractions:

Abstraction: ignore “unimportant” data

Compression: work with *sets* of states; efficient data structures for storing and manipulating sets

Approximation: find over- or underapproximations of the reachable states

We will see some examples of these techniques during the course.

Kripke structures

Idea: Extract from each state a valuation.

$$\mathcal{K} = (S, \to, r, AP, \nu)$$

$(S, \to, r)$	$\widehat{=}$	underlying transition system
AP	$\widehat{=}$	set of atomic propositions
$\nu\colon S \to 2^{AP}$	$\widehat{=}$	Interpretation of atomic propositions (“valuation”)

Remarks:

2^{AP} denotes the *powerset* of AP.

Valuations are represented here as subsets of AP rather than functions; the propositions contained in the set are those that are deemed true.

Example of a Kripke structure

Transition system $(S, \rightarrow, r)$ as in Example 1.

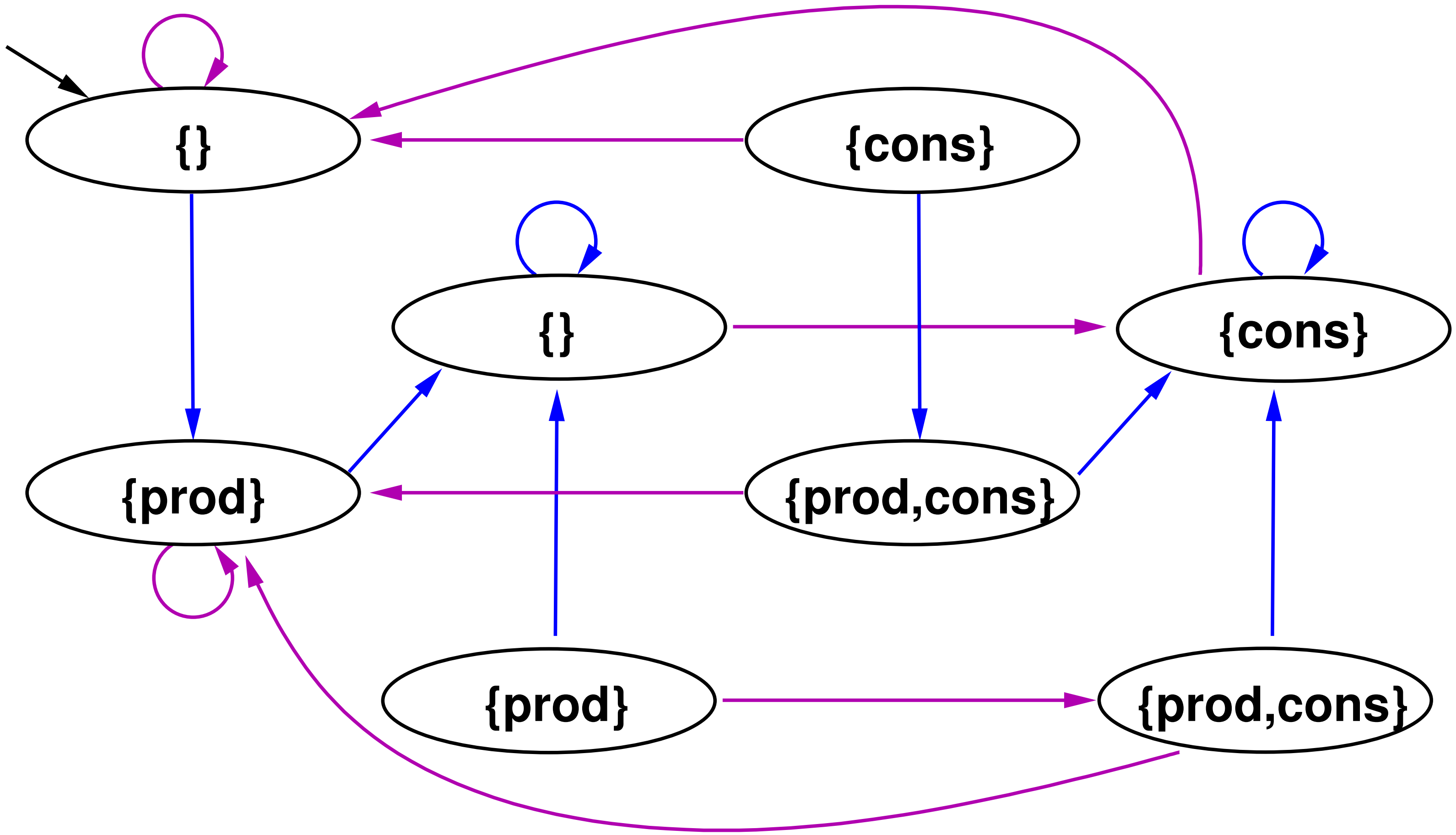
Suppose we are interested in the acts of production and consumption.

Let $AP = \{prod, cons\}$;

$\nu^{-1}(prod) = \{p_0\} \times \{w_1, c_1\} \times \{0, 1\}$;

$\nu^{-1}(cons) = \{w_0, p_0\} \times \{c_1\} \times \{0, 1\}$.

The valuations in Example 1:

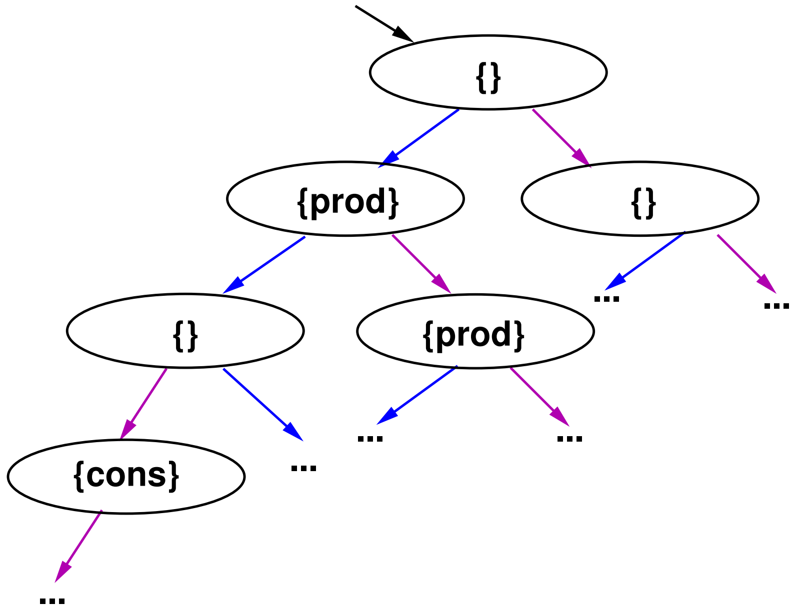


Sequences and trees of valuations

In linear-time logic we consider the possible valuation sequences:

e.g. $\emptyset\ \emptyset\ \{prod\}\ \emptyset\ \{cons\} \ldots$ or $\emptyset\ \{prod\}\ \{prod\}\ \{prod\}\ \ldots$

In computation-tree logic we consider the “tree-wise unfolding” of the Kripke structure:



Examples of temporal-logic properties

"It is never possible that *prod* and *cons* hold at the same time."

Intuitively, this property holds because no state in which both *prod* and *cons* holds is reachable from the beginning, which can be verified by inspecting the sequences and trees.

A property of this form is also called an invariant.

"Whenever something is produced it is also consumed afterwards."

We may take the view that this property does not hold because of the following sequence: $\emptyset\ \{prod\}\ \emptyset\ \emptyset\ \emptyset \ldots$ Thus, something is produced but followed by an infinite loop.

A property of this form is also called a reactivity property.

Fairness

We may also take the view that the second property merely fails because of overly simplistic modelling:

In the counterexample, only one process is acting, making “empty” steps, while the second process does not do anything.

Such a behaviour is usually unrealistic in concurrent systems; even if one process may be faster than another and execute multiple steps, a “fair” scheduler will eventually grant execution time to either process.

We may therefore wish to exclude such unrealistic (“unfair”) executions and only consider “fair” ones. In other words, we work under certain “fairness assumptions”.

Under a reasonable fairness assumption, the second property holds.

Part 3: Linear-time logic

Preliminaries

Linear-time logic in general:

- any logic working with sequences of valuations
- model: time progresses in discrete steps and in linear fashion, each point in time has exactly one possible future
- origins in philosophy/logic

Most prominent species: LTL

- in use for verification since end of the 1970s
- specification of correctness properties

Recap

Let AP be a set of atomic propositions.

2^{AP} denotes the powerset of AP, i.e. its set of subsets.

$(2^{AP})^{\omega}$ denotes the set of (infinite) sequences of valuations (of AP).

Syntax of LTL

Let *AP* be a set of atomic propositions. The set of LTL formulae over *AP* is inductively defined as follows:

If $p \in AP$ then p is a formula.

If ϕ_1, ϕ_2 are formulae then so are

$$\neg\phi_1, \qquad \phi_1 \vee \phi_2, \qquad \mathbf{X}\,\phi_1, \qquad \phi_1\,\mathbf{U}\,\phi_2$$

Intuitive meaning: $\mathbf{X} \mathrel{\widehat{=}}$ "next", $\mathbf{U} \mathrel{\widehat{=}}$ "until".

Remarks

This is a minimal syntax that we will use for proofs etc.

For added expressiveness, we will later define some abbreviations based on the minimal syntax.

Comparision of propositional logic (PL) and LTL:

	PL	LTL
Syntax	atomic proposition, logic operators	+ temporal operators
Evaluated on…	valuations	sequences of valuations
Semantics	set of valuations	set of valuation sequences

Semantics of LTL

Let ϕ be an LTL formula and σ a valuation sequence.
We write $\sigma \models \phi$ for "σ satisfies ϕ."

$$
\begin{array}{lll}
\sigma \models & p & \text{if } p \in AP \text{ and } p \in \sigma(0) \\
\sigma \models & \neg\phi & \text{if } \sigma \not\models \phi \\
\sigma \models & \phi_1 \vee \phi_2 & \text{if } \sigma \models \phi_1 \text{ or } \sigma \models \phi_2 \\
\sigma \models & \mathbf{X}\,\phi & \text{if } \sigma^1 \models \phi \\
\sigma \models & \phi_1 \,\mathbf{U}\, \phi_2 & \text{if } \exists i : \left(\sigma^i \models \phi_2 \;\wedge\; \forall k < i : \sigma^k \models \phi_1\right)
\end{array}
$$

Semantics of ϕ: $[\![\phi]\!] = \{\, \sigma \mid \sigma \models \phi \,\}$

Examples

Let $AP = \{p, q, r\}$. Find out whether the sequence

$$\sigma = \{p\}\ \{q\}\ \{p\}^\omega$$

satisfies the following formulae:

$$p$$

$$q$$

$$\mathbf{X}\, q$$

$$\mathbf{X}\, \neg p$$

$$p\ \mathbf{U}\ q$$

$$q\ \mathbf{U}\ p$$

$$(p \vee q)\ \mathbf{U}\ r$$

Extended syntax

We will commonly use the following abbreviations:

$$\begin{aligned} \phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2) \\ \phi_1 \rightarrow \phi_2 &\equiv \neg\phi_1 \vee \phi_2 \\ \mathbf{true} &\equiv p \vee \neg p \\ \mathbf{false} &\equiv \neg\mathbf{true} \end{aligned}$$

$$\begin{aligned} \mathbf{F}\,\phi &\equiv \mathbf{true}\,\mathbf{U}\,\phi \\ \mathbf{G}\,\phi &\equiv \neg\,\mathbf{F}\,\neg\phi \\ \phi_1\,\mathbf{W}\,\phi_2 &\equiv (\phi_1\,\mathbf{U}\,\phi_2) \vee \mathbf{G}\,\phi_1 \\ \phi_1\,\mathbf{R}\,\phi_2 &\equiv \neg(\neg\phi_1\,\mathbf{U}\,\neg\phi_2) \end{aligned}$$

Meaning: $\mathbf{F} \mathrel{\widehat{=}}$ “finally” (eventually), $\mathbf{G} \mathrel{\widehat{=}}$ “globally” (always),
$\mathbf{W} \mathrel{\widehat{=}}$ “weak until”, $\mathbf{R} \mathrel{\widehat{=}}$ “release”.

Some example formulae

Invariant: $\mathbf{G}\,\neg(cs_1 \wedge cs_2)$

cs_1 and cs_2 are never true at the same time.

Remark: This particular form of invariant is also called mutex property (“mutual exclusion”).

Safety: $(\neg p)\ \mathbf{W}\ q$

p does not occur before *q* has happend.

Remark: It may happen that *q* never happens in which case *p* also never happens.

More examples

Liveness:

$\mathbf{G}\,\mathbf{F}\,p$

p occurs infinitely often.

$\mathbf{F}\,\mathbf{G}\,p$

At some point *p* will continue to hold forever.

$\mathbf{G}(try_1 \rightarrow \mathbf{F}\,cs_1)$

For mutex algorithms: Whenever process 1 tries to enter its critical section it will eventually succeed.

Conjunction of safety and liveness: $(\neg p)\ \mathbf{U}\ q$

p does not occur before *q* and *q* eventually happens.

Tautology, equivalence

Certain concepts from propositional logic can be transferred to LTL.

Tautology: A formula ϕ with $[\![\phi]\!] = (2^{AP})^\omega$ is called tautology.

Unsatisfiability: A formula ϕ with $[\![\phi]\!] = \emptyset$ is called unsatisfiable.

Equivalence: Two formulae ϕ_1, ϕ_2 are called equivalent iff $[\![\phi_1]\!] = [\![\phi_2]\!]$.
Denotation: $\phi_1 \equiv \phi_2$

Equivalences: relations between operators

$$
\begin{aligned}
\mathbf{X}(\phi_1 \vee \phi_2) &\equiv \mathbf{X}\,\phi_1 \;\vee\; \mathbf{X}\,\phi_2 \\
\mathbf{X}(\phi_1 \wedge \phi_2) &\equiv \mathbf{X}\,\phi_1 \;\wedge\; \mathbf{X}\,\phi_2 \\
\mathbf{X}\,\neg\phi &\equiv \neg\,\mathbf{X}\,\phi \\
\mathbf{F}(\phi_1 \vee \phi_2) &\equiv \mathbf{F}\,\phi_1 \;\vee\; \mathbf{F}\,\phi_2 \\
\neg\,\mathbf{F}\,\phi &\equiv \mathbf{G}\,\neg\phi \\
\mathbf{G}(\phi_1 \wedge \phi_2) &\equiv \mathbf{G}\,\phi_1 \;\wedge\; \mathbf{G}\,\phi_2 \\
\neg\,\mathbf{G}\,\phi &\equiv \mathbf{F}\,\neg\phi \\
(\phi_1 \wedge \phi_2)\,\mathbf{U}\,\psi &\equiv (\phi_1\,\mathbf{U}\,\psi) \;\wedge\; (\phi_2\,\mathbf{U}\,\psi) \\
\phi\,\mathbf{U}\,(\psi_1 \vee \psi_2) &\equiv (\phi\,\mathbf{U}\,\psi_1) \;\vee\; (\phi_1\,\mathbf{U}\,\psi_2)
\end{aligned}
$$

Equivalences: idempotence and recursion laws

$$\begin{aligned}
\mathbf{F}\,\phi &\equiv \mathbf{F}\,\mathbf{F}\,\phi \\
\mathbf{G}\,\phi &\equiv \mathbf{G}\,\mathbf{G}\,\phi \\
\phi\,\mathbf{U}\,\psi &\equiv \phi\,\mathbf{U}\,(\phi\,\mathbf{U}\,\psi)
\end{aligned}$$

$$\begin{aligned}
\mathbf{F}\,\phi &\equiv \phi \vee \mathbf{X}\,\mathbf{F}\,\phi \\
\mathbf{G}\,\phi &\equiv \phi \wedge \mathbf{X}\,\mathbf{G}\,\phi \\
\phi\,\mathbf{U}\,\psi &\equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{U}\,\psi)) \\
\phi\,\mathbf{W}\,\psi &\equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{W}\,\psi))
\end{aligned}$$

Interpretation of LTL on Kripke structures

Let $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ be a Kripke structure.
We are interested in the valuation sequences generated by $\mathcal{K}$.

Let ρ in S^ω be an infinite path of $\mathcal{K}$.

We assign to ρ an "image" $\nu(\rho)$ in $(2^{AP})^\omega$; for all $i \geq 0$ let

$$\nu(\rho)(i) = \nu(\rho(i))$$

i.e. $\nu(\rho)$ is the corresponding valuation sequence.

Let $[\![\mathcal{K}]\!]$ denote the set of all such sequences:

$$[\![\mathcal{K}]\!] = \{\, \nu(\rho) \mid \rho \text{ is an infinite path of } \mathcal{K} \,\}$$

The LTL model-checking problem

Problem: Given a Kripke structure $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ and an LTL formula ϕ over AP, does $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ hold?

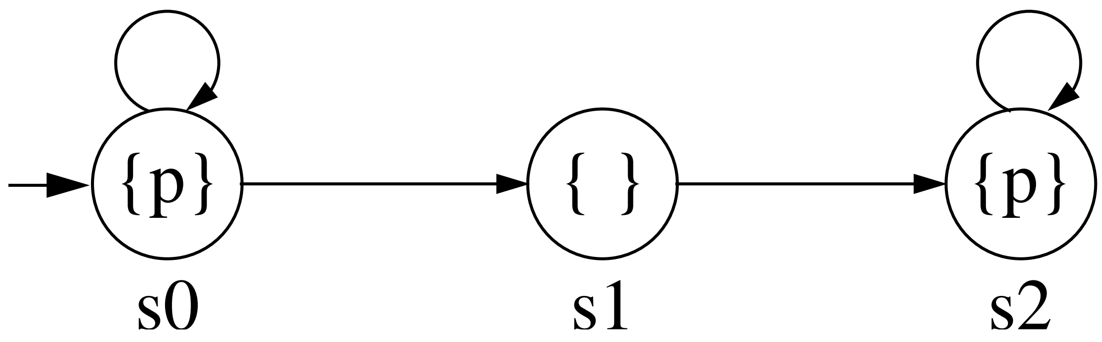
Definition: If $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ then we write $\mathcal{K} \models \phi$.

Interpretation: Every execution of $\mathcal{K}$ must satisfy ϕ for $\mathcal{K} \models \phi$ to hold.

Remark: We may have $\mathcal{K} \not\models \phi$ and $\mathcal{K} \not\models \neg\phi$!

Example

Consider the following Kripke structure $\mathcal{K}$ with $AP = \{p\}$:



There are two classes of infinite paths in $\mathcal{K}$:

(i) Either the system stays in s_0 forever,

(ii) or it eventually reaches s_2 via s_1 and remains there.

We have:

$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$ because all runs eventually end in a state satisfying p.

$\mathcal{K} \not\models \mathbf{G}\,p$ because executions of type (ii) contain a non-p state.

Dealing with deadlocks

The definition of the model-checking problem only considers the infinite sequences!

Thus, executions reaching a deadlock (i.e. a state without any successor) will be ignored, with possibly unforeseen consequences:

Suppose $\mathcal{K}$ contains an error, so that every execution reaches a deadlock.

Then $[\![\mathcal{K}]\!] = \emptyset$, so $\mathcal{K}$ satisfies *every* formula, according to the definition.

Possible alternatives

Remove deadlocks by design:

equip deadlock states with a self loop

Interpretation: system stays in deadlock forever

adapt formula accordingly, if necessary

Treat deadlocks specially:

Check for deadlocks before LTL model checking, deal with them separately.

Tool demonstration: Spin

Demonstration of Spin

Spin is a versatile model-checking tool written by Gerard Holzmann at Bell Labs.

Received the ACM Software System Award in 2002

URL: http://spinroot.com

Book: Holzmann, The Spin Model Checker

Modelling with Spin

System description using Promela (Protocol Meta Language)

Suitable for describing finite systems

Concurrent processes, synchronous/asynchronous communication, variables, data types

LTL model checking (with fairness and reduction techniques)

Example: Dekker's Mutex algorithm

Model of a protocol for mutual exclusion:

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;

active proctype p0() {
...
}

active proctype p1() {
...
}
```

Dekker: contents of process p0

```
active proctype p0() {
again:  flag0 = true;
        do
        :: flag1 ->
                if
                :: turn == 1 ->
                        flag0 = false;
                        (turn != 1) -> flag0 = true;
                :: else -> skip;
                fi
        :: else -> break;
        od;

        crit0 = true; /* critical section */ crit0 = false;

        turn = 1; flag0 = false;
        goto again;
}
```

Process p1: like p0, but all 0s and 1s exchanged

What's going on here? (Promela syntax I)

Variable declarations:

```
bit turn;
bool flag0, flag1;
bool crit0, crit1;
```

`turn` can take values 0 or 1.

`flag1` can become `true` or `false`.

intial values: by default 0 and `false`, resp.

Other data types: `byte`, user-defined types, ...

Promela syntax II

Process declaration:

```
active proctype p0() {
...
}
```

`proctype` defines a process *type*. `active` means that one instance of this process type shall be active initially. It is also possible to activate more than one instance initially, e.g.

```
active [2] proctype my_process() {
...
}
```

Concurrent processes are combined by interleaving: In each step of the system one process makes a step while the others remain stationary.

Promela syntax III

labels / assignments / jumps

```
again:  flag0 = true;
...
        goto again;
```

empty statement:

```
skip
```

Promela syntax IV

Loop:

```
do
:: flag1 -> ...
:: else -> break;
od;
```

`flag1` and `else` are "guards"

Execution branches non-deterministically to some branch whose guard is satisfied.

The `else` branch can only be taken if no other guard is satisfied.

`break` leaves the `do` block.

Promela syntax V

Branching:

```
if
:: turn == 1 -> ...
:: else -> ...;
fi
```

Syntax and semantics as in `do` but without repetition.

```
(turn != 1) -> ...
```

Guarded command: blocks the process until the guard is satisfied.

Model checking using Spin (Example 1)

In the Dekker algorithm the two processes should never be in their critical sections at the same time. This can be expressed by:

$$\mathbf{G}\ \neg(\texttt{crit0} \wedge \texttt{crit1})$$

(where the atomic propositions `crit0` and `crit1` mean that the corresponding Boolean variables are true).

LTL syntax in Spin: `[] !(crit0 && crit1)`

Model checking using Spin (Example 1)

In the Dekker algorithm the two processes should never be in their critical sections at the same time. This can be expressed by:

$$\mathbf{G}\ \neg(\mathtt{crit0} \wedge \mathtt{crit1})$$

(where the atomic propositions `crit0` and `crit1` mean that the corresponding Boolean variables are true).

LTL syntax in Spin: `[] !(crit0 && crit1)`

Checking the property with Spin (spinLTL script):

Property satisfied!

Model checking using Spin (Example 2)

In the Dekker algorithm, a process wanting to enter its critical section should eventually succeed.

$$\mathbf{G}(\texttt{flag0} \rightarrow \mathbf{F}\,\texttt{crit0})$$

(analogously for the other process).

Syntax in Spin: `[] (flag0 -> <> crit0)`

Model checking using Spin (Example 2)

In the Dekker algorithm, a process wanting to enter its critical section should eventually succeed.

$$\mathbf{G}(\texttt{flag0} \rightarrow \mathbf{F}\,\texttt{crit0})$$

(analogously for the other process).

Syntax in Spin: `[] (flag0 -> <> crit0)`

Checking the property with Spin (spinLTL script):

Property not satisfied!

Fairness in Spin

Process 0 cannot enter its critical section if process 1 gets no share of the computation time to set `flag1` back to `false`.

Such an execution is “unfair”.

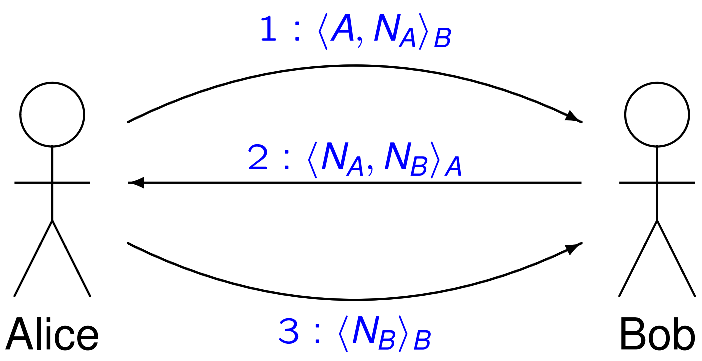
Fairness assumption: Consider only those executions in which both processes infinitely often perform a step.

Spin has got a special switch for this (activated by the script spinFairLTL).

Larger example: Needham-Schröder protocol

Source: Stephan Merz, Model Checking: A Tutorial Overview, 2001

Goal of the protocol: Alice and Bob try to agree on a “secret”.



- Secret represented by nonces $\langle N_A, N_B \rangle$
- Messages can be intercepted
- Assumption: Alice and Bob can communicate using secure public-key cryptography

Is the protocol secure?

Analysis of the protocol using Spin

Representation as a finite system

finite number of principals	Alice, Bob, Intruder
finite models of the principals	one (symbolic) nonce per principal
	intruder can only remember one single message
simple net model	common message channel
	messages as tuples $\langle \mathit{recipient}, \mathit{data} \rangle$
cryptography simulated	compare keys rather than numerical computation

More about Promela

Declaration of enumeration types:

```
mtype = {red,green,blue};
mtype x;
```

The first line declares a bunch of symbolic constants.

The second line declares variable `x`, which can take values `0` (uninitialized), `red`, `green`, `blue`.

Declaration of a record:

```
typedef newtype { bit b; mtype m; }
```

More about Promela

Message channels:

```
chan c = [3] of mtype;
```

`c` is the name of the channel.

The number in brackets gives the capacity; `[0]` means synchronous communication.

`of` is followed by the type of data items that may be sent on the channel.

Write: `c!red;`

Read: `mtype color; c?color;`

Promela model: Declarations

```
#define success         (statusA == ok && statusB == ok)
#define aliceBob        partnerA == bob
#define bobAlice        partnerB == alice
```

Definition of an enumeration type

```
mtype = {msg1, msg2, msg3, alice, bob, intruder,
         nonceA, nonceB, nonceI, keyA, keyB, keyI, ok};
```

```
typedef Crypt { mtype key, d1, d2; }
```
Record type for messages

```
chan network = [0] of {mtype, /* message number */
                       mtype, /* recipient */
                       Crypt};
```
Common message channel

```
mtype partnerA, partnerB;
mtype statusA, statusB;
```
State of Alice and Bob

```
bool knowNA, knowNB;
```
Nonces known to the intruder

Promela model for Alice

```
active proctype Alice() {
  if          choose partner
  :: partnerA = bob; partner_key = keyB;
  :: partnerA = intruder; partner_key = keyI;
  fi;
              send the first msg
  network ! msg1, partnerA, ⟨partner_key, alice, nonceA⟩;

              wait for reply (second msg)
  network ? msg2, alice, data;
              check key and nonce
  (data.key == keyA) && (data.d1 == nonceA);
  partner_nonce = data.d2;
              send the third msg
  network ! msg3, partnerA, ⟨partner_key, partner_nonce⟩;
  statusA = ok;
}
```

The model for Bob is similar

Promela model for intruder (1)

```
active proctype Intruder() {
  do          receive/intercept msg
  :: network ? msg, _, data ->
     if       remember msg if undecryptable
     :: intercepted = data;
     :: skip;
     fi;
     if       evaluate msg if decryptable
     :: (data.key == keyI) ->
        if
        :: (data.d1 == nonceA || data.d2 == nonceA) -> knowNA = true;
        :: else -> skip;
        fi;
        if
        :: (data.d1 == nonceB || data.d2 == nonceB) -> knowNB = true;
        :: else -> skip;
        fi;
     :: else -> skip;
     fi;
  :: ...
```

Promela model for intruder (2)

```
  :: ...
  :: if            send first msg to bob
     :: network ! msg1, bob, intercepted;     repeat intercepted msg
     :: data.key = keyB;                      compose a new msg
        if     the intruder can pose as Alice or himself
        :: data.d1 = alice;
        :: data.d1 = intruder;
        fi;
        if     uses only known nonces
        :: knowsNA -> data.d2 = nonceA;
        :: knowsNB -> data.d2 = nonceB;
        :: data.d2 = nonceI;
        fi;
        network ! msg1, bob, data;
     fi;
  :: ...           similar code for other msgs
  od;
}
```

Analysis of the protocol using Spin

Desirable properties:

$$\mathbf{G}\left(\mathit{statusA} = \mathit{ok} \wedge \mathit{statusB} = \mathit{ok} \ \Rightarrow \right.$$
$$\left. (\mathit{partnerA} = \mathit{bob} \Leftrightarrow \mathit{partnerB} = \mathit{alice})\right)$$

$$\mathbf{G}(\mathit{statusA} = \mathit{ok} \wedge \mathit{partnerA} = \mathit{bob} \ \Rightarrow \neg \mathit{knowsNA})$$

$$\mathbf{G}(\mathit{statusB} = \mathit{ok} \wedge \mathit{partnerB} = \mathit{alice} \ \Rightarrow \neg \mathit{knowsNB})$$

Analysis of the protocol using Spin

Desirable properties:

$$\mathbf{G}\left(\mathit{statusA} = \mathit{ok} \wedge \mathit{statusB} = \mathit{ok} \;\Rightarrow\; (\mathit{partnerA} = \mathit{bob} \Leftrightarrow \mathit{partnerB} = \mathit{alice})\right)$$

$$\mathbf{G}(\mathit{statusA} = \mathit{ok} \wedge \mathit{partnerA} = \mathit{bob} \;\Rightarrow \neg \mathit{knowsNA})$$

$$\mathbf{G}(\mathit{statusB} = \mathit{ok} \wedge \mathit{partnerB} = \mathit{alice} \;\Rightarrow \neg \mathit{knowsNB})$$

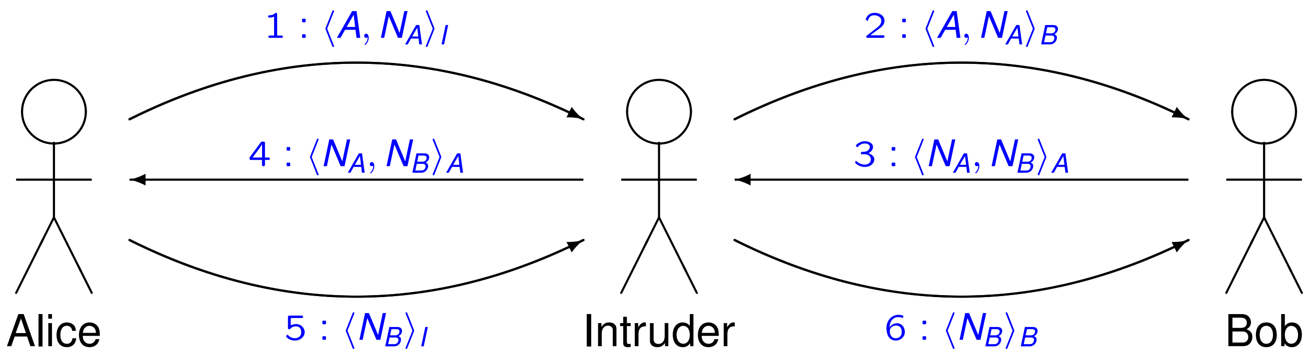
Result of the analysis:

– Property is violated!

The bug in the protocol

Alice opens a connection to Intruder.

Bob mistakenly believes he is talking to Alice.



This bug was found only 18 years after the protocol was invented!

[Needham, Schröder (1978), Lowe (1996)]

Part 4: Büchi automata

Preview

Model-checking problem: $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ – how can we check this algorithmically?

(Historically) first approach: Translate $\mathcal{K}$ into an LTL formula $\psi_{\mathcal{K}}$, check whether $\psi_{\mathcal{K}} \rightarrow \phi$ is a tautology. Problem: very inefficient.

Language-/automata-theoretic approach: $[\![\mathcal{K}]\!]$ and $[\![\phi]\!]$ are languages (of infinite words).

- Find a suitable class of automata for representing these languages.
- Define suitable operations on these automata for solving the problem.

This is the approach we shall follow.

Büchi automata

A Büchi automaton is a tuple

$$\mathcal{B} = (\Sigma, S, s_0, \Delta, F),$$

where:

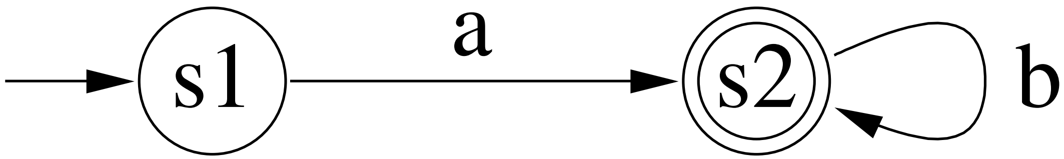
Σ	is a finite alphabet;
S	is a finite set of states;
$s_0 \in S$	is an initial state;
$\Delta \subseteq S \times \Sigma \times S$	are transitions;
$F \subseteq S$	are accepting states.

Remarks:

- Definition and graphical representation like for finite automata.
- However, Büchi automata are supposed to work on *infinite* words, requiring a different acceptance condition.

Example

Graphical representation of a Büchi automaton:



The components of this automaton are $(\Sigma, S, s_1, \Delta, F)$, where:

- $\Sigma = \{a, b\}$ (symbols on the edges)
- $S = \{s_1, s_2\}$ (circles)
- s_1 (indicated by arrow)
- $\Delta = \{(s_1, a, s_2), (s_2, b, s_2)\}$ (edges)
- $F = \{s_2\}$ (with double circle)

Language of a Büchi automaton

Let $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ be a Büchi automaton.

A run of $\mathcal{B}$ over an infinite word $\sigma \in \Sigma^\omega$ is an infinite sequences of states $\rho \in S^\omega$ where $\rho(0) = s_0$ and $(\rho(i), \sigma(i), \rho(i+1)) \in \Delta$ for $i \geq 0$.

We call ρ accepting iff $\rho(i) \in F$ for infinitely many values of i.

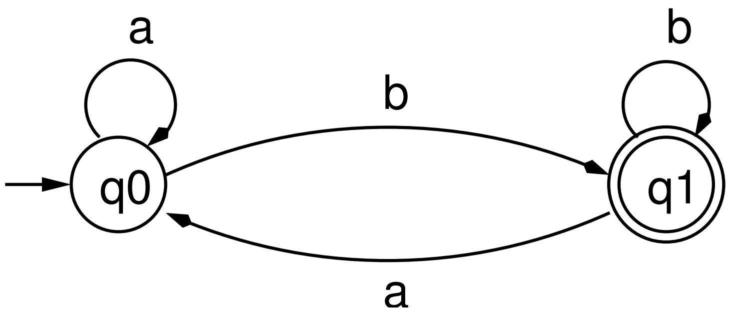
I.e., ρ infinitely often visits accepting states.
(By the pigeon-hole principle: at least one accepting state is visited infinitely often.)

$\sigma \in \Sigma^\omega$ is accepted by $\mathcal{B}$ iff there exists an accepting run over σ in $\mathcal{B}$.

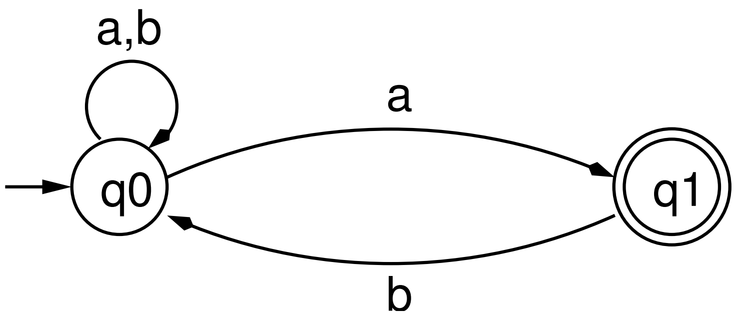
The language of $\mathcal{B}$, denoted $\mathcal{L}(\mathcal{B})$, is the set of all words accepted by $\mathcal{B}$.

Büchi automata: examples

"infinitely often b"



"infinitely often ab"



Büchi automata and LTL

Let *AP* be a set of atomic propositions.

A Büchi automaton with alphabet 2^{AP} accepts a sequence of valuations.

Claim: For every LTL formula ϕ there exists a Büchi automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$.

(We shall prove this claim later.)

Examples:

$\mathbf{F}\, p$

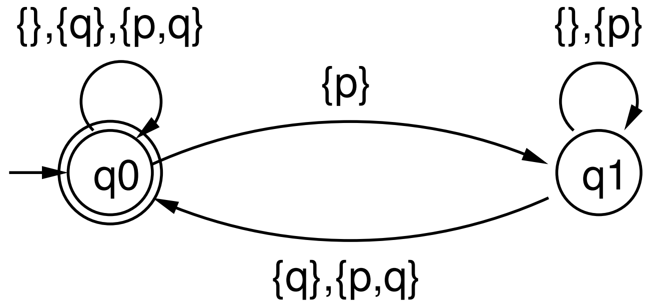
$\mathbf{G}\, p$

$\mathbf{G}\,\mathbf{F}\, p$

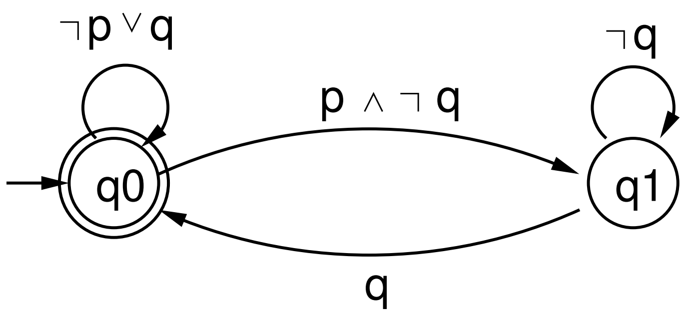
$\mathbf{F}\,\mathbf{G}\, p$

$\mathbf{G}(p \rightarrow \mathbf{F}\, q)$

Example automaton for $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$, with $AP = \{p, q\}$.



Alternatively we can label edges with formulae of propositional logic; in this case, a formula F stands for all elements of $[\![F]\!]$. In this case:



Operations on Büchi automata

The languages accepted by Büchi automata are also callled ω-regular languages.

Like the usual regular languages, ω-regular languages are also closed under Boolean operations.

I.e., if $\mathcal{L}_1$ and $\mathcal{L}_2$ are ω-regular, then so are

$$\mathcal{L}_1 \cup \mathcal{L}_2, \qquad \mathcal{L}_1 \cap \mathcal{L}_2, \qquad \mathcal{L}_1^c.$$

We shall now define operations that take Büchi automata accepting some languages $\mathcal{L}_1$ and $\mathcal{L}_2$ and produce automata for their union or intersection.

In the following slides we assume $\mathcal{B}_1 = (\Sigma, S, s_0, \Delta_1, F)$ and $\mathcal{B}_2 = (\Sigma, T, t_0, \Delta_2, G)$ (with $S \cap T = \emptyset$).

Union

"Juxtapose" $\mathcal{B}_1$ and $\mathcal{B}_2$ and add a new initial state.

In other words, the automaton $\mathcal{B} = (\Sigma,\ S \cup T \cup \{u\},\ u,\ \Delta_1 \cup \Delta_2 \cup \Delta_u,\ F \cup G)$ accepts $\mathcal{L}(\mathcal{B}_1) \cup \mathcal{L}(\mathcal{B}_2)$, where

- u is a "fresh" state ($u \notin S \cup T$);
- $\Delta_u = \{\ (u, \sigma, s) \mid (s_0, \sigma, s) \in \Delta_1\ \} \cup \{\ (u, \sigma, t) \mid (t_0, \sigma, t) \in \Delta_2\ \}$.

Intersection I (a special case)

We first consider the case where all states in $\mathcal{B}_2$ are accepting, i.e. $G = T$.

Idea: Construct a cross-product automaton (like for FA), check whether F is visited infinitely often.

Let $\mathcal{B} = (\Sigma,\ S \times T,\ \langle s_0, t_0 \rangle,\ \Delta,\ F \times T)$, where

$$\Delta = \{\, (\langle s, t \rangle, a, \langle s', t' \rangle) \mid a \in \Sigma,\ (s, a, s') \in \Delta_1,\ (t, a, t') \in \Delta_2 \,\}.$$

Then: $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$.

Intersection II (the general case)

Principle: We again construct a cross-product automaton.

Problem: The acceptance condition needs to check whether *both* accepting sets are visited infinitely often.

Idea: create two copies of the cross product.

- In the first copy we wait for a state from F.
- In the second copy we wait for a state from G.
- In both copies, once we've found one of the states we're looking for, we switch to the other copy.

We will choose the acceptance condition in such a way that an accepting run switches back and forth between the copies infinitely often.

Let $\mathcal{B} = (\Sigma, U, u, \Delta, H)$, where

$$U = S \times T \times \{1, 2\}, \quad u = \langle s_0, t_0, 1\rangle, \quad H = F \times T \times \{1\}$$

$$(\langle s, t, 1\rangle, a, \langle s', t', 1\rangle) \in \Delta \text{ iff } (s, a, s') \in \Delta_1,\ (t, a, t') \in \Delta_2,\ s \notin F$$
$$(\langle s, t, 1\rangle, a, \langle s', t', 2\rangle) \in \Delta \text{ iff } (s, a, s') \in \Delta_1,\ (t, a, t') \in \Delta_2,\ s \in F$$
$$(\langle s, t, 2\rangle, a, \langle s', t', 2\rangle) \in \Delta \text{ iff } (s, a, s') \in \Delta_1,\ (t, a, t') \in \Delta_2,\ t \notin G$$
$$(\langle s, t, 2\rangle, a, \langle s', t', 1\rangle) \in \Delta \text{ iff } (s, a, s') \in \Delta_1,\ (t, a, t') \in \Delta_2,\ t \in G$$

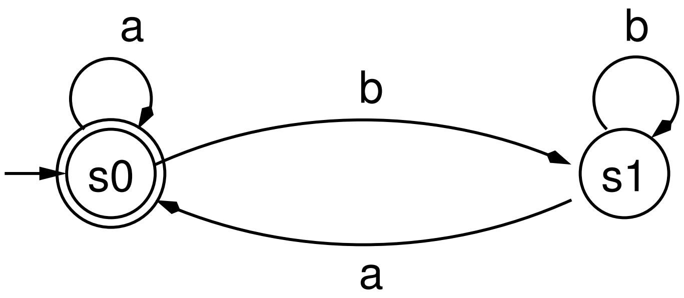
Remarks:

The automaton starts in the first copy.

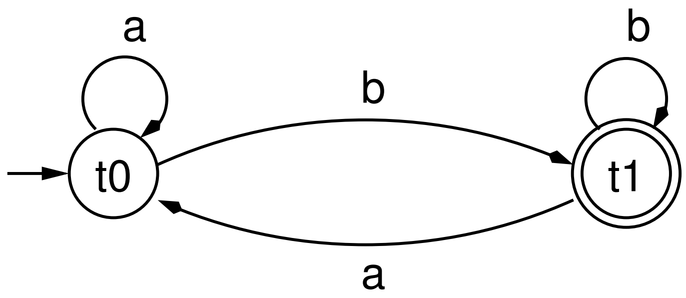
We could have chosen other acceptance conditions such as $S \times G \times \{2\}$.

The construction can be generalized to intersecting n automata.

Intersection: example

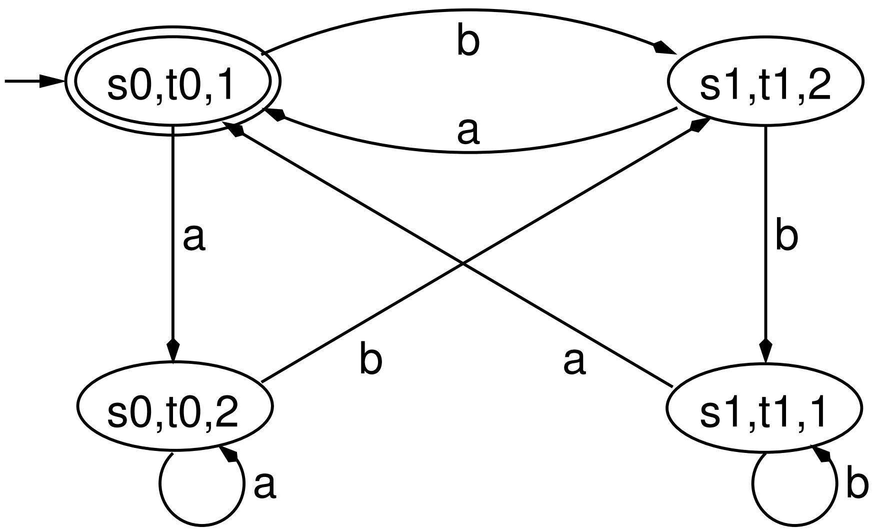


B1



B2

B1 x B2



Complement

Problem: Given $\mathcal{B}_1$, construct $\mathcal{B}$ with $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1)^c$.

Such a construction is possible (but rather complicated). We will not require it for the purpose of this course.

Additional literature:

Wolfgang Thomas, *Automata on Infinite Objects*, Chapter 4 in *Handbook of Theoretical Computer Science*,

Igor Walukiewicz, lecture notes on *Automata and Logic*, chapter 3, `www.labri.fr/Perso/~igw/Papers/igw-eefss01.ps`

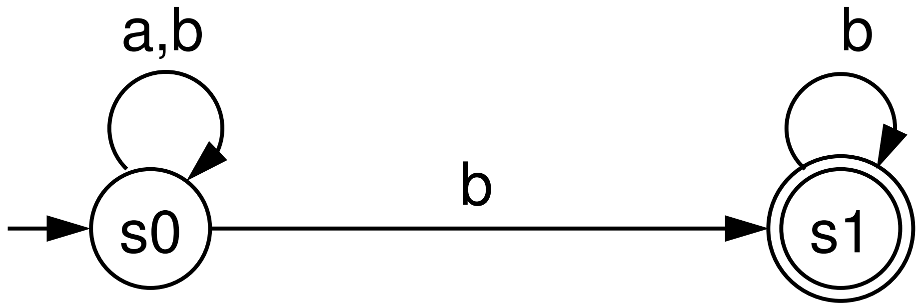
Deterministic Büchi automata

For finite automata (known from *regular language theory*), it is known that every language expressible by a finite automaton can also be expressed by a deterministic automaton, i.e. one where the transition relation Δ is a function $S \times \Sigma \to S$.

Such a procedure does not exist for Büchi automata.

In fact, there is no *deterministic* Büchi automaton accepting the same language as the automaton below:

"Only finitely many *a* s."



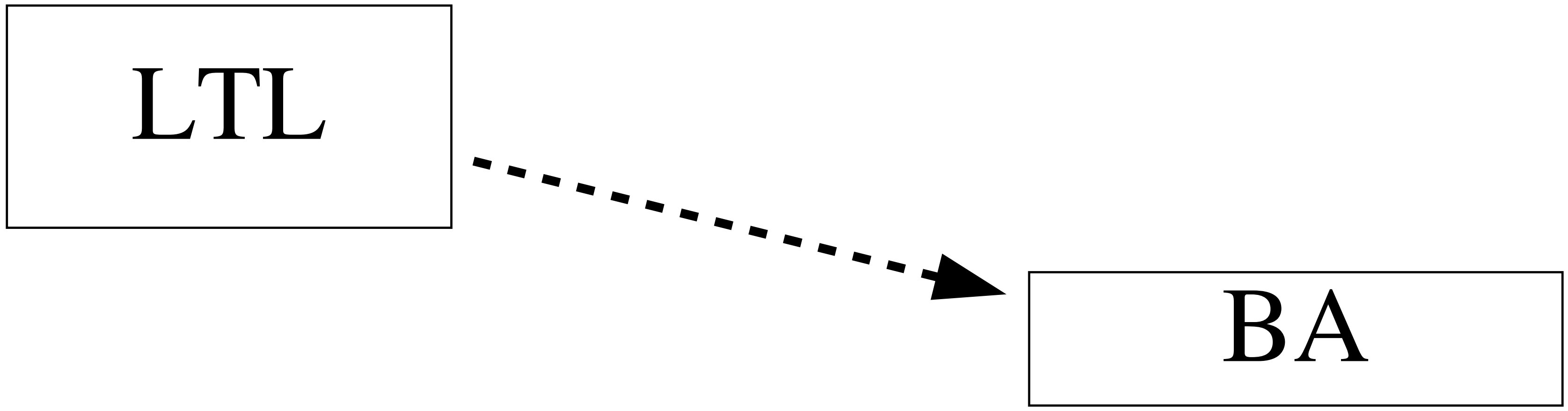
Proof: Let $\mathcal{L}$ be the language of infinite words over $\{a, b\}$ containing only finitely many as. Assume that a deterministic Büchi automaton $\mathcal{B}$ with $\mathcal{L}(\mathcal{B}) = \mathcal{L}$ exists, and let n be the number of states in $\mathcal{B}$.

We have $b^\omega \in \mathcal{L}$, so let α_1 be the (unique) accepting run for b^ω. Suppose that an accepting state is first reached after n_1 letters, i.e. $s_1 := \alpha_1(n_1)$ is the first accepting state in α_1.

We now regard the word $b^{n_1}ab^\omega$, which is still in $\mathcal{L}$, therefore accepted by some run α_2. Since $\mathcal{B}$ is deterministic, α_1 and α_2 must agree on the first n_1 states. Now, watch for the second occurrence of an accepting state in α_2, i.e. let $s_2 := \alpha_2(n_1 + 1 + n_2)$ be an accepting state for n_2 minimal. Then, $s_1 \neq s_2$ because otherwise there would be a loop around an accepting state containing a transition with an a.

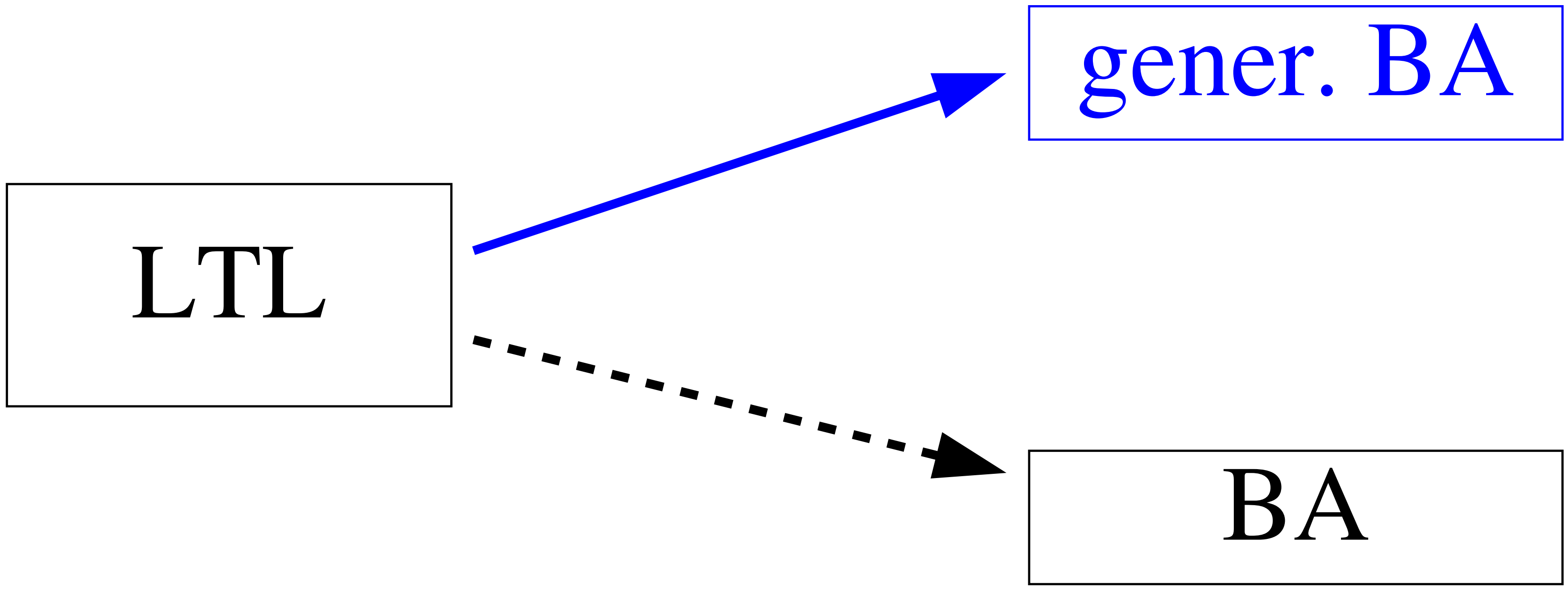
We now repeat the argument for $b^{n_1}ab^{n_2}ab^\omega$, derive the existence of a third distinct state, etc. After doing this $n+1$ times, we conclude that $\mathcal{B}$ must have more than n distinct states, a contradiction.

Preview



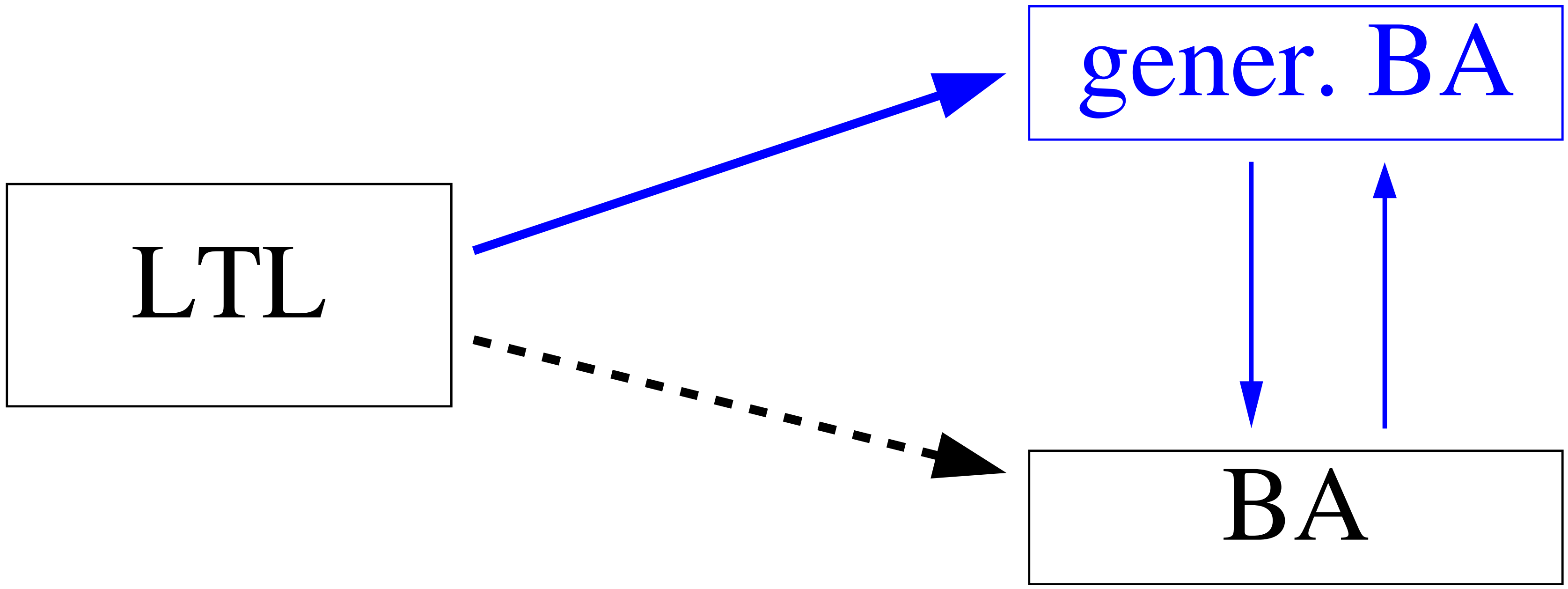
We desire to translate LTL formulae into Büchi automata.

Preview



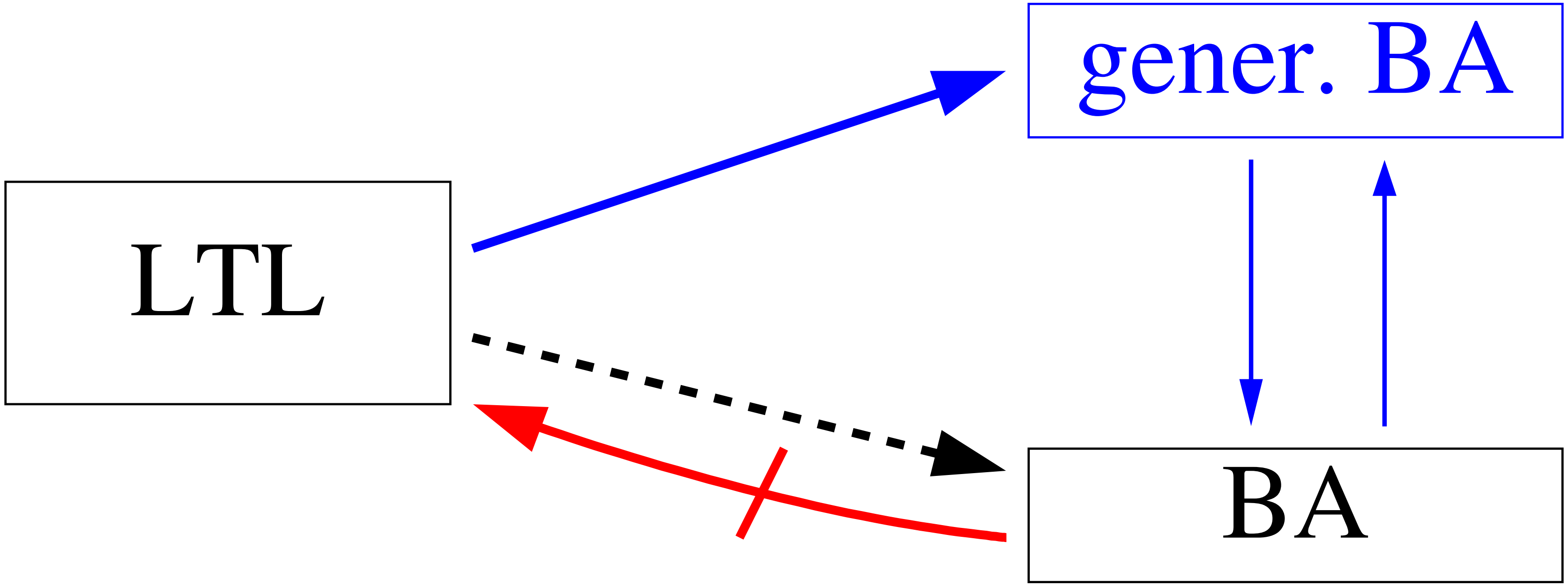
Detour: We translate them into so-called *generalized* Büchi automata (GBA).

Preview



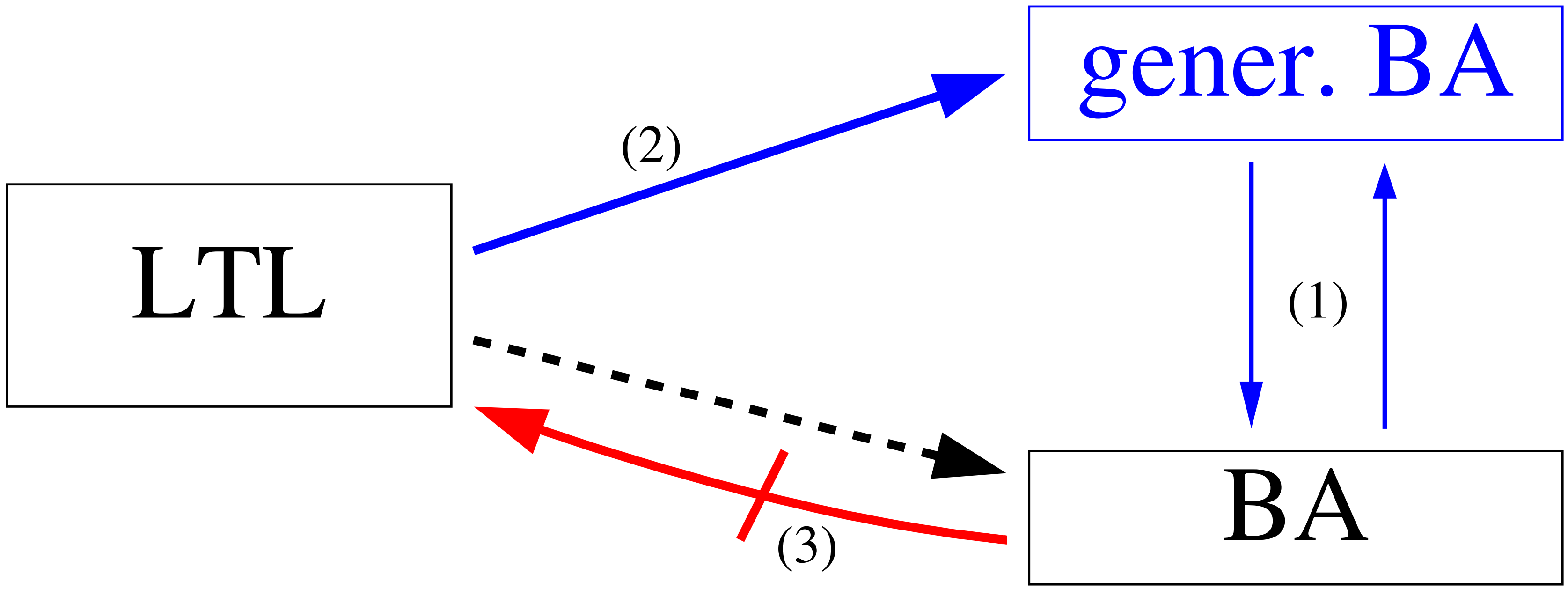
GBA accept the same class of languages as BA.

Preview



Translation from BA to LTL not possible in general.

Preview



We shall proceed in the order indicated above.

Generalized Büchi automata

A generalized Büchi automaton (GBA) is a tuple $\mathcal{G} = (\Sigma, S, s_0, \Delta, \mathcal{F})$.

There is only one difference w.r.t. normal BA:

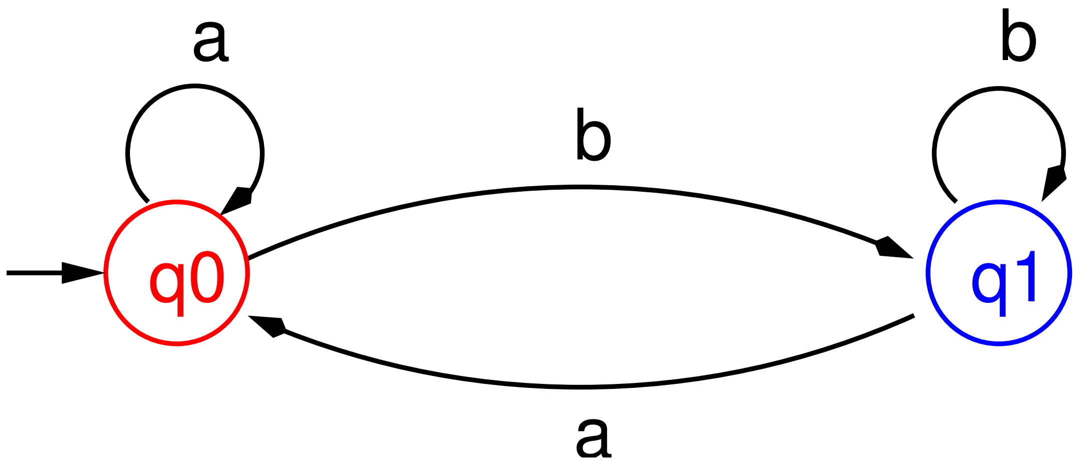
The acceptance condition $\mathcal{F} \subseteq 2^S$ is a *set of sets of states*.

E.g., let $\mathcal{F} = \{F_1, \ldots, F_n\}$. A run ρ of $\mathcal{G}$ is called accepting iff for every F_i ($i = 1, \ldots, n$), ρ visits infinitely many states of F_i.

Put differently: many acceptance conditions at once.

GBA: Example

For the GBA shown below, let $\mathcal{F} = \{ \{q_0\}, \{q_1\} \}$.



Language of the automaton: “infinitely often *a and* infinitely often *b*”

Note: In general, the acceptance conditions need not be pairwise disjoint.

Advantage: GBA may be more succinct than BA.

Translations BA $\leftrightarrow$ GBA

GBA accept the same class of languages as BA.

I.e., for every BA there is a GBA accepting the same language, and vice versa.

Part 1 of the claim (BA $\rightarrow$ GBA):

Let $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ be a BA.

Then $\mathcal{G} = (\Sigma, S, s_0, \Delta, \{F\})$ is a GBA with $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{B})$.

Part 2 of the claim (GBA $\rightarrow$ BA):

Let $\mathcal{G} = (\Sigma, S, s_0, \Delta, \{F_1, \ldots, F_n\})$ be a GBA.

We construct $\mathcal{B} = (\Sigma, S', s_0', \Delta', F)$ as follows:

$S' = S \times \{1, \ldots, n\}$

$s_0' = (s_0, 1)$

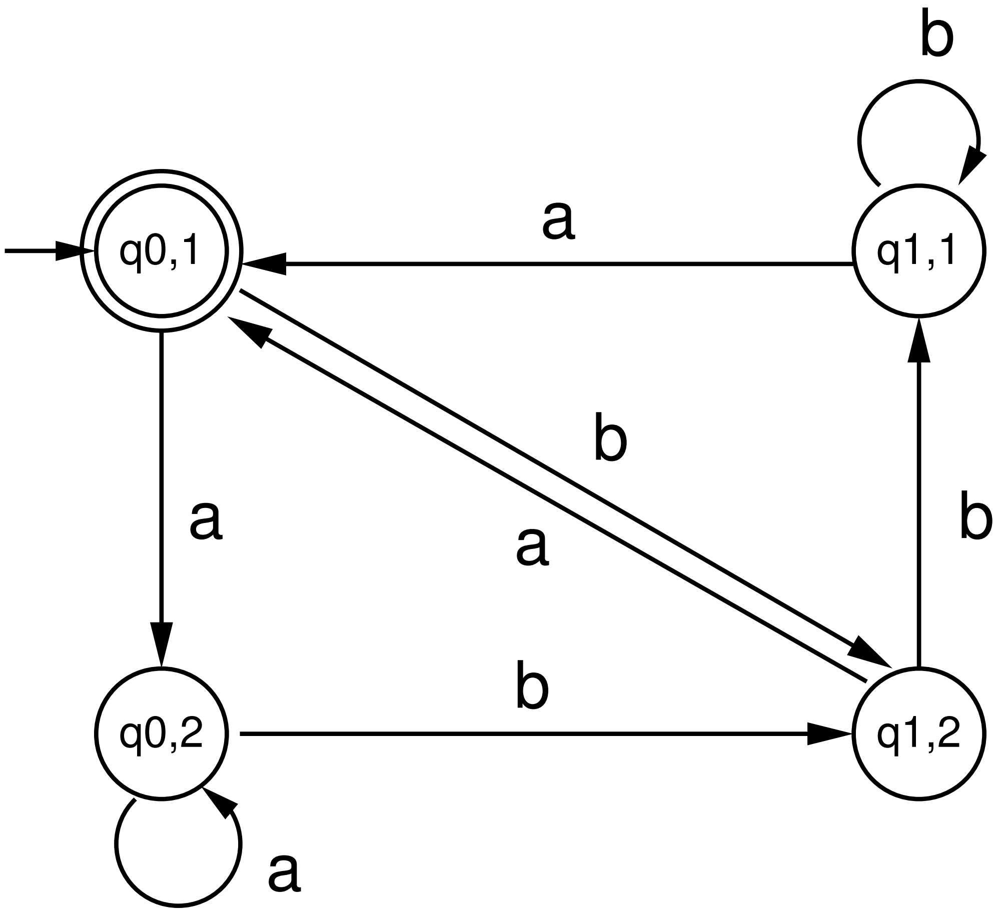
$F = F_1 \times \{1\}$

$((s, i), a, (s', k)) \in \Delta'$ iff $1 \leq i \leq n,\ (s, a, s') \in \Delta$
and $k = \begin{cases} i & \text{if } s \notin F_i \\ (i \bmod n) + 1 & \text{if } s \in F_i \end{cases}$

Then we have $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{G})$. (Idea: *n*-fold intersection)

GBA → BA: example

The BA corresponding to the previous GBA ("infinitely often *a and* infinitely often *b*") is as follows:



Remark: Multiple initial states

Our definitions of BA and GBA require exactly one initial state.

For the translation LTL $\rightarrow$ BA it will be convenient to use GBA with multiple initial states.

> Intended meaning: A word is regarded as accepted if it is accepted starting from *any* initial state.

Obviously, every (G)BA with multiple initial states can easily be converted into a (G)BA with just one initial state.

Part 5: LTL and Büchi automata

Overview

In this part, we shall solve the following problem:

Given an LTL formula ϕ over *AP*, we shall construct a GBA $\mathcal{G}$ (with multiple initial states) such that $\mathcal{L}(\mathcal{G}) = [\![\phi]\!]$.

($\mathcal{G}$ can then be converted to a normal BA.)

Remarks:

Analogous operation for regular languages: reg. expression $\rightarrow$ NFA

The crucial difference: it is not possible to provide an LTL $\rightarrow$ BA translation in modular fashion.

The automaton may have to check multiple subformulae *at the same time* (e.g.: $(\mathbf{G}\,\mathbf{F}\,p) \rightarrow (\mathbf{G}(q \rightarrow \mathbf{F}\,r))$ or $(p\,\mathbf{U}\,q)\,\mathbf{U}\,r$).

More remarks:

The construction shown in the following is comparatively simplistic.

It will produce rather suboptimal automata (size *always* exponential in $|\phi|$).

Obviously, this is quite inefficient, and not meant to be done by pen and paper, only as a "proof of concept".

There are far better translation procedures but the underlying theory is rather beyond the scope of the course.

Interesting, on-going research area!

Structure of the construction

1. We first convert ϕ into a certain normal form.

2. States will be "responsible" for some set of subformulae.

3. The transition relation will ensure that "simple" subformulae such as p or $\mathbf{X}\, p$ are satisfied.

4. The acceptance condition will ensure that $\mathbf{U}$-subformulae are satisfied.

Negation normal form

Let *AP* be a set of atomic propositions. The set of NNF formulae over *AP* is inductively defined as follows:

If $p \in AP$ then p and $\neg p$ are NNF formulae.
(Remark: Negations occur *exclusively* in front of atomic propositions.)

If ϕ_1 and ϕ_2 are NNF formulae then so are

$$\phi_1 \vee \phi_2, \quad \phi_1 \wedge \phi_2, \quad \mathbf{X}\,\phi_1, \quad \phi_1\,\mathbf{U}\,\phi_2, \quad \phi_1\,\mathbf{R}\,\phi_2, \quad \textbf{true}, \quad \textbf{false}.$$

Claim: For every LTL formula ϕ there is an equivalent NNF formula:

$$\neg(\phi_1\,\mathbf{R}\,\phi_2) \equiv \neg\phi_1\,\mathbf{U}\,\neg\phi_2 \qquad \neg(\phi_1\,\mathbf{U}\,\phi_2) \equiv \neg\phi_1\,\mathbf{R}\,\neg\phi_2$$

$$\neg(\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \vee \neg\phi_2 \qquad \neg(\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$$

$$\neg\,\mathbf{X}\,\phi \equiv \mathbf{X}\,\neg\phi \qquad \neg\neg\phi \equiv \phi$$

NNF: Example

Translation into an NNF formula:

$$
\begin{aligned}
\mathbf{G}(p \rightarrow \mathbf{F}\, q) &\equiv \neg\, \mathbf{F}\, \neg(p \rightarrow \mathbf{F}\, q) \\
&\equiv \neg(\mathbf{true}\ \mathbf{U}\ \neg(p \rightarrow \mathbf{F}\, q)) \\
&\equiv \neg\mathbf{true}\ \mathbf{R}\ (p \rightarrow \mathbf{F}\, q) \\
&\equiv \mathbf{false}\ \mathbf{R}\ (\neg p \vee \mathbf{F}\, q) \\
&\equiv \mathbf{false}\ \mathbf{R}\ (\neg p \vee (\mathbf{true}\ \mathbf{U}\ q))
\end{aligned}
$$

Remark: Because of this, we shall henceforth assume that the LTL formula in the translation procedure is given in NNF.

Remark: **true** and **false** could be treated as syntactic sugar.

Subformulae

Let ϕ be an NNF formula. The set $Sub(\phi)$ is the smallest set satisfying:

$\phi \in Sub(\phi)$;

if $\mathbf{X}\, \phi_1 \in Sub(\phi)$ then $\phi_1 \in Sub(\phi)$;

if $\phi_1 \,\mathbf{U}\, \phi_2 \in Sub(\phi)$ then $\phi_1, \phi_2 \in Sub(\phi)$;

if $\phi_1 \,\mathbf{R}\, \phi_2 \in Sub(\phi)$ then $\phi_1, \phi_2 \in Sub(\phi)$;

if $\phi_1 \vee \phi_2 \in Sub(\phi)$ then $\phi_1, \phi_2 \in Sub(\phi)$;

if $\phi_1 \wedge \phi_2 \in Sub(\phi)$ then $\phi_1, \phi_2 \in Sub(\phi)$;

if $\neg\phi_1 \in Sub(\phi)$ then $\phi_1 \in Sub(\phi)$;

if $\phi_1 \in Sub(\phi)$ and ϕ_1 is not of the form $\neg\phi_1'$ then $\neg\phi_1 \in Sub(\phi)$;

Note: We have $|Sub(\phi)| = \mathcal{O}(|\phi|)$ (two subformulae per syntactic element).

Consistent sets

Recall item 2 of the construction:

> Every state will be labelled with a subset of $Sub(\phi)$.

Idea: A state labelled by set *M* will accept a sequence iff it satisfies every single subformula contained in *M* and violates every single subformula contained in $Sub(\phi) \setminus$ *M*.

For this reason, we will a priori exclude some sets *M* which would obviously lead to empty languages.

The other states will be called consistent.

Definition: We call a set $M \subset Sub(\phi)$ *consistent* if it satisfies the following conditions:

- if $\phi_1 \wedge \phi_2 \in Sub(\phi)$ then $\phi_1 \wedge \phi_2 \in M$ iff $\phi_1 \in M$ and $\phi_2 \in M$;
- if $\phi_1 \vee \phi_2 \in Sub(\phi)$ then $\phi_1 \vee \phi_2 \in M$ iff $\phi_1 \in M$ or $\phi_2 \in M$;
- if $\neg\phi_1 \in Sub(\phi)$ then $\neg\phi_1 \in M$ iff $\phi_1 \notin M$;
- if **true** $\in Sub(\phi)$ then **true** $\in M$;
- **false** $\notin M$;

By $CS(\phi)$ we denote the set of all consistent subsets of $Sub(\phi)$.

Translation (1)

Let ϕ be an NNF formula and $\mathcal{G} = (\Sigma, S, S_0, \Delta, \mathcal{F})$ be a GBA such that:

$\Sigma = 2^{AP}$
(i.e. the valuations over *AP*)

$S = CS(\phi)$
(i.e. every state is a consistent set)

$S_0 = \{ M \in S \mid \phi \in M \}$
(i.e. the initial states admit sequences satisfying ϕ)

Δ and $\mathcal{F}$: see next slide

Translation (2)

Transitions: $(M, \sigma, M') \in \Delta$ iff $\sigma = M \cap AP$ and:

- if $\mathbf{X}\,\phi_1 \in Sub(\phi)$ then $\mathbf{X}\,\phi_1 \in M$ iff $\phi_1 \in M'$;
- if $\phi_1 \,\mathbf{U}\, \phi_2 \in Sub(\phi)$ then $\phi_1 \,\mathbf{U}\, \phi_2 \in M$
  iff $\phi_2 \in M$ or ($\phi_1 \in M$ and $\phi_1 \,\mathbf{U}\, \phi_2 \in M'$);
- if $\phi_1 \,\mathbf{R}\, \phi_2 \in Sub(\phi)$ then $\phi_1 \,\mathbf{R}\, \phi_2 \in M$
  iff ($\phi_1 \in M$ and $\phi_2 \in M$) or ($\phi_2 \in M$ and $\phi_1 \,\mathbf{R}\, \phi_2 \in M'$).

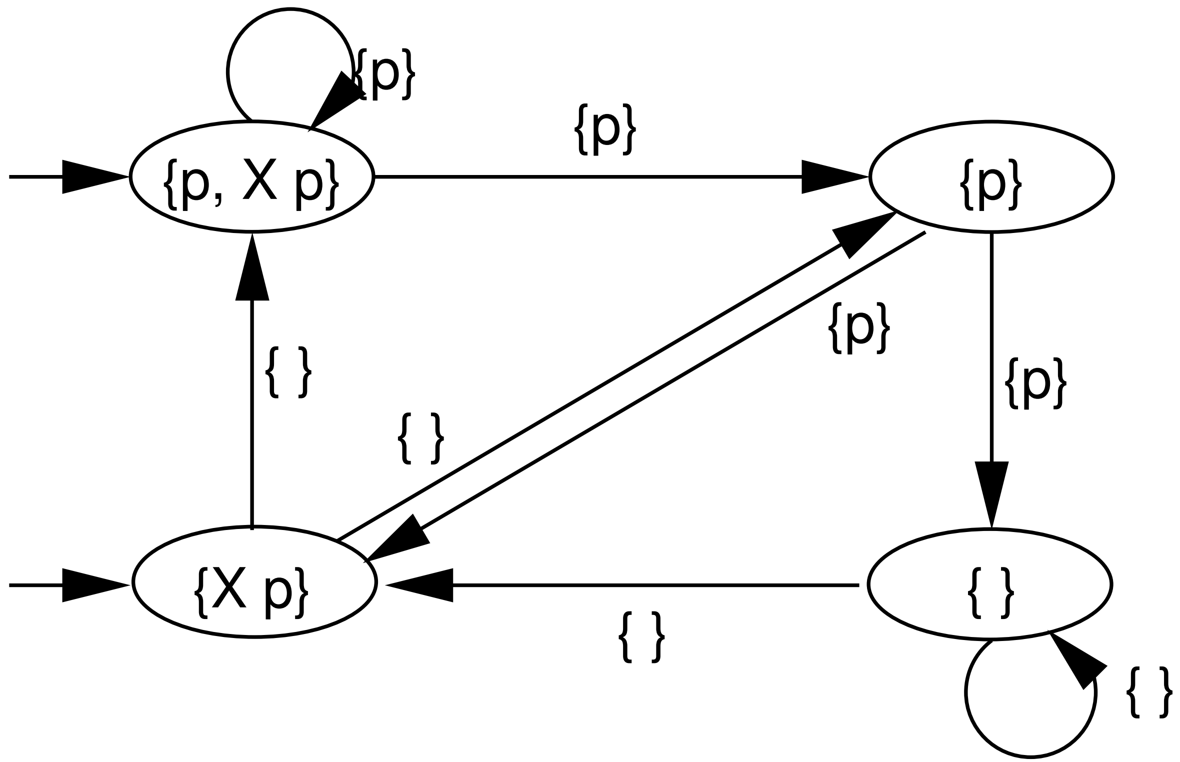
Acceptance condition:

$\mathcal{F}$ contains a set F_ψ for every subformula ψ of the form $\phi_1 \,\mathbf{U}\, \phi_2$, where

$$F_\psi = \{\, M \in CS(\phi) \mid \phi_2 \in M \text{ or } \neg(\phi_1 \,\mathbf{U}\, \phi_2) \in M \,\}.$$

Translation: Example 1

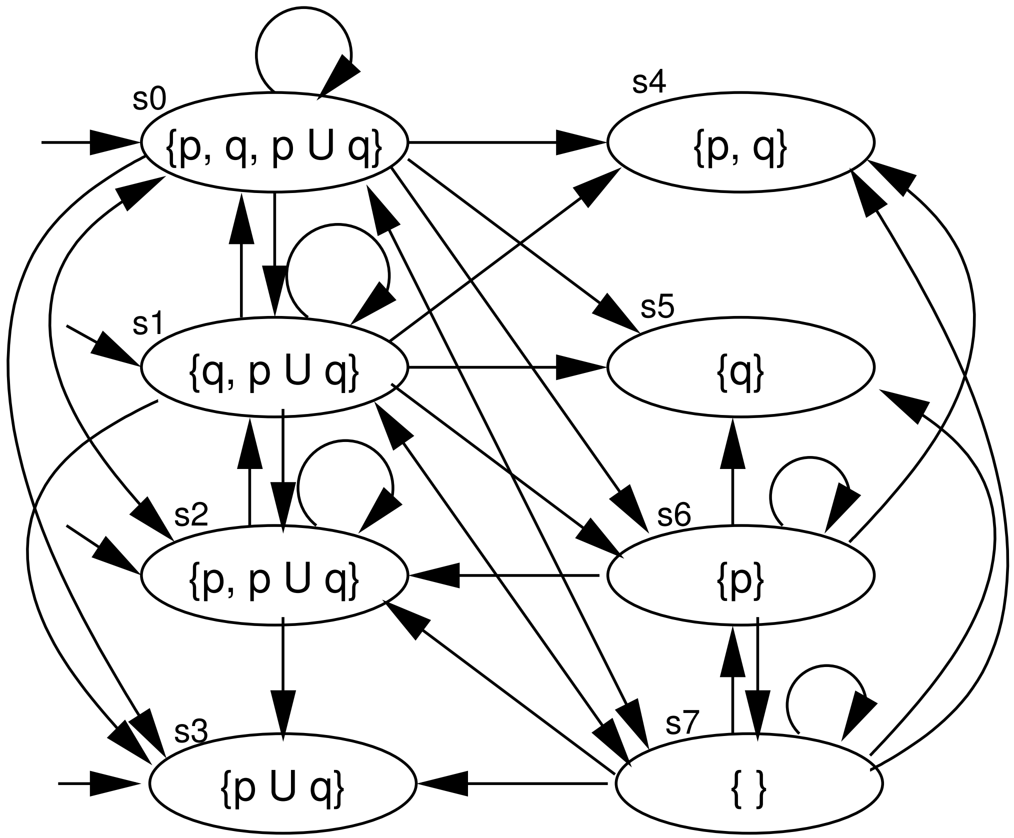
$\phi = \mathbf{X}\, p$



This GBA has got two initial states and the acceptance condition $\mathcal{F} = \emptyset$, i.e. every infinite run is accepting.

Translation: Example 2

$\phi \equiv p \, \mathbf{U} \, q$



GBA with $\mathcal{F} = \{\{s_0, s_1, s_4, s_5, s_6, s_7\}\}$, transition labels also omitted.

Proof of correctness

We want to prove the following:

$$\sigma \in \mathcal{L}(\mathcal{G}) \quad \text{iff} \quad \sigma \in [\![\phi]\!]$$

To this aim, we shall prove the following stronger lemma:

> Let α be a sequence of consistent sets (i.e., states of $\mathcal{G}$) and let σ be a sequence of valuations over *AP*.
>
> α is an accepting run of $\mathcal{G}$ over σ from some (possibly *non-initial*) state
> iff $\alpha(i) = \{\psi \in Sub(\phi) \mid \sigma^i \in [\![\psi]\!]\}$ for all $i \geq 0$.

The desired proof then follows from the choice of initial states:

⇒: For $\sigma \in \mathcal{L}(\mathcal{G})$, let α be an accepting run on σ hence $\phi \in \alpha(0)$. By the property (for $\psi = \phi, i = 0$), we have $\sigma^0 \in [\![\phi]\!]$.

⇐: For any σ, let α be defined by the property. Hence (i) α is an accepting run (from α_0), and (ii) if $\sigma \in [\![\phi]\!]$ then $\phi \in \alpha(0)$; by the latter, $\alpha(0)$ is initial.

Correctness of the lemma

Remark: By construction, we have $\sigma(i) = \alpha(i) \cap AP$ for all $i \geq 0$.

Proof of $\Leftarrow$ by the semantics of the operators.

Proof of $\Rightarrow$ via structural induction over ψ:

for **true** and **false**: by consistency of $\alpha(i)$.

for $p \in AP$: by $p \in \alpha(i)$ iff $p \in \sigma(i)$ iff $\sigma^i \in [\![p]\!]$.

for $\neg p$: by $\neg p \in \alpha(i)$ iff $p \notin \alpha(i)$ iff (by IH) $\sigma^i \notin [\![p]\!]$ iff $\sigma^i \in [\![\neg p]\!]$.

for $\psi_1 \vee \psi_2$ and $\psi_1 \wedge \psi_2$: follows from consistency of $\alpha(i)$ and from the induction hypothesis for ψ_1 and ψ_2.

Correctness of the lemma

for $\psi = \mathbf{X}\,\psi_1$: follows from the construction of Δ and induction hypothesis for ψ_1.

for $\psi = \psi_1 \,\mathbf{R}\, \psi_2$:

Follows from the construction of Δ, the recursion equation for $\mathbf{R}$ and the induction hypothesis.

for $\psi = \psi_1 \,\mathbf{U}\, \psi_2$:

Analogous to $\mathbf{R}$, but additionally we must ensure that $\psi_2 \in \alpha(k)$ for some $k \geq i$. Assume that this is not the case, then we have $\psi_1 \,\mathbf{U}\, \psi_2 \in \alpha(k)$ for all $k \geq i$. However, none of these states is in F_ψ, therefore α cannot be accepting, which is a contradiction.

Complexity of the translation

For a formula ϕ, the translation procedure produces an automaton of size $\mathcal{O}(2^{|\phi|})$.

Question: Is there a better translation procedure?

Answer 1: No (not in general). There exist formulae for which any Büchi automaton has necessarily exponential size.

Example: The following LTL formula over $\{p_0, \ldots, p_{n-1}\}$ simulates an n-bit counter.

$$\mathbf{G}(p_0 \not\leftrightarrow \mathbf{X}\, p_0) \wedge \bigwedge_{i=1}^{n-1} \mathbf{G}\left(\left(p_i \not\leftrightarrow \mathbf{X}\, p_i\right) \leftrightarrow \left(p_{i-1} \wedge \neg\, \mathbf{X}\, p_{i-1}\right)\right)$$

The formula has size $\mathcal{O}(n)$. Obviously, any automaton for this formula must have at least 2^n states.

Answer 2: Yes (sometimes). There are translation procedures that produce smaller automata *for most cases*.

Some tools:

- Spin (command `spin -f 'p U q'`)
- LTL2BA (also web applet)
- Spot (currently most efficient)

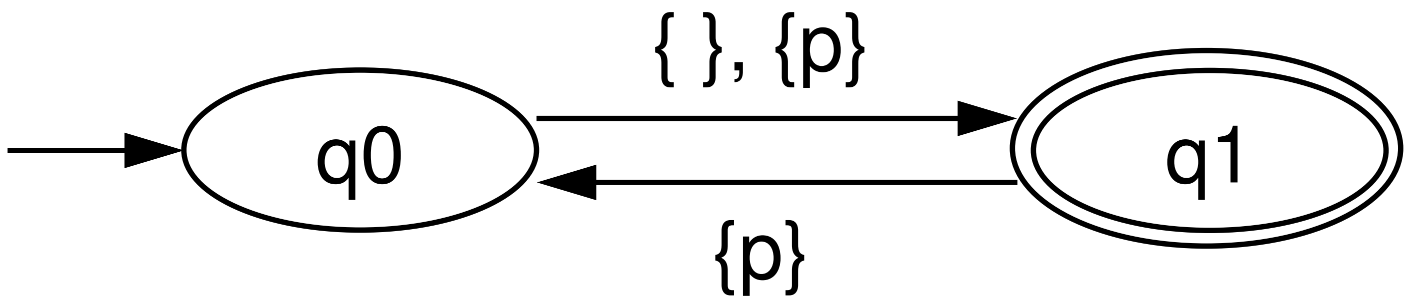
Literature:

- Gerth, Peled, Vardi, Wolper: *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, 1996
- Oddoux, Gastin: *Fast LTL to Büchi Automata Translation*, 2001

Translation BA $\rightarrow$ LTL

The reverse translation (BA $\rightarrow$ LTL) is not possible in general.

I.e., there are Büchi automata $\mathcal{B}$ such that there is no formula ϕ with $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$ (Wolper, 1983).



The property "*p* holds in every second step" is not expressible in LTL (proof: next slide).

Proof (BA $\not\to$ LTL)

We first show a more general lemma:

Let ϕ be an arbitrary LTL formula over AP and n the number of $\mathbf{X}$ operators in ϕ. We regard the sequences

$$\sigma_i = \{p\}^i \, \emptyset \, \{p\}^\omega$$

for $i \geq 0$. For all pairs $i, k > n$ we have: $\sigma_i \models \phi$ iff $\sigma_k \models \phi$.

Proof by structural induction over ϕ:

If $\phi = p$, for $p \in AP$, then $n = 0$ and $i, k \geq 1$.
Thus, $\sigma_i \models p$ and $\sigma_k \models p$.

For the other cases, the induction hypothesis assumes that the property holds for ϕ_1 and ϕ_2, i.e. if ϕ_1, ϕ_2 contain n_1 and n_2 occurrences of $\mathbf{X}$, resp., then for all $i_1, k_1 > n_1$ we have $\sigma_{i_1} \models \phi_1$ iff $\sigma_{k_1} \models \phi_1$, and analogously for ϕ_2.

If $\phi = \neg\phi_1$, then the proof follows directly from the induction hypothesis.

For $\phi = \phi_1 \vee \phi_2$: same

If $\phi = \mathbf{X}\,\phi_1$, then $n_1 = n - 1$. Since $i - 1, k - 1 > n - 1 = n_1$, the induction hypothesis implies: $\sigma_i^1 = \sigma_{i-1} \models \phi_1$ iff $\sigma_k^1 = \sigma_{k-1} \models \phi_1$, which implies the proof.

For $\phi = \phi_1 \,\mathbf{U}\, \phi_2$: Let $m > n$. We have:

$$\phi_1 \,\mathbf{U}\, \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \,\mathbf{U}\, \phi_2))$$

Applying this law recursively we obtain:

$$\sigma_m \models \phi \quad \text{iff} \quad \sigma_m \models \phi_2 \vee (\sigma_m \models \phi_1 \wedge (\sigma_{m-1} \models \phi_2 \vee (\ldots \\ (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2))))$$

According to the induction hypothesis, we can replace indices bigger than n equivalently by $n+1$:

$$\sigma_m \models \phi \quad \text{iff} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge (\sigma_{n+1} \models \phi_2 \vee (\ldots \\ (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \, \mathbf{U} \, \phi_2))))$$

This can be simplified to the following:

$$\sigma_m \models \phi \quad \text{iff} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \, \mathbf{U} \, \phi_2)$$

Thus, the validity of $\sigma_m \models \phi$ is completely independent of m, leading to the desired property for i and k, which concludes the proof of the lemma.

Proof of the main claim (BA $\not\to$ LTL)

Let us now assume that there exists an LTL formula ϕ expressing the property of the aforementioned BA ("*p* holds in every second step"). Let *n* be the number of occurrences of $\mathbf{X}$ in ϕ.

Let us consider the sequences σ_{n+1} and σ_{n+2}.

If *n* is even then $\sigma_{n+1} \not\models \phi$ and $\sigma_{n+2} \models \phi$. If *n* is odd, then vice versa.

However, the previous lemma tells us that this is impossible: either σ_{n+1} and σ_{n+2} both satisfy ϕ, or none of them does. Therefore, such a formula ϕ cannot exist.

The model-checking problem for LTL (preview)

Problem: Given a Kripke structure $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ and an LTL formula ϕ over AP, we ask whether $\mathcal{K} \models \phi$.

Solution: (sketch)

We re-interpret $\mathcal{K}$ as a Büchi automaton $\mathcal{B}_{\mathcal{K}}$:

$\mathcal{B}_{\mathcal{K}} = (2^{AP}, S, r, \Delta, S)$, where $\Delta = \{\, (s, \nu(s), t) \mid s \rightarrow t \,\}$

Obviously, $[\![\mathcal{K}]\!] = \mathcal{L}(\mathcal{B}_{\mathcal{K}})$.

Moreover, we translate $\neg\phi$ into a Büchi automaton $\mathcal{B}_{\neg\phi}$.

We have:

$$
\begin{array}{rrcll}
& \mathcal{K} & \models & \phi & \\
\iff & [\![\mathcal{K}]\!] & \subseteq & [\![\phi]\!] & \\
\iff & [\![\mathcal{K}]\!] & \cap & [\![\neg\phi]\!] & = \emptyset \\
\iff & \mathcal{L}(\mathcal{B}_{\mathcal{K}}) & \cap & \mathcal{L}(\mathcal{B}_{\neg\phi}) & = \emptyset
\end{array}
$$

Therefore:

We construct Büchi automata $\mathcal{B}_{\mathcal{K}}$ and $\mathcal{B}_{\neg\phi}$.

We intersect both automata (using the special-case construction).

Thus, the model-checking problem reduces to the problem of deciding whether the product automaton accepts the empty language.

Part 6: Efficient Emptiness Test

for Büchi Automata

Overview

As we have seen, the model-checking problem reduces to checking whether the language of a certain Büchi automaton $\mathcal{B}$ is *empty*.

Reminder: $\mathcal{B}$ arises from the intersection of a Kripke structure $\mathcal{K}$ with a BA for the *negation* of ϕ.

If $\mathcal{B}$ accepts some word, we call such a word a counterexample.

$\mathcal{K} \models \phi$ iff $\mathcal{B}$ accepts the empty language.

Typical instances:

Size of $\mathcal{K}$: between several hundreds to millions of states.

Size of $\mathcal{B}_{\neg\phi}$: usually just a couple of states

Typical setting (e.g., in Spin):

$\mathcal{K}$ indirectly given in some description language (C, Java / in Spin: Promela); model-checking tools will generate $\mathcal{K}$ internally.

$\mathcal{B}_{\neg\phi}$ generated from ϕ before start of emptiness check.

Typical setting:

$\mathcal{B}$ generated “on-the-fly” from (the description of) $\mathcal{K}$ and from $\mathcal{B}_{\neg\phi}$ and tested for emptiness *at the same time*.

As a consequence, the size of $\mathcal{K}$ (and of $\mathcal{B}$) is not known initially!

At the beginning, only the initial state is known, and we have a function $\mathsf{succ}\colon S \to 2^S$ for computing the immediate successors of a given state (the function implements the semantics of the description).

Memory requirements

Transitions not stored explicitly, will be explored “on demand” by calling succ (calls to succ will be comparatively costly).

Hash table for explored states.

Information stored for each state:

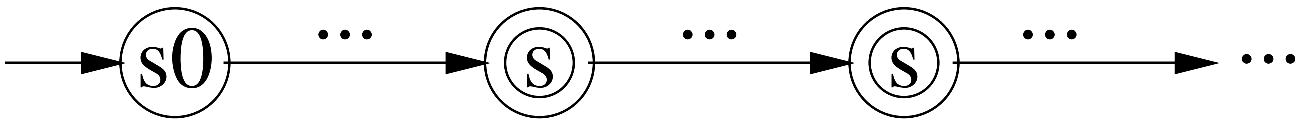
Descriptor: program counter, variable values, active processes, etc (often dozens or hundreds of bytes)

Auxiliary information: Data needed by the emptiness test (a couple of bytes)

Simple solution I: Check for Lassos

Let $\mathcal{B} = (\Sigma, S, s_0, \delta, F)$ be a Büchi automaton.

$\mathcal{L}(\mathcal{B}) \neq \emptyset$ iff there is $s \in F$ such that $s_0 \rightarrow^* s \rightarrow^+ s$



Naïve solution:

Check for each $s \in F$ whether there is a cycle around s; let $F_\circ \subseteq F$ denote the set of states with this property.

Check whether s_0 can reach some state in $F_\circ$.

Time requirement: Each search takes linear time in the size of $\mathcal{B}$, altogether quadratic run-time $\rightarrow$ unacceptable for millions of states.

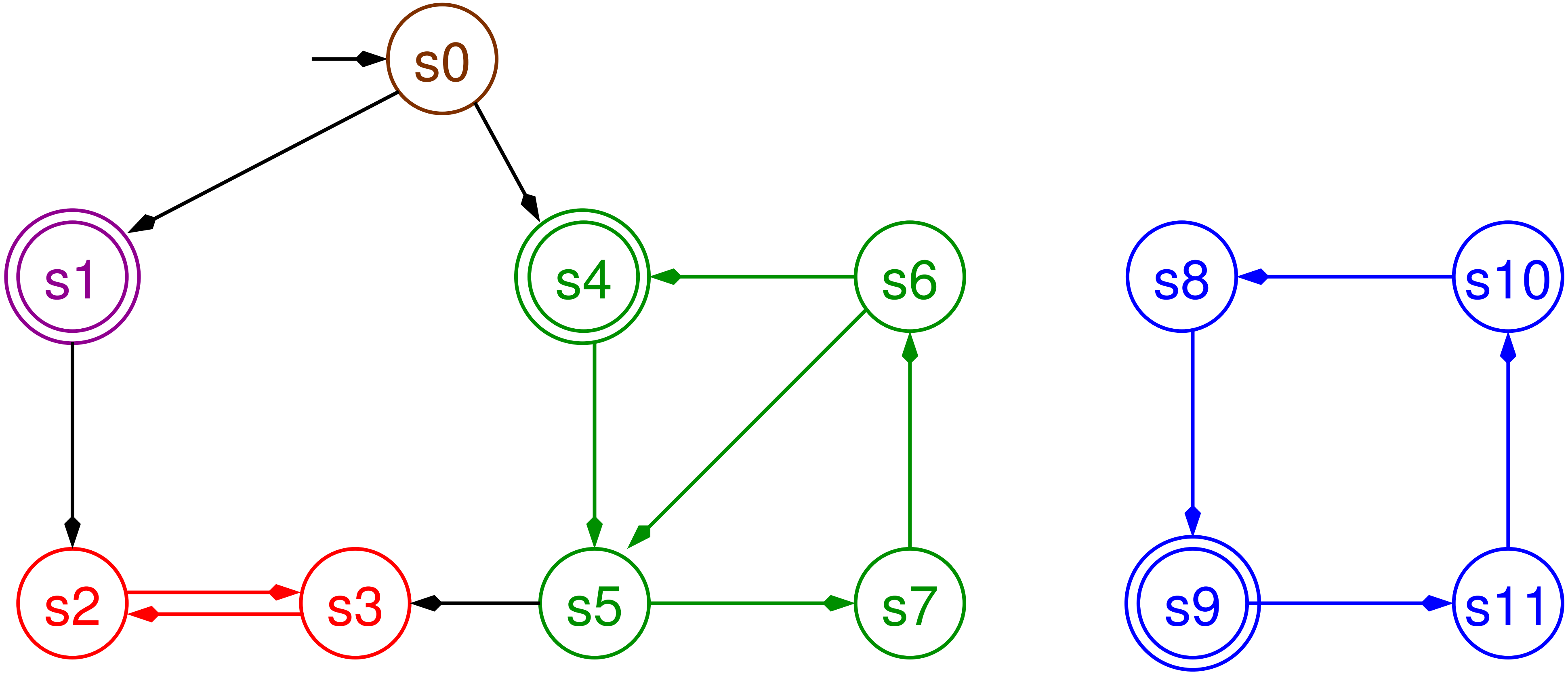
Strongly connected components

$C \subseteq S$ is called a strongly connected component (SCC) iff

- $s \to^* s'$ for all $s, s' \in C$;
- C is maximal w.r.t. the above property, i.e. there is no proper superset of C satisfying the above.

An SCC C is called trivial if $|C| = 1$ and for the unique state $s \in C$ we have $s \not\to s$ (single state without loop).

Example: SCCs



The SCCs $\{s_0\}$ and $\{s_1\}$ are trivial.

Simple algorithm II: SCCs

Observation: $\mathcal{L}(\mathcal{B}) \neq \emptyset$ iff $\mathcal{B}$ has a non-trivial SCC that is reachable from s_0 and contains an accepting state.

Simple algorithm: for every accepting state s

compute the set V_s of the predecessors of s;

compute the set N_s of the successors of s;

$V_s \cap N_s$ is the SCC containing s;

test whether $V_s \cap N_s \supset \{s\}$ or $s \to s$.

Running time: again quadratic

Efficient solution

In the following, we shall discuss a solution whose run-time is linear in $|\mathcal{B}|$ (i.e. proportional to $|S| + |\delta|$).

The solution is based on depth-first search (DFS) and on partitioning $\mathcal{B}$ into its SCCs.

Literature: [Tarjan 1972], Couvreur 1999, Gabow 2000

Depth-first search (basic version)

```
nr = 0;
hash = {};
dfs(s0);
exit;

dfs(s) {
   add s to hash;
   nr = nr+1;
   s.num = nr;

   for (t in succ(s)) {
      // deal with transition s -> t
      if (t not yet in hash) { dfs(t); }
   }
}
```

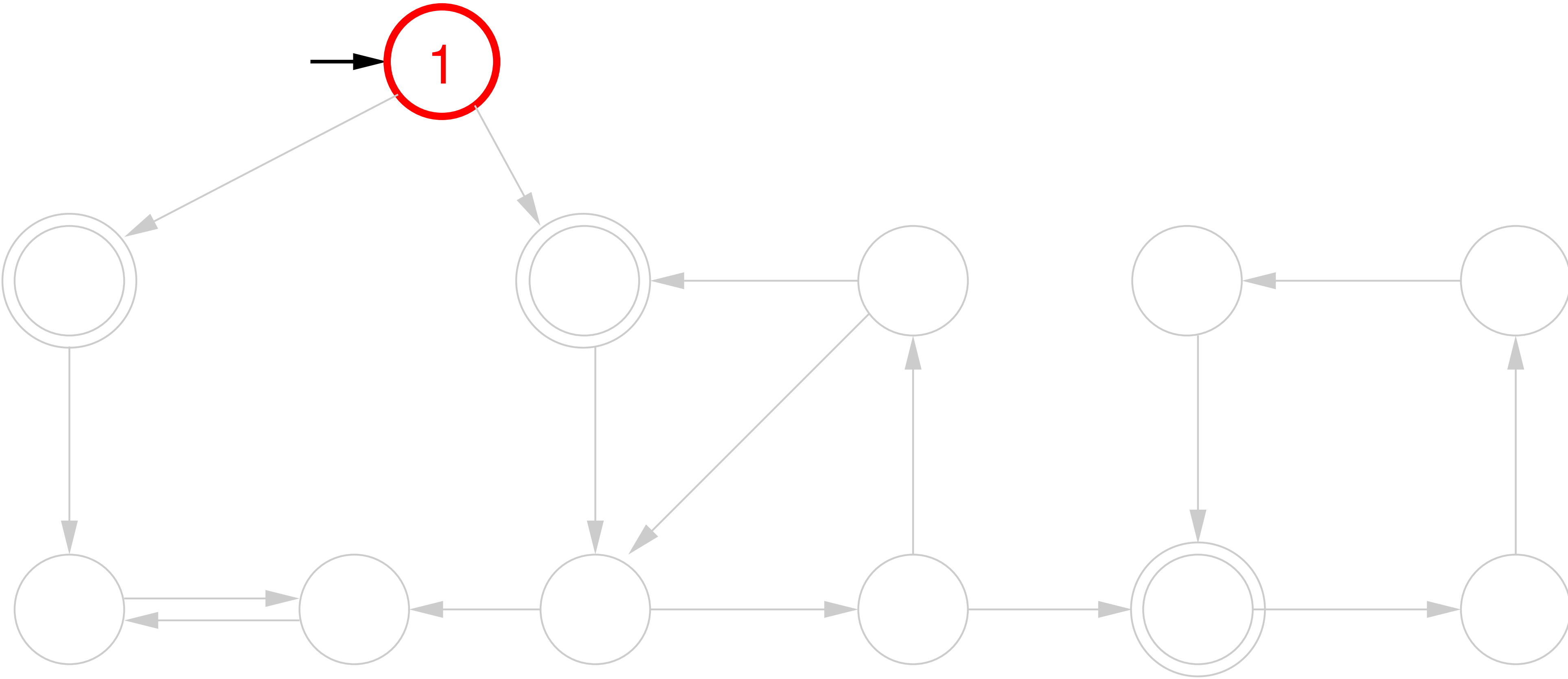
Memory usage

Global variables: counter *nr*, hash table for states

Auxiliary information: “DFS number” *s.num*

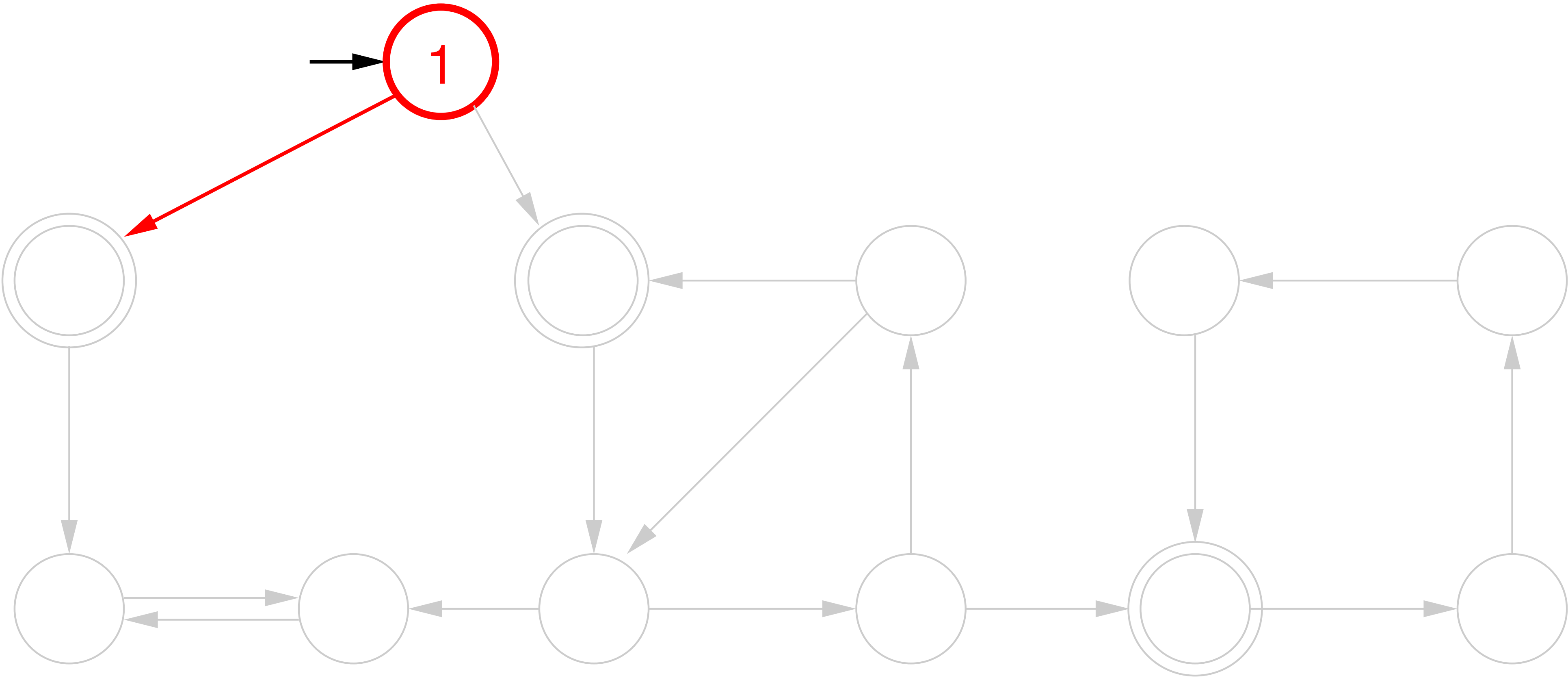
search path: Stack for memorizing the “unfinished” calls to *dfs*

Example: Depth-first search



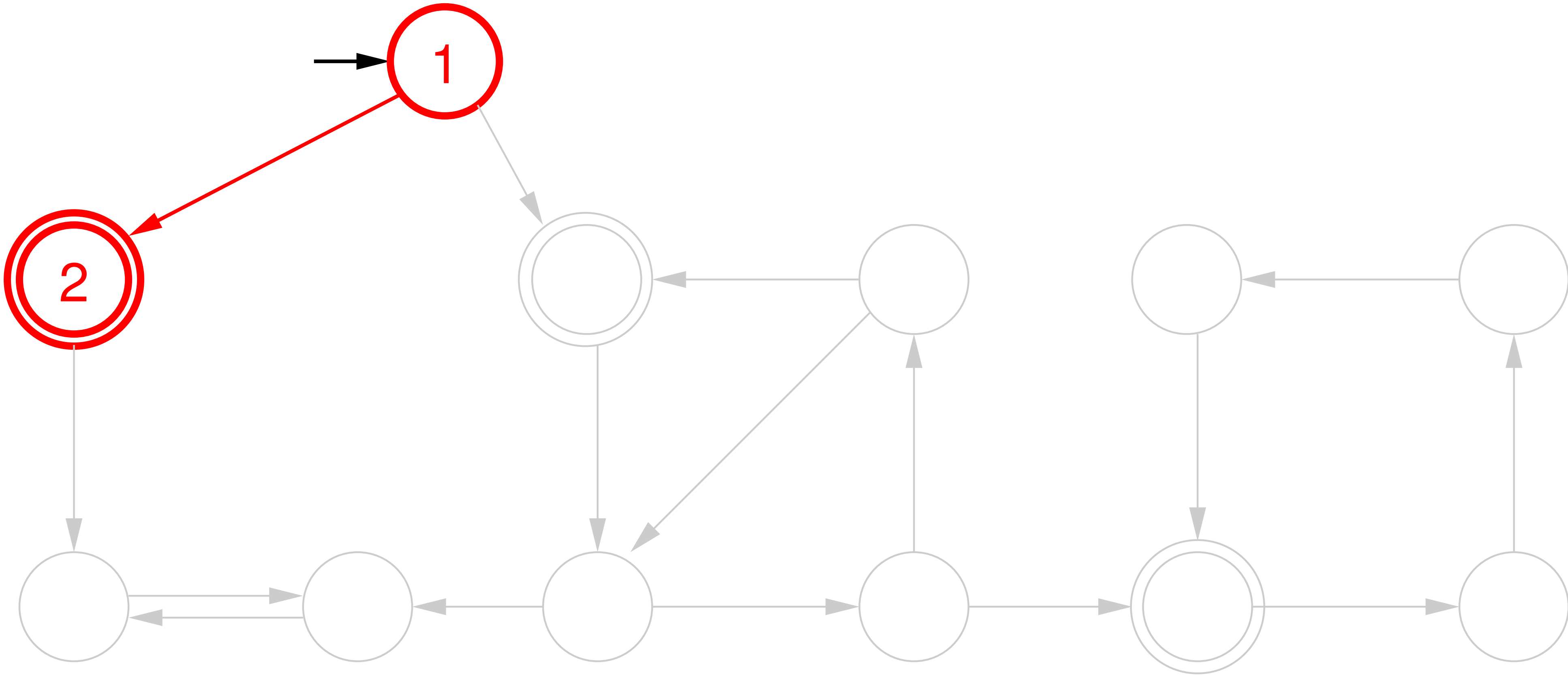
Search path shown in red, other visited states black, states not yet seen grey.

Example: Depth-first search



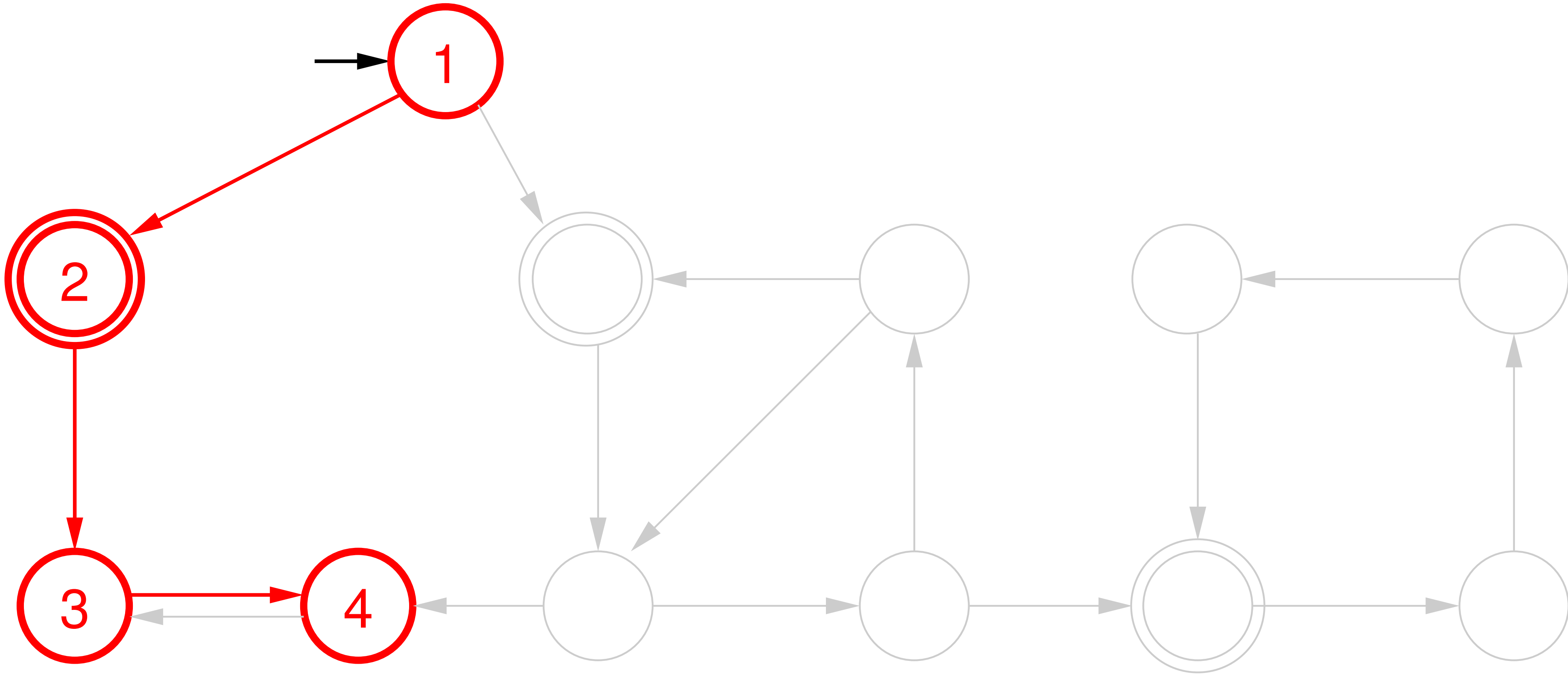
DFS starts at initial state and explores some immediate successor.

Example: Depth-first search



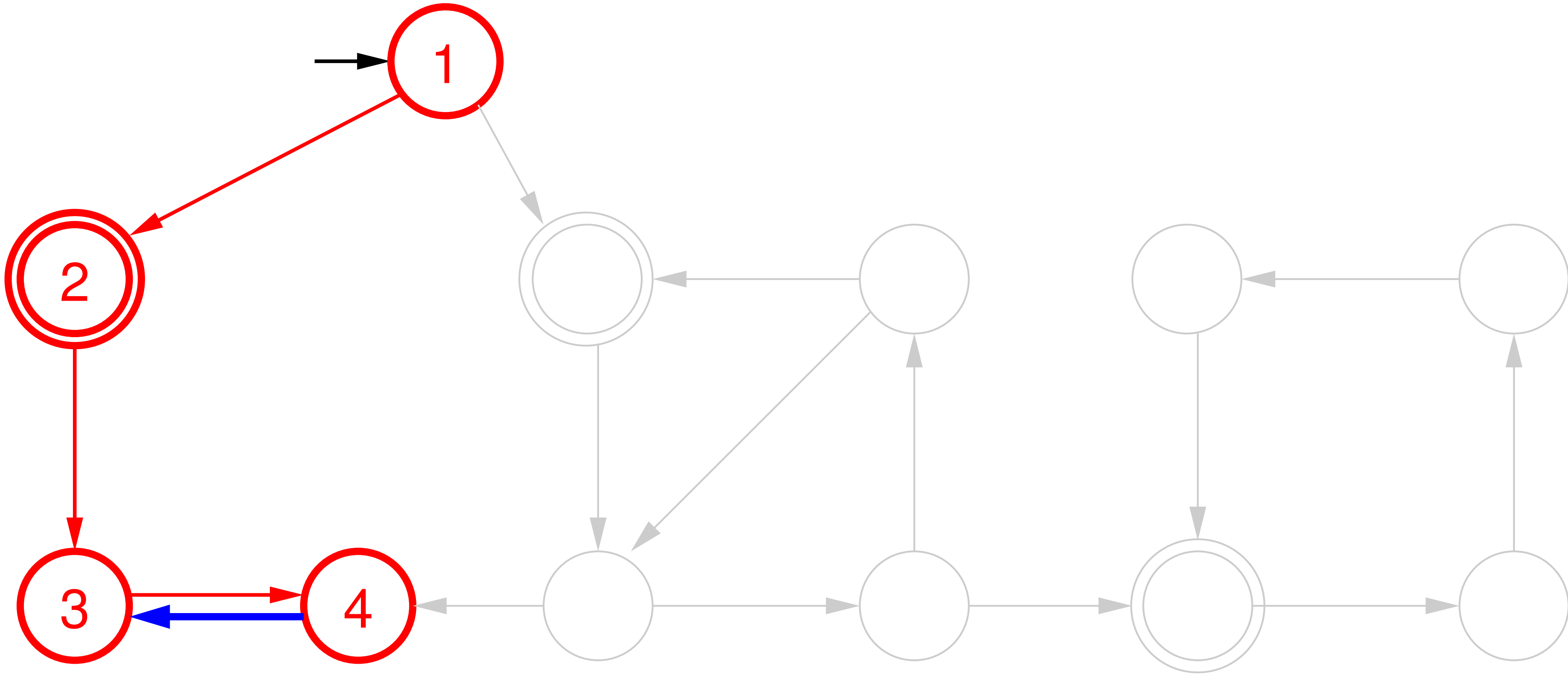
Successor state not yet visited; recursive call, assigned to number 2.

Example: Depth-first search



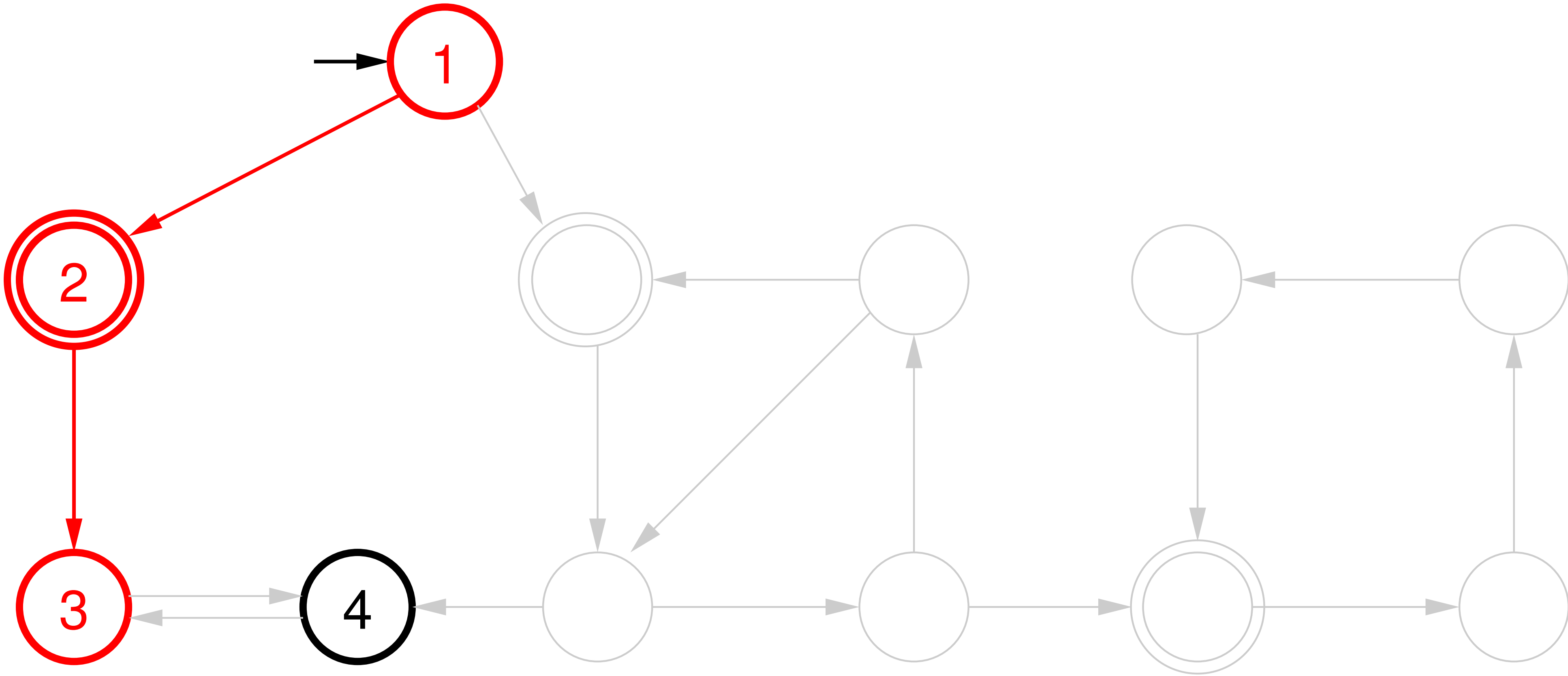
More unvisited states are being explored. . .

Example: Depth-first search



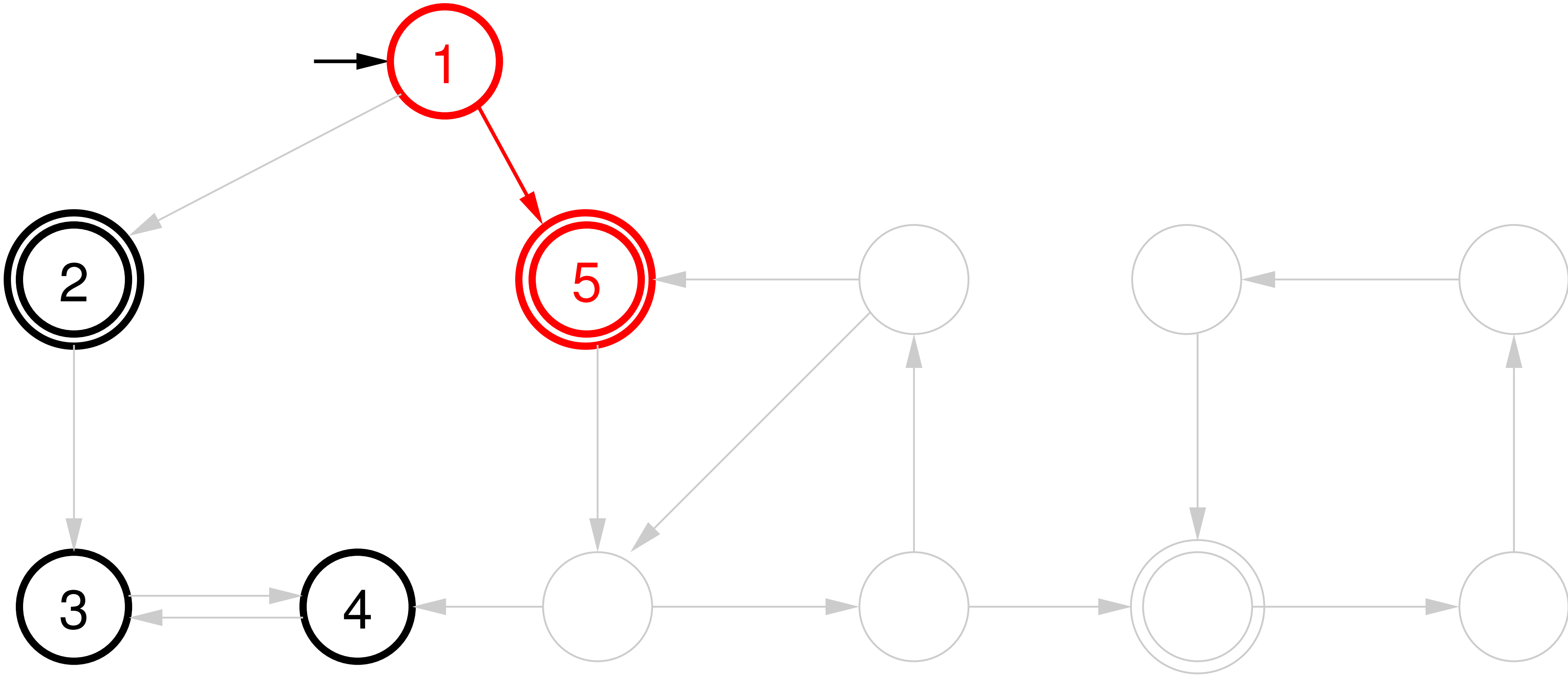
Edge from 4 to 3: target state already known, no recursive call

Example: Depth-first search



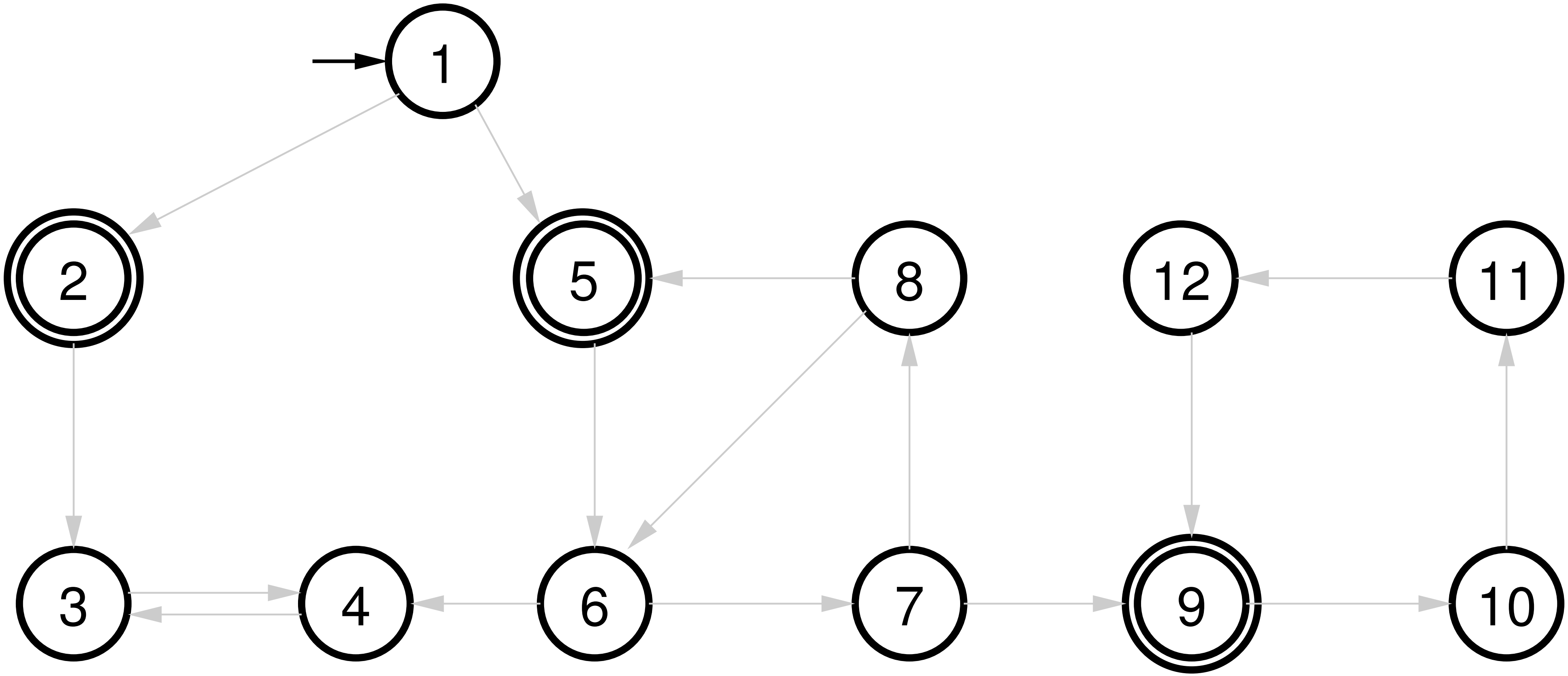
All immediate successors of 4 have been explored; backtrack.

Example: Depth-first search



Backtracking proceeds to state 1, next successor gets number 5.

Example: Depth-first search



Possible numbering at the end of DFS.

Properties of the search path

(1) Let $s_0 s_1 \ldots s_n$ be the search path at some point during DFS.
Then we have $s_i.num < s_j.num$ iff $i < j$.
Moreover, $s_i \to^* s_j$ if $i < j$.

Proof: follows from the logic of the program and the order of recursive calls.

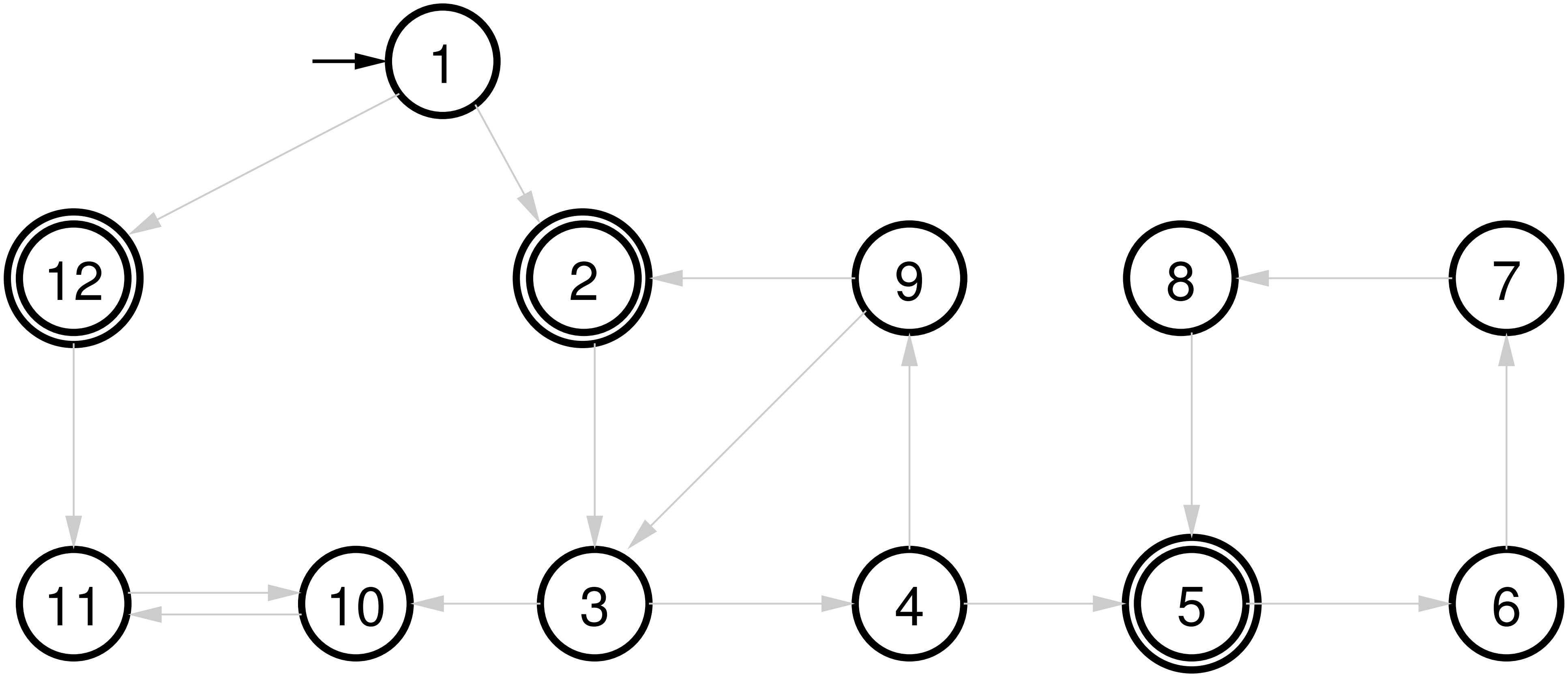
Search order

If a state has got multiple immediate successors, they must be explored in *some* order.

The DFS numbering therefore depends on the order in which these successors are explored; multiple different numberings are possible.

The search order may influence how quickly a counterexample is found (if one exists)!

Example: Search order



Possible alternative numbering for a different search order.

Search order

If a state has got multiple immediate successors, they must be explored in *some* order.

The DFS numbering therefore depends on the order in which these successors are explored; multiple different numberings are possible.

The search order may influence how quickly a counterexample is found (if one exists)!

Assumption: search-order non-deterministic (or fixed from outside)

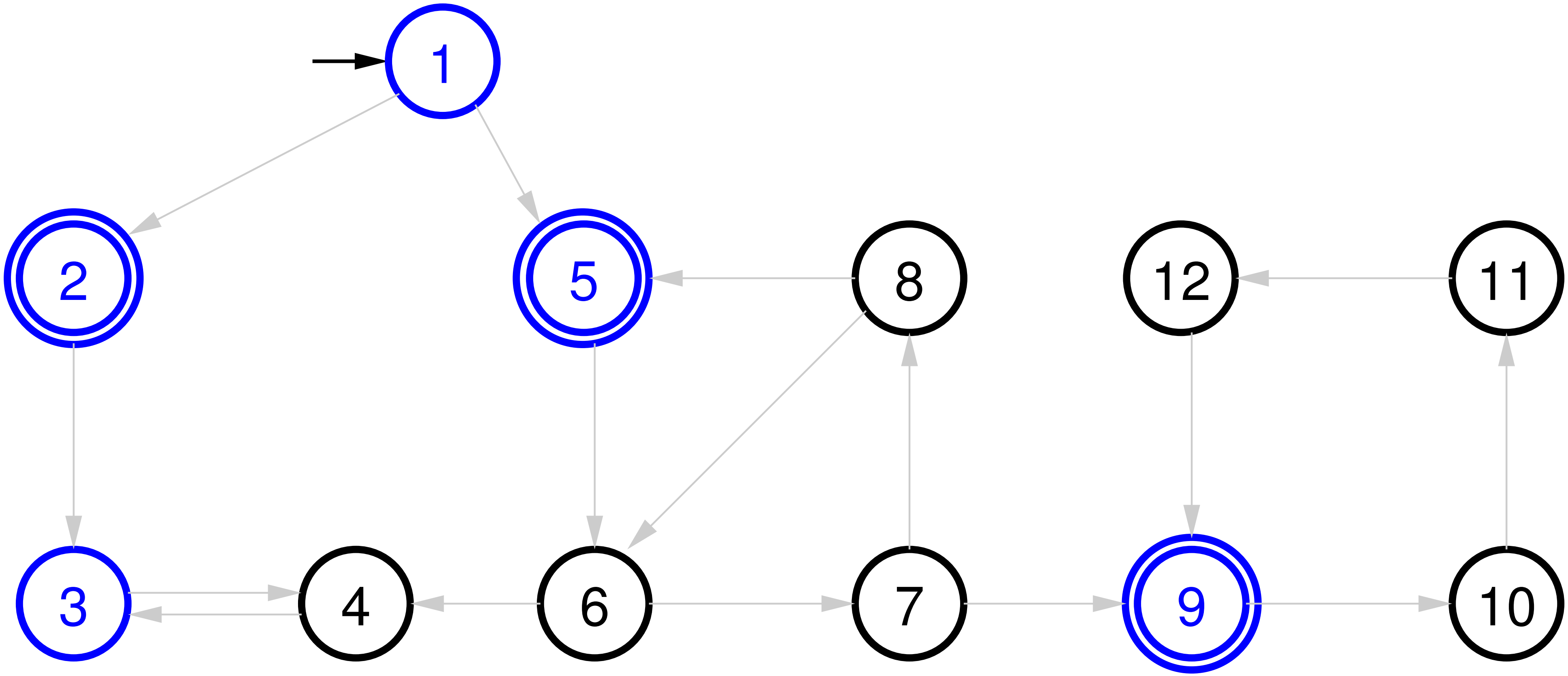
Possible extension: “intelligent” search order exploiting additional knowledge about the model to find counterexamples more quickly.

Roots

The unique (w.r.t. a fixed search order) state of an SCC that is visited first during DFS is called its root.

Remark: Different search orders may lead to different states designated as roots.

Example: Search order



Roots shown in blue when using the previous search order.

Properties of roots

(2) A root has the smallest DFS number within its SCC.

Proof: obvious

(3) Within each SCC, the root is the last state from which DFS backtracks, and, at that point, the SCC has been explored completely (i.e., all states and edges have been considered).

Proof: Suppose the DFS first reaches a root r. At that point, no other state of the SCC has been visited so far, and all are reachable from r. Therefore, the DFS will visit all those states (and possibly others) and backtrack from them before it can backtrack from r.

Explored/active Subgraph

At each point during the DFS, let us distinguish two specific subgraphs of $\mathcal{B}$.

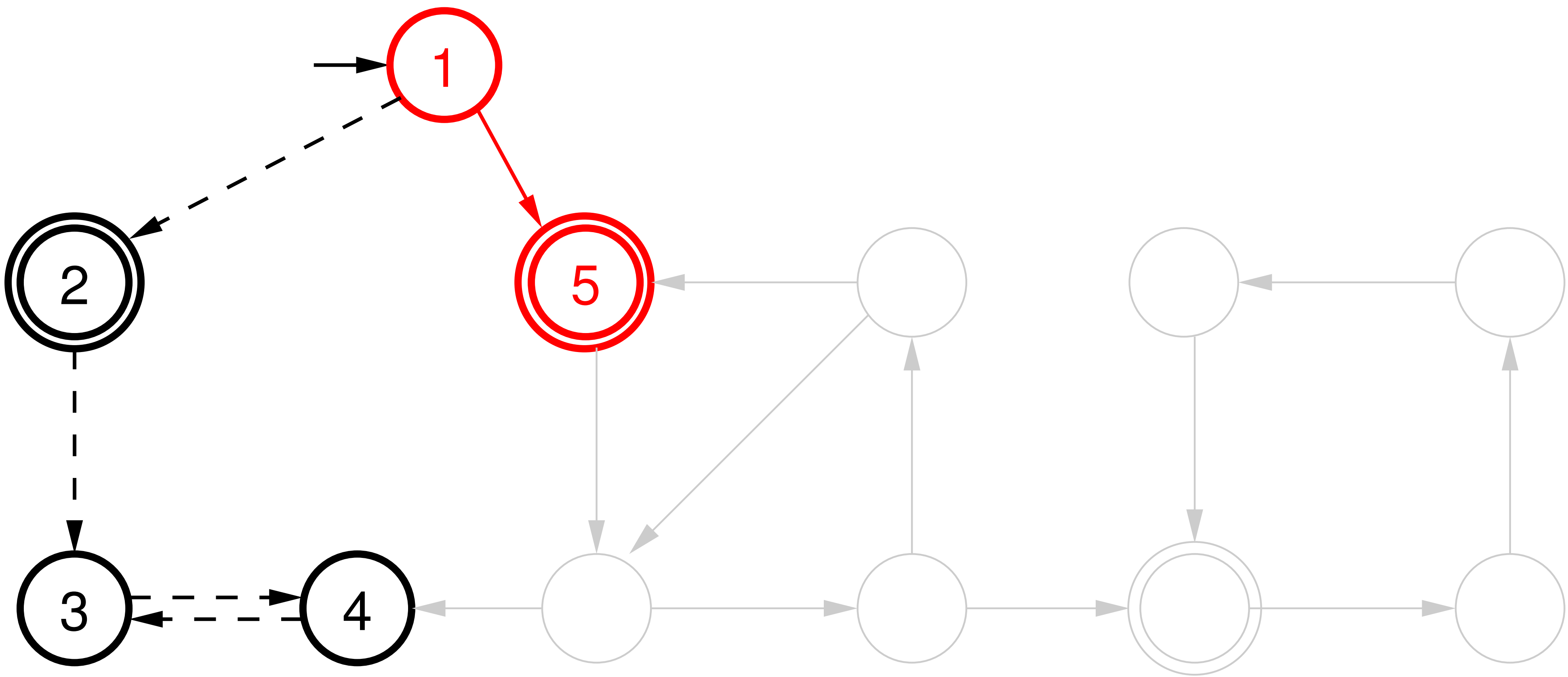
The explored graph of $\mathcal{B}$ denotes the subgraph containing all visited states and explored transitions.

We call an SCC *of the explored graph*(!) active, if the search path contains at least one of its states (whose DFS call has not yet terminated).

A state is called active if it is part of an active SCC (it is not necessary for the state itself to be on the search path).

The active graph is the subgraph of the explored graph induced by the active states.

Example: Explored/active subgraph



Here: explored graph shown in red and black, active SCCs: $\{1\}$ and $\{5\}$, inactive SCCs $\{2\}$ and $\{3, 4\}$.

Properties of the active graph

(4) An SCC becomes inactive when we backtrack from its root.

Proof: follows from (3).

(5) An inactive SCC of the explored graph is also an SCC of $\mathcal{B}$.

Proof: Follows immediately from (3) and (4).

(6) The roots of the active graph are a subsequence of the search path.

Proof: Follows from (4) because the root of an active SCC must be on the search path.

(7) Let s be an active state and t (where $t.num \leq s.num$) the root of its SCC in the active graph. Then there is no active root u with $t.num < u.num < s.num$.

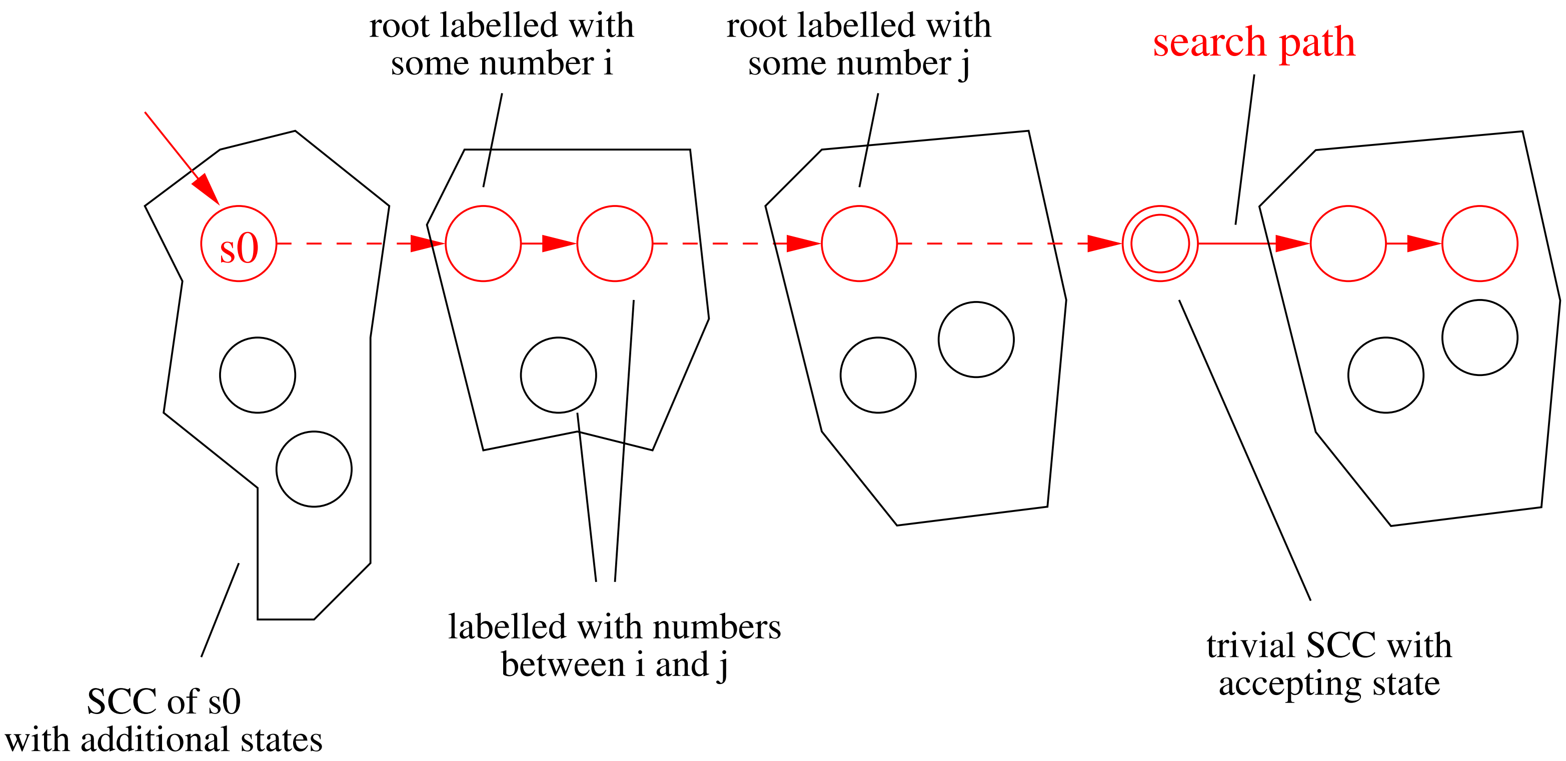
Proof: Assume that such an active root u exists. Since u is active, it is on the search stack, just like t, see (4). Then, because of (1), we have $t \rightarrow^* u$. As `dfs(u)` has not yet terminated and $u.num < s.num$, s must have been reached from u, i.e. $u \rightarrow^* s$. Because s, t are in the same SCC, $s \rightarrow^* t$ holds. But then, t, u are in the same SCC and cannot both be its root.

(8) Let s and t be two active states with $s.num \leq t.num$. Then $s \rightarrow^* t$.

Let s', t' be the (active) roots for s and t, resp. Because of (7) we have $s'.num \leq t'.num$, thus because of (1) $s' \rightarrow^* t'$, and therefore $s \rightarrow^* t$.

Visualization

From the properties we've just proved, it follows that the active graph and its SCCs are always of the following form, at any time during DFS:



Properties of our emptiness-checking algorithm

Run-time linear in $|S| + |\delta|$.

Explores $\mathcal{B}$ using DFS; reports a counterexample *as soon as the explored graph contains one*. (*)

For every explored state s the algorithm computes $\mathsf{succ}(s)$ only once.

$\rightarrow$ saves time because succ is the most expensive operation in practice.

Additional memory usage

Stack W with elements of the form (s, C), where

- s is the root of an active SCC;
- C is the set of state in the SCC of s.

(C may be implemented as a linked list, one additional pointer for each state.)

One bit per state indicating whether a state is active or not.

How the algorithm works

Actions of the algorithm:

- Initialization
- Discovering new edges (to old or new states)
- Backtracking

With each action, we

- update the contents of W and the “active” bits;
- check whether the explored graph contains a counterexample.

Initialization

Explored graph consists just of the initial state s_0, no edges.

One single element in W: the tuple $(s_0, \{s_0\})$

s_0 is active.

Dealing with new edges

Suppose we discover an edge $s \to t$. We distinguish five cases:

Case 1: t was never seen before:

The explored graph is extended by the state t and the edge $s \to t$.

t is active and forms a trivial SCC within the active graph.

Extend W by $(t, \{t\})$.

Recursively start DFS on t.

Dealing with new edges

Suppose we discover an edge $s \to t$. We distinguish five cases:

Case 2: t has been visited before and is inactive.

If t is inactive, then its SCC has been completely explored, see (3) and (4). Therefore, s, t must belong to different SCCs, in particular, $t \to^* s$ cannot hold. Therefore, the edge $s \to t$ cannot be part of a lasso, and we can ignore it.

No recursive call, W and the “active” bits remain unchanged.

Dealing with new edges

Suppose we discover an edge $s \to t$. We distinguish five cases:

Case 3: t was visited before and is active, and $t.num > s.num$.

From (8) we already know that $s \to^* t$ holds, therefore the SCCs of the active graph do not change, and no new counterexample can be generated in this way. Thus, we ignore the edge.

No recursive call, W and the "active" bits remain unchanged.

Dealing with new edges

Suppose we discover an edge $s \to t$. We distinguish five cases:

Case 4: t was visited before and $t.num = s.num$.

Then $s = t$.

A counterexample has been discovered iff s is accepting.

Otherwise: no recursive call, W and the "active" bits remain unchanged.

Dealing with new edges

Suppose we discover an edge $s \to t$. We distinguish five cases:

Case 5: t was seen before and is active, $t.num < s.num$.

Then because of (8) we have $t \to^* s$. Thus, s, t belong to the same SCC. Let u, with $u.num \leq t.num$, be the root of the SCC to which t belongs. Since s is the latest element on the search path, it follows from (1) that all SCCs stored on W from u upwards must be merged into one SCC.

We find u by removing elements from W until we find a root whose number is no larger than $t.num$, compare (7).

A new counterexample is generated only iff one of the merged SCCs was hitherto trivial consisting of an accepting state. Therefore, while removing elements from W we simply check whether any of the roots is an accepting state.

Backtracking

Suppose that all elements in $\mathsf{succ}(s)$ have been explored.

Case 1: s is a root.

Then s and its entire SCC become inactive, see (4).

Moreover, we remove the topmost element from W.

Case 2: s is not a root.

Then the root of its SCC is still active.

W and the “active” bits remain unchanged.

Et voilà. . .

```
nr = 0; hash = {}; W = {}; dfs(s0); exit;

dfs(s) {
   add s to hash; s.active = true;
   nr = nr+1; s.num = nr;
   push (s,{s}) onto W;
   for (t in succ(s)) {
      if (t not yet in hash) { dfs(t); }
      else if (t.active) {
         D = {};
         repeat
            pop (u,C) from W;
            if u is accepting { report success; halt; }
            merge C into D;
         until u.num <= t.num;
         push (u,D) onto W;
   }  }
   if s is the top root in W {
      pop (s,C) from W;
      for all t in C { t.active = false; }
   }
}
```

Remarks on the algorithm

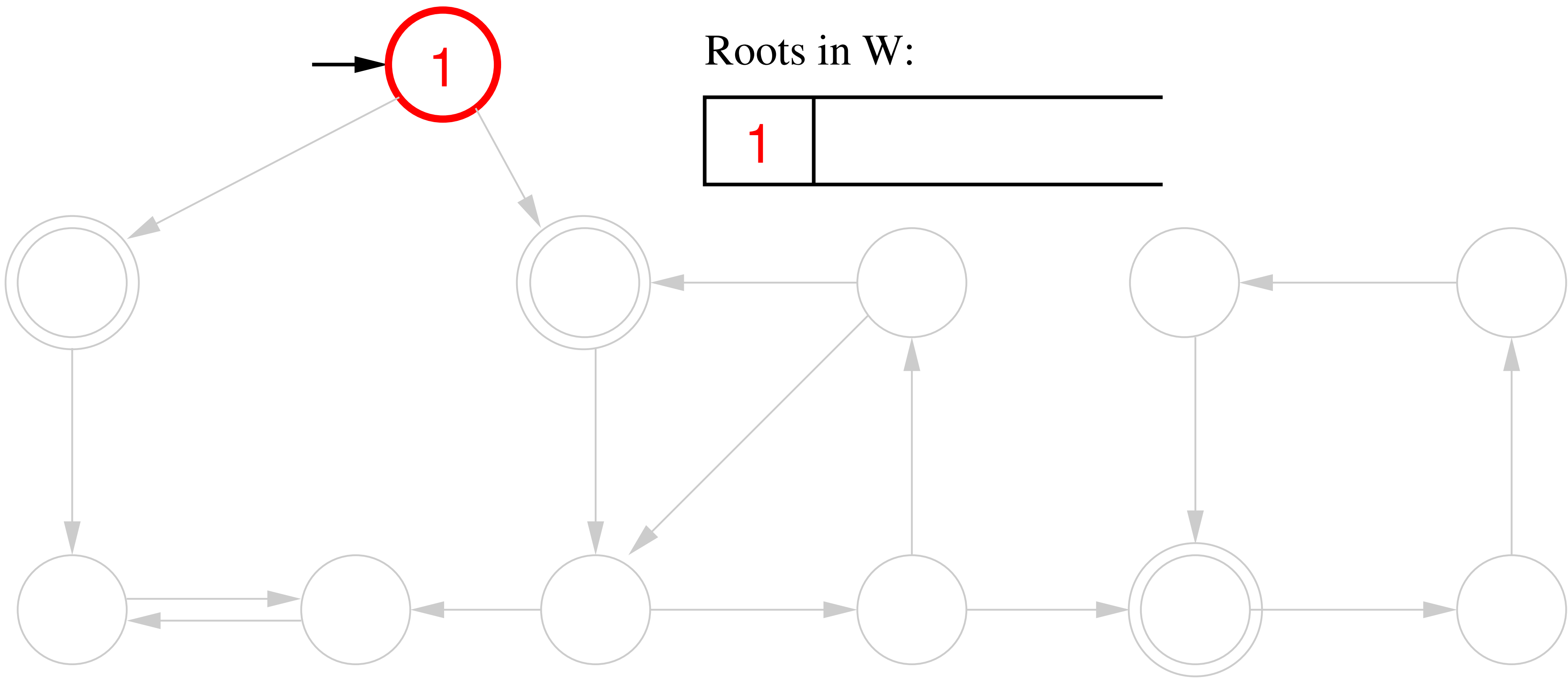
Cases 3 to 5 for handling edges are dealt with uniformly in the repeat-until loop.

The statement `report success` symbolizes the discovery of a counterexample.

If `dfs(s0)` terminates, no counterexample exists.

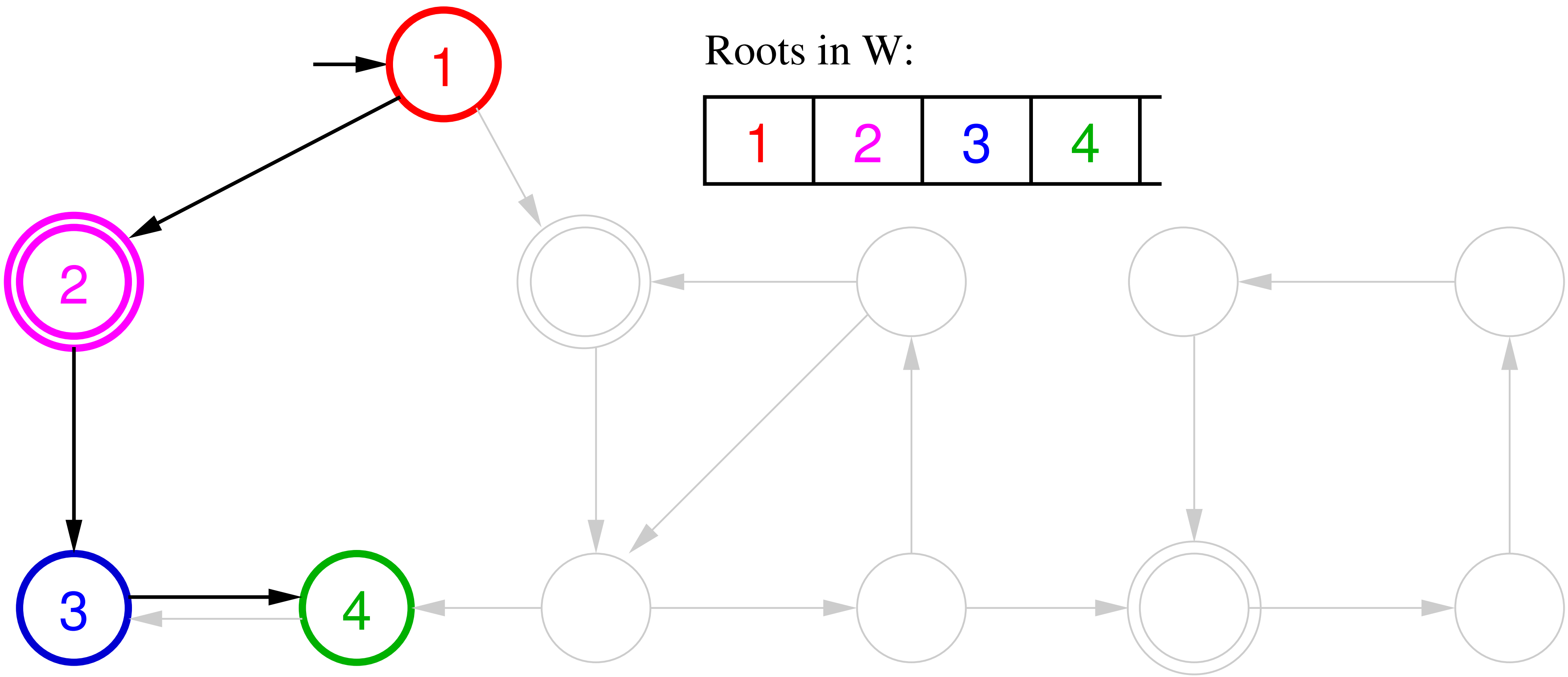
Run-time linear in number of states plus number of transitions.

Example: Execution of the algorithm



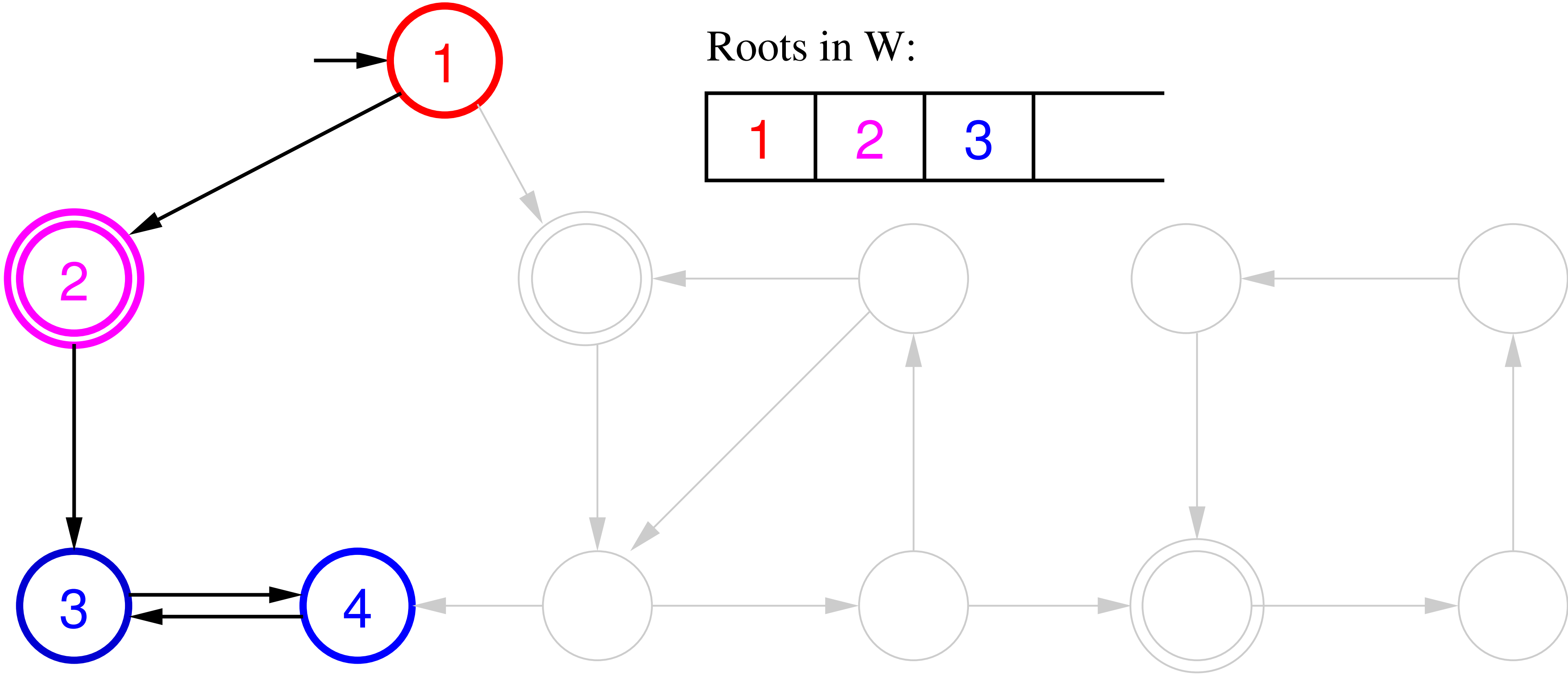
Situation at the beginning, only s_0 explored.

Example: Execution of the algorithm



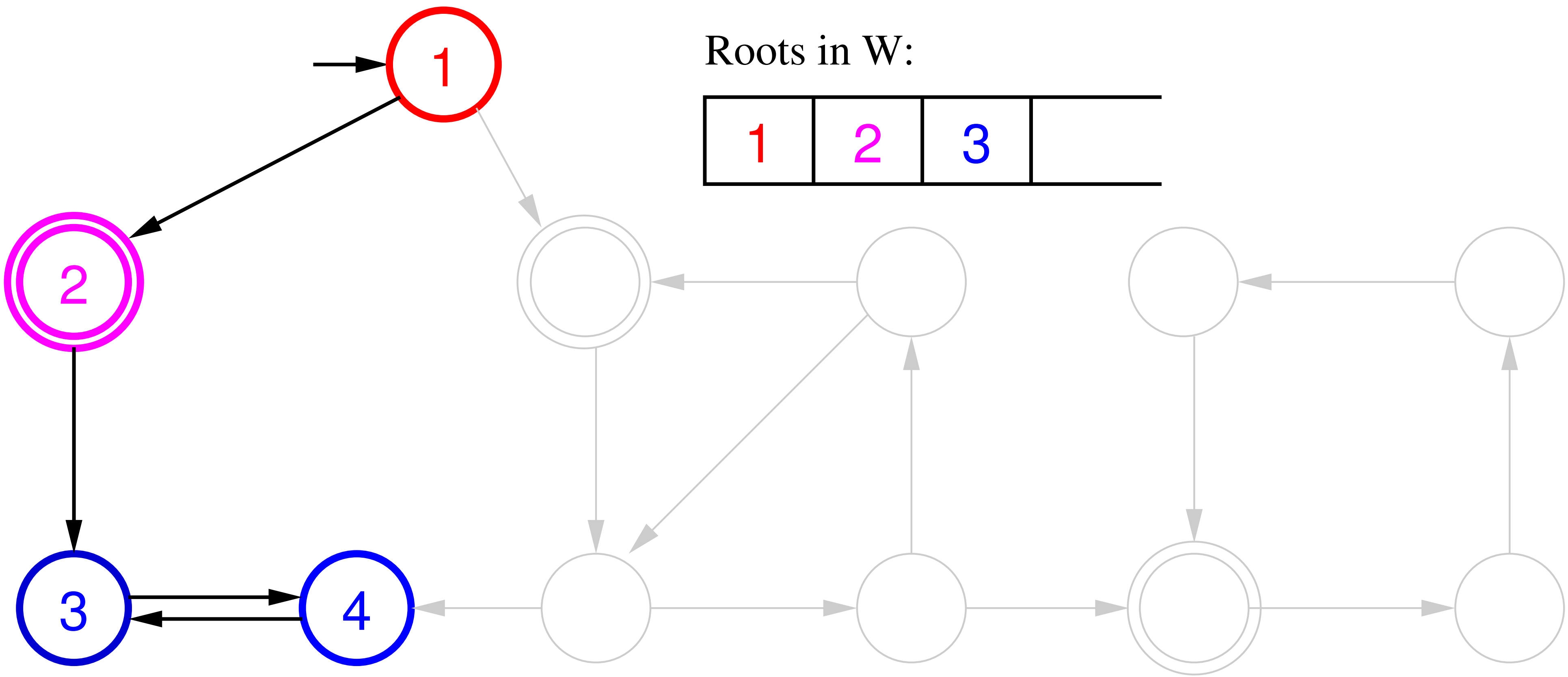
Situation after discovering three edges.

Example: Execution of the algorithm



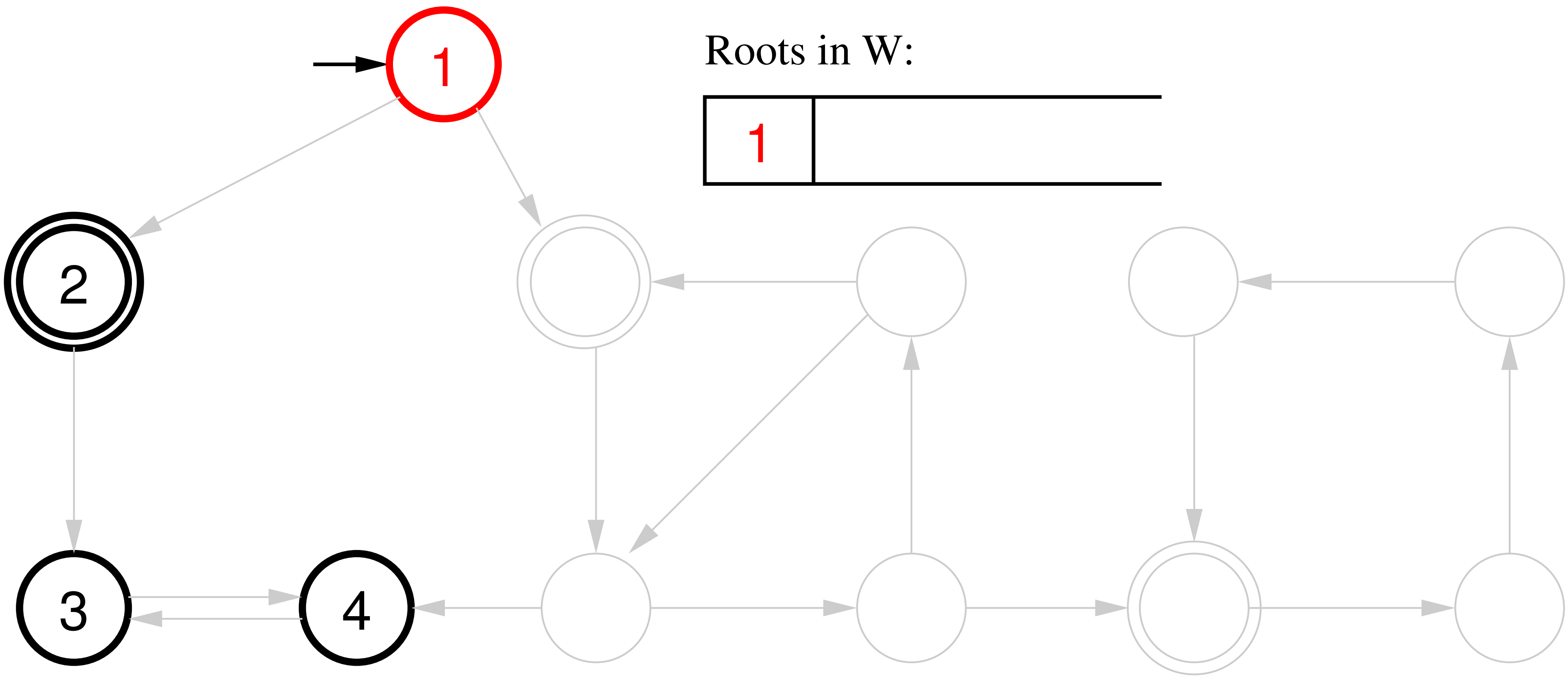
Edge 4 $\rightarrow$ 3 leads to merger of two SCCs.

Example: Execution of the algorithm



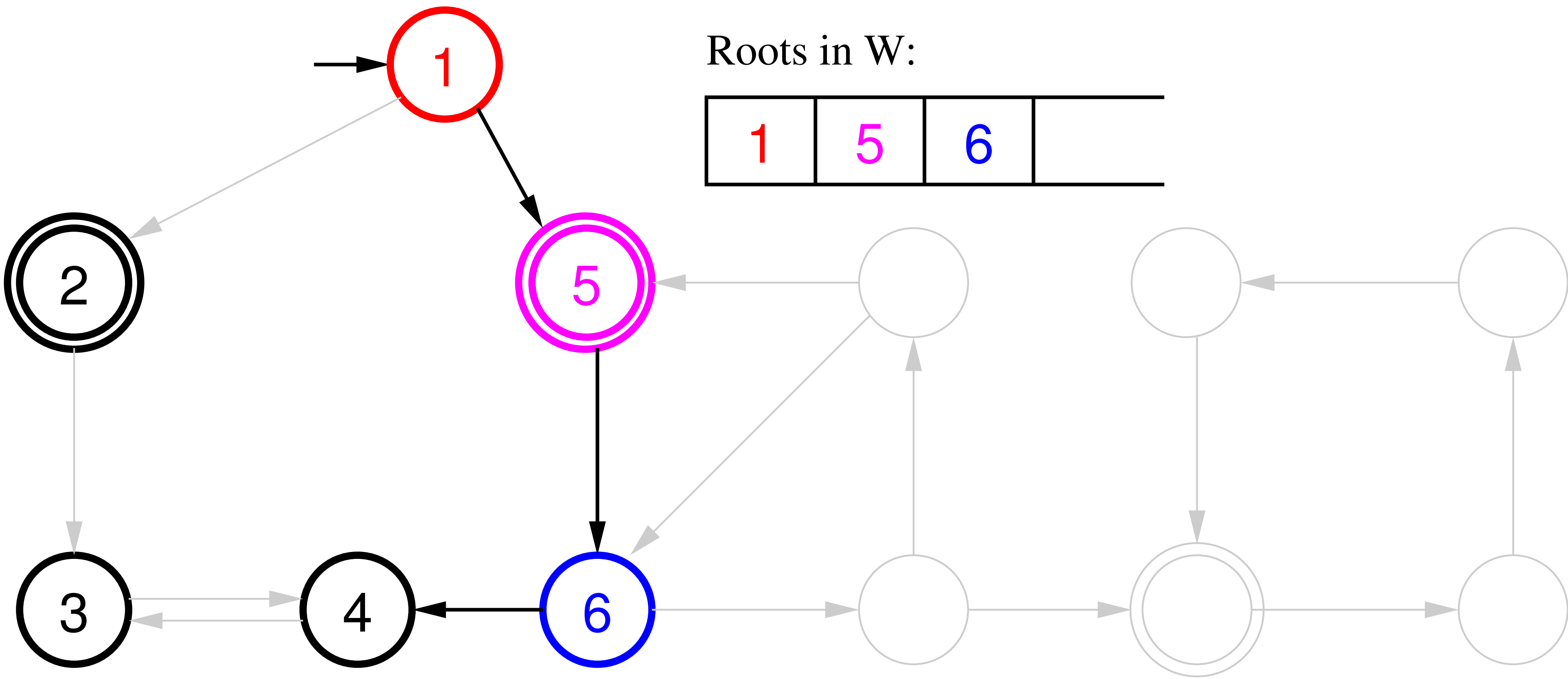
(set component of each entry in *W* indicated by colours)

Example: Execution of the algorithm



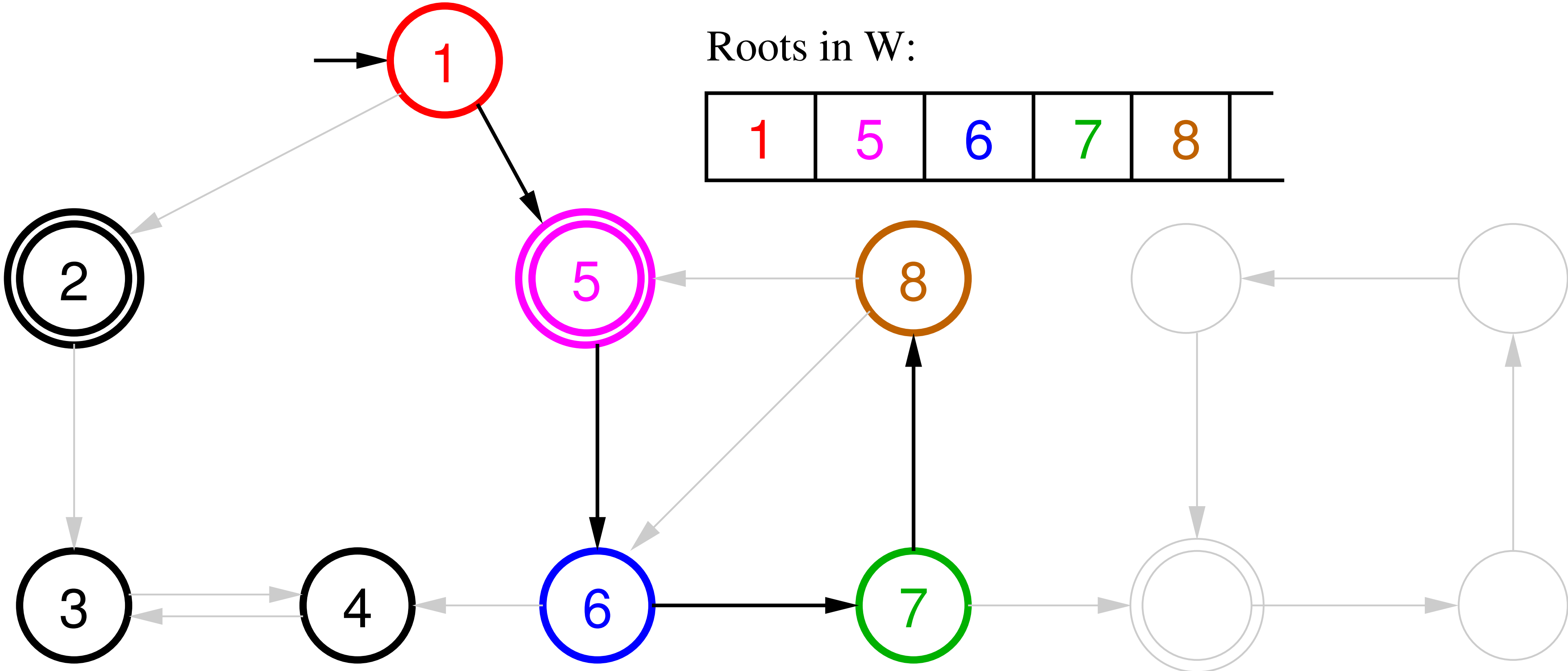
Backtracking makes 2, 3, and 4 inactive (shown in black).

Example: Execution of the algorithm



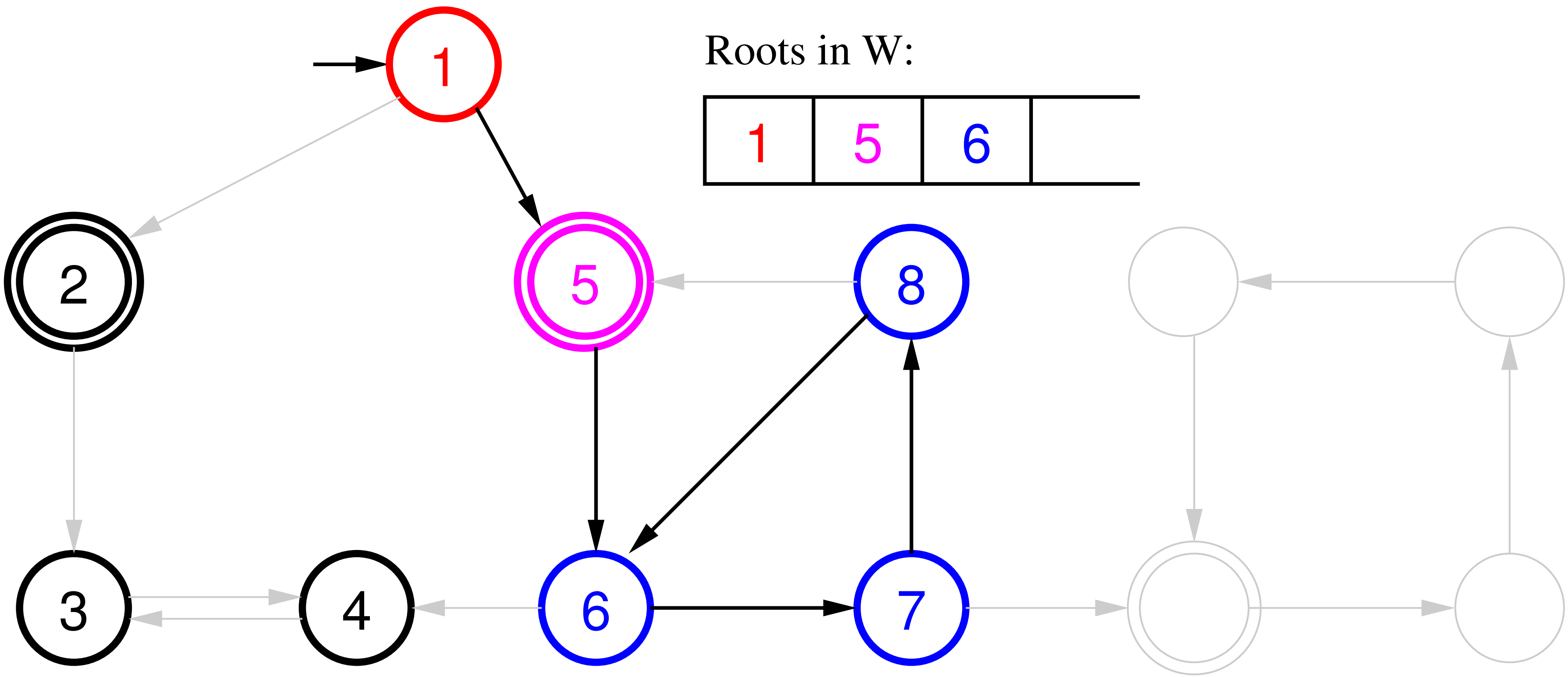
Edge $6 \rightarrow 4$ is an example of Case 2 and may be ignored.

Example: Execution of the algorithm



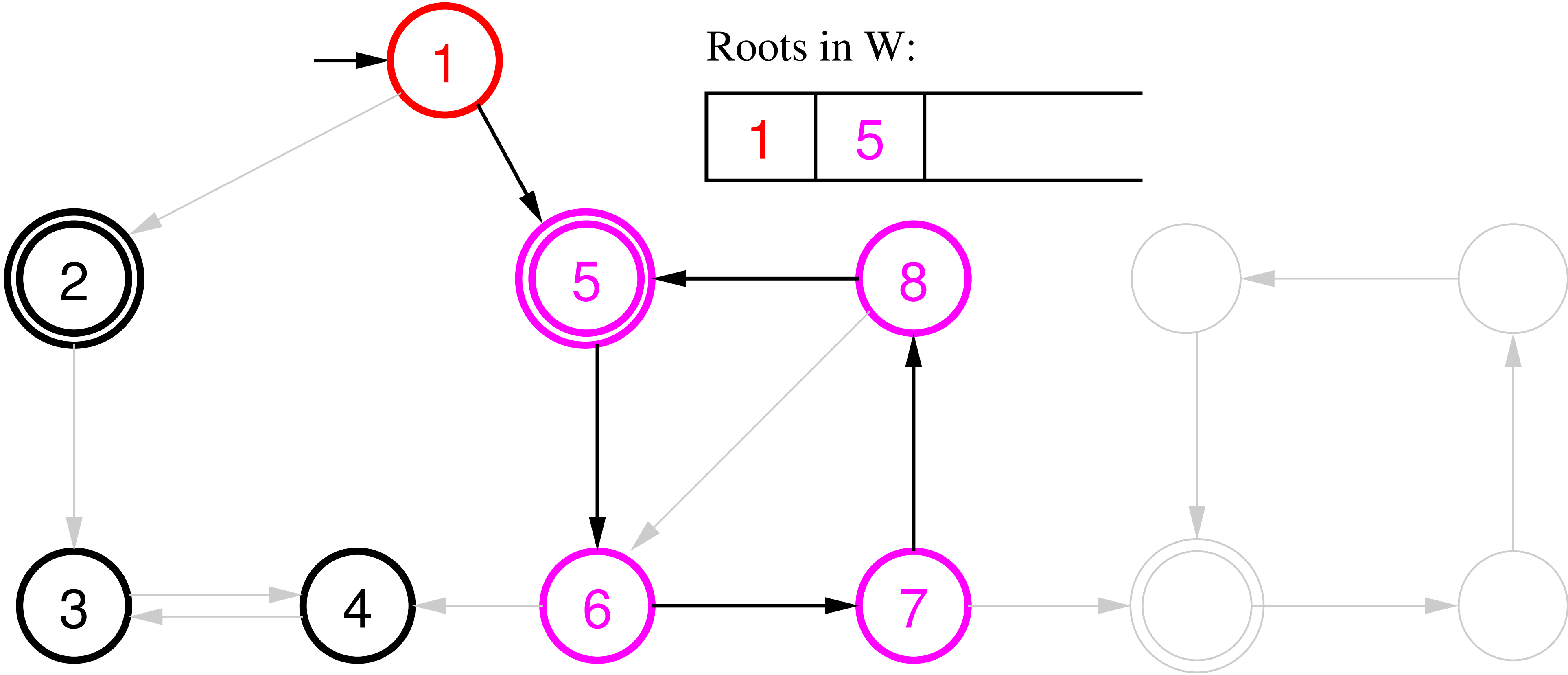
Situation when reaching 8.

Example: Execution of the algorithm



Edge 8 → 6 leads to a merger.

Example: Execution of the algorithm



Edge 8 → 5: Counterexample discovered because root 5 is accepting.

Extension to generalized Büchi automata

Let $\mathcal{G}$ be a GBA with n acceptance sets $F_1, \ldots, F_n$.

$\mathcal{L}(\mathcal{G})$ is non-empty iff there exists a non-trivial SCC intersecting each set F_i ($1 \leq i \leq n$).

Let us label each state s with the index set of the acceptance sets it is contained in, denoted M_s. (E.g., if s in F_1 and in F_3, but in no other acceptance set, then $M_s = \{1, 3\}$.)

We extend W by a third component, an index set, i.e. a subset of $\{1, \ldots, n\}$.

During the algorithm, we uphold the following invariant: if W has an entry (s, C, M), then $M = \bigcup_{t \in C} M_t$.

When two SCCs are merged, we take the union of the index sets.

A counterexample is discovered if this leads to an index set $\{1, \ldots, n\}$.

If n is “small”, the required operations can be implemented using bit vectors (constant time).

Modification for computing SCCs

The algorithm can also be used to partition the BA (or, in fact, any directed graph) into its SCCs.

For this, we simply omit the acceptance test when merging active SCCs.

The algorithm may output a complete SCC as soon as one backtracks from its root.

Part 7: Partial-order reduction

The story so far

Given:

a system S described as Promela model, C program, Petri net, ...
$\Rightarrow$ obtain a Kripke structure $\mathcal{K}$

specification as LTL formula ϕ
$\Rightarrow$ obtain a Büchi automaton $\mathcal{B}$

Approach:

Construct the product of $\mathcal{K}$ and $\mathcal{B}$ and analyze it “on the fly”

run-time linear in $|\mathcal{K}| \cdot |\mathcal{B}|$

State-space explosion

Size of $\mathcal{B}$:

worst-case exponential in $|\phi|$, but mostly harmless

unavoidable (in general)

Size of $\mathcal{K}$:

often exponential in $|S|$

"State-space explosion" caused by concurrency, data, . . .

In the following, we will consider an improvement that tackles the effect of concurrency.

Example: Leader-election protocol

The following protocol is due to Dolew, Klawe, Rodeh (1982). A model of it is contained in the Spin distribution.

The protocol consists of n participants (where n is a parameter). The participants are connected by a ring of unidirectional message channels. Communication is asynchronous, and the channels are reliable. Each participant has a unique ID (e.g., some random number).

Goal: The participants communicate to elect a "leader" (i.e., some distinguished participant). The protocol shown here ensures low communication overhead ($\mathcal{O}(n \log n)$ messages).

Leader-election protocol

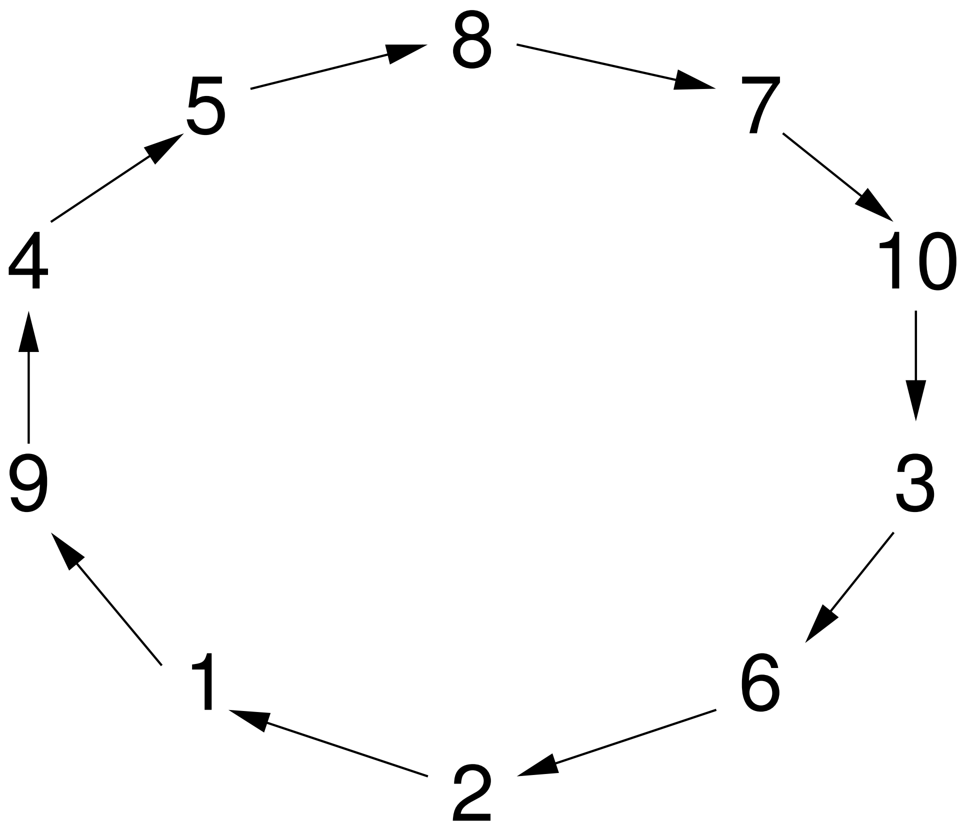
Participants are either active or inactive. Initially, all participants are *active*.

The protocol proceeds in rounds. In each round, at least half of the participants will become inactive. (As a consequence, there are at most $\mathcal{O}(\log n)$ rounds.

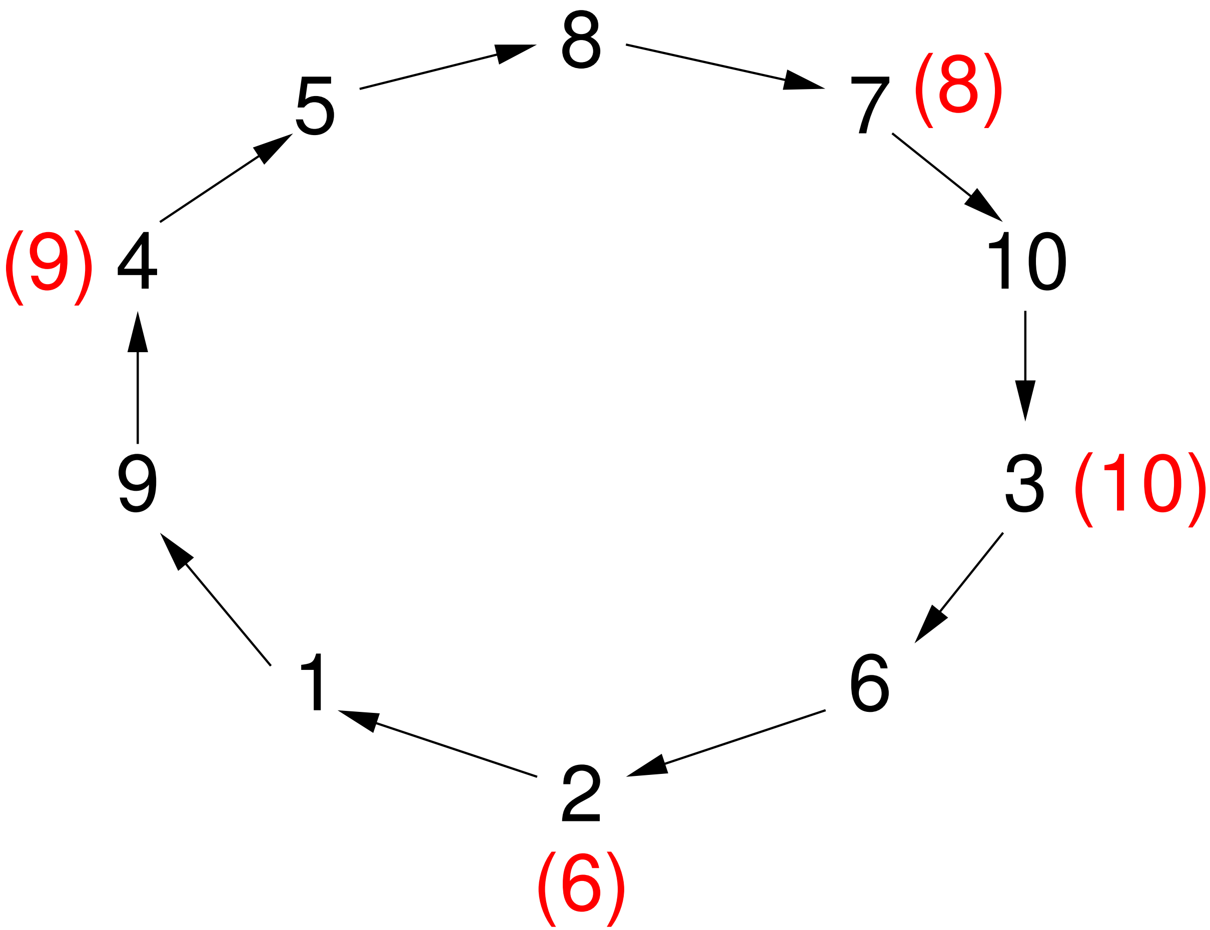
In each round every active participant receives the numbers of the two nearest active participants (in incoming direction). A participant remains active only if the value of the nearest neighbour is the largest of the three. In this case, the participant adopts this largest number as its own.

The last remaining active participant is declared the leader.

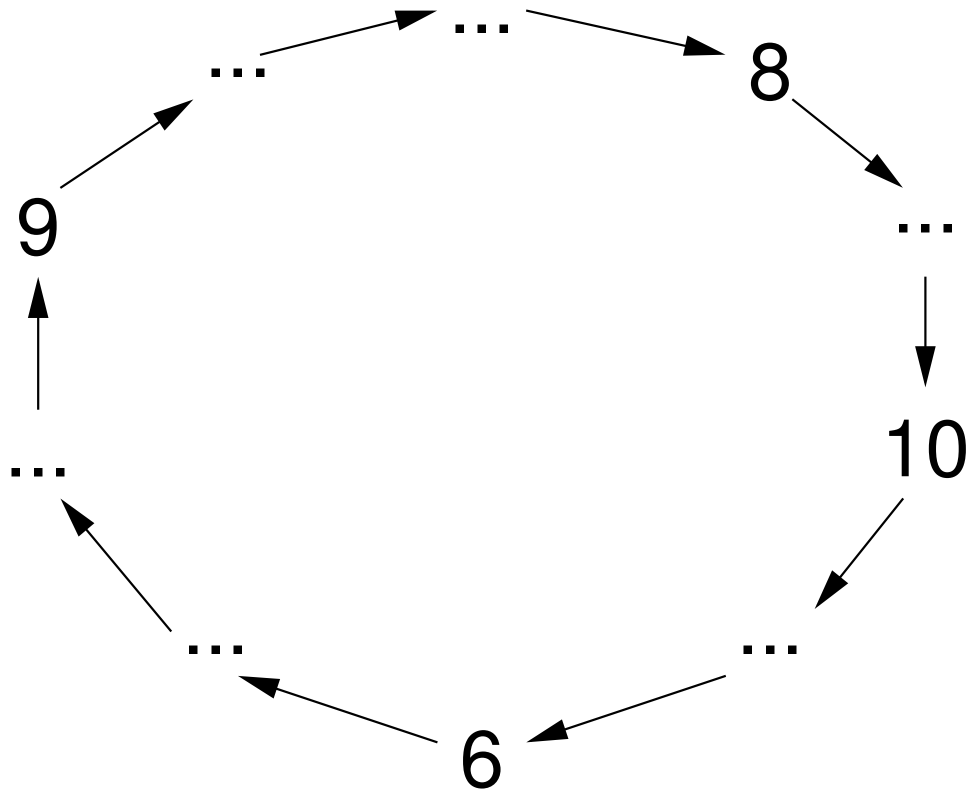
Leader Election: Example



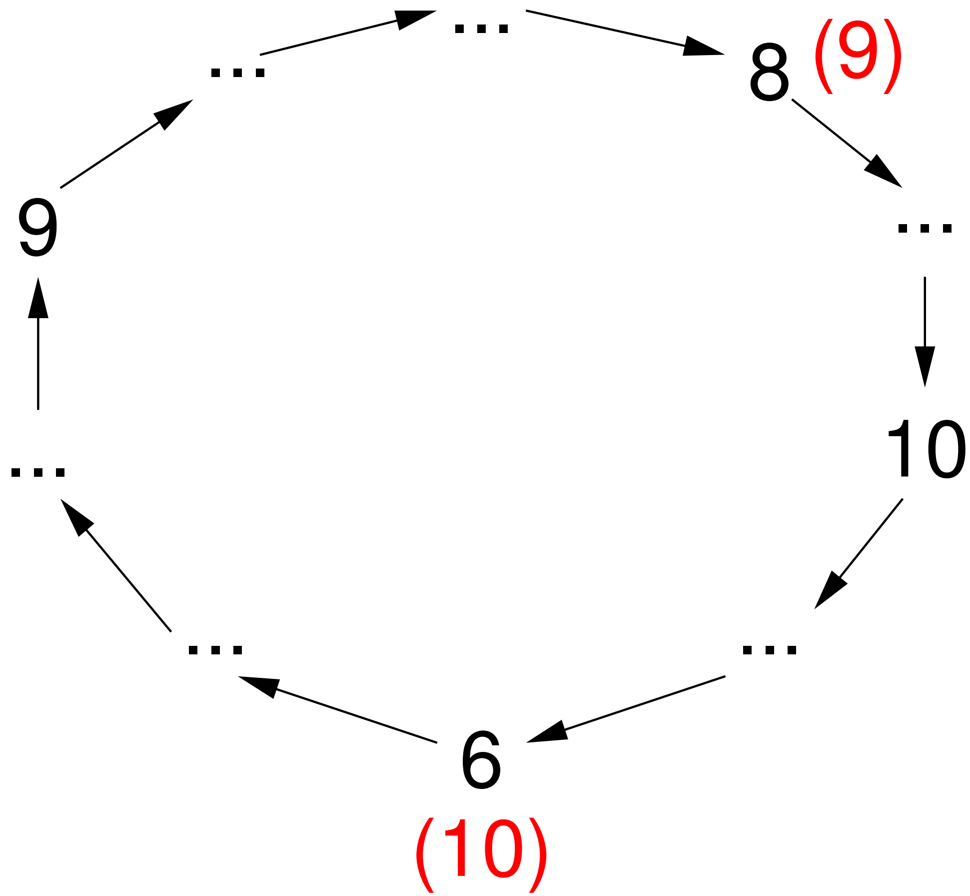
Leader Election: First round



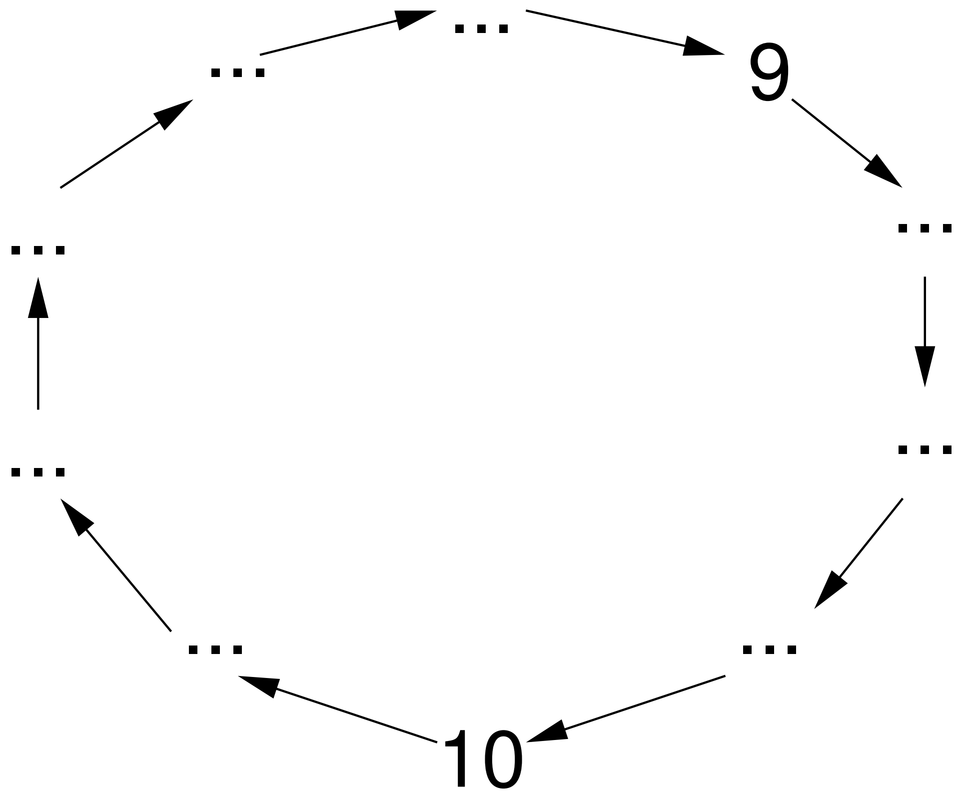
Leader Election: Result of the first round



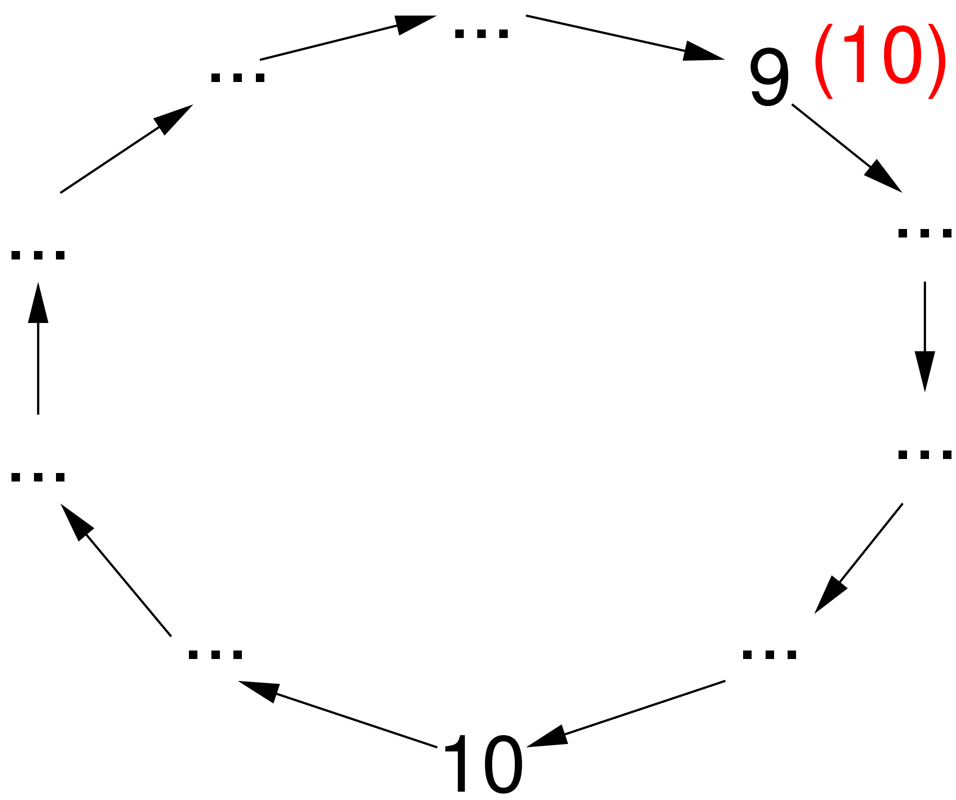
Leader Election: Second round



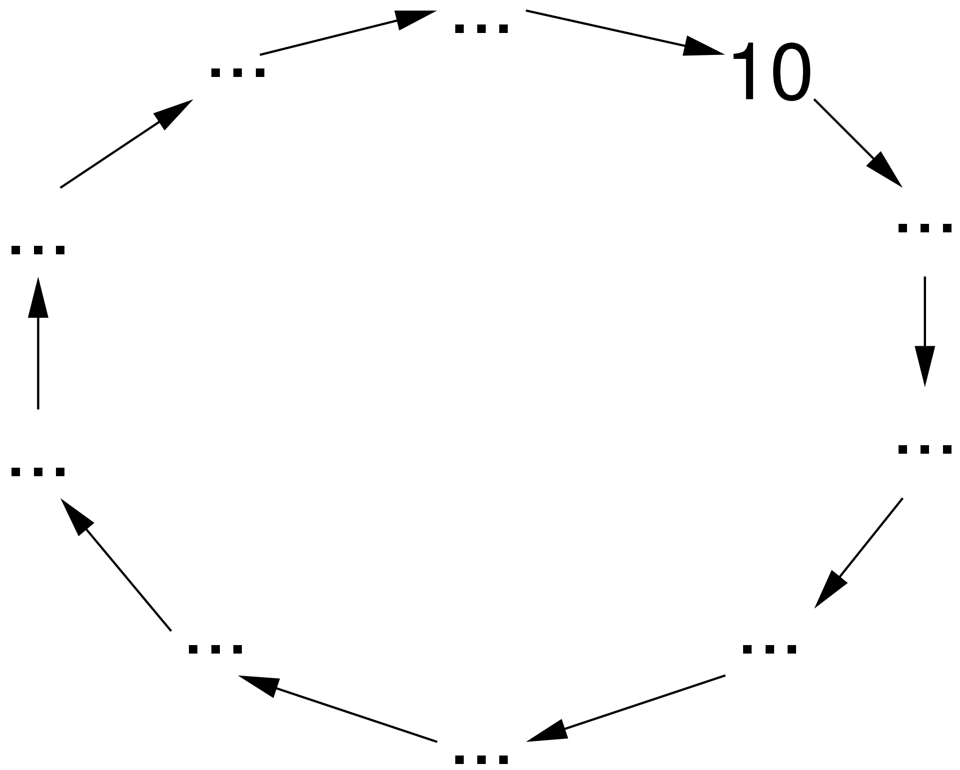
Leader Election: Result of the second round



Leader Election: Third round



Leader Election: Final result



Leader-Election protocol

Motivation: Low message overhead ($\mathcal{O}(n \log n)$ messages).
(Most naïve approaches require quadratically many messages.)

We generate the state space of this example

- with the methods we've seen so far
- using Spin

We will see that the methods we see so far will explore a number of states exponential in n. It will run out of memory for fewer than 10 participants.

In contrast, Spin generates very few states (linearly many in n). This is due to the way Spin handles concurrent processes. Let us take a closer look at what's happening.

Example

Pseudocode program with three concurrent processes:

int *x,y,z* **init** 0;
cobegin $\{P \parallel Q \parallel R\}$ **coend**

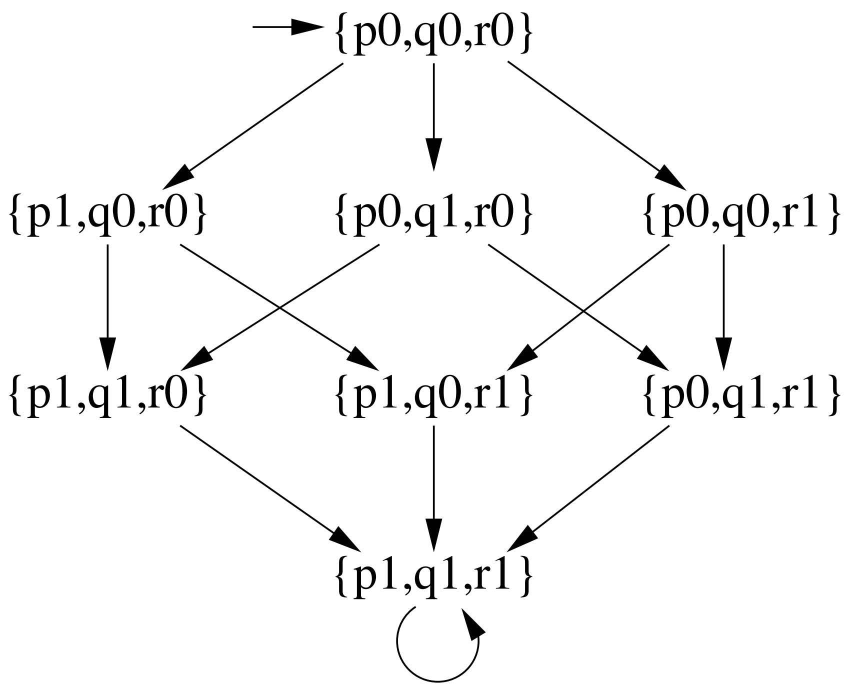
$P =$ `p0:` $x := 1;$ `p1:` **end**

$Q =$ `q0:` $y := 1;$ `q1:` **end**

$R =$ `r0:` $z := 1;$ `r1:` **end**

The transition system has got $8 = 2^3$ states and $6 = 3!$ possible paths.

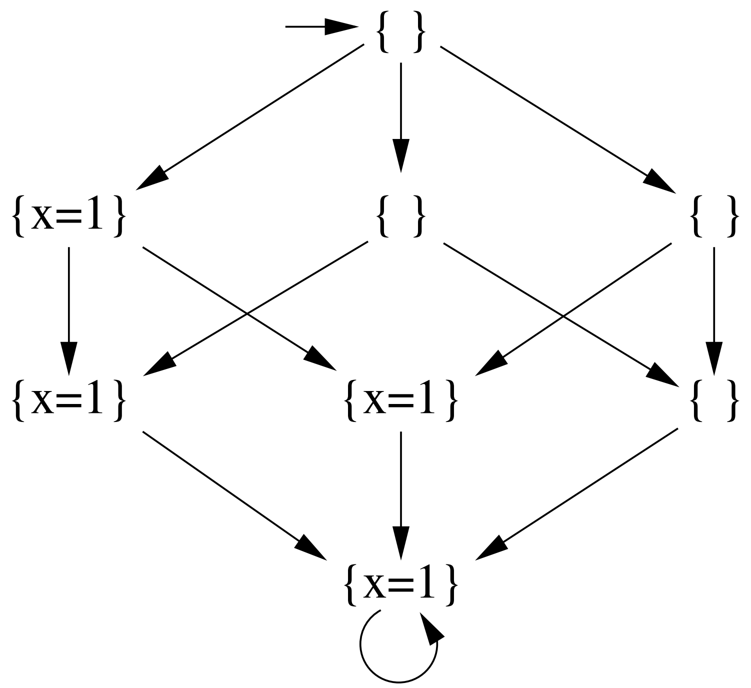
We omit the values of the variables in the states



For n components we have 2^n states and $n!$ paths.

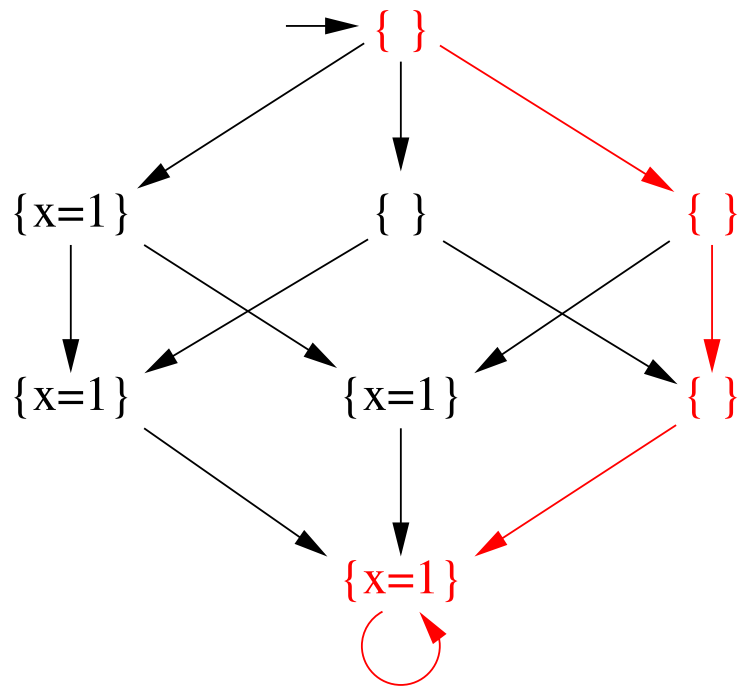
Consider properties like $\mathbf{F}(x = 1)$ or $\mathbf{G}\,\mathbf{F}(x = 1)$.

The Kripke structure for $AP = \{x = 1\}$ is:



Idea: reduce size by considering only one path

Caution: Obviously, this is only possible if the paths are “equivalent”.



I.e., in the eliminated states nothing “interesting” happens.

Partial-order techniques

Partial-order techniques aim to reduce state-space explosion due to concurrency.

One tries to exploit *independences* between transitions, e.g.

Assignments of variables that do not depend upon each other: .

$$\mathtt{x} := \mathtt{z} + 5 \quad \| \quad \mathtt{y} := \mathtt{w} + \mathtt{z}$$

Send and receive on channels that are neither empty nor full.

Idea: avoid exploring all interleavings of independent transitions

correctness depends on whether the property of interest does not distinguish between different such interleavings

may reduce the state space by an exponential factor

Methods: ample sets, stubborn sets, persistent sets, sleep sets

On-the-fly Model Checking

Important: It would be pointless to construct $\mathcal{K}$ first and then reduce its size.

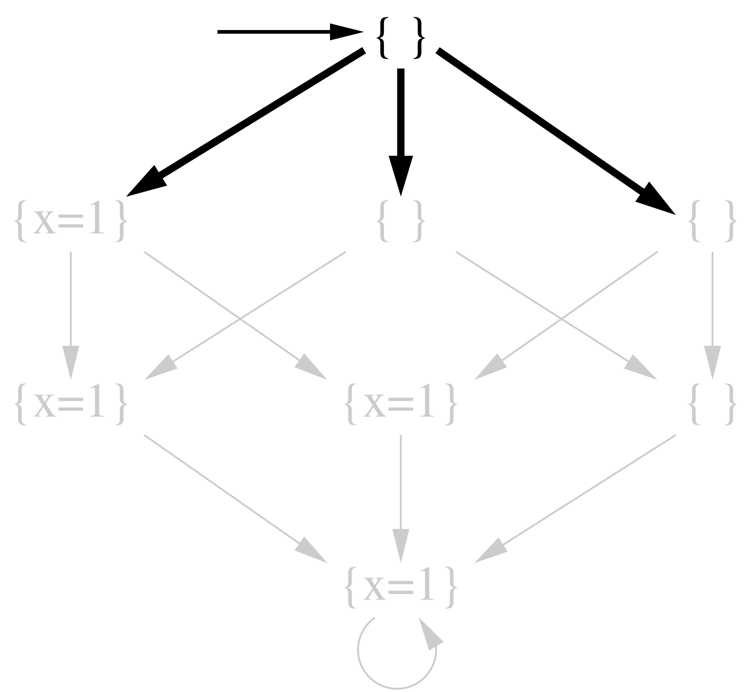
(does not save space, we can analyze $\mathcal{K}$ during construction anyway)

Thus: The reduction must be done "on-the-fly", i.e. while $\mathcal{K}$ is being constructed (from a compact description such as a Promela model) and during analysis.

$\Rightarrow$ combination with depth-first search

Reduction and DFS

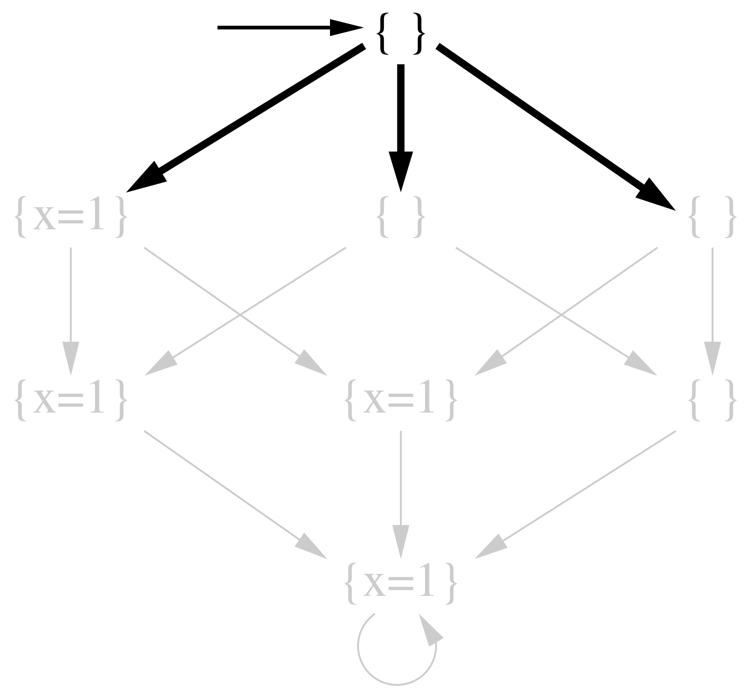
We must decide which paths to explore *at this moment*.



I.e. before having constructed (or "seen") the rest!

Reduction and DFS

We must decide which paths to explore *at this moment*.



$\rightarrow$ only possible with additional information!

Additional information

Transitions labelled with actions.

extracted from the underlying description of $\mathcal{K}$, e.g. the statements of a Promela model etc

Information about independence between actions

Do two actions influence each other?

Information about visibility of actions

Can an action influence the validity of any atomic proposition?

Labelled Kripke structures

We extend our model with actions:

$$\mathcal{K} = (S, A, \rightarrow, r, AP, \nu)$$

S, r, AP, ν as before, A is a set of actions, and $\rightarrow \subseteq S \times A \times S$.

We assume forthwith that transitions are deterministic, i.e. for each $s \in S$ and $a \in A$ there is at most one $s' \in S$ such that $(s, a, s') \in \rightarrow$.

$en(s) := \{\, a \mid \exists s' \colon (s, a, s') \in \rightarrow \,\}$ are called the enabled actions in s.

Independence

$I \subseteq A \times A$ is called independence relation for $\mathcal{K}$ if:

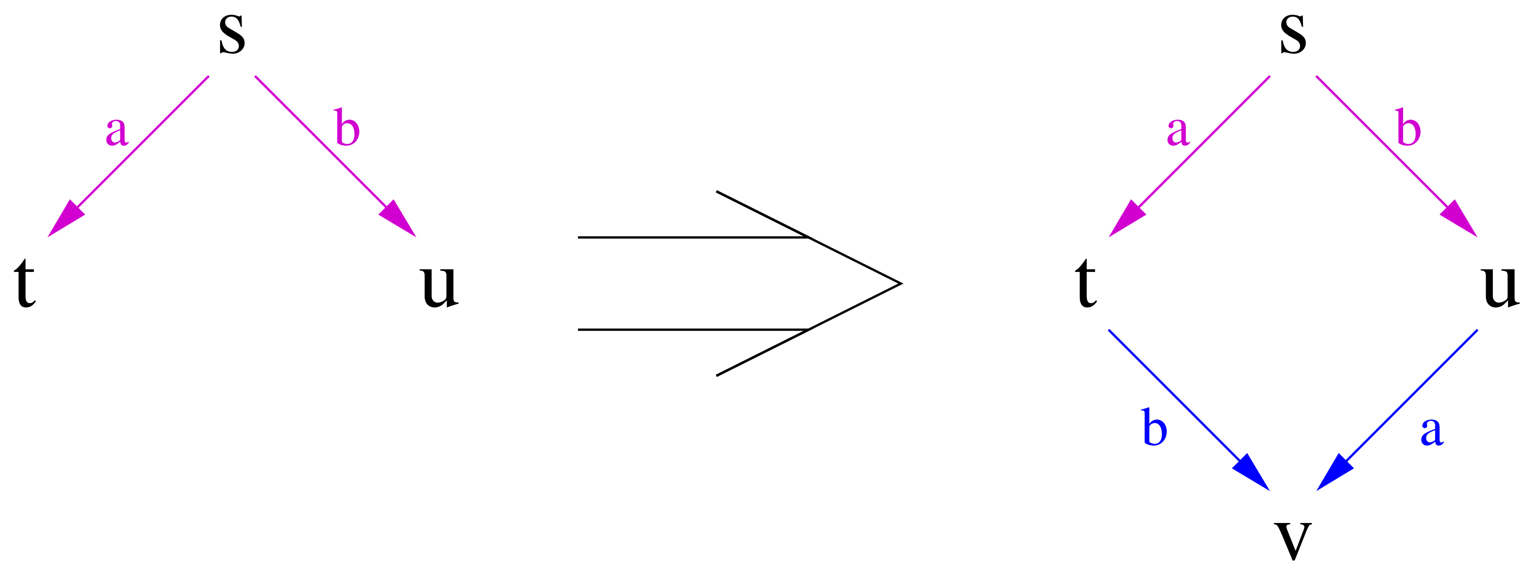
for all $a \in A$ we have $(a, a) \notin I$ (irreflexivity);

for all $(a, b) \in I$ we have $(b, a) \in I$ (symmetry);

for all $(a, b) \in I$ and all $s \in S$ we have:

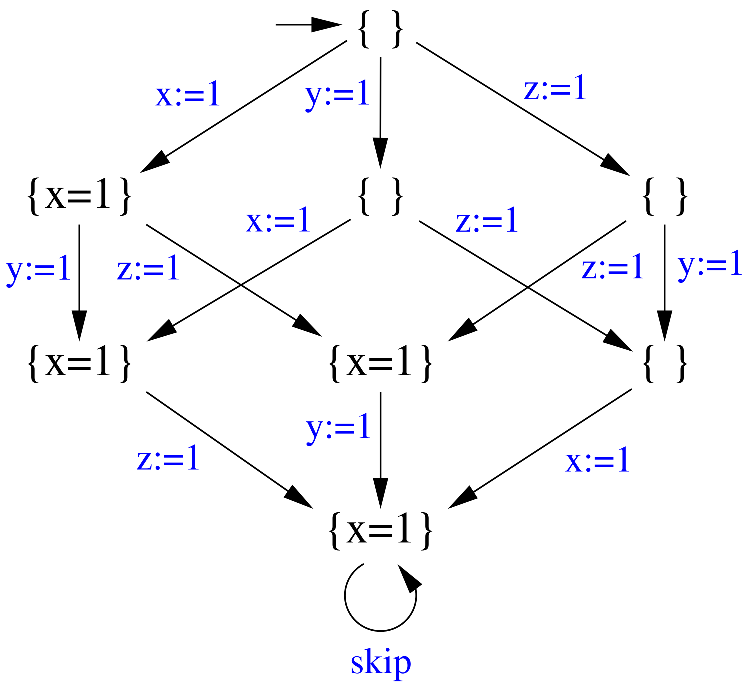
if $a, b \in en(s)$, $s \xrightarrow{a} t$, and $s \xrightarrow{b} u$,
then there exists v such that $a \in en(u)$, $b \in en(t)$, $t \xrightarrow{b} v$ and $u \xrightarrow{a} v$.

Independence



Independence: Example

In the example all pairs of distinct actions are independent.



Remark: In general, an independence relation may not be transitive!

(In-)Visibility

$U \subseteq A$ is called an invisibility set, if all $a \in U$ have the following property:

for all $(s, a, s') \in \rightarrow$ we have: $\nu(s) = \nu(s')$.

I.e., no action in U ever changes the validity of an atomic proposition.

In the example: $\{y := 1, z := 1\}$ (or every subset of it) is an invisibility set.

Motivation: Interleavings of visible actions may not be eliminated because they might influence the validity of LTL properties.

Remarks

Sources for I and U: “external” knowledge about the model and the actions possible in it

e.g. instruction only touches local variables,. . .

will *not* be obtained from first constructing all of $\mathcal{K}$!

Every (symmetric) subset of an independence relation remains an independence relation, every subset of an invisibility set remains an invisibility set.

$\rightarrow$ conservative approximation possible

But: The bigger I and U are, the more information we have at hand to improve the reduction.

In the following, we assume some fixed independence relation I and invisibility set U.

We call a and b independent if $(a, b) \in I$, and dependent otherwise.

We call a invisible if $a \in U$, and visible otherwise.

In the example we take: $I = \{x := 1, y := 1, z := 1\}^2 \setminus Id$

$U = \{y := 1, z := 1\}$

Preview

We first define a notion of "equivalent" runs.

We then consider some conditions guaranteeing that every equivalence class is preserved in the reduced system.

Finally, we consider some practical implementation issues as solved in Spin.

Stuttering equivalence

Definition: Let σ, ρ be infinite sequences over 2^{AP}. We call σ and ρ stuttering equivalent iff there are integer sequences

$$0 = i_0 < i_1 < i_2 < \cdots \text{ and } 0 = k_0 < k_1 < k_2 < \cdots,$$

such that for all $\ell \geq 0$:

$$\sigma(i_\ell) = \sigma(i_\ell + 1) = \cdots = \sigma(i_{\ell+1} - 1) =$$
$$\rho(k_\ell) = \rho(k_\ell + 1) = \cdots = \rho(k_{\ell+1} - 1)$$

(I.e., σ and ρ can be partitioned into "blocks" of possibly differing sizes, but with the same valuations.)

In our example: all infinite sequences of the Kripke structure are stuttering equivalent.

We extend this notion to Kripke structures:

Let $\mathcal{K}, \mathcal{K}'$ be two Kripke structures with the same set of atomic propositions *AP*.

$\mathcal{K}$ and $\mathcal{K}'$ are called stuttering equivalent iff for every sequence in $[\![\mathcal{K}]\!]$ there exists a stuttering equivalent sequence in $[\![\mathcal{K}']\!]$, and vice versa.

I.e., $[\![\mathcal{K}]\!]$ and $[\![\mathcal{K}']\!]$ contain the same equivalence classes of runs.

In our example: The Kripke structure containing only “the rightmost path” is stuttering equivalent to the original one.

Invariance under stuttering

Let ϕ be an LTL formula. We call ϕ invariant under stuttering iff for all stuttering-equivalent pairs of sequences σ and ρ:

$$\sigma \in [\![\phi]\!] \quad \text{iff} \quad \rho \in [\![\phi]\!].$$

Put differently: ϕ cannot distinguish stuttering-equivalent sequences.
(And neither stuttering-equivalent Kripke structures.)

Theorem: Any LTL formula that does not contain an $\mathbf{X}$ operator is invariant under stuttering.
Proof: Exercise.

Strategy

Assume ϕ does not contain any $\mathbf{X}$.

We replace $\mathcal{K}$ by a stuttering-equivalent, smaller structure $\mathcal{K}'$.

Then we check whether $\mathcal{K}' \models \phi$, which is equivalent to $\mathcal{K} \models \phi$ (since ϕ is invariant under stuttering).

We generate $\mathcal{K}'$ by performing a DFS on $\mathcal{K}$, and in each step eliminating certain successor states, based on the knowledge about properties of actions that is imparted by *I* and *U*.

The method presented here is called the ample set method.

Ample sets

For every state s we compute a set $red(s) \subseteq en(s)$; $red(s)$ contains the actions whose corresponding successor states will be explored.

(partially conflicting) goals:

- $red(s)$ must be chosen in such a way that stuttering equivalence is guaranteed.
- The choice of $red(s)$ should reduce $\mathcal{K}$ strongly.
- The computation of $red(s)$ should be efficient.

Conditions for Ample Sets

C0: $red(s) = \emptyset$ iff $en(s) = \emptyset$

C1: Every path of $\mathcal{K}$ starting at a state s satisfies the following: an action dependent on some action in $red(s)$ cannot be executed without an action from $red(s)$ occurring first.

C2: If $red(s) \neq en(s)$ then all actions in $red(s)$ are invisible.

C3: For all cycles in $\mathcal{K}'$ the following holds: if $a \in en(s)$ for some state s in the cycle, then $a \in red(s')$ for some (possibly other) state s' in the cycle.

Idea

C0 ensures that no additional deadlocks are introduced.

C1 and C2 ensure that every stuttering-equivalence class of runs is preserved.

C3 ensures that enabled actions cannot be omitted forever.

Example

Pseudocode program with two concurrent processes:

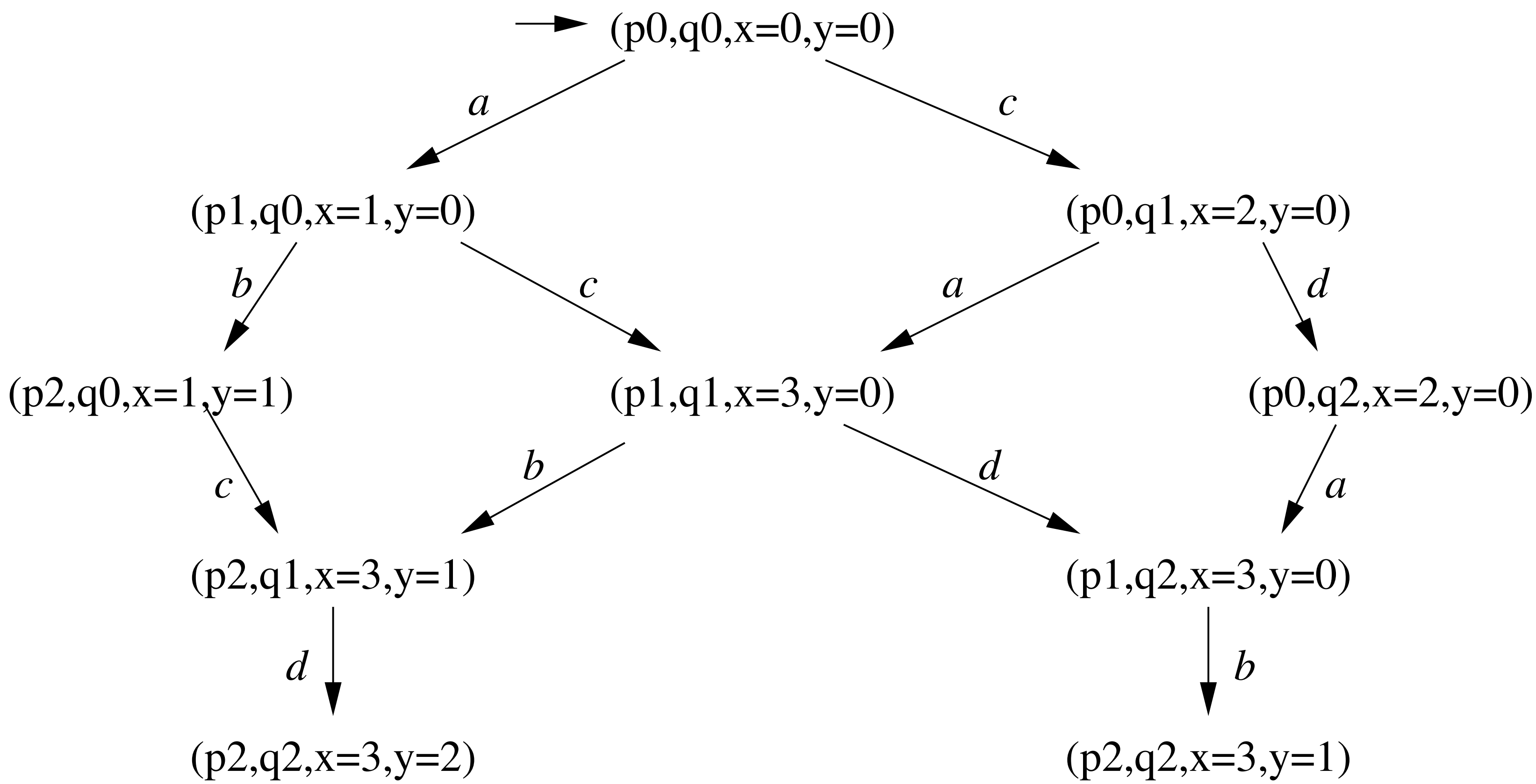
int *x*,*y* **init** 0;
cobegin $\{P \parallel Q\}$ **coend**

$P =$ p0: $x := x + 1$; (action *a*)
p1: $y := y + 1$; (action *b*)
p2: **end**

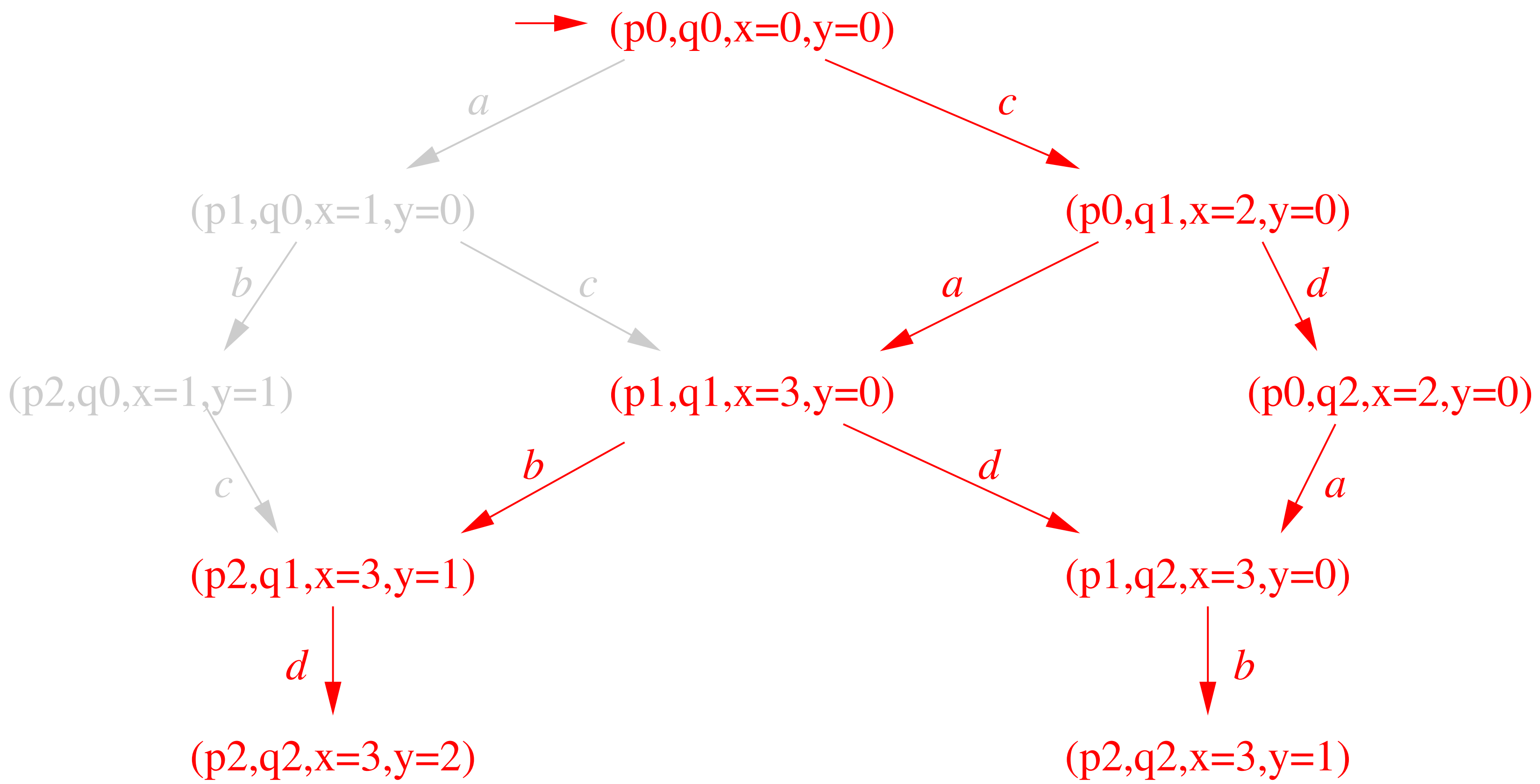
$Q =$ q0: $x := x + 2$; (action *c*)
q1: $y := y * 2$; (action *d*)
q2: **end**

Independent actions: all pairs but (*b*,*d*) and (*d*,*b*).

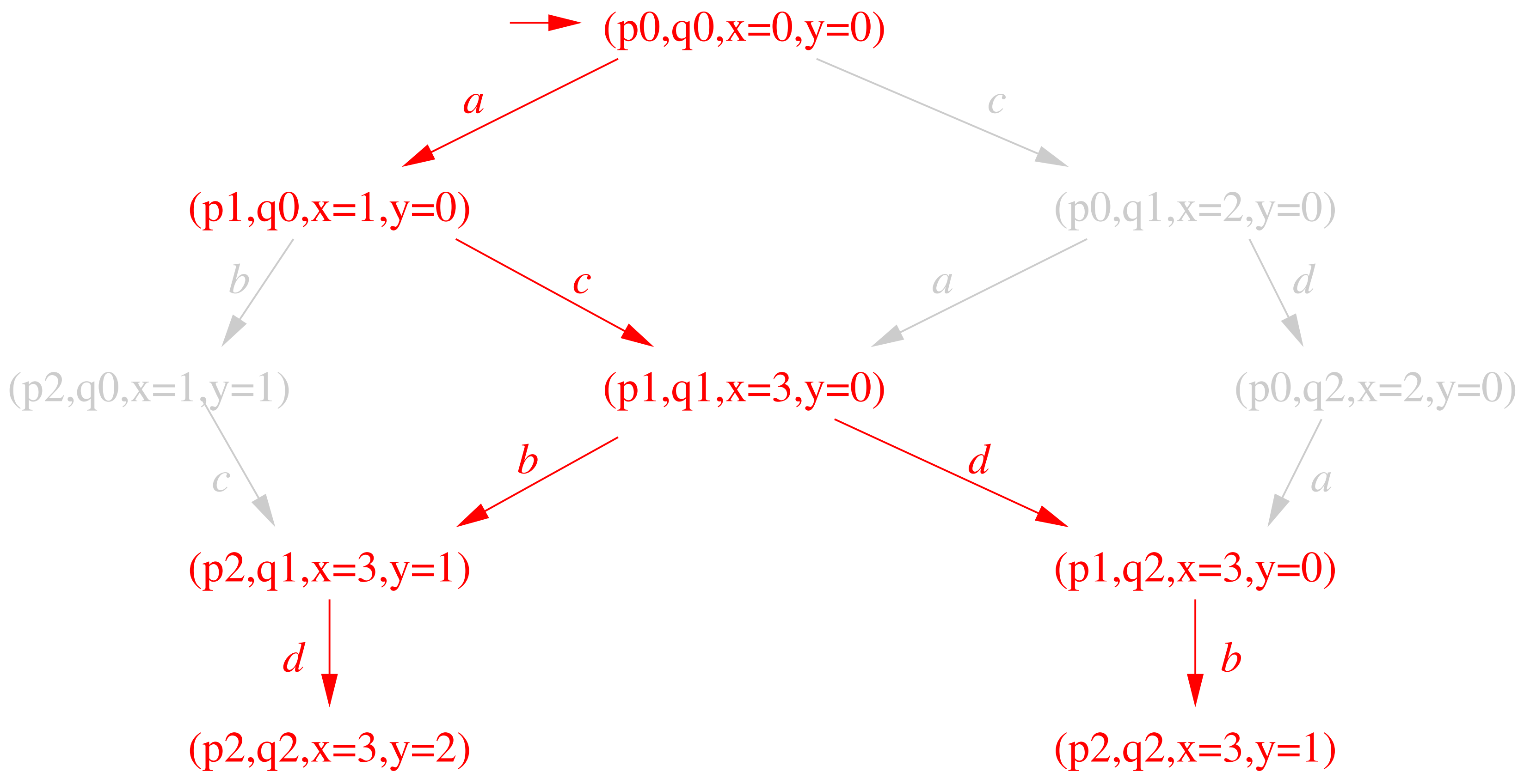
Transition system of the example:



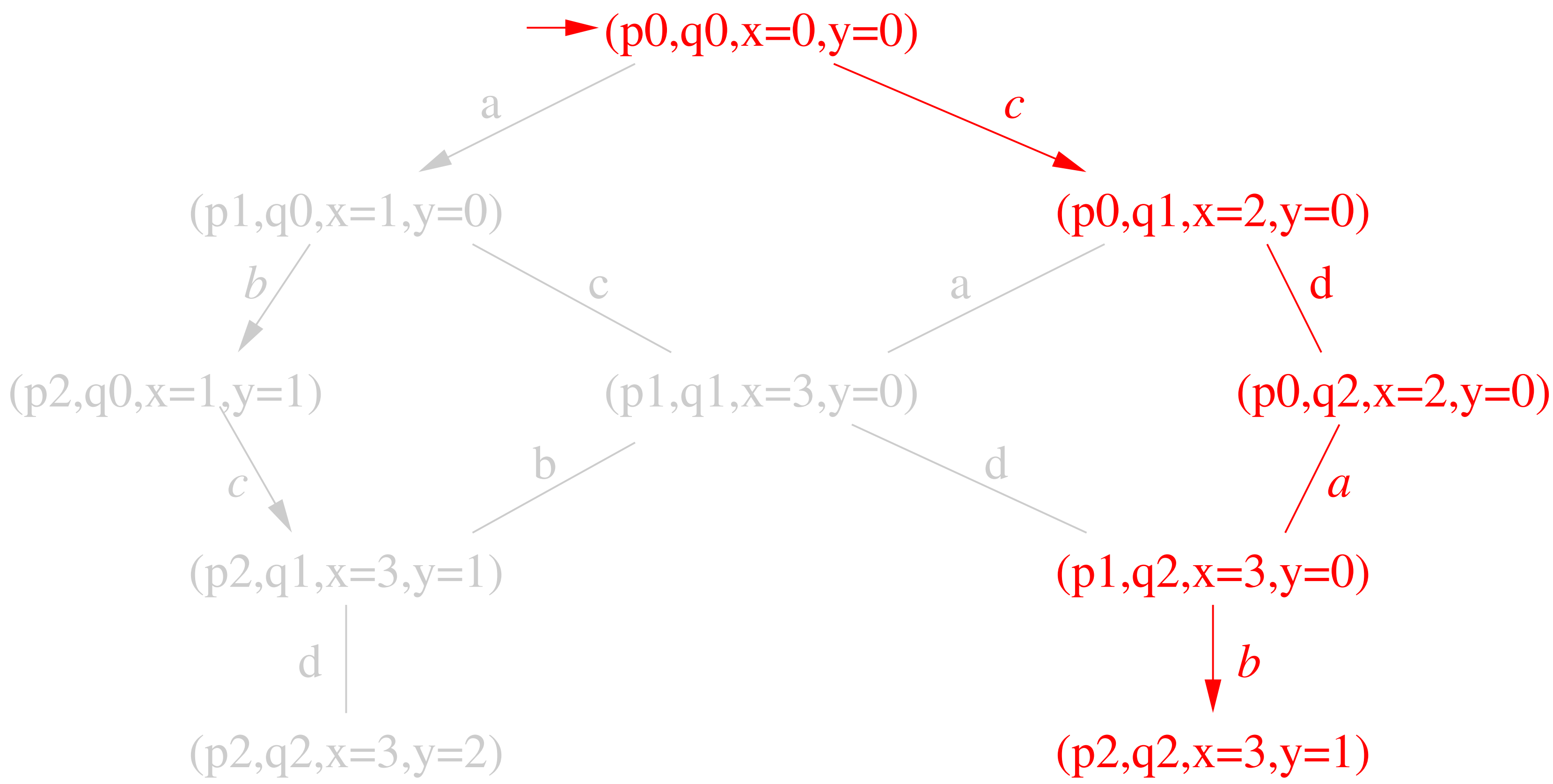
Possible reduced structure if *b*, *d* are visible:



Possible reduced structure if at most *d* is visible:



Can we do even better if no action is visible?



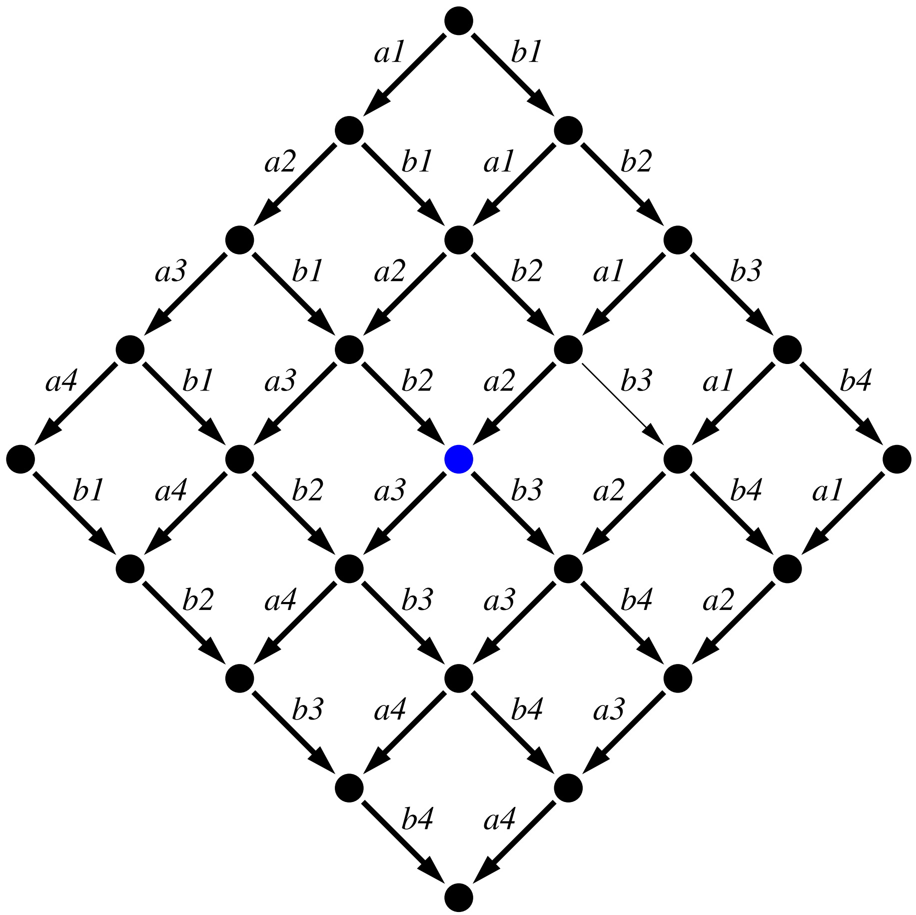
(Non-)Optimality

An ideal reduction would retain only one execution from each stuttering equivalence class.

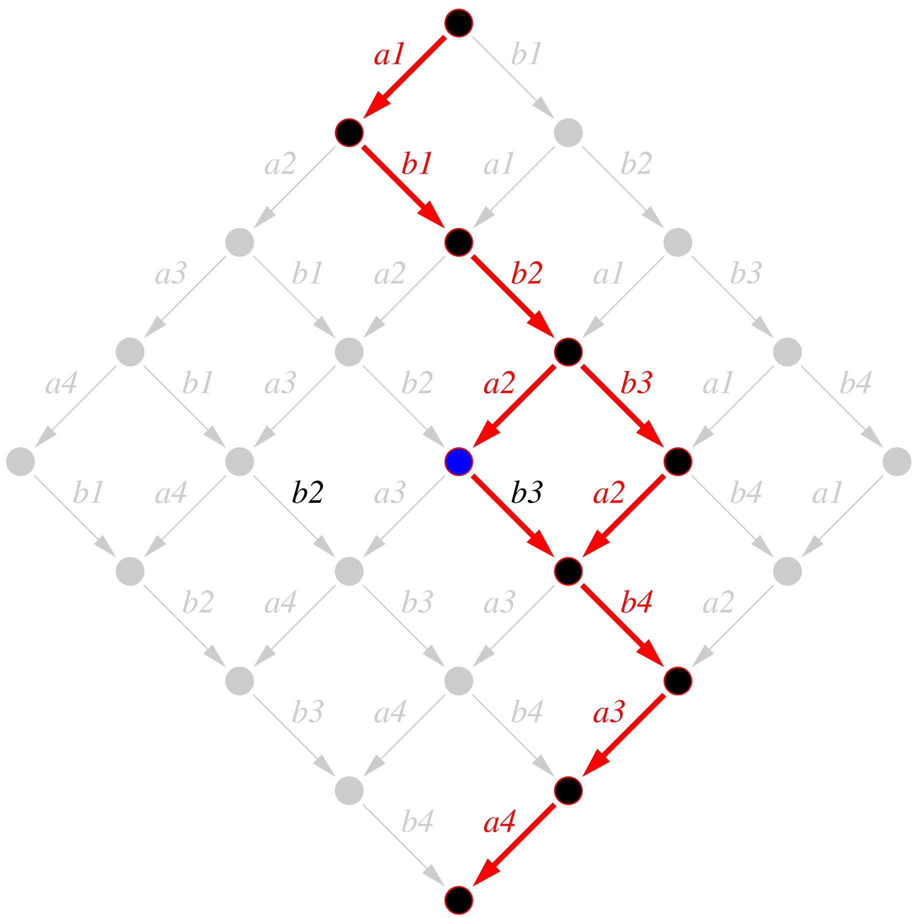
C0–C3 do *not* ensure such an ideal reduction, i.e. the resulting reduced structure is not minimal in general.

Example (see next slide): two parallel processes with four actions each ($a_1, \ldots, a_4$ or $b_1, \ldots, b_4$, resp.), all independent.

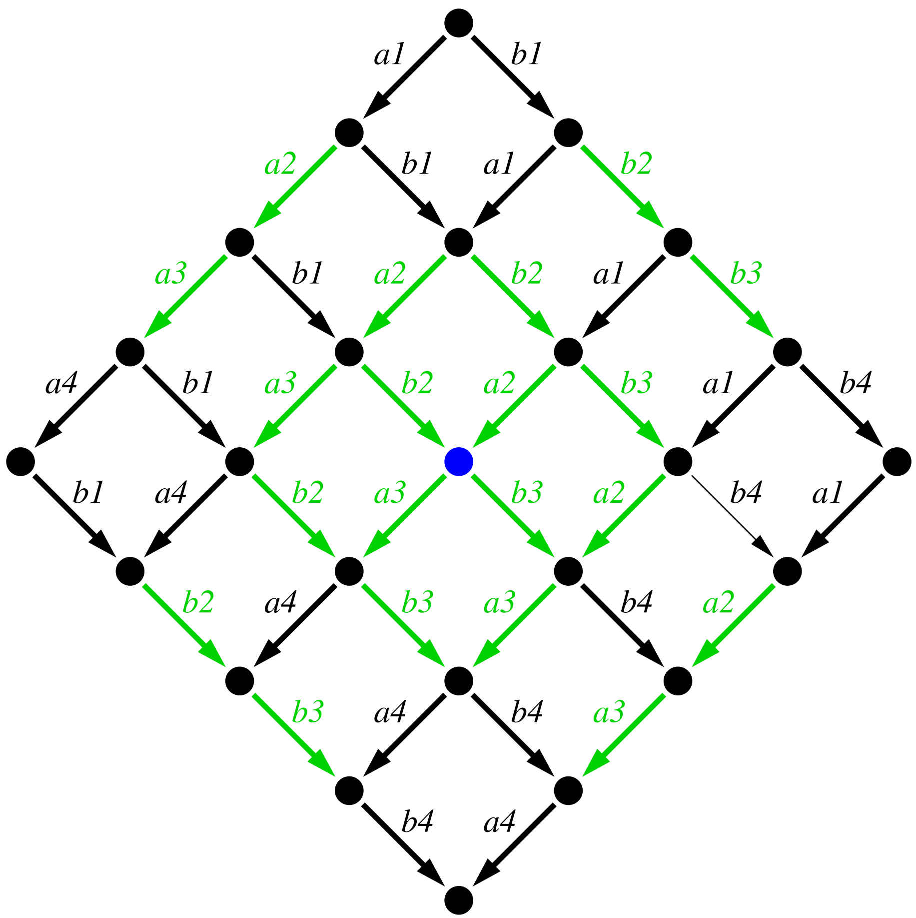
The valuation of the blue state differs from the others:



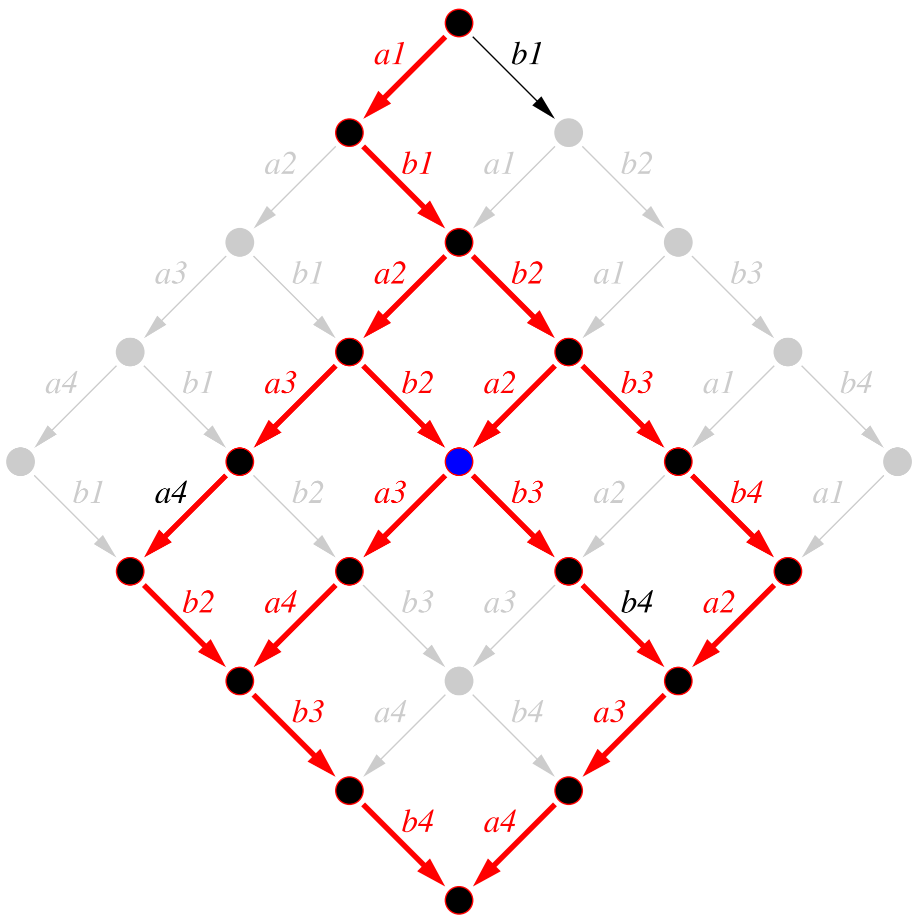
Minimal stuttering-equivalent structure:



Visible actions: a_2, a_3, b_2, b_3 (in green):



Smallest structure satisfying C0–C3:

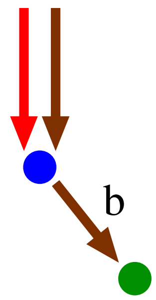


Correctness

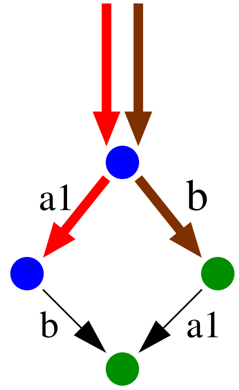
Claim: If red satisfies conditions C0 through C3, then $\mathcal{K}'$ is stuttering-equivalent to $\mathcal{K}$.

Proof (idea): Let σ be an infinite path in $\mathcal{K}$. We show that in $\mathcal{K}'$ there exists an infinite path τ such that $\nu(\sigma)$ and $\nu(\tau)$ are stuttering-equivalent.

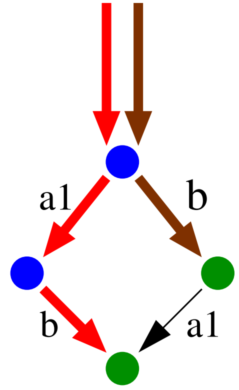
In the following, σ is shown in brown and τ in red. States known to fulfil the same atomic propositions are drawn in the same colours.



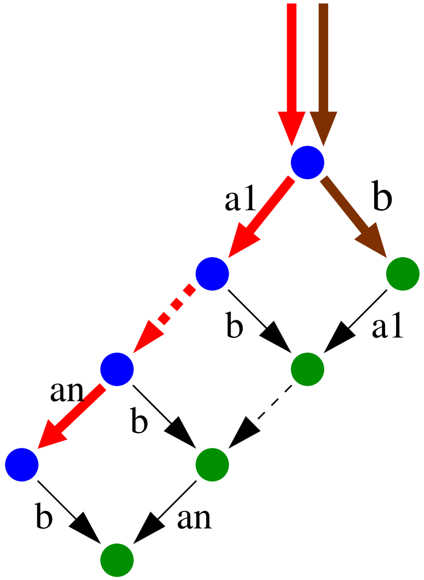
Suppose that the transition labelled with b is the first in σ that is not present in $\mathcal{K}'$.



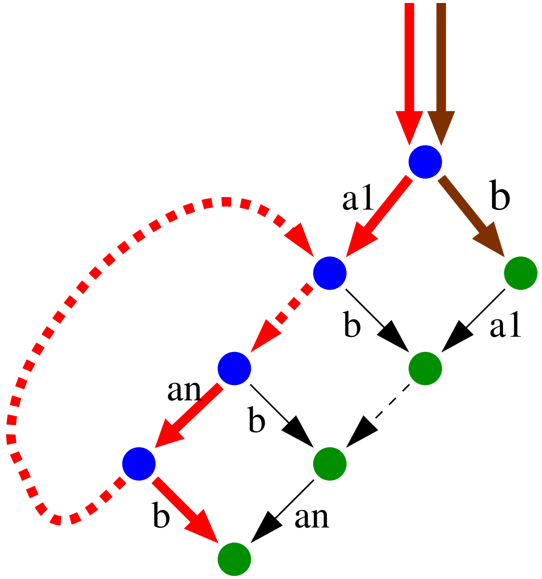
Because of C0 the blue state must have another enabled action, let us call it a_1. a_1 is independent of b (C1) and invisible (C2).



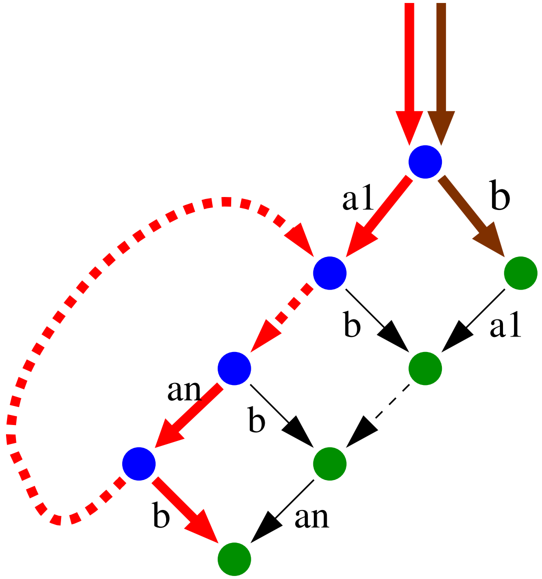
Either the second b-transition is in $\mathcal{K}'$, then we take τ to be the sequence of red edges. . .



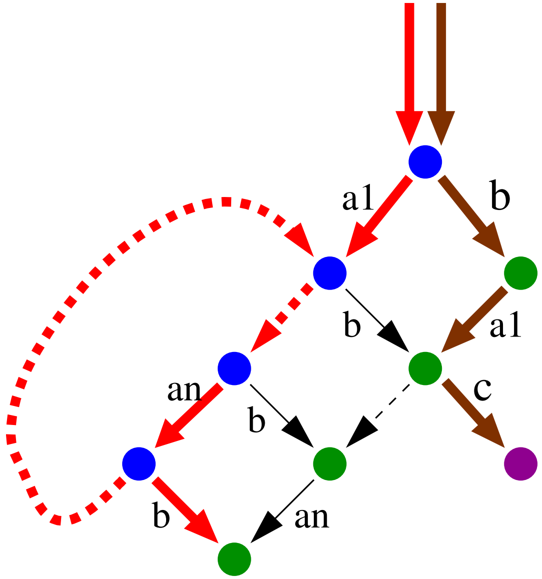
...or *b* will be "deferred" in favour of $a_2, \ldots, a_n$, all of which are also invisible and independent of *b*.



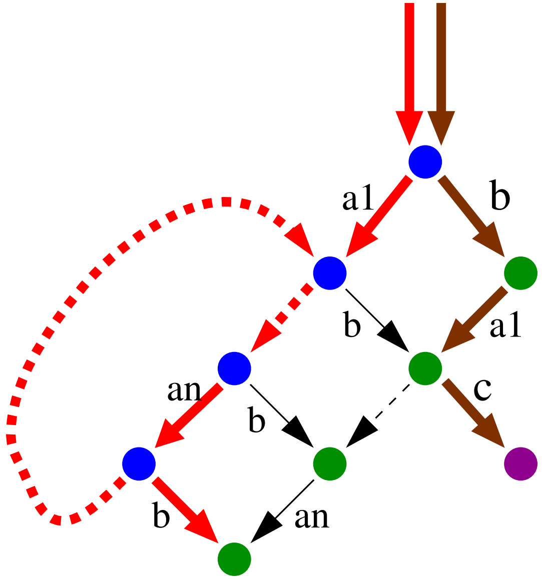
Since $\mathcal{K}$ is finite, this process must either end or create a cycle (in $\mathcal{K}'$). Because of C3, b must be activated in some state along the cycle.



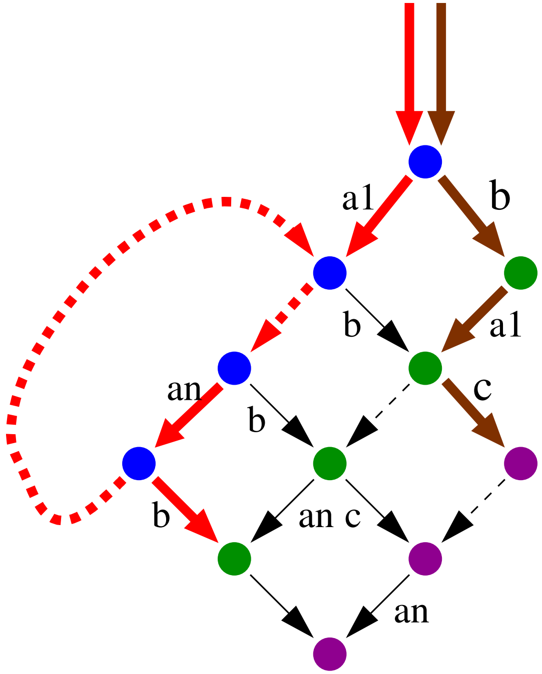
Both σ and τ contain blue states followed by green ones.



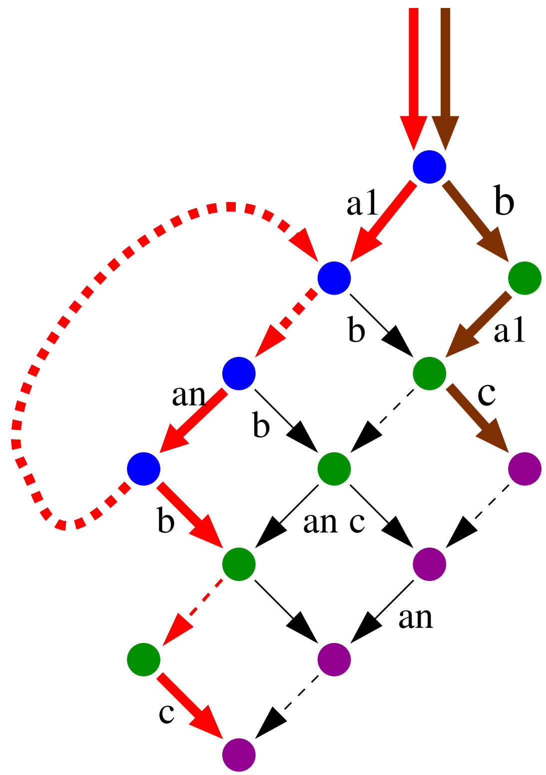
σ either continues with $a_1, \ldots, a_n$ until the paths “converge”, or it “diverges” again with an action c.



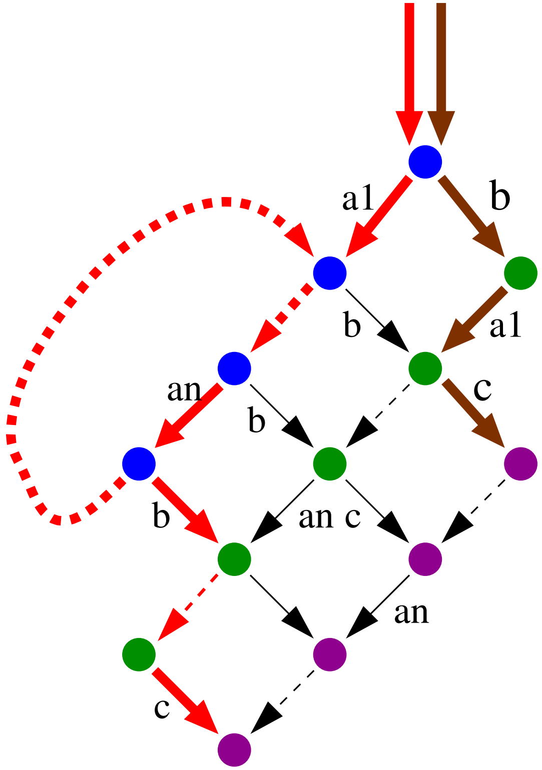
Then, however, c must be independent from a_2 by C1.



By the independence, c is enabled in the next green state and again independent of a_3 and, by induction, independent of all $a_2 \ldots, a_n$.



Repeating the previous arguments, we can conclude that $\mathcal{K}'$ also has a c-labelled transition along the red path.



Both the red and the brown path again contains blue, green, and purple states, in that order. The previous arguments can be repeated ad infinitum.

The approach in Spin

Implementing $red(s)$ depends on the underlying description of the system; here, we will discuss what Spin is doing for Promela models.

In general, a Promela model will contain multiple concurrent processes $P_1, \ldots, P_n$ communicating via global variables and message channels.

Let $E_i(s)$ denotes the actions of process P_i activated in s.

Spin tests the sets $E_i(s)$, for $i = 1, \ldots, n$, as candidates for $red(s)$. If all of them fail, it "gives up" and takes $red(s) = en(s)$.

Checking conditions: Implementation issues

C0 and C2: obvious

C1 and C3 depend on the complete state graph

We shall find conditions that are stronger than C1 and C3. These will exclude certain reductions but can be efficiently implemented during DFS.

Replace C3 by C3':

> If $red(s) \neq en(s)$ for some state s, then no action of $red(s)$ may lead to a state that is currently on the search stack of the DFS.

Heuristic for C1 in Spin

Recall: $E_i(s)$ satisfies C_1 if no action a that depends on $E_i(s)$ may be executed before an action from $E_i(s)$ itself.

A sufficient condition for $E_i(s)$ to satisfy $C1$ is the conjunction of

(i) No action of other processes depends on $E_i(s)$
(so a cannot be an action from other process)

(ii) No action of P_i outside $E_i(s)$ can become activated by an action of another process.
(so a cannot be an action of P_i either)

Heuristic for C1 in Spin: Dependent actions

$Dep(a) := \{ b \mid (a, b) \notin I \}$ contains the actions dependent on a.

In Spin, if a is an action of P_i, then $Dep(a)$ will be overapproximated by:

- all other actions in P_i;
- actions in other processes that write to a variable from which a reads, or vice versa;
- if a reads from a message channel, then all actions in other processes that read from the same channel.
- if a writes to a message channel, then all actions in other processes that write to the same channel.

Heuristic for C1 in Spin: Activation

Let A_i denote the possible actions in process P_i.

Let $Pc_i(s)$ denote the actions possible in P_i at label $pc_i(s)$.

Observe that some actions of $Pc_i(s)$ may not be activated in s itself!

- Their guards may not be enabled at s, but may become enabled due to an action from another process.

Heuristic for C1 in Spin: Activation

Let $Pre(a)$ be the set of actions that might activate a, i.e. (some overapproximation of) the set

$$\{\, b \mid \exists s\colon a \notin en(s),\ b \in en(s),\ a \in en(b(s)) \,\}$$

In Spin: if a is an action of P_i then we can choose $Pre(a)$ as the set containing:

- all actions of P_i leading to a control-flow label in which a can be executed;

- if the guard of a uses global variables, all actions in other processes that modify these variables;

- if a reads from a message channel or writes into it, then all actions in other processes that do the opposite (write/read).

Test for C1 in Spin

function $\mathit{check\_C1}(s, P_i)$

for all $P_j \neq P_i$ **do**

if $\mathit{Dep}(E_i(s)) \cap A_j \neq \emptyset$

```
\* some action of P_j     *\
\* is dependent of E_i(s) *\
```

or $\mathit{Pre}(\mathit{Pc}_i(s) \setminus E_i(s)) \cap A_j \neq \emptyset$

```
\* some action of P_i outside E_i(s) can *\
\* become activated by an action of P_j  *\
```

then return False;

return True;

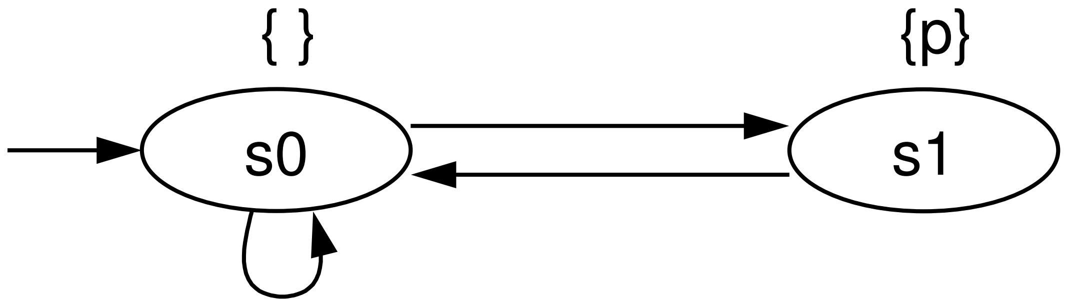
end function

Part 8: Branching-time logics

Motivation

Linear-time logic describes properties of runs. However, it cannot describe the *options* that are possible in a system.

Example: The behaviour of the structure below cannot be adequately described using LTL. At any time, the system *may* change to a state satisfying p, however, it may never do so.



Branching-time logics allow to speak about the branching behaviour, i.e. about multiple possible futures. Hence, branching-time logics are evaluated over *trees* of valuations.

We shall first extend LTL into a branching-time logic called CTL*. CTL* is quite powerful, however, the associated model-checking problem is correspondingly hard.

We then introduce a restriction of CTL*, called CTL. CTL is probably the most popular branching-time logic in verification because

- it is still expressive enough for most applications;
- it has useful algorithmic properties;
- correspondingly, it is often used in automated verification.

Valuation trees

Let $\mathcal{T} = (V, \rightarrow, r, AP, \nu)$ be a Kripke structure (where V may be an infinite set). We call $\mathcal{T}$ a valuation tree iff

> $(V, \rightarrow)$ is a directed tree with root r (i.e. for every node $v \in V$ there is exactly one path from r to v).

$\lceil v \rceil$ denotes the subtree whose root is $v \in V$.

Computation tree

Let $\mathcal{K} = (S, \rightarrow, s_0, AP, \nu)$ be a Kripke structure and $s \in S$.

By $\mathcal{T}_{\mathcal{K}}(s)$ we denote the (unique) valuation tree with root r and the following properties:

$\nu(r) = \nu(s)$

$s \rightarrow s'$ holds in $\mathcal{K}$ iff in $\mathcal{T}_{\mathcal{K}}(s)$ there is a transition $r \rightarrow r'$ such that $\lceil r' \rceil$ is isomorphic to $\mathcal{T}_{\mathcal{K}}(s')$.

We call $\mathcal{T}_{\mathcal{K}}(s)$ the computation tree of s.
(Note: $\mathcal{T}_{\mathcal{K}}(s_0)$ is also sometimes called the (acyclic) unfolding of $\mathcal{K}$.)

CTL*: Syntax

Let *AP* be a set of atomic propositions. The set of CTL* formulae over *AP* is inductively defined as follows:

- if $p \in AP$, then p is a formula;
- if ϕ_1, ϕ_2 are formulae, then $\neg\phi_1$ and $\phi_1 \vee \phi_2$ are formulae.

Moreover, let ϕ be an 'extended' LTL formula where atomic propositions are replaced by CTL* formulae. Then $\mathbf{E}\,\phi$ is a CTL* formula.

Note: We use $\mathbf{A}\,\phi$ as an abbreviation for $\neg\,\mathbf{E}\,\neg\phi$.

CTL*: Semantics

Let $\mathcal{T}$ be a valuation tree with root r and ϕ a CTL* formula. We write $\mathcal{T} \models \phi$ for "$\mathcal{T}$ satisfies ϕ."

$\mathcal{T} \models$	p	iff $p \in AP$ and $p \in \nu(r)$
$\mathcal{T} \models$	$\neg\phi$	iff $\mathcal{T} \not\models \phi$
$\mathcal{T} \models$	$\phi_1 \vee \phi_2$	iff $\mathcal{T} \models \phi_1$ or $\mathcal{T} \models \phi_2$
$\mathcal{T} \models$	$\mathbf{E}\,\phi$	iff $\mathcal{T}$ contains some infinite path σ starting at r such that $\nu(\sigma) \models \phi$. (Note: When ϕ contains $\mathbf{E}\,\phi'$ as an "atomic proposition" then we deem $\nu(\sigma)^i \models \mathbf{E}\,\phi'$ iff $\lceil\sigma(i)\rceil \models \mathbf{E}\,\phi'$.)

We say $\mathcal{K} \models \phi$ iff $\mathcal{T}_{\mathcal{K}}(s_0) \models \phi$, where $\mathcal{K}$ is a Kripke structure with initial state s_0.

A model-checking algorithm for CTL*

Let $\mathcal{K}$ be a Kripke structure. By $\mathcal{K}[s]$, where s is a state of $\mathcal{K}$ we denote the same structure as $\mathcal{K}$, but with s as initial state.

Given an *LTL* formula ϕ we require an algorithm that solves the global model-checking problem for LTL: Find the set $[\![\phi]\!]_{\mathcal{K}}$ of all states in $\mathcal{K}$ such that $s \in [\![\phi]\!]_{\mathcal{K}}$ iff $\mathcal{K}[s] \models \phi$.

Now, let ϕ be a CTL* formula. We follow these steps:

Either ϕ does not contain any $\mathbf{E}$-subformula. Then the procedure is obvious.

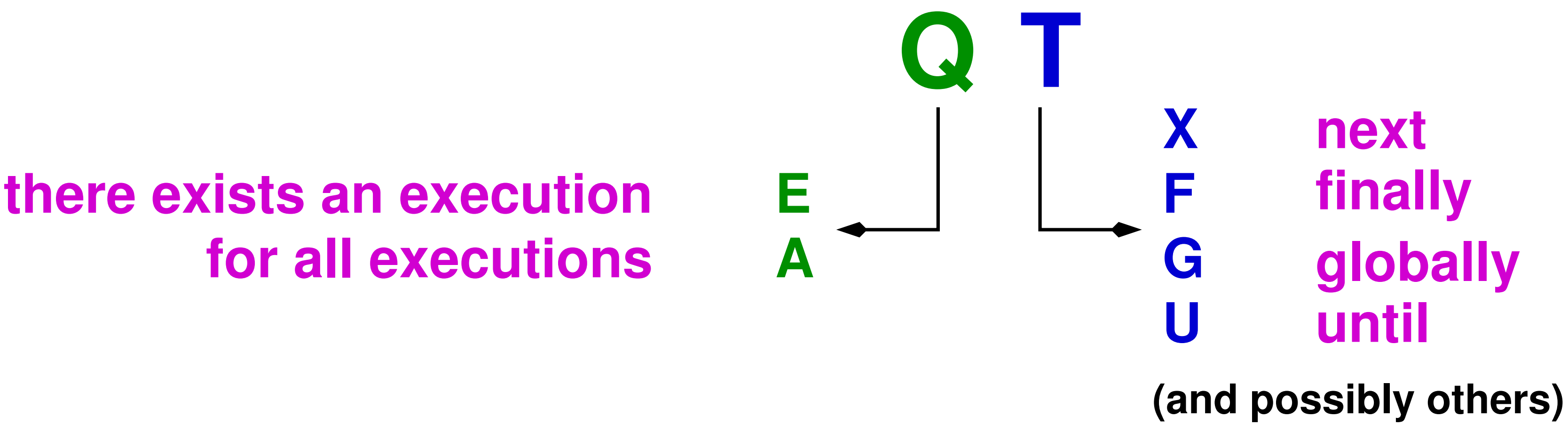
Otherwise, let $\phi' = \mathbf{E}\,\psi$ be a subformula of ϕ such that ψ does not contain another $\mathbf{E}$-subformula. We compute $M := S \setminus [\![\neg\psi]\!]_{\mathcal{K}}$. We then replace ϕ' by a "fresh" atomic proposition p and modify ν such that for all states s, we have $p \in \nu(s)$ iff $s \in M$. We then repeat the procedure until all $\mathbf{E}$s are eliminated.

Part 9: CTL

CTL: Overview

We now define CTL (Computation-Tree Logic) as a syntactic restriction of CTL*.

Operators are restricted to the following form:



CTL: Syntax

We define a minimal syntax first. Later we define additional operators with the help of the minimal syntax.

Let *AP* be a set of atomic propositions: The set of CTL formulas over *AP* is as follows:

if $a \in AP$, then a is a CTL formula;

if ϕ_1, ϕ_2 are CTL formulas, then so are

$$\neg\phi_1, \qquad \phi_1 \vee \phi_2, \qquad \mathbf{EX}\,\phi_1, \qquad \mathbf{EG}\,\phi_1, \qquad \phi_1\ \mathbf{EU}\ \phi_2$$

It is easy to see that every CTL formula is also a CTL* formula.

Previously, we defined the satisfaction relationship between valuation trees and CTL* formulae. Since each state of a Kripke structure has a clearly defined computation tree, we may just as well say that a *state* satisfies a CTL/CTL* formula, meaning that its computation tree does.

Let $\mathcal{K}$ be a Kripke structure, let s one of its states, and let ϕ be a CTL formula. On the following slide, we define a set $[\![\phi]\!]_\mathcal{K}$ in such a way that $s \in [\![\phi]\!]_\mathcal{K}$ iff $\mathcal{T}_\mathcal{K}(s) \models \phi$.

CTL: Semantics

Let $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ be a Kripke structure.

We define the semantic of every CTL formula ϕ over AP w.r.t. $\mathcal{K}$ as a set of states $[\![\phi]\!]_{\mathcal{K}}$, as follows:

$$
\begin{aligned}
[\![a]\!]_{\mathcal{K}} &= \{\, s \mid a \in \nu(s) \,\} \qquad \text{for } a \in AP \\
[\![\neg\phi_1]\!]_{\mathcal{K}} &= S \setminus [\![\phi_1]\!]_{\mathcal{K}} \\
[\![\phi_1 \vee \phi_2]\!]_{\mathcal{K}} &= [\![\phi_1]\!]_{\mathcal{K}} \cup [\![\phi_2]\!]_{\mathcal{K}} \\
[\![\mathbf{EX}\,\phi_1]\!]_{\mathcal{K}} &= \{\, s \mid \text{there is a } t \text{ s.t. } s \rightarrow t \text{ and } t \in [\![\phi_1]\!]_{\mathcal{K}} \,\} \\
[\![\mathbf{EG}\,\phi_1]\!]_{\mathcal{K}} &= \{\, s \mid \text{there is a run } \rho \text{ with } \rho(0) = s \\
&\qquad\qquad \text{and } \rho(i) \in [\![\phi_1]\!]_{\mathcal{K}} \text{ for all } i \geq 0 \,\} \\
[\![\phi_1\,\mathbf{EU}\,\phi_2]\!]_{\mathcal{K}} &= \{\, s \mid \text{there is a run } \rho \text{ with } \rho(0) = s \text{ and } k \geq 0 \text{ s.t.} \\
&\qquad\qquad \rho(i) \in [\![\phi_1]\!]_{\mathcal{K}} \text{ for all } i < k \text{ and } \rho(k) \in [\![\phi_2]\!]_{\mathcal{K}} \,\}
\end{aligned}
$$

We say that $\mathcal{K}$ satisfies ϕ (denoted $\mathcal{K} \models \phi$) iff $r \in [\![\phi]\!]_{\mathcal{K}}$.

The local model-checking problem is to check whether $\mathcal{K} \models \phi$.

The global model-checking problem is to compute $[\![\phi]\!]_{\mathcal{K}}$.

We declare two formulas equivalent (written $\phi_1 \equiv \phi_2$) iff for every Kripke structure $\mathcal{K}$ we have $[\![\phi_1]\!]_{\mathcal{K}} = [\![\phi_2]\!]_{\mathcal{K}}$.

In the following, we omit the index $\mathcal{K}$ from $[\![\cdot]\!]_{\mathcal{K}}$ if $\mathcal{K}$ is understood.

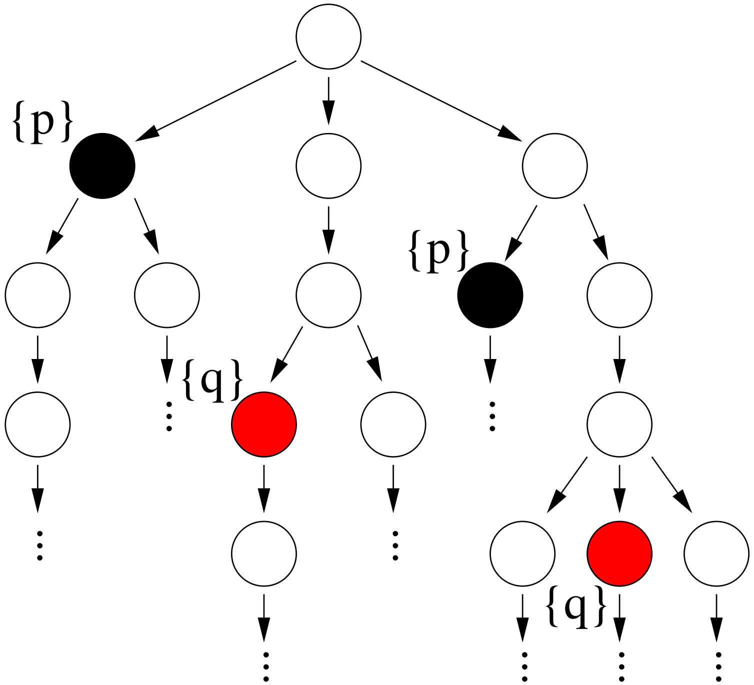
CTL: Extended syntax

$$
\begin{aligned}
\phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2) \\
\mathbf{true} &\equiv a \vee \neg a \\
\mathbf{false} &\equiv \neg\mathbf{true} \\
\phi_1\ \mathbf{EW}\ \phi_2 &\equiv \mathbf{EG}\,\phi_1 \vee (\phi_1\ \mathbf{EU}\ \phi_2) \\
\mathbf{EF}\,\phi &\equiv \mathbf{true}\,\mathbf{EU}\,\phi
\end{aligned}
\qquad
\begin{aligned}
\mathbf{AX}\,\phi &\equiv \neg\,\mathbf{EX}\,\neg\phi \\
\mathbf{AG}\,\phi &\equiv \neg\,\mathbf{EF}\,\neg\phi \\
\mathbf{AF}\,\phi &\equiv \neg\,\mathbf{EG}\,\neg\phi \\
\phi_1\ \mathbf{AW}\ \phi_2 &\equiv \neg(\neg\phi_2\,\mathbf{EU}\,\neg(\phi_1 \vee \phi_2)) \\
\phi_1\ \mathbf{AU}\ \phi_2 &\equiv \mathbf{AF}\,\phi_2 \wedge (\phi_1\ \mathbf{AW}\ \phi_2)
\end{aligned}
$$

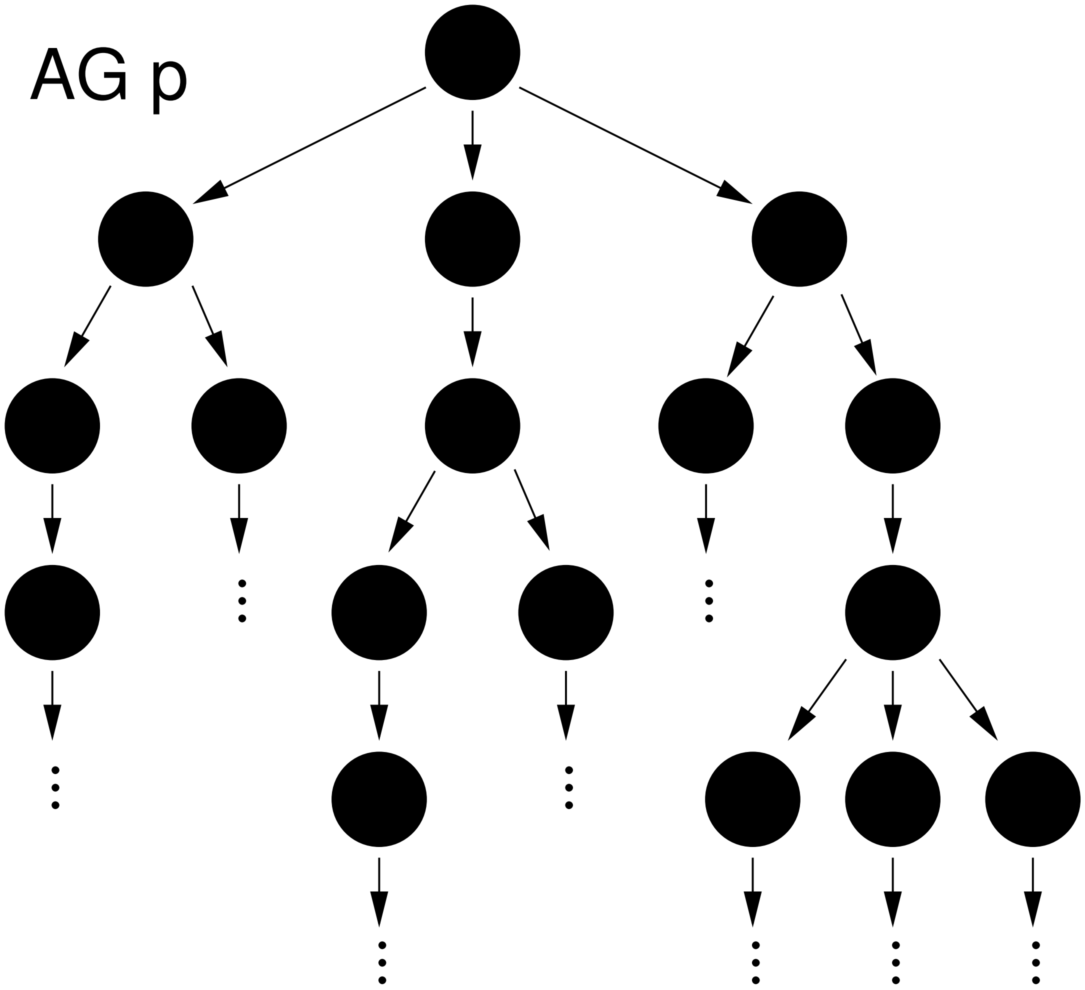
Other logical and temporal operators (e.g. $\rightarrow$, $\mathbf{ER}$, $\mathbf{AR}$), ... may also be defined.

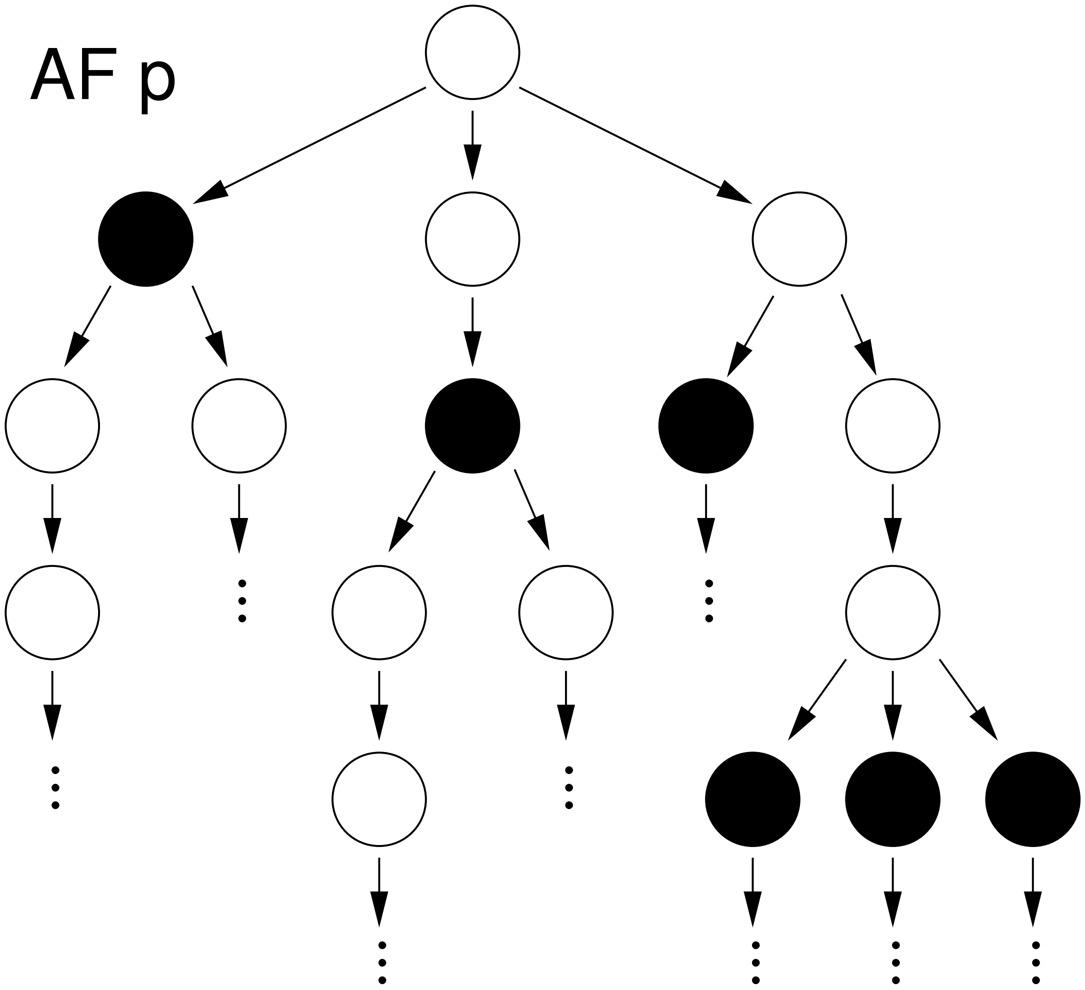
CTL: Examples

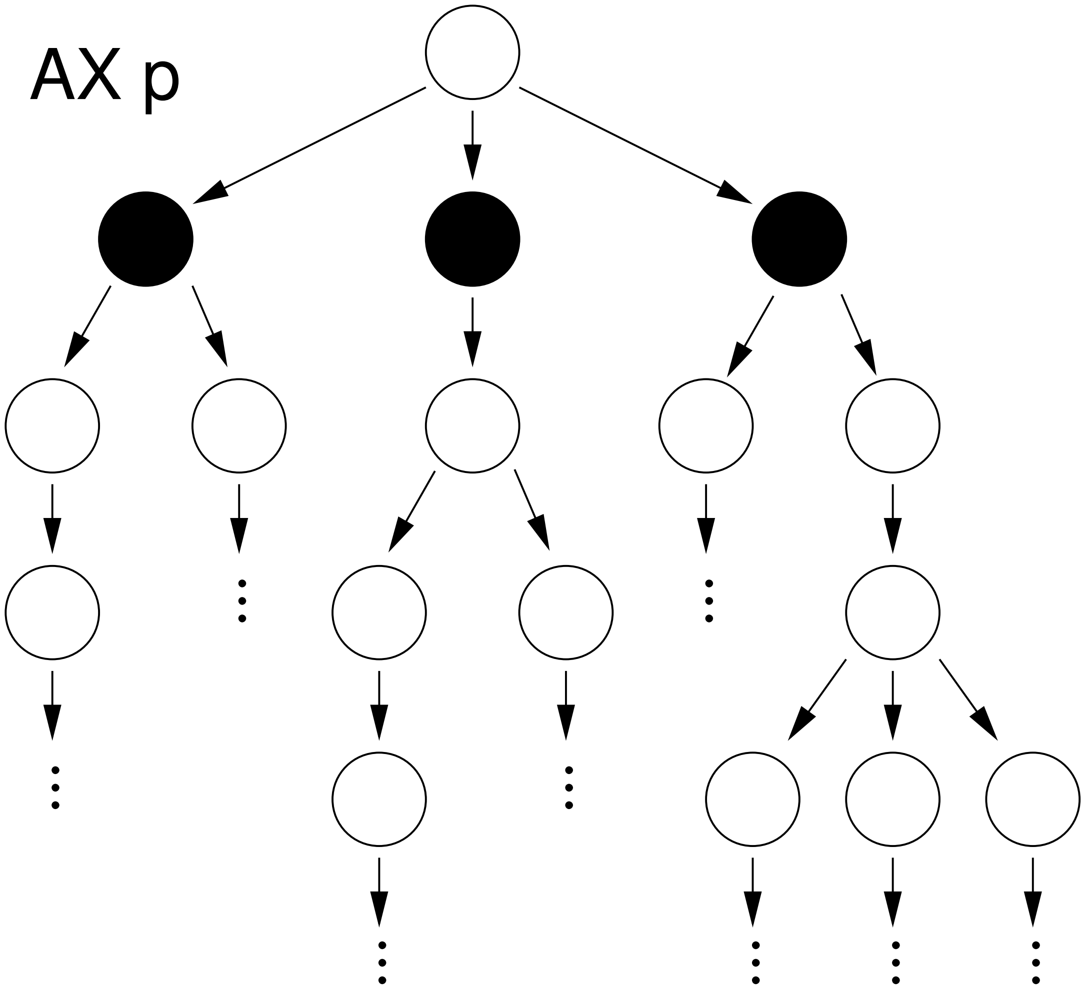
We use the following computation tree as a running example (with varying distributions of red and black states):

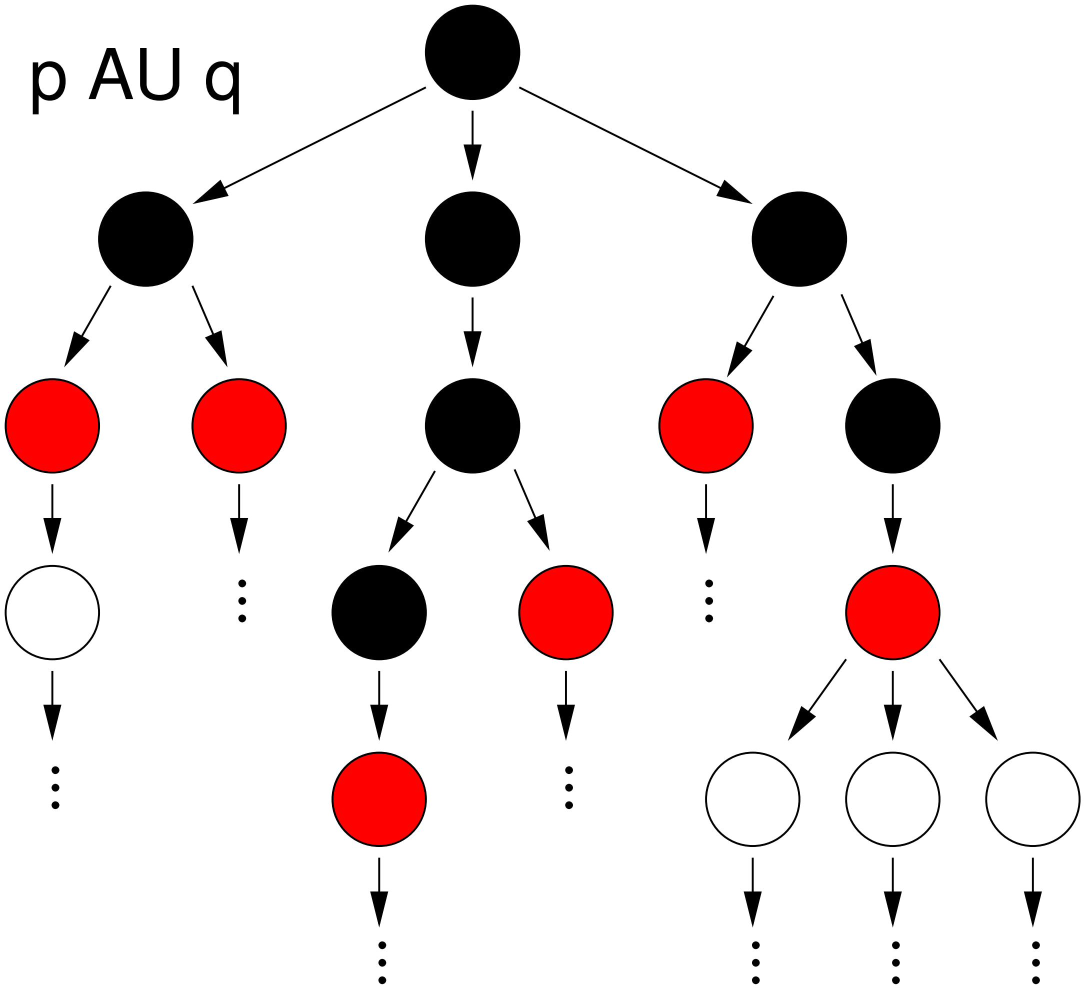


In the following slides, the topmost state satisfies the given formula if the black states satisfy p and the red states satisfy q.

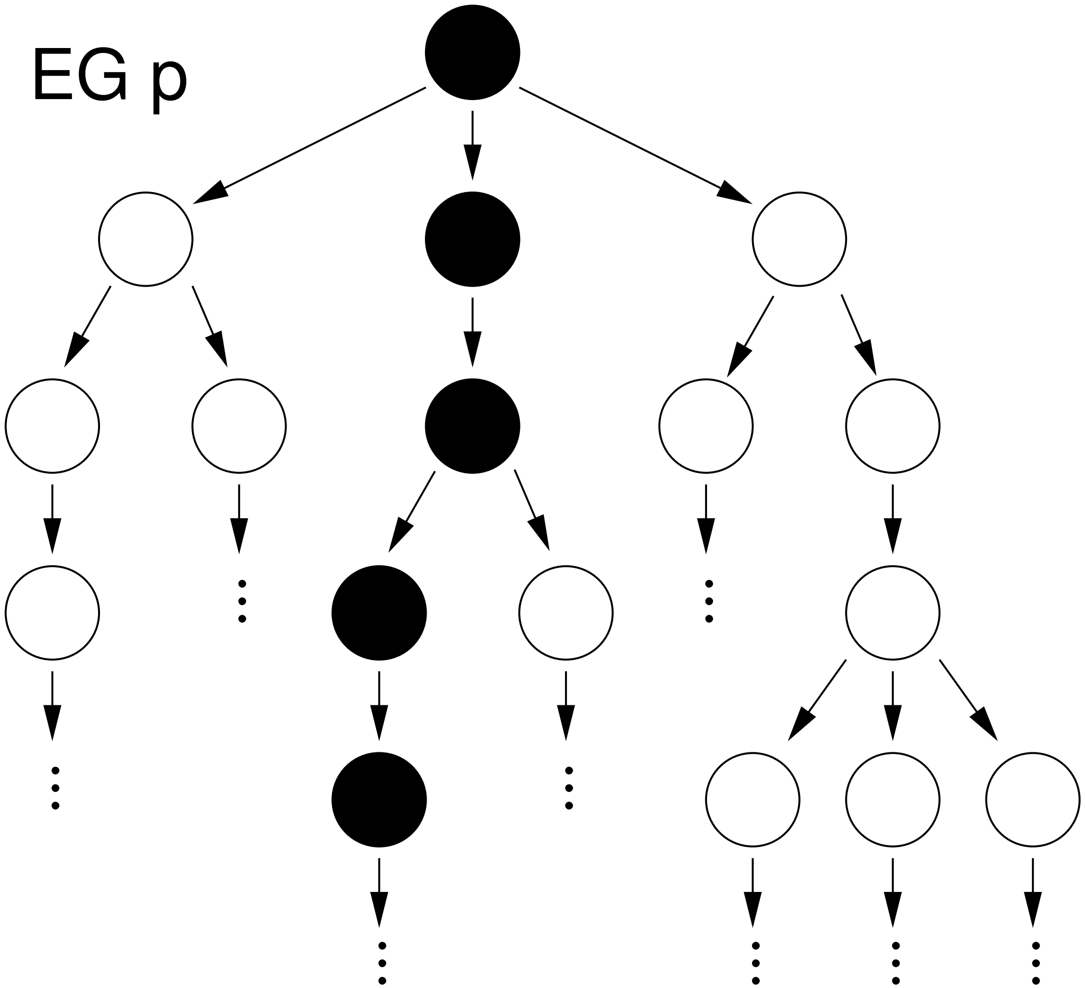

AG p

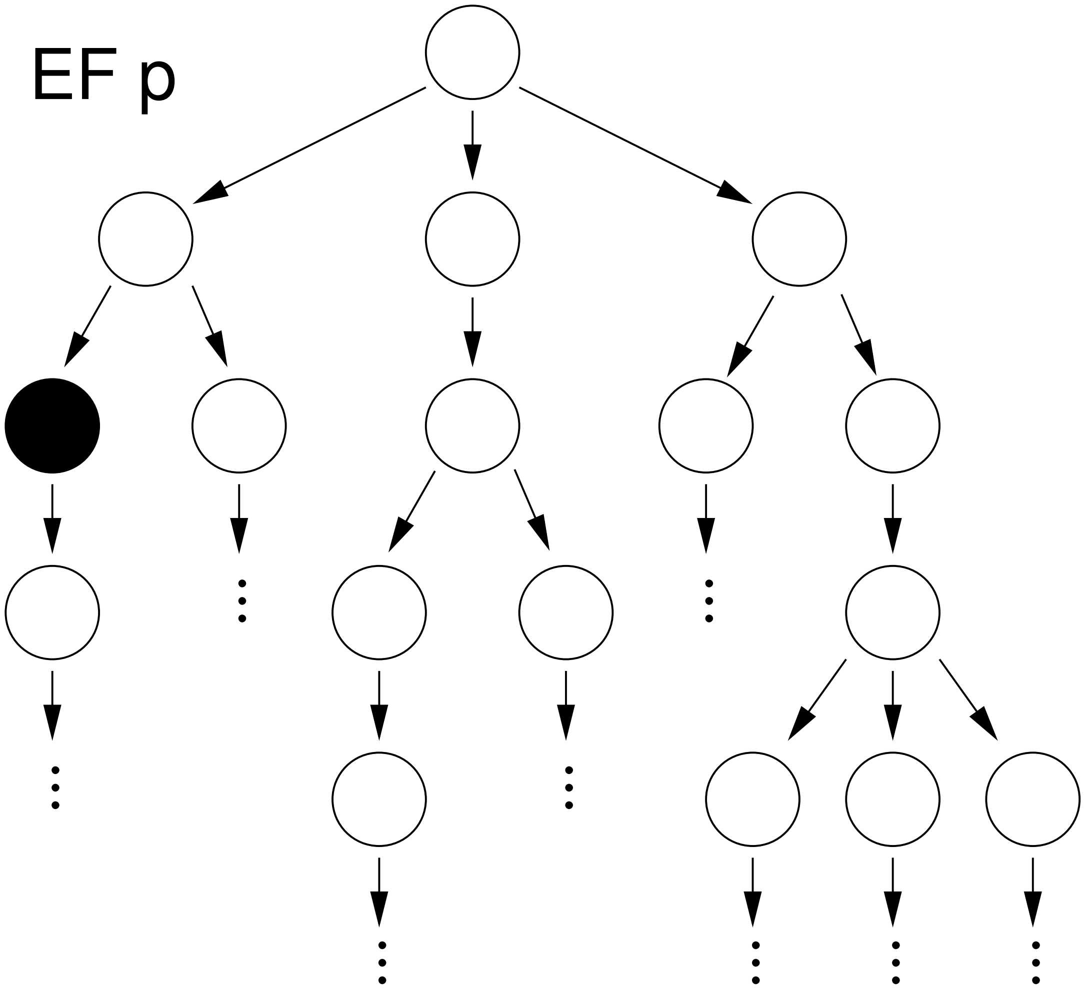

AF p

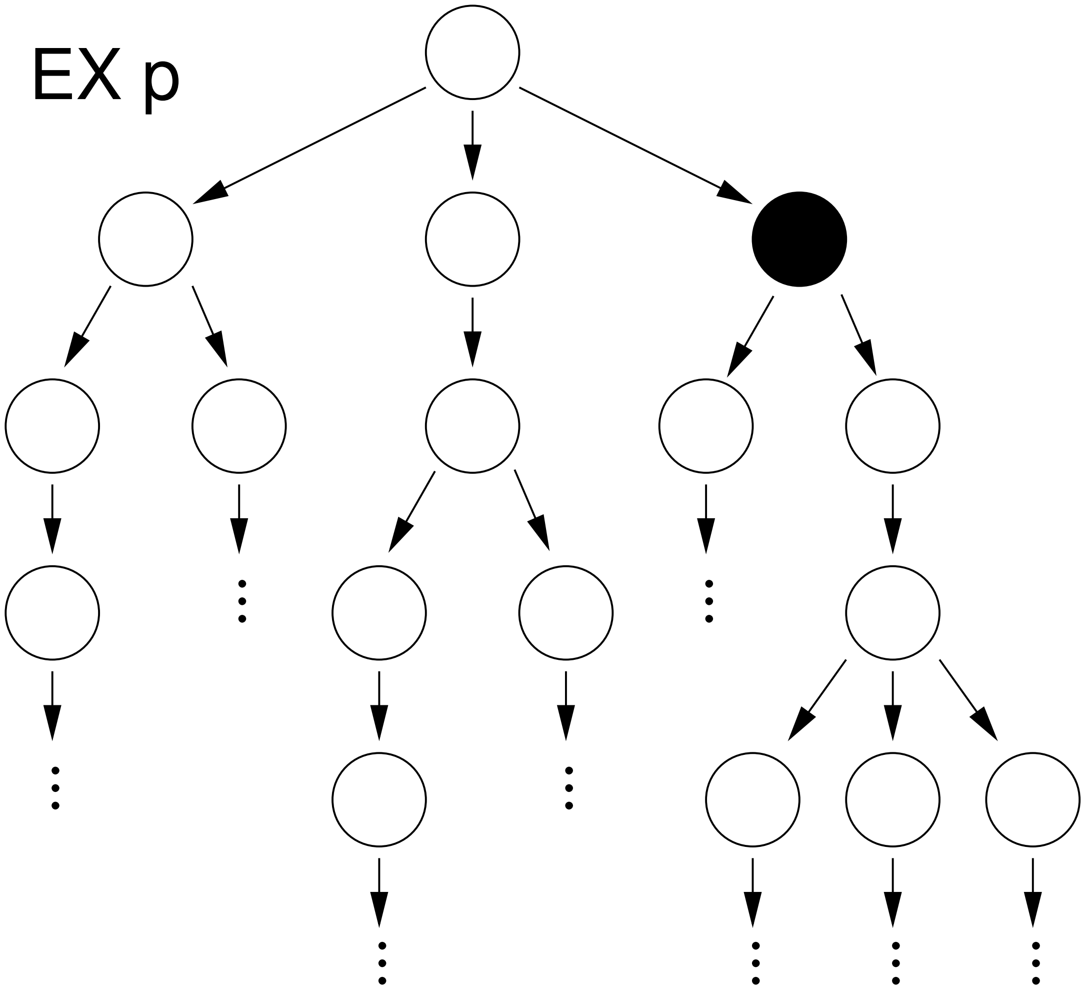

AX p

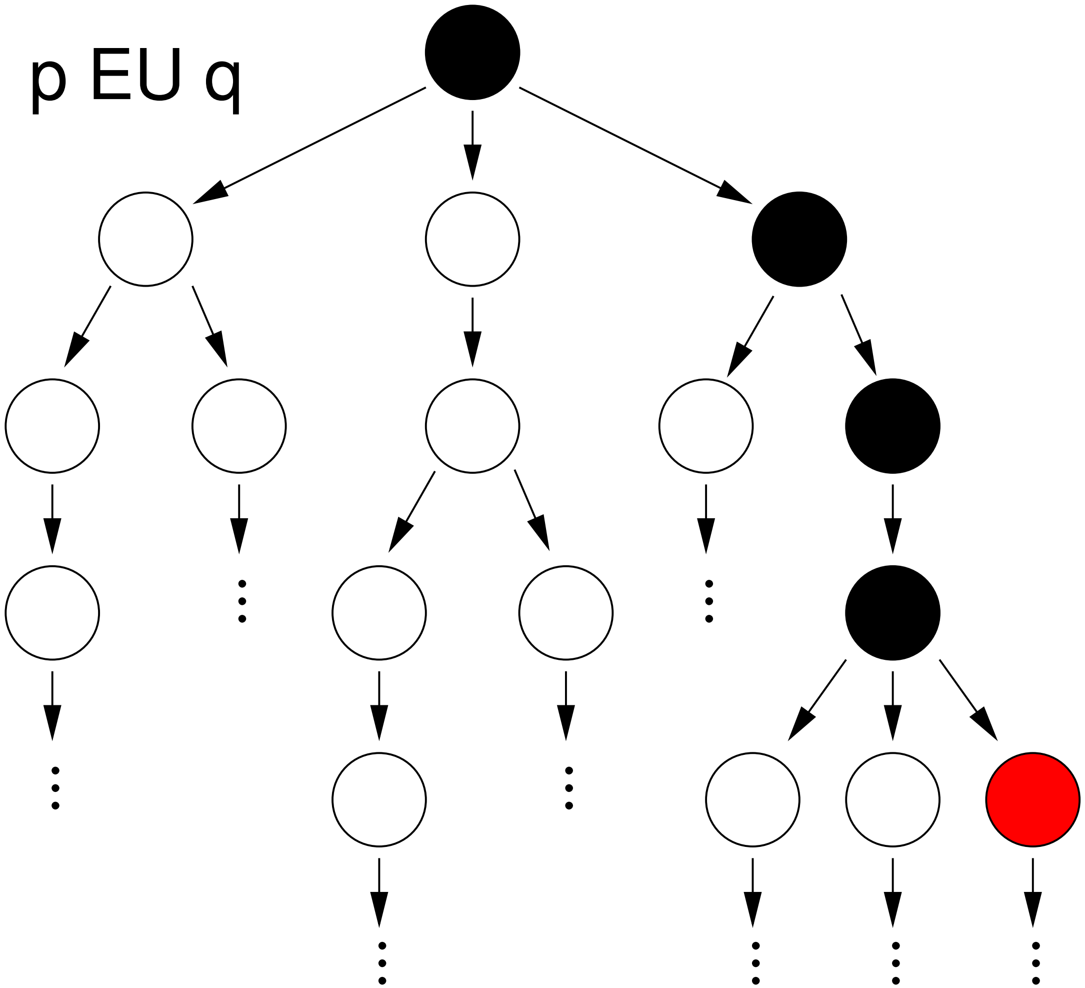

p AU q

EG p

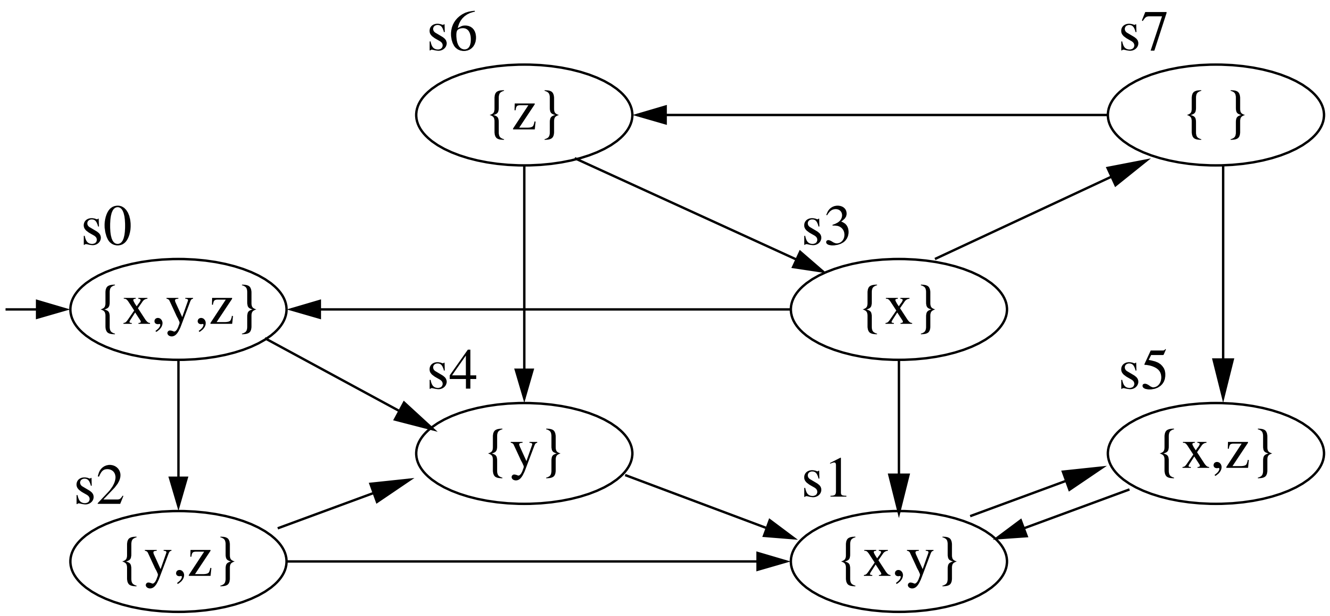



EF p


EX p


p EU q

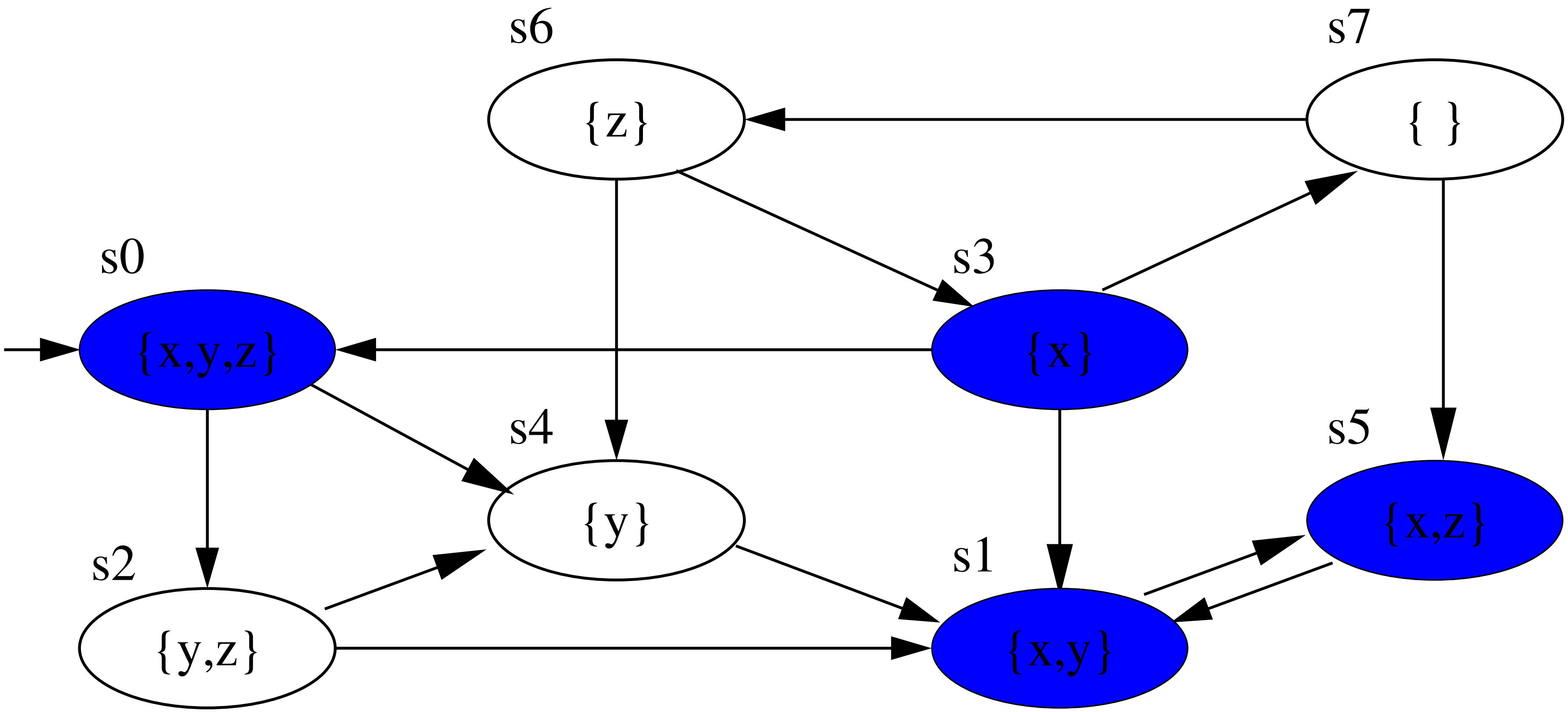
Solving nested formulas: Is $s_0 \in [\![\mathbf{AF}\ \mathbf{AG}\ x]\!]$?



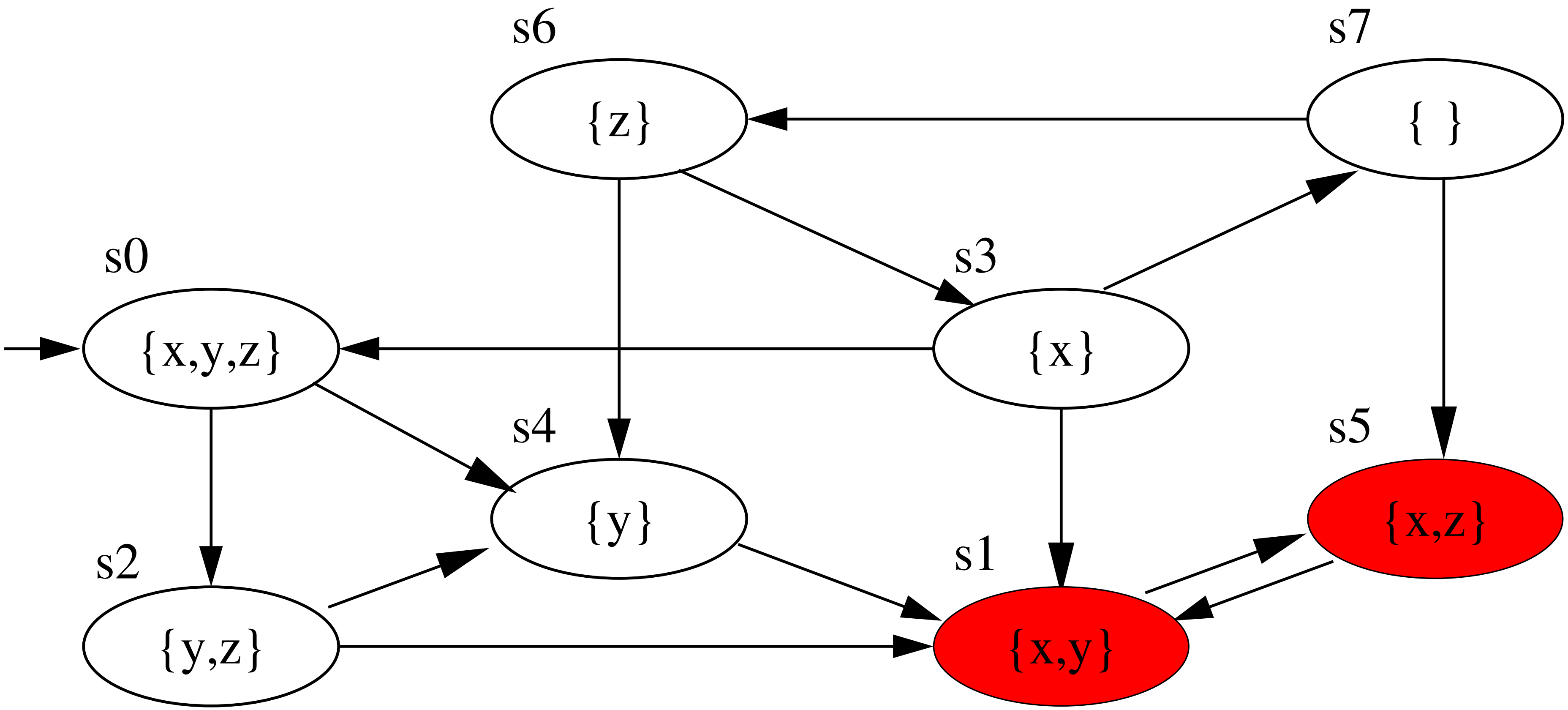
To compute the semantics of formulas with nested operators, we first compute the states satisfying the innermost formulas; then we use those results to solve progressively more complex formulas.

In this example, we compute $[\![x]\!]$, $[\![\mathbf{AG}\ x]\!]$, and $[\![\mathbf{AF}\ \mathbf{AG}\ x]\!]$, in that order.

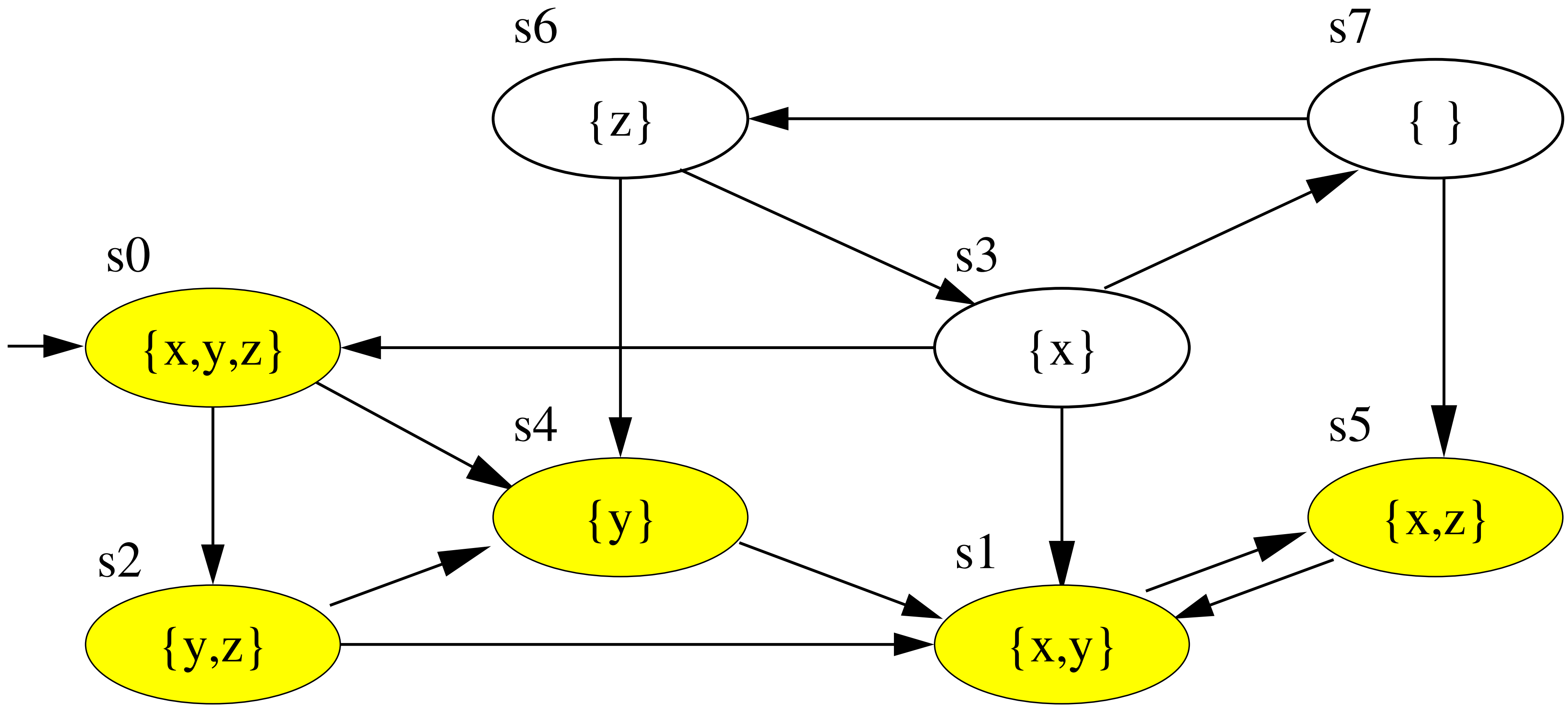
Bottom-up method (1): Compute [[x]]



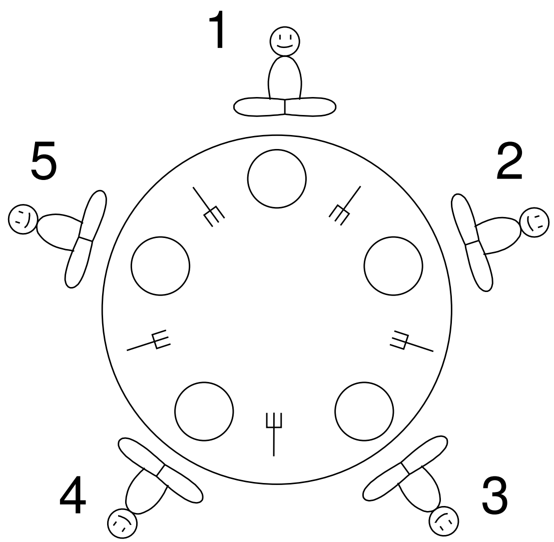
Bottom-up method (2): Compute $[\![\mathbf{AG}\ x]\!]$



Bottom-up method (3): Compute $[\![\mathbf{AF}\ \mathbf{AG}\ x]\!]$



Example: Dining Philosophers



Five philosophers are sitting around a table, taking turns at thinking and eating.

We shall express a couple of properties in CTL. Let us assume the following atomic propositions:

$e_i \ \widehat{=}\ $ philosopher i is currently eating

Properties of the Dining Philosophers

“Philosophers 1 and 4 will never eat at the same time.”

Properties of the Dining Philosophers

“Philosophers 1 and 4 will never eat at the same time.”

$$\mathbf{AG}\,\neg(e_1 \land e_4)$$

“It is possible that Philosopher 3 never eats.”

Properties of the Dining Philosophers

"Philosophers 1 and 4 will never eat at the same time."

$$\mathbf{AG}\,\neg(e_1 \wedge e_4)$$

"It is possible that Philosopher 3 never eats."

$$\mathbf{EG}\,\neg e_3$$

"From every situation on the table it is possible to reach a state where only philosopher 2 is eating."

Properties of the Dining Philosophers

"Philosophers 1 and 4 will never eat at the same time."

$$\mathbf{AG}\,\neg(e_1 \wedge e_4)$$

"It is possible that Philosopher 3 never eats."

$$\mathbf{EG}\,\neg e_3$$

"From every situation on the table it is possible to reach a state where only philosopher 2 is eating."

$$\mathbf{AG}\,\mathbf{EF}(\neg e_1 \wedge e_2 \wedge \neg e_3 \wedge \neg e_4)$$

Part 10: Algorithms for CTL

CTL-Model-Checking

In the following, let $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ be a Kripke structure (where S is finite) and ϕ a CTL formula over AP.

We shall solve the *global* model-checking problem for CTL, i.e. to compute $[\![\phi]\!]_\mathcal{K}$ (all states of $\mathcal{K}$ whose computation tree satisfies ϕ).

Our solution works “bottom-up”, i.e. it considers simple subformulae first, and then successively more complex ones.

The solution shown here considers only the minimal syntax. For additional efficiency one could extend it by treating some cases of the extended syntax more directly.

The ‘bottom-up’ algorithm for CTL

The algorithm reduces ϕ step by step to a single atomic proposition. Reminder: $[\![p]\!]_\mathcal{K} = \{\, s \mid p \in \nu(s) \,\}$ for $p \in AP$. In the following, we abbreviate this set as $\mu(p)$.

1. Check whether $\phi = p$, where $p \in AP$. If yes, output $\mu(p)$ and stop.

2. Otherwise, ϕ contains some subformula ψ of the form $\neg p$, $p \vee q$, $\mathbf{EX}\, p$, $\mathbf{EG}\, p$, or $p\, \mathbf{EU}\, q$, where $p, q \in AP$. Compute $[\![\psi]\!]_\mathcal{K}$ using the algorithms on the following slides.

3. Let $p' \notin AP$ be a “fresh” atomic proposition. Add p' to AP and set $\mu(p') := [\![\psi]\!]_\mathcal{K}$. Replace all occurrences of ψ in ϕ by p' and continue at step 1.

Computation of $[\![\psi]\!]_{\mathcal{K}}$: simple cases

Case 1: $\psi \equiv \neg p, \quad p \in AP$

By definition, $[\![\psi]\!]_{\mathcal{K}} = S \setminus \mu(p)$.

Case 2: $\psi \equiv p \vee q, \quad p, q \in AP$

Then $[\![\psi]\!]_{\mathcal{K}} = \mu(p) \cup \mu(q)$.

Case 3: $\psi \equiv \mathbf{EX}\, p, \quad p \in AP$

In the following, let $pre(X)$, for $X \subseteq S$, denote the set

$$pre(X) := \{\, s \mid \exists t \in X \colon s \to t \,\}.$$

Then by definition $[\![\psi]\!]_{\mathcal{K}} = pre(\mu(p))$.

Computation of $[\![\psi]\!]_{\mathcal{K}}$: $\mathbf{EU}$ and $\mathbf{EG}$

We shall first define $\mathbf{EU}$ and $\mathbf{EG}$ in terms of fixed points.

$\mathbf{EU}$ is characterized by a smallest fixed point: We first assume that no state satisfies the EU formula and then, one by one, identify those that do satisfy it after all.

By contrast, $\mathbf{EG}$ can be characterized by a largest fixed point: We first assume that all states satisfy a given EG formula and then, one by one, eliminate those that do not.

Based on this, we then derive algorithms for EG and EU.

Computation of $[\![\psi]\!]_{\mathcal{K}}$: $\mathbf{EG}$

Case 4: $\psi \equiv \mathbf{EG}\, p, \quad p \in AP$

Lemma 1: $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ is the largest solution (w.r.t. $\subseteq$) of the equation

$$X = \mu(p) \cap pre(X).$$

Proof: We proceed in two steps:

1. We show that $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ *is* indeed a solution of the equation, i.e.

$$[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \mu(p) \cap pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}}).$$

Reminder: $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \{\, s \mid \exists \rho\colon \rho(0) = s \wedge \forall i \geq 0\colon \rho(i) \in \mu(p) \,\}$.

"$\Rightarrow$" Let $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ and ρ a "witness" path. Then obviously $s \in \mu(p)$. Moreover, $\rho(1) \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ (because of ρ^1), hence $s \in pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}})$.

Continuation of the proof of Lemma 1:

1. "$\Leftarrow$" Let $s \in \mu(p) \cap pre([\![\mathbf{EG}\, p]\!]_{\mathcal{K}})$. Then s has a direct successor t, where a path ρ starts proving that $t \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$. Thus, $s\rho$ is a path witnessing that $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

2. We show that $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$ is indeed the *largest* solution, i.e., if M is a solution of the equation, then $M \subseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

Let $M \subseteq S$ be a solution of the equation, i.e. $M = \mu(p) \cap pre(M)$, and let $s \in M$. We shall show $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

- Since $s \in M$, we have $s \in \mu(p)$ and $s \in pre(M)$.
- Since $s \in pre(M)$, there exist $s_1 \in M$ with $s \to s_1$.
- Repeating this argument, we can construct an infinite path $\rho = ss_1 \cdots$ in which all states are contained in $\mu(p)$. Therefore, $s \in [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

Lemma 2: Consider the sequence $S,\ \pi(S),\ \pi(\pi(S)),\ \ldots$, i.e. $\left(\pi^i(S)\right)_{i\geq 0}$,
where $\pi(X) := \mu(p) \cap pre(X)$.
For all $i \geq 0$ we have $\pi^i(S) \supseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

We state the following two facts:

(1) π is *monotone*: if $X \supseteq X'$, then $\pi(X) \supseteq \pi(X')$.

(2) The sequence is *descending*: $S \supseteq \pi(S) \supseteq \pi(\pi(S)) \ldots$ (follows from (1)).

Proof of Lemma 2: (induction over *i*)
Base: $i = 0$: obvious.
Step: $i \to i + 1$:

$$\begin{aligned} \pi^{i+1}(S) &= \mu(p) \cap pre(\pi^i(S)) \\ &\supseteq \mu(p) \cap pre([\![\mathbf{EG}\, \varphi]\!]_{\mathcal{K}}) \quad \text{(i.h. and monotonicity)} \\ &= [\![\mathbf{EG}\, p]\!]_{\mathcal{K}} \end{aligned}$$

Lemma 3: There exists an index i such that $\pi^i(S) = \pi^{i+1}(S)$, and $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}} = \pi^i(S)$.

Proof: Since S is finite, the descending sequence must reach a fixed point, say after i steps. Then we have $\pi^i(S) = \pi(\pi^i(S)) = \mu(p) \cap pre(\pi^i(S))$. Therefore, $\pi^i(S)$ is a solution of the equation from Lemma (1).

Because of Lemma 1, we have $\pi^i(S) \subseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.
Because of Lemma 2, we have $\pi^i(S) \supseteq [\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$.

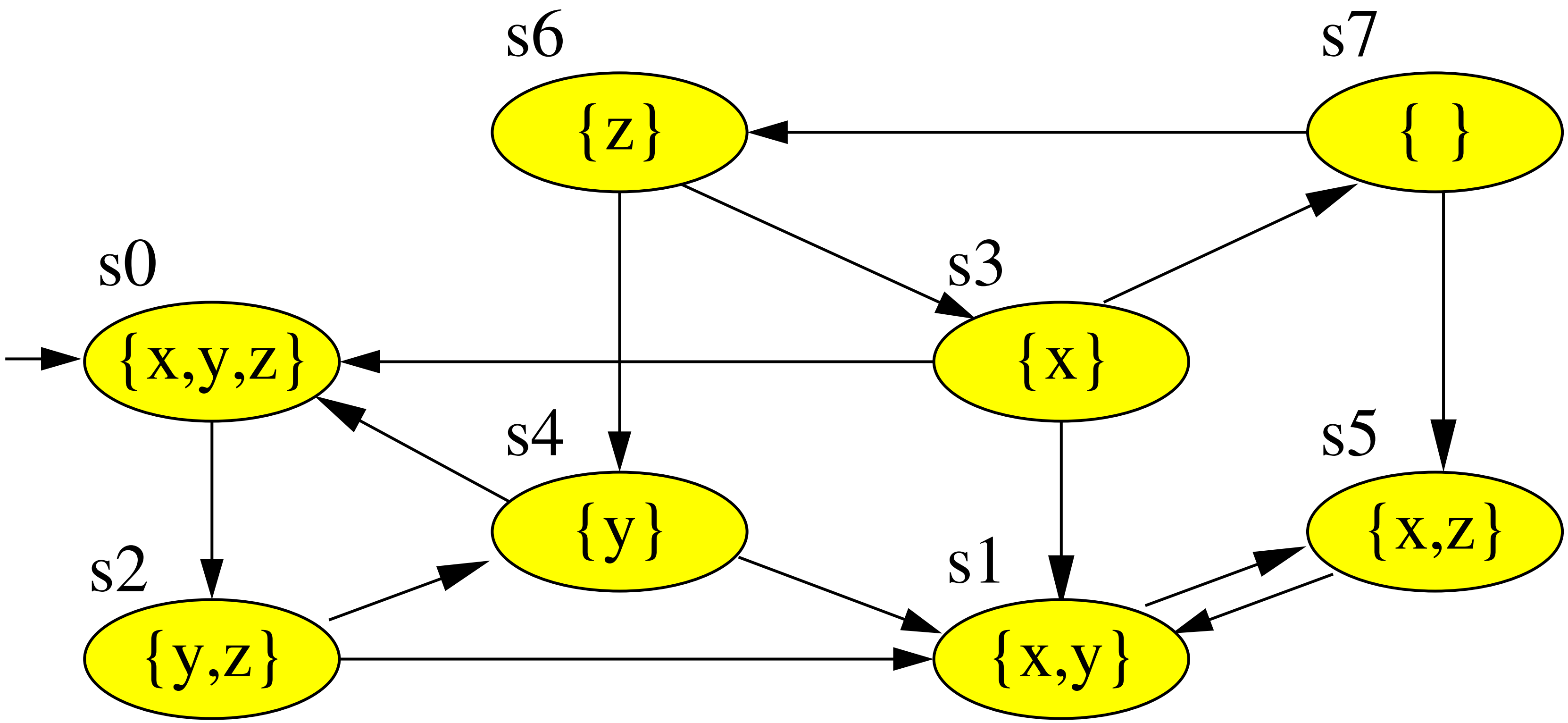
An algorithm for $\mathbf{EG}$

Lemma 3 gives us a strategy for computing $[\![\mathbf{EG}\, p]\!]_{\mathcal{K}}$: compute the sequence $S,\ \pi(S),\ \cdots$ until a fixed point is reached.

For practicality, one would start immediately with $X := \mu(p)$. Then, in each round, one eliminates those states having no successors in X.

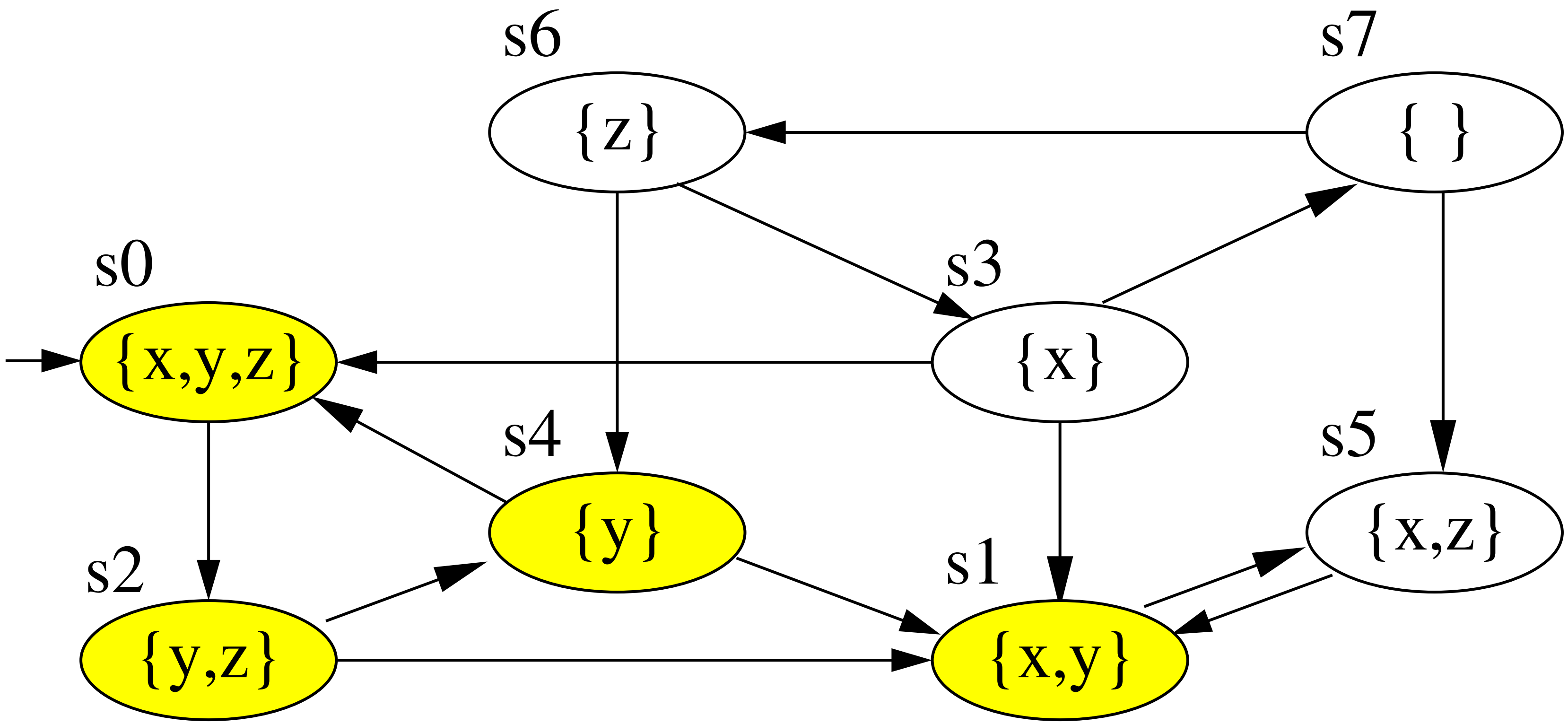
This can be efficiently implemented in $\mathcal{O}(|\mathcal{K}|)$ time ("reference counting").

Example: Computation of $[\![\mathbf{EG}\ y]\!]_{\mathcal{K}}$ (1/4)



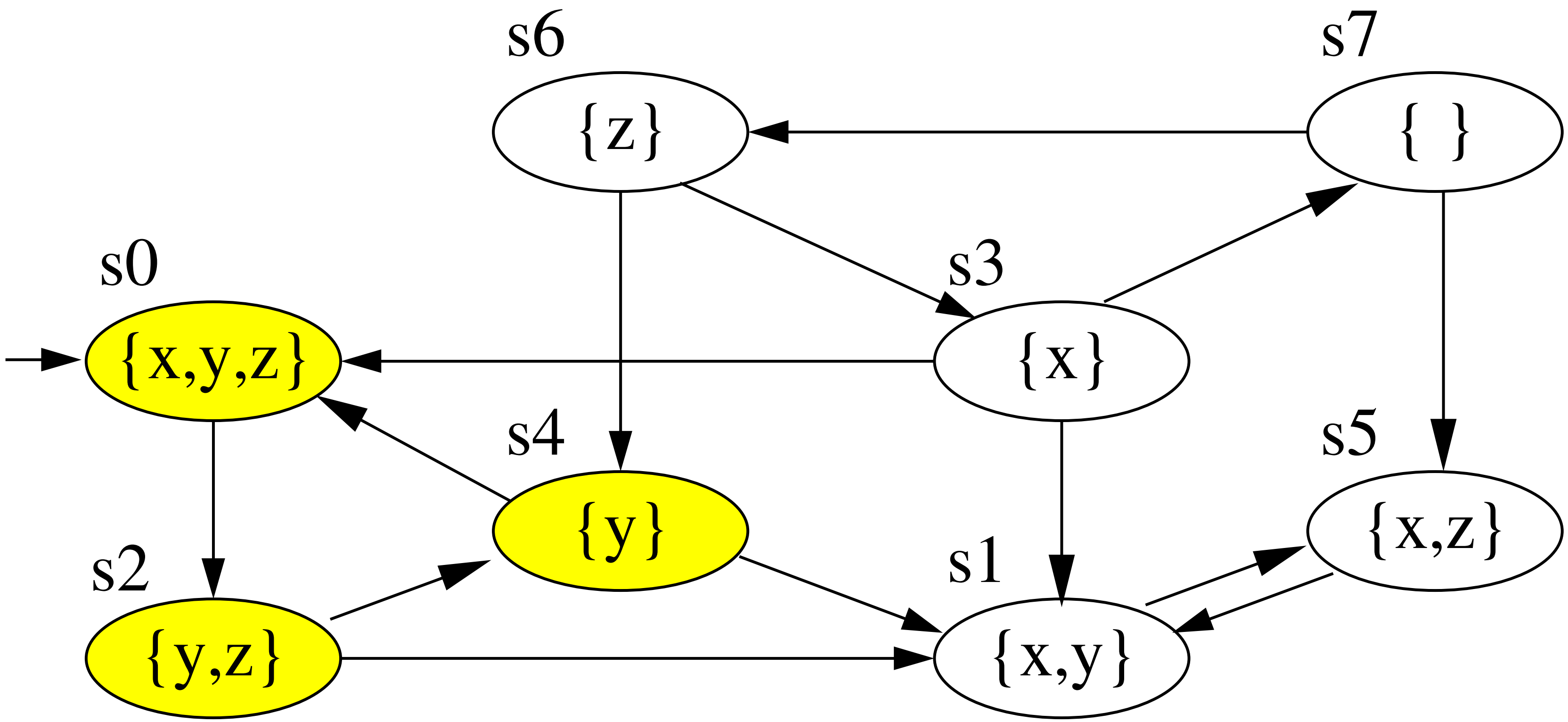
$$\pi^0(S) \ = \ S$$

Example: Computation of $[\![\mathbf{EG}\ y]\!]_{\mathcal{K}}$ (2/4)



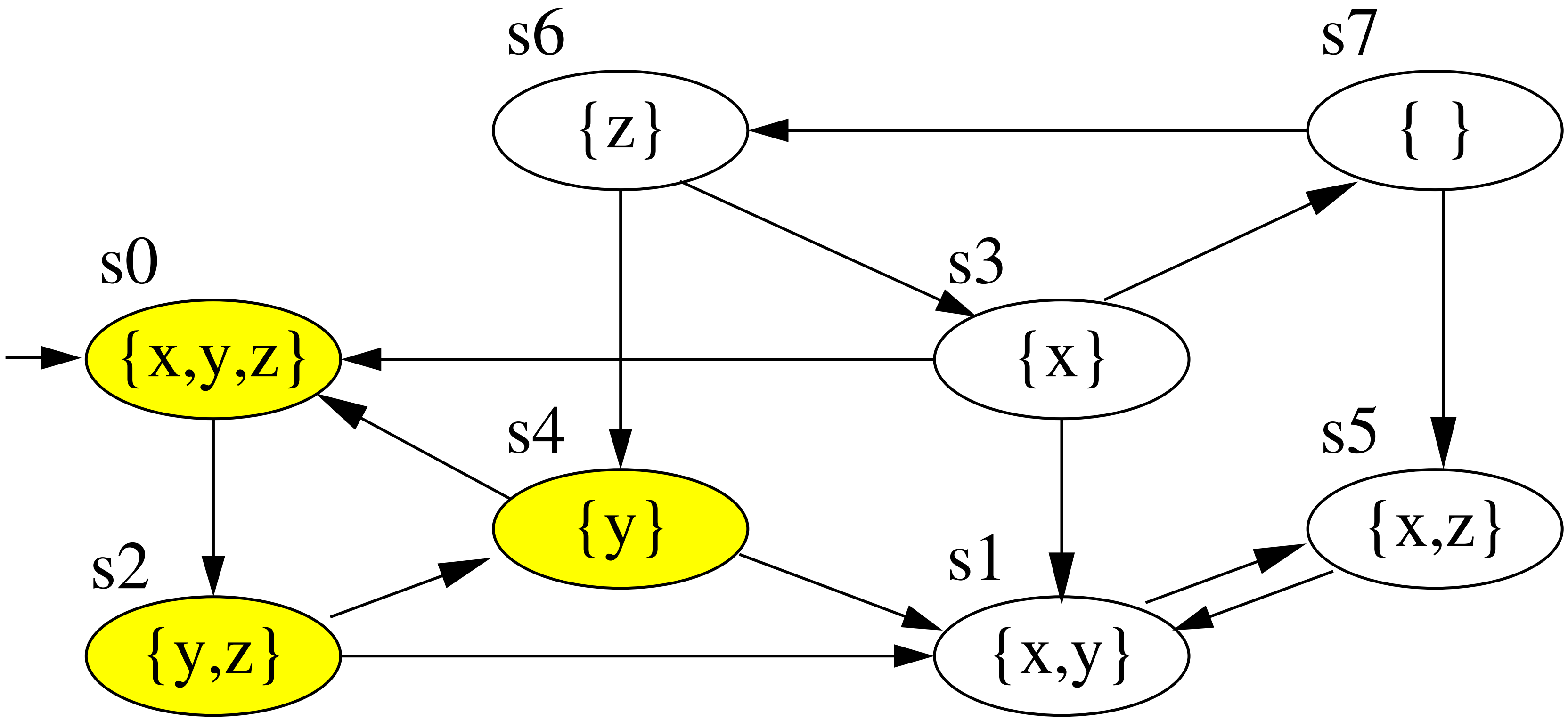
$$\pi^1(S) \;=\; \mu(y) \cap pre(S)$$

Example: Computation of $[\![\mathbf{EG}\ y]\!]_{\mathcal{K}}$ (3/4)



$$\pi^2(S) \;=\; \mu(y) \cap pre(\pi^1(S))$$

Example: Computation of $[\![\mathbf{EG}\ y]\!]_{\mathcal{K}}$ (4/4)



$\pi^3(S) \;=\; \mu(y) \cap pre(\pi^2(S)) \;=\; \pi^2(S)$: $\quad [\![\mathbf{EG}\ y]\!]_{\mathcal{K}} = \{s_0, s_2, s_4\}$

Computation of $\mathbf{EU}$

Case 5: $\psi \equiv p\,\mathbf{EU}\,q$, $\quad p, q \in AP$

Analogous to $\mathbf{EG}$ (proofs omitted):

Lemma 4: $[\![p\,\mathbf{EU}\,q]\!]_\mathcal{K}$ is the smallest solution (w.r.t. $\subseteq$) of the equation

$$X = \mu(q) \cup (\mu(p) \cap pre(X)).$$

Lemma 5: $[\![p\,\mathbf{EU}\,q]\!]_\mathcal{K}$ is the fixed point of the sequence

$\emptyset,\ \ \xi(\emptyset),\ \ \xi(\xi(\emptyset)),\ \ \ldots$ where $\quad \xi(X) := \mu(q) \cup (\mu(p) \cap pre(X))$

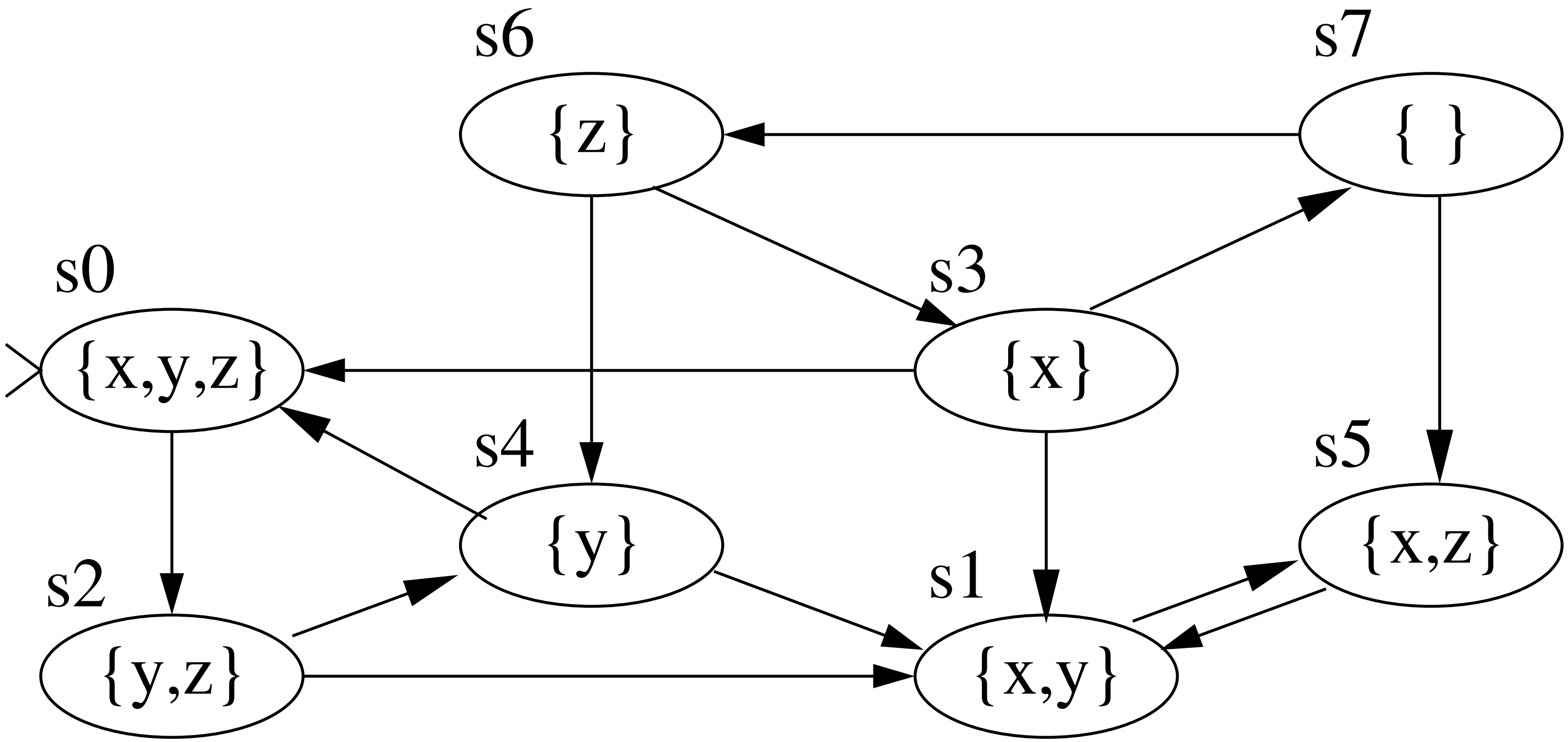
An algorithm for $\mathbf{EU}$

Lemma 5 proposes a strategy: Compute the sequence $\emptyset,\ \xi(\emptyset),\ \cdots$ until a fixed point is reached.

In practice one would start with $X := \mu(q)$. Then, in each step, one can add those direct predecessors that are in $\mu(p)$.

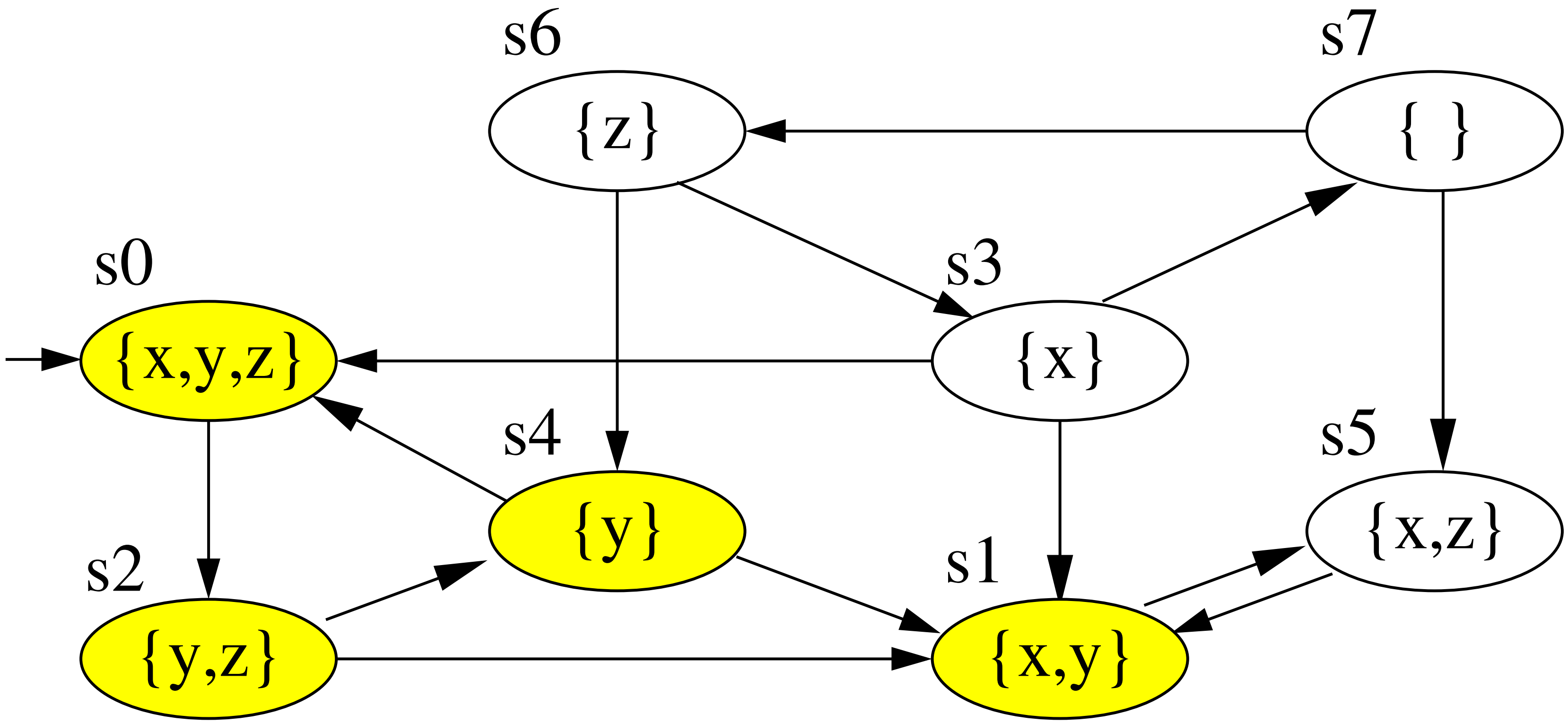
Can be done efficiently in $\mathcal{O}(|\mathcal{K}|)$ time (multiple backwards DFS).

Example: Computation of $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (1/4)



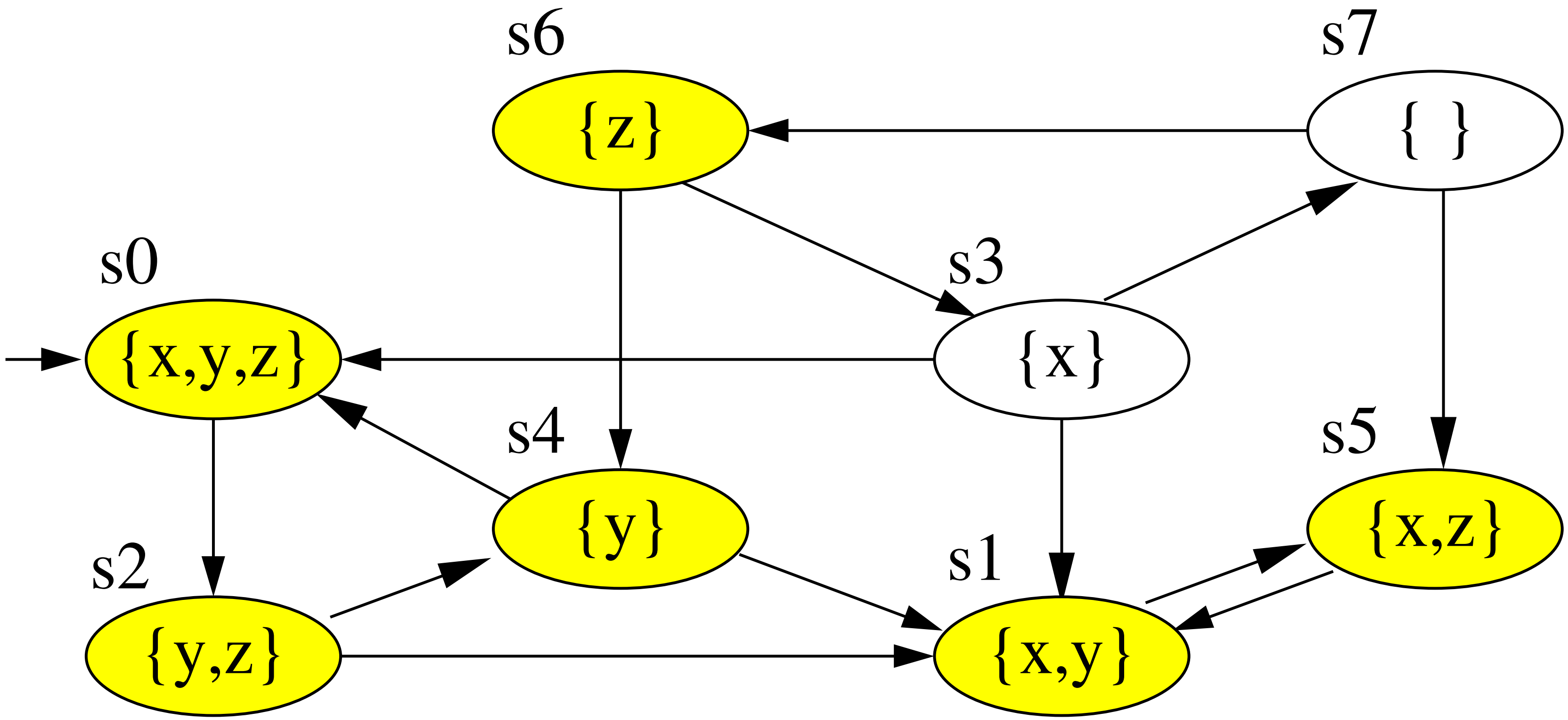
$$\xi^0(\emptyset) = \emptyset$$

Example: Computation of $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (2/4)



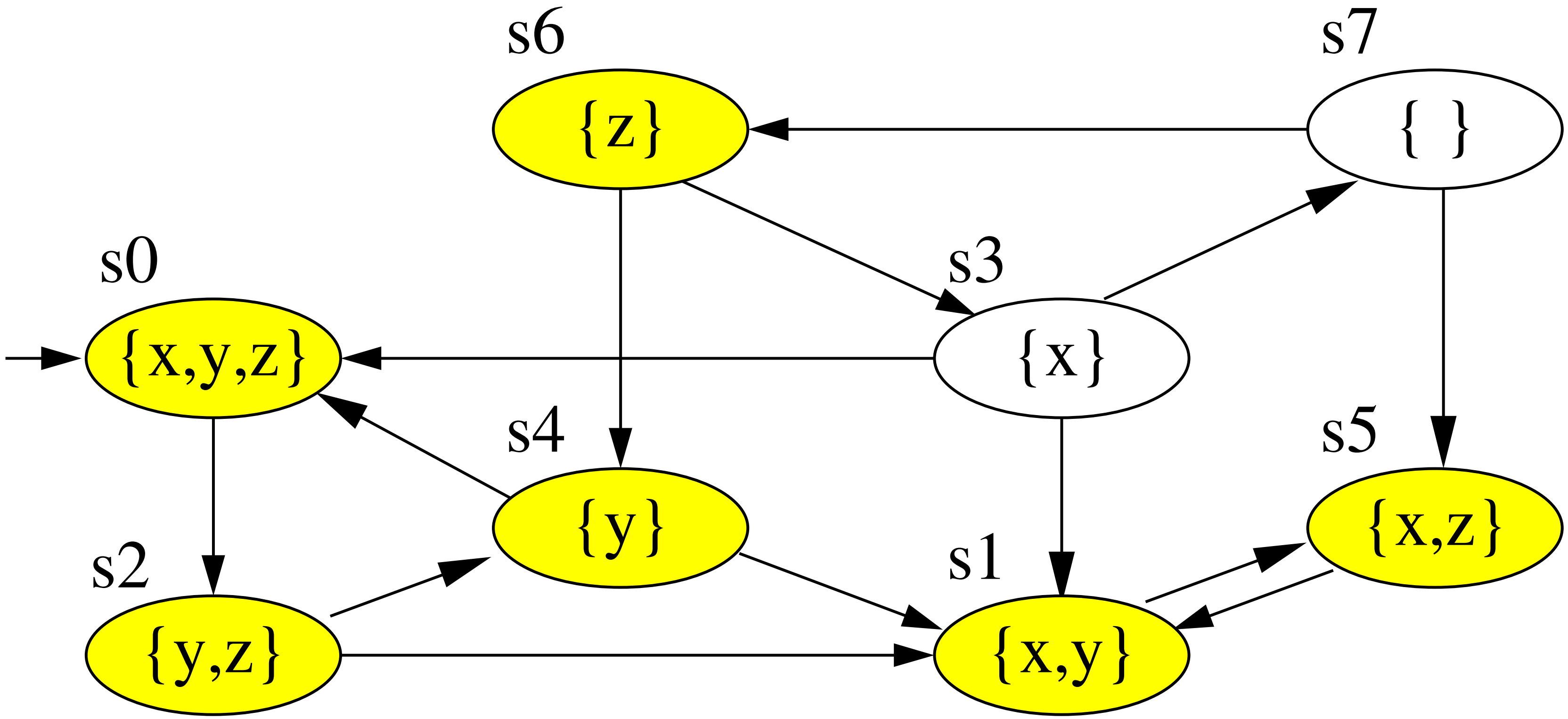
$$\xi^1(\emptyset) \ = \ \mu(y) \cup (\mu(z) \cap pre(\xi^0(\emptyset)))$$

Example: Computation of $[\![z \ \mathbf{EU} \ y]\!]_{\mathcal{K}}$ (3/4)



$$\xi^2(\emptyset) \ = \ \mu(y) \cup (\mu(z) \cap pre(\xi^1(\emptyset)))$$

Example: Computation of $[\![z\ \mathbf{EU}\ y]\!]_{\mathcal{K}}$ (4/4)



$$\xi^3(\emptyset) \ = \ \mu(y) \cup (\mu(z) \cap pre(\xi^2(\emptyset))) \ = \ \xi^2(\emptyset)$$

$[\![z\ \mathbf{EU}\ y]\!]_{\mathcal{K}} = \{s_0, s_1, s_2, s_4, s_5, s_6\}$

Example: Dekker's mutex algorithm

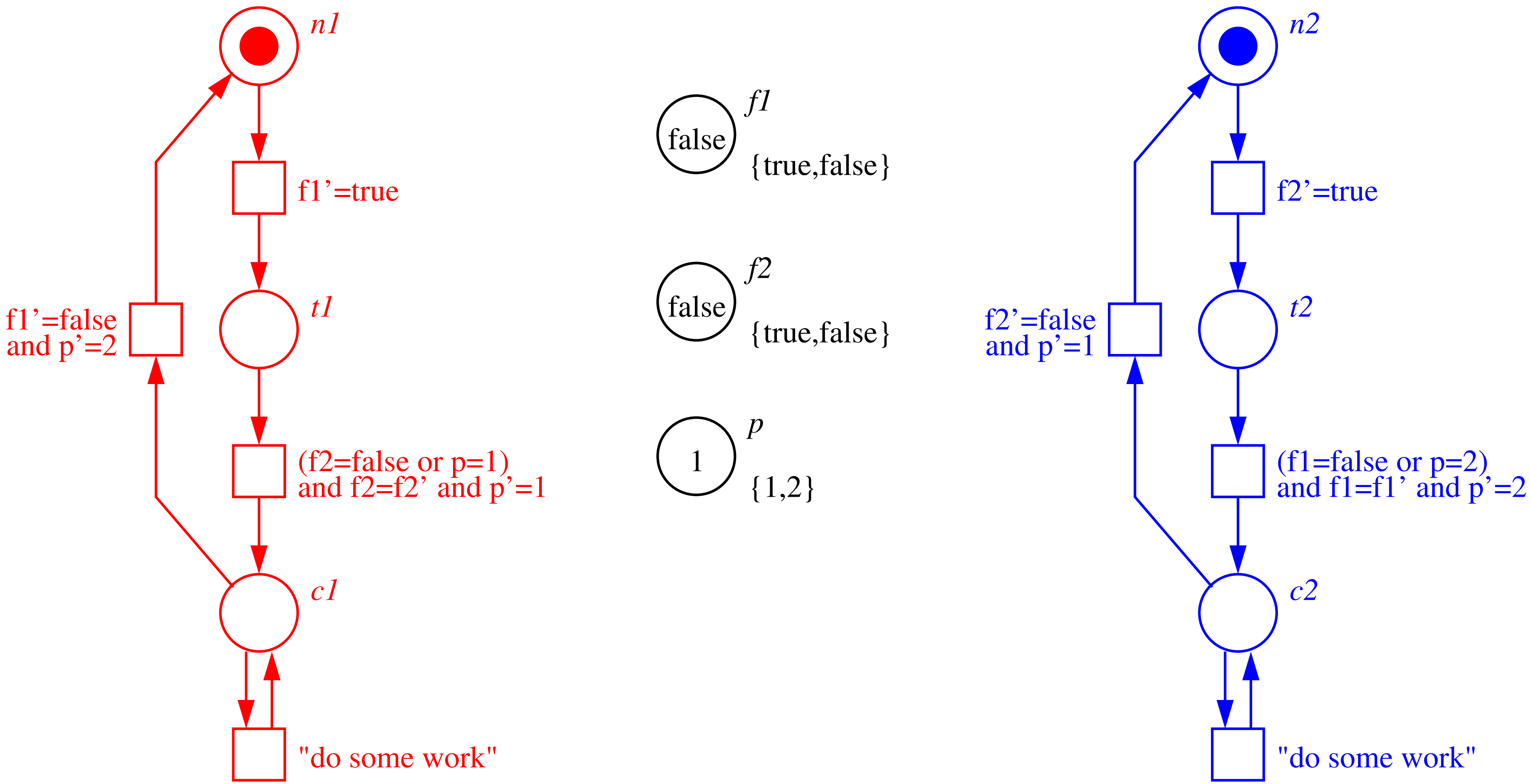
The next slides show a Petri net implementing a (fair) mutual exclusion protocol for two processes (red and blue), and the reachability graph of the net.

The places $n1, n2$ denote the non-critical sections, $c1, c2$ the critical sections, and $t1, t2$ indicate that a process is *trying* to enter its critical section.

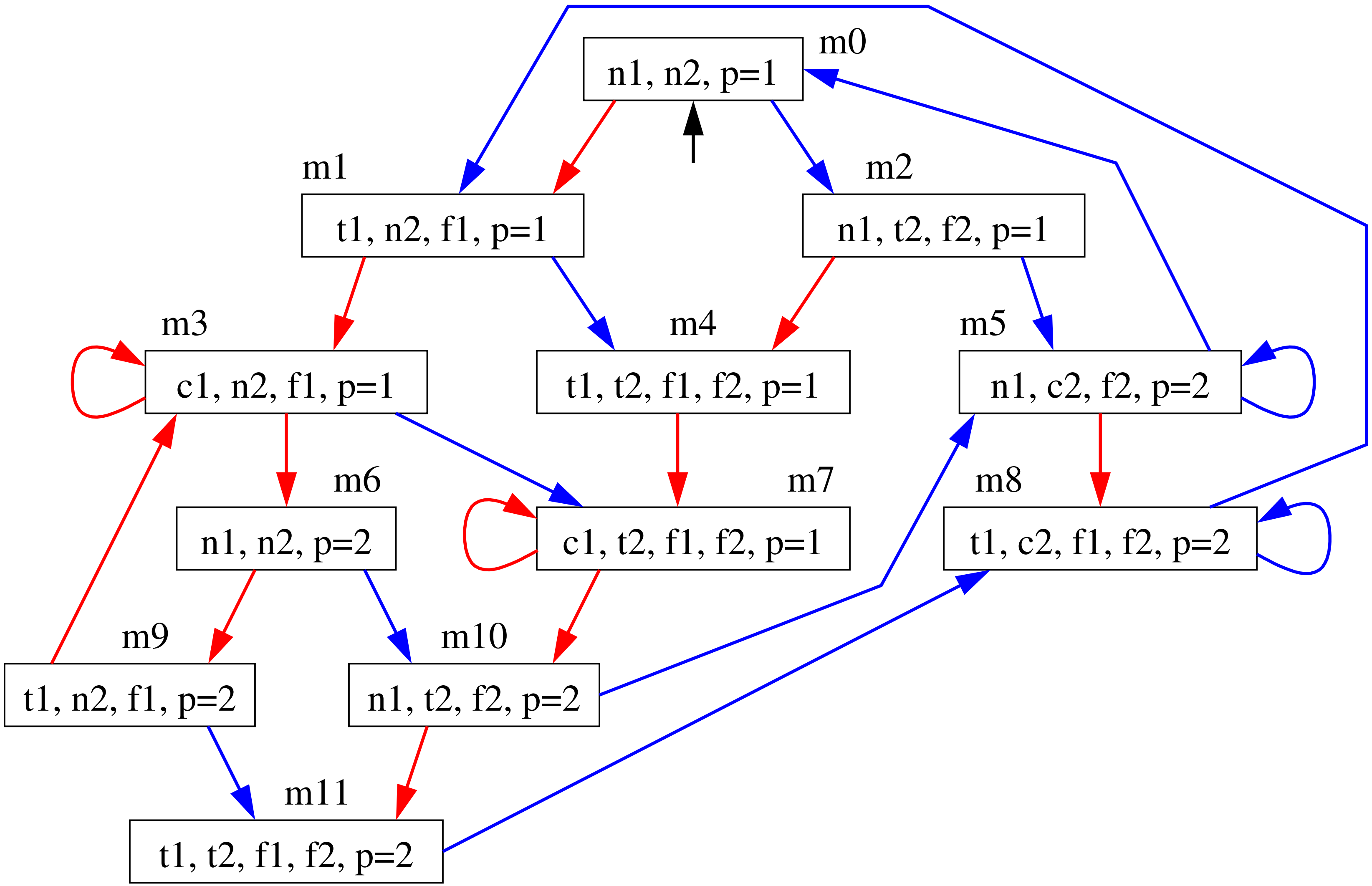
For space reason, the arcs to places in the middle ($f1, f2, p$) were omitted.

In the reachability graph, $f1$ and $f2$ are only mentioned when true.

Example: Petri net



Example: Reachability graph



Specification

The satisfaction mutual exclusion property is directly observable in the reachability graph. Moreover, we might be interested in the following property:

> "Whenever a process wants to enter a critical section, it will eventually succeed in doing so."

A plausible formulation in CTL is as follows:

$$\mathbf{AG}(t1 \rightarrow \mathbf{AF}\, c1) \qquad \text{and} \qquad \mathbf{AG}(t2 \rightarrow \mathbf{AF}\, c2)$$

For this, we extend the reachability graph into a Kripke structure with four atomic propositions:

$c1$ with $\mu(c1) := \{m_3, m_7\}$; $\quad$ $t1$ with $\mu(t1) := \{m_1, m_4, m_8, m_9, m_{11}\}$

$c2$ with $\mu(c2) := \{m_5, m_8\}$; $\quad$ $t2$ with $\mu(t2) := \{m_2, m_4, m_7, m_{10}, m_{11}\}$

Checking the formula

We rewrite the first formula into the minimal syntax:

$$\neg\Big(\mathbf{true}\ \mathbf{EU}\ (t1 \wedge \mathbf{EG}\,\neg c1)\Big)$$

(The second formula is analogous, so we consider just the first.)

When checking this formula, we shall observe that it does not hold – the system may remain in states m_5 and m_8 forever.

This happens, when one process remains in its critical section forever.

Fairness

Thus, the property is not satisfied because the blue process may exhibit an "unfair" behaviour, i.e. does not leave the critical section.

Can we – in analogy to LTL – consider only those runs that satisfy some fairness constraint (here: processes eventually leave their critical section)?

Answer 1: No. This is not expressible in CTL (e.g., $(\mathbf{AG}\,\mathbf{AF}\,\textit{fair}) \rightarrow \phi$ won't do).

Answer 2: Yes. We can extend CTL accordingly.

CTL with fairness

Let $\mathcal{K}$ and ϕ be the same as before. Additionally, let $F_1, \ldots, F_n \subset S$ be fairness constraints.

In the following, we shall call a path fair iff it visits each fairness constraint infinitely often.

In our example, the fairness constraints would be as follows:

$$F_1 = S \setminus \mu(c1)$$

$$F_2 = S \setminus \mu(c2)$$

Our problem is to compute $[\![\phi]\!]_{\mathcal{K}}$ for the case where $\mathbf{EG}$ and $\mathbf{EU}$ quantify only over "fair" paths. We introduce modified operators $\mathbf{EG_f}$ and $\mathbf{EU_f}$ with the following meaning:

$\mathcal{T} \models$	$\mathbf{EG_f}\,\phi$	iff $\mathcal{T}$ has a fair infinite path $r = v_0 \to v_1 \to \ldots$,
		where for all $i \geq 0$ we have: $\lceil v_i \rceil \models \phi$
$\mathcal{T} \models$	$\phi_1\,\mathbf{EU_f}\,\phi_2$	iff $\mathcal{T}$ has a fair infinite path $r = v_0 \to v_1 \to \ldots$,
		s.t. $\exists i\colon \lceil v_i \rceil \models \phi_2 \wedge \forall k < i\colon \lceil v_k \rceil \models \phi_1$

We make the following observations:

(1) ρ is fair iff ρ^i is fair for all $i \geq 0$.

(2) ρ is fair iff there is a fair suffix ρ^i for some $i \geq 0$.

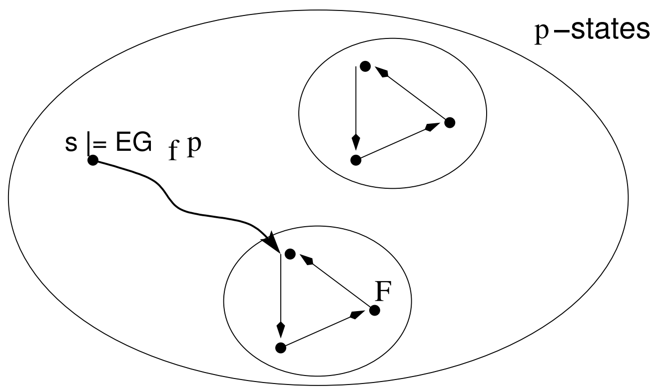
Thus, we can rewrite $\mathbf{EU_f}$ as follows:

$$\phi_1 \, \mathbf{EU_f} \, \phi_2 \quad \equiv \quad \phi_1 \, \mathbf{EU} \, (\phi_2 \wedge \mathbf{EG_f} \, \mathbf{true})$$

It therefore suffices to find a modified algorithm for $\mathbf{EG_f}$.

An algorithm for $[\![\mathbf{EG_f}\, p]\!]_{\mathcal{K}}$

1. Let $\mathcal{K}_p$ be the restriction of $\mathcal{K}$ to the states $\mu(p)$.
2. Compute the SCCs of $\mathcal{K}_p$.
3. Find the non-trivial SCCs intersecting all fairness constraints.
4. $[\![\mathbf{EG_f}\, p]\!]_{\mathcal{K}}$ contains exactly those states in $\mathcal{K}_p$ from which such an SCC is reachable.



Complexity: still linear in $|\mathcal{K}|$.

Comparison of CTL and LTL

Many properties can be expressed equivalently in both CTL and LTL, e.g.

Invariants (e.g., "p never holds.")

$$\mathbf{AG}\,\neg p \qquad \text{or} \qquad \mathbf{G}\,\neg p$$

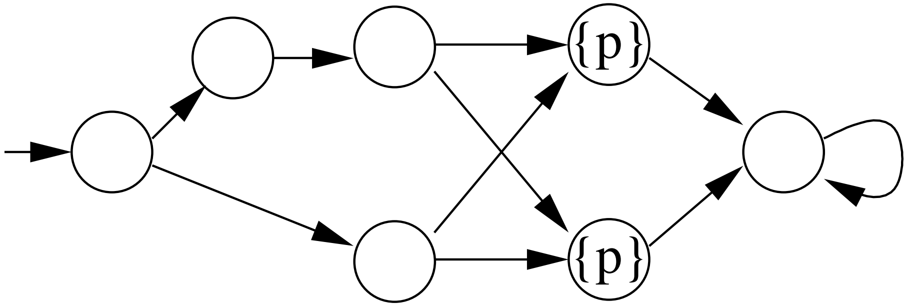
Reactivity ("Whenever p happens, eventually q will happen.")

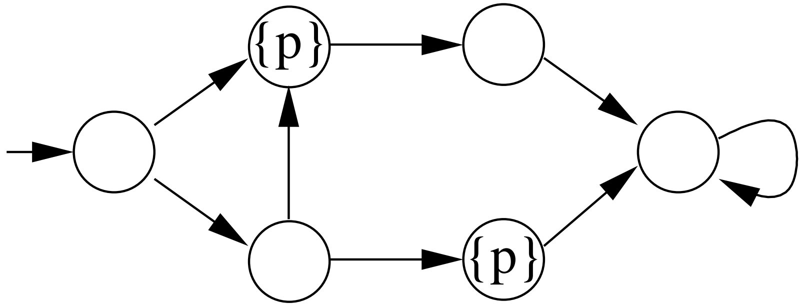
$$\mathbf{AG}(p \rightarrow \mathbf{AF}\,q) \qquad \text{or} \qquad \mathbf{G}(p \rightarrow \mathbf{F}\,q)$$

CTL considers the whole computation tree, LTL the set of runs. Hence, CTL can reason about the branching behaviour (the *possibilities*), which LTL cannot. Examples:

The CTL formula $\mathbf{AG}\,\mathbf{EF}\,p$ ("reset") is not expressible in LTL.

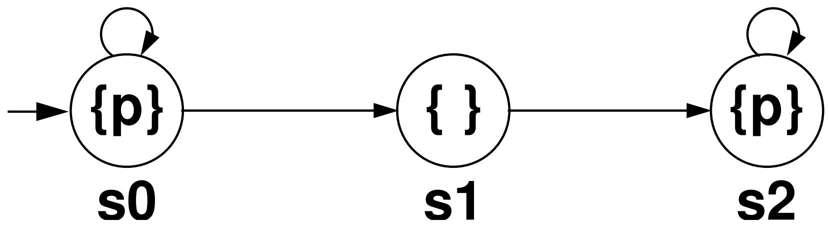
The CTL formula $\mathbf{AF}\,\mathbf{AX}\,p$ distinguishes the following structures, but the LTL formula $\mathbf{F}\,\mathbf{X}\,p$ does not:





However, the syntactic restriction in CTL (paired quantifiers and path operators) means that, in turn, some LTL properties are inexpressible in CTL. Hence, the two logics are *incomparable* in their expressive power.

The LTL property $\mathbf{F}\,\mathbf{G}\,p$ is inexpressible in CTL:



$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$ but $\mathcal{K} \not\models \mathbf{AF}\,\mathbf{AG}\,p$

(Emerson, Halpern 1986)

Computational complexity

CTL model checking: $\mathcal{O}(|\mathcal{K}| \cdot |\phi|)$
LTL model checking: $\mathcal{O}(|\mathcal{K}| \cdot 2^{|\phi|})$

Does this mean that CTL model checking is more efficient?
(for properties expressible in both logics)

Answer: not necessarily

LTL enables on-the-fly model checking and partial-order reduction

CTL: whole structure must be constructed and traversed multiple times

CTL: instead of p.o.reduction $\rightarrow$ efficient data structures for sets (to be done)

Literature

More about this topic:

M. Vardi, *Branching vs. Linear Time: Final Showdown*, 2001

Specification Patterns Database:

`http://patterns.projects.cis.ksu.edu/`

Tool demonstration: SMV

The SMV system

SMV was designed by Ken McMillan (CMU, 1992)

Model-checking tool for CTL (with fairness)

Useful for describing finite structures, especially synchronous or asynchronous circuits.

SMV was the first tool suitable for verifying large hardware systems (by employing BDDs).

Information/downloading SMV

In the world-wide web:

`http://www-2.cs.cmu.edu/~modelcheck/smv.html`

(includes extensive manual)

CTL model checking in SMV

The problem: Given $\mathcal{K}$ (with multiple initial states) and ϕ,
do all initial states of $\mathcal{K}$ satisfy ϕ?

In SMV, structures consist of "modules" and "processes", that manipulate a number of variables.

Transitions are specified by declaring their 'next' value, depending on their current value.

Atomic propositions may talk about variables and are interpreted 'naturally' (as in Spin).

Demonstration: short.smv

Syntax example (1/3)

```
-- This is a comment.

MODULE main
VAR
   x : boolean;
   y : {q1,q2};

-- to be continued
```

Remarks:

- Data types: boolean, integer, enumerations
- all variable with *finite* range

Syntax example (2/3)

```
ASSIGN
  init(x) := 1;                       -- initial value
  next(x) := case                     -- transition relation
               x: 0;
              !x: 1;
             esac;

  next(y) := case
               x & y=q1: q2;
               x & y=q2: {q1,q2};  -- non-determinism
               1       : y;
             esac;
```

Syntax example: Remarks

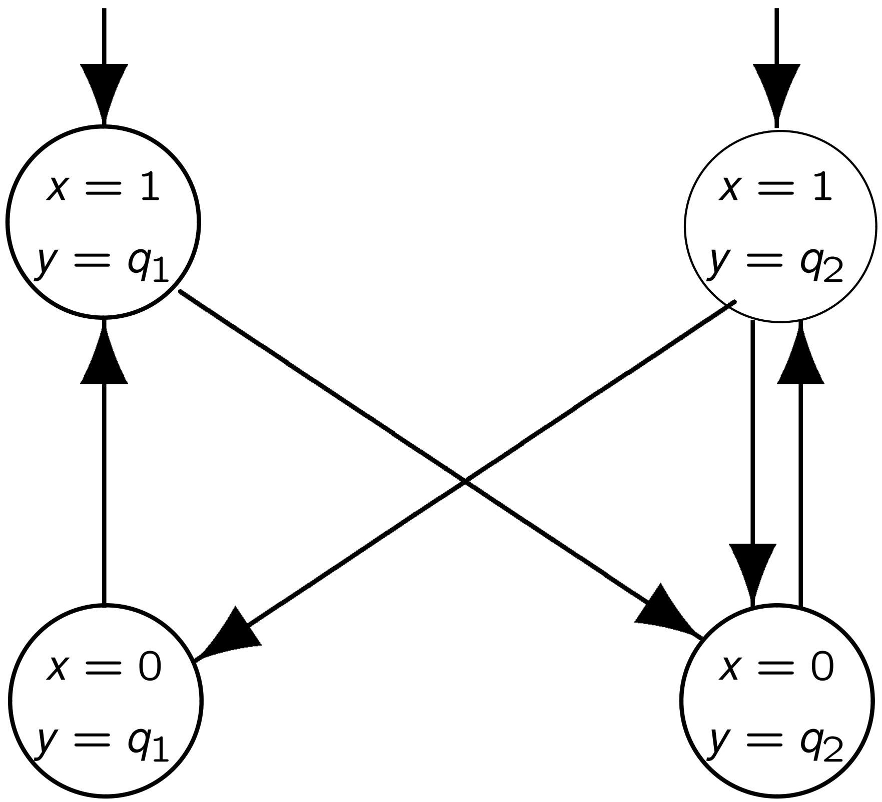
Initial states are given by the init predicates; uninitialized variables can take any initial value.

The transitions are given by the synchronous composition of the next predicates.

case expressions are evaluated top to bottom, the first case that fits is taken.

Non-determinism can be introduced by giving multiple successor values.

The resulting structure



Syntax example (3/3)

```
-- Is q1 always reachable from q2?
SPEC AG (y=q2 -> EF y=q1)

-- Is x true infinitely often in every execution?
SPEC AG AF x
```

Remarks:

- One can give multiple CTL formulae, SMV will check them all one by one.

Modules (1/2)

Modules may be parametrized. Example:

```
MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := (value + carry_in) mod 2;
DEFINE
  carry_out := value & carry_in;
```

Remark:

- The parameter of this module is `carry_in`.
- `DEFINE` declares a ‘macro’.

Modules (2/2)

Modules may be instantiated like variables:

```
MODULE main
VAR
  bit0 : counter_cell(1);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);
```

This system behaves like a three-bit counter, i.e. “counts” from 0 to 7 and then resets.

Remark: There must be one module named `main`, and SMV will evaluate the specifications of this module.

Demonstration: counter.smv

Asynchronous systems

All examples up to now were synchronous, i.e. all variables and modules take a transition *at the same time*.

When modules are instantiated with the keyword `process` (see next example), then in each step one process makes a step while the others do nothing (*asynchronous* composition, interleaving).

Alternatively, no process may make a step (“stuttering”).

Example: Mutex

var turn : $\{0,1\}$;

while true do	**while true do**
q_0 non-critical section	r_0 non-critical section
q_1 **await** (turn=0);	r_1 **await** (turn=1);
q_2 critical section	r_2 critical section
q_3 turn:=1;	r_3 turn:=0;
od	**od**

Mutex in SMV (1/2)

```
MODULE main
VAR
  turn: boolean;
  p0: process p(0,turn);
  p1: process p(1,turn);

SPEC
  AG !(p0.state = critical & p1.state = critical)
```

Mutex in SMV (2/2)

```
MODULE p (nr,turn)
VAR
  state: {non_critical, critical};
ASSIGN
  init(state) := non_critical;
  next(state) := case
     state = non_critical & turn != nr: non_critical;
     state = non_critical & turn = nr : critical;
     state = critical: {critical,non_critical};
  esac;
  next(turn) := case
     state = critical & next(state) = non_critical: !nr;
     1                                              : turn;
  esac;
```

Fairness

In the mutex example, the following specification is evaluated to false.

```
SPEC
  AG (p0.state = non_critical -> AF p0.state = critical)
```

This is because SMV allows the system to “stutter” forever (i.e. to do nothing). One can exclude such behaviours using the keyword `FAIRNESS`, e.g. as follows:

```
FAIRNESS
  p0.running & p1.running
```

The internal variable `running` becomes true whenever the corresponding process makes a step. With this addition, the verification succeeds.

Part 12: Binary Decision Diagrams

Literature

Some pointers:

H.R. Andersen, *An Introduction to Binary Decision Diagrams*, Lecture notes, Department of Information Technology, IT University of Copenhagen

URL:
https://www.cmi.ac.in/ madhavan/courses/verification-2011/andersen-bdd.pdf
or https://www.cs.utexas.edu/∼isil/cs389L/bdd.pdf

Libraries:

CUDD, BuDDy, JavaBDD, JDD, JBDD...

parallel: ParaBDD, Sylvan

Set representations

As we have seen, the solution to the model-checking problem for CTL can be expressed by operations on sets:

states satisfying some atomic proposition: $\mu(p)$ for $p \in AP$

states satisfying (sub)formulae: $[\![\psi]\!]_{\mathcal{K}}$

computation by set operations: pre, $\cap$, $\cup$, ...

How can such sets be represented:

explicit list: $S = \{s_1, s_2, s_4, \ldots\}$

symbolic representation: compact notation or data structure

Symbolic representation

There are many ways of representing sets symbolically. Some are commonly used in mathematics:

Intervals: $[1, 10]$ for $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Characterizations: "odd numbers" for $\{1, 3, 5, \ldots\}$

Every symbolic representation is suitable for some sets and less so for others (for instance, intervals for odd numbers).

We are interested in a data structure suitable for representing sets of states in hardware systems, and where the necessary operations (pre, $\cap$ etc) can be implemented efficiently.

In the following, we assume that states can be represented as Boolean vectors

$$S = \{0, 1\}^m \qquad \text{for some } m \geq 1$$

Example:

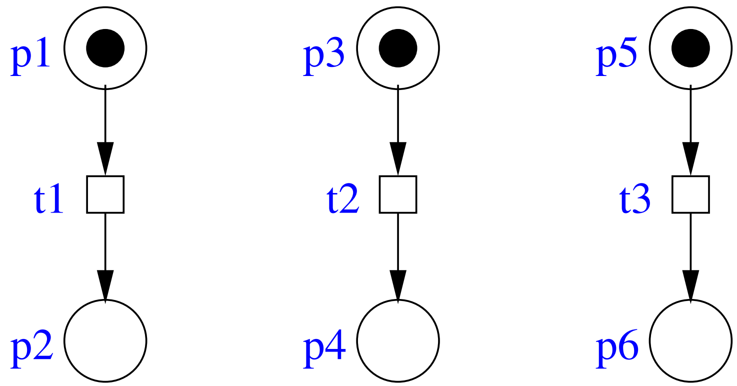
1-safe Petri nets (every place marked with at most one token)

circuits (all inputs and outputs are 0 or 1)

Remark: In general, the elements of *any* finite set can be represented by Boolean vectors if m is chosen large enough. (However, this may not be adequate for all sets.)

Example 1: Petri-net

Consider the following Petri net:

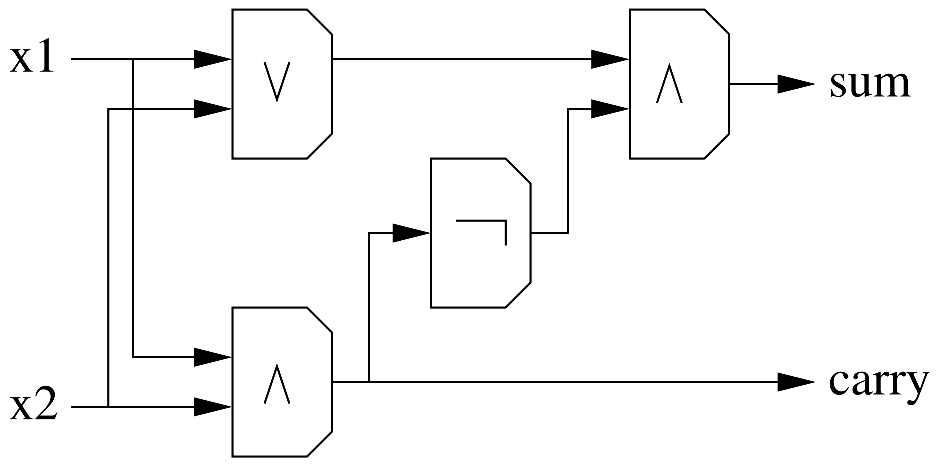


A state can be written as $(p_1, p_2, \ldots, p_6)$, ewhere p_i, $1 \leq i \leq 6$ indicates whether there is a token on P_i.

Initial state$(1, 0, 1, 0, 1, 0)$;
other reachable states are, e.g., $(0, 1, 1, 0, 1, 0)$ or $(1, 0, 0, 1, 0, 1)$.

Example 2: Circuit

Half-adder:



The circuit has got two inputs (x_1, x_2) and two outputs (*carry*, *sum*). Their admissible combinations can be denoted by Boolean 4-tuples, e.g. $(1, 0, 0, 1)$ ($x_1 = 1$, $x_2 = 0$, $carry = 0$, $sum = 1$) is a possible combination.

Characteristic functions

Let $C \subseteq S = \{0,1\}^m$.
(i.e., a set of Boolean vectors.)

The set C is uniquely defined by its characteristic function $f_C \colon S \to \{0,1\}$ given by

$$f_C(s) := \begin{cases} 1 & \text{if } s \in C \\ 0 & \text{if } s \notin C \end{cases}$$

Remark: f_C is a Boolean function with m inputs and therefore corresponds to a formula F of propositional logic with m atomic propositions.

The characteristic function of the admissible combinations in Example 2 corresponds to the following formula of propositional logic:

$$F \equiv \Big(carry \leftrightarrow (x_1 \wedge x_2) \Big) \wedge \Big(sum \leftrightarrow (x_1 \vee x_2) \wedge \neg carry \Big)$$

In the following, we shall treat

- sets of states (i.e. sets of Boolean vectors)
- characteristic functions
- formulae of propositional logic

simply as different representations of the same objects.

Representing formulae

Truth table:

x_1	x_2	*carry*	*sum*	F
0	0	0	0	1
0	0	0	1	0
		...		
0	1	0	1	1
		...		

A truth table is obviously *not* a compact representation.

However, we use it as a starting point for a graphical, more compact representation.

Binary decision graphs

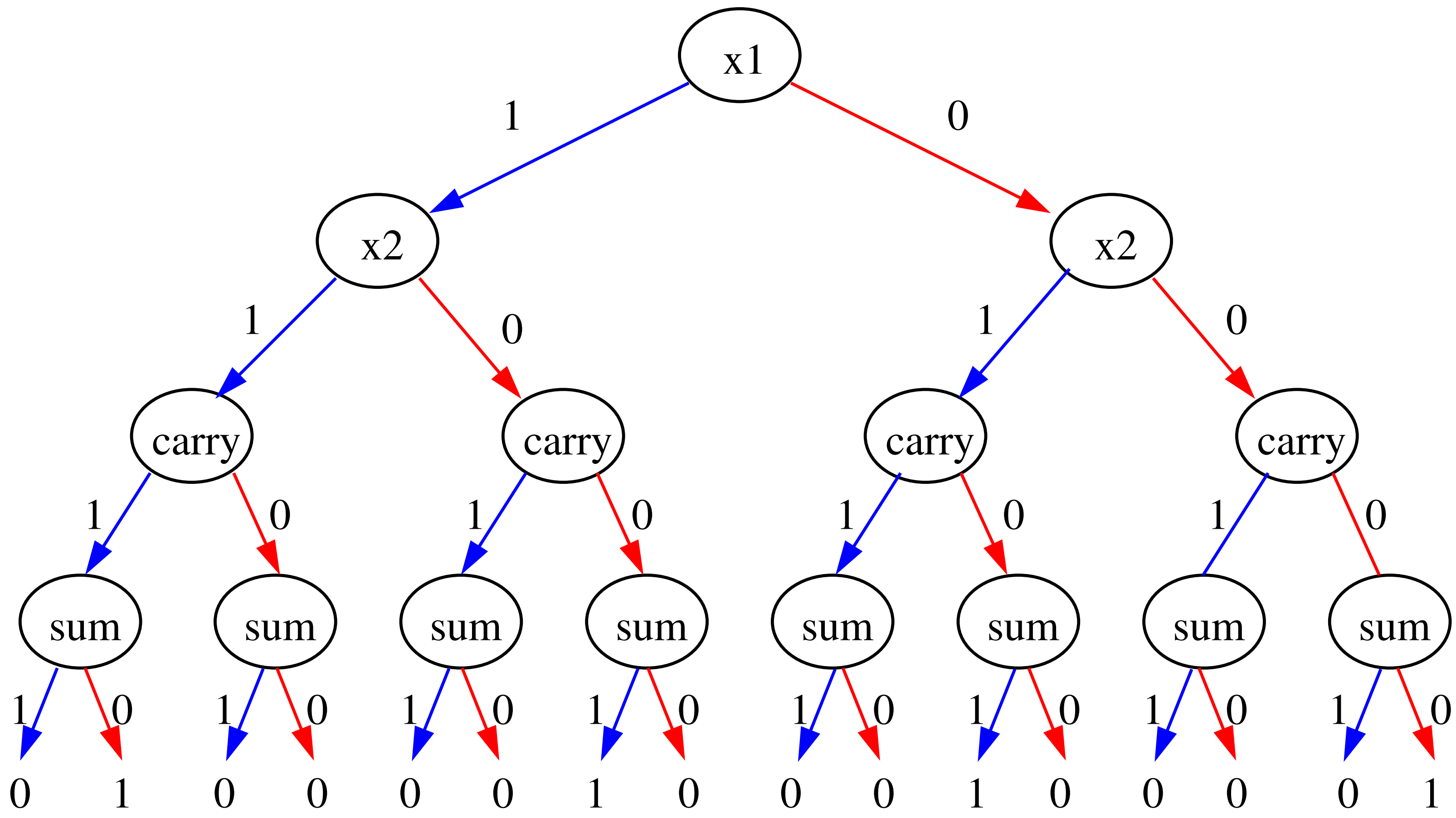
Let V be a set of variables (atomic propositions) and $<$ a total order on V, e.g.

$$x_1 < x_2 < carry < sum$$

A binary decision graph (w.r.t. $<$) is a directed, connected, acyclic graph with the following properties:

- there is exactly one root, i.e. a node without incoming arcs;
- there are at most two leaves, labelled by $\mathbf{0}$ or $\mathbf{1}$;
- all non-leaves are labelled with variabls from V;
- every non-leaf has two outgoing arcs labelled by 0 and 1;
- if there is an edge from an x-labelled node to a y-labelled node, then $x < y$.

Example 2: Binary decision graph (here: a full tree)



Paths ending in 1 correspond to vectors whose entry in the truth table is 1.

Binary decision diagrams

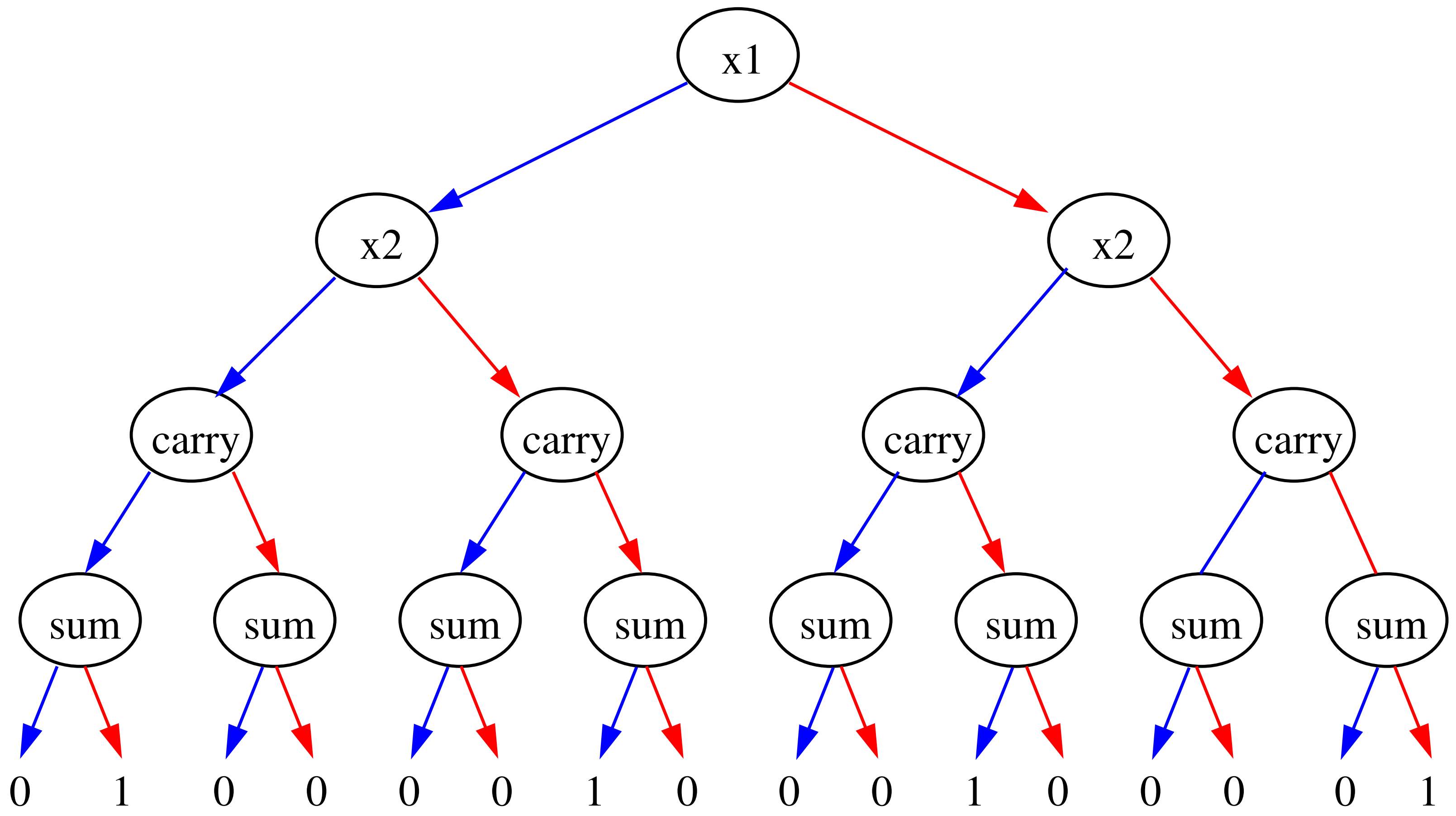
A binary decision diagram (BDD) is a binary decision graph with two additional properties:

- no two subgraphs are isomorphic;
- there are no *redundant* nodes, where both outgoing edges lead to the same target node.

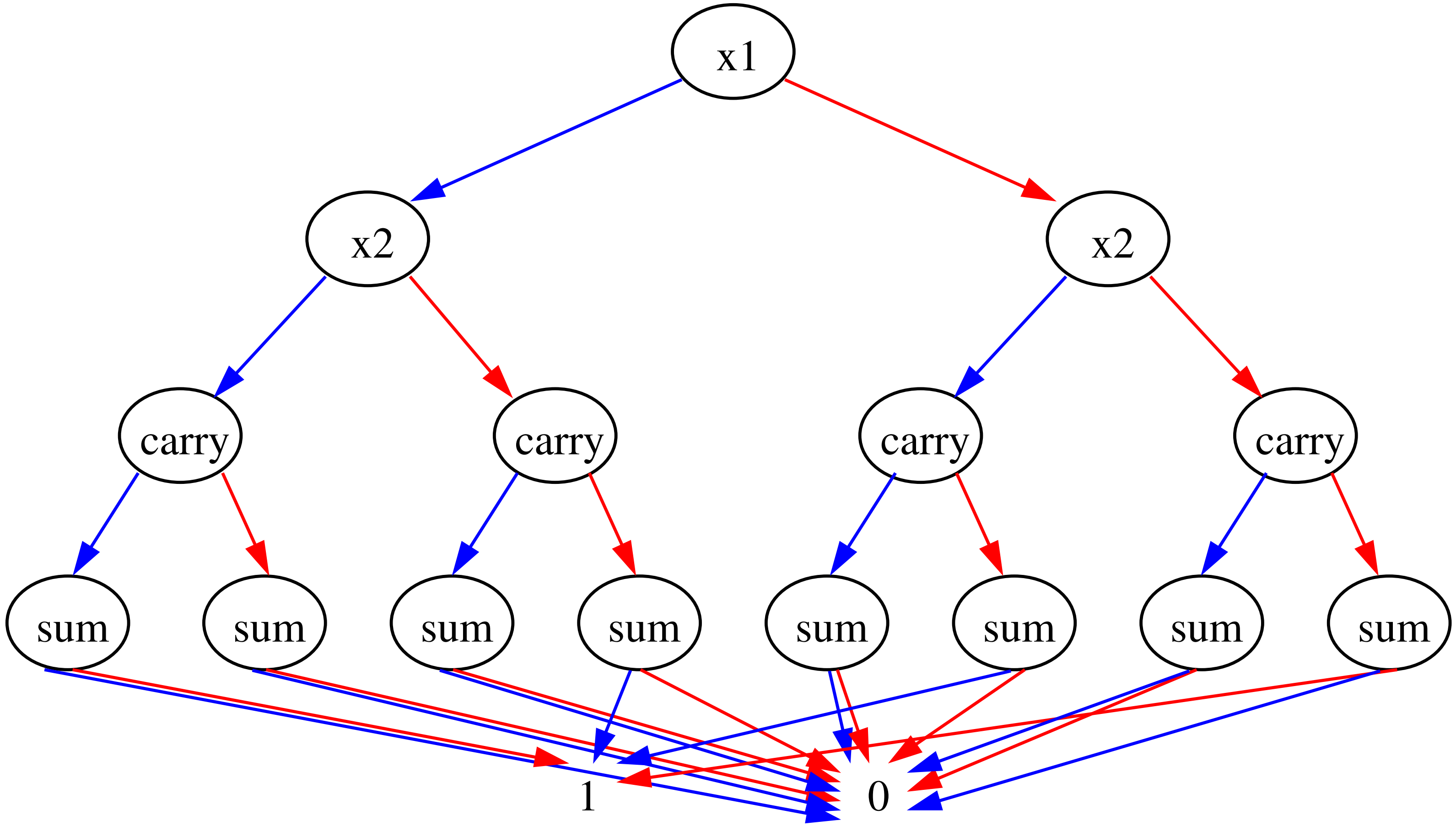
We also allow to omit the **0**-node and the edges leading there.

Remarks: On the following slides, the blue edges are meant to be labelled by 1, the red edges by 0.

Example 2: Eliminate isomorphic subgraphs (1/4)

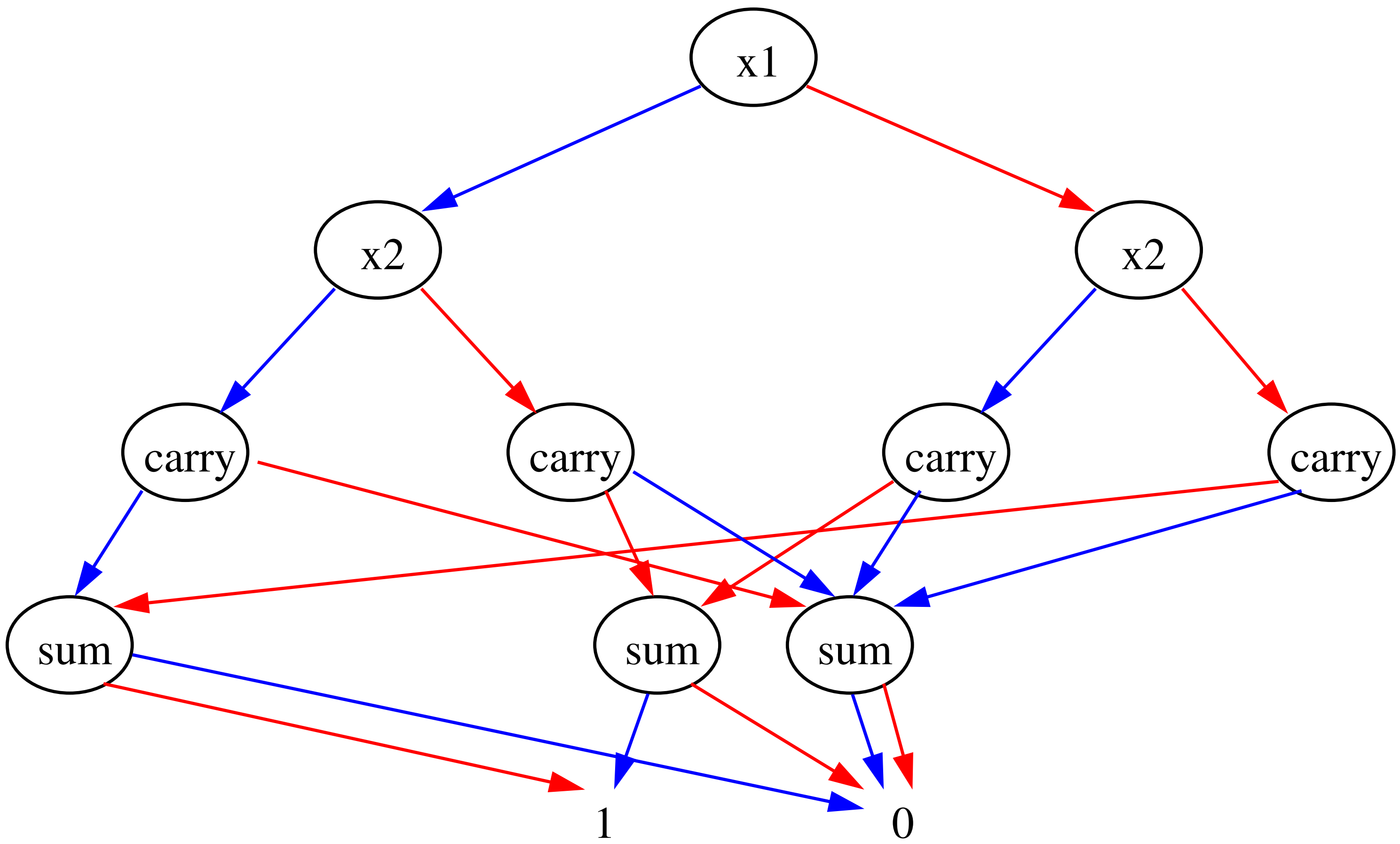


Example 2: Eliminate isomorphic subgraphs (2/4)



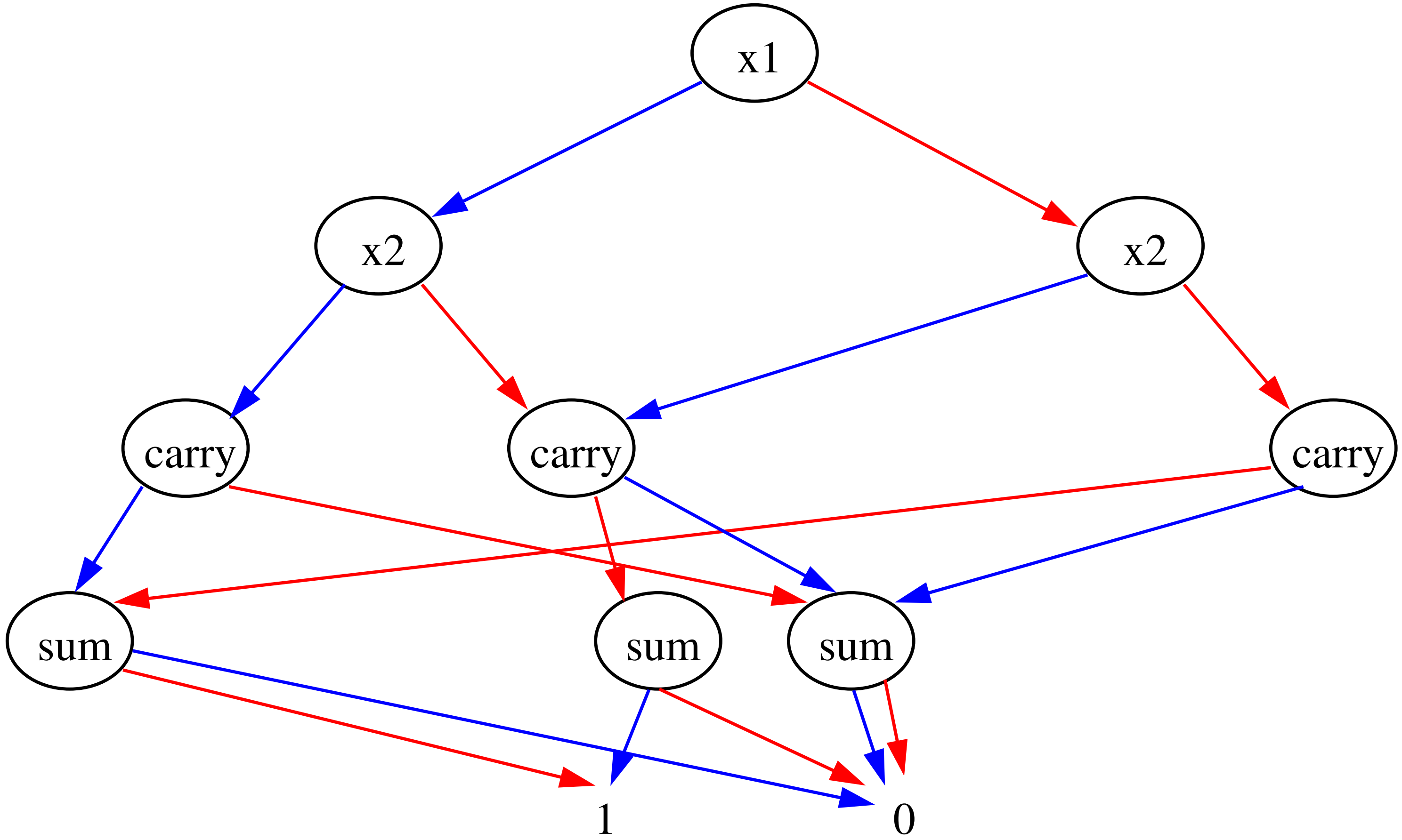
0-nodes and 1-nodes merged, respectively.

Example 2: Eliminate isomorphic subgraphs (3/4)



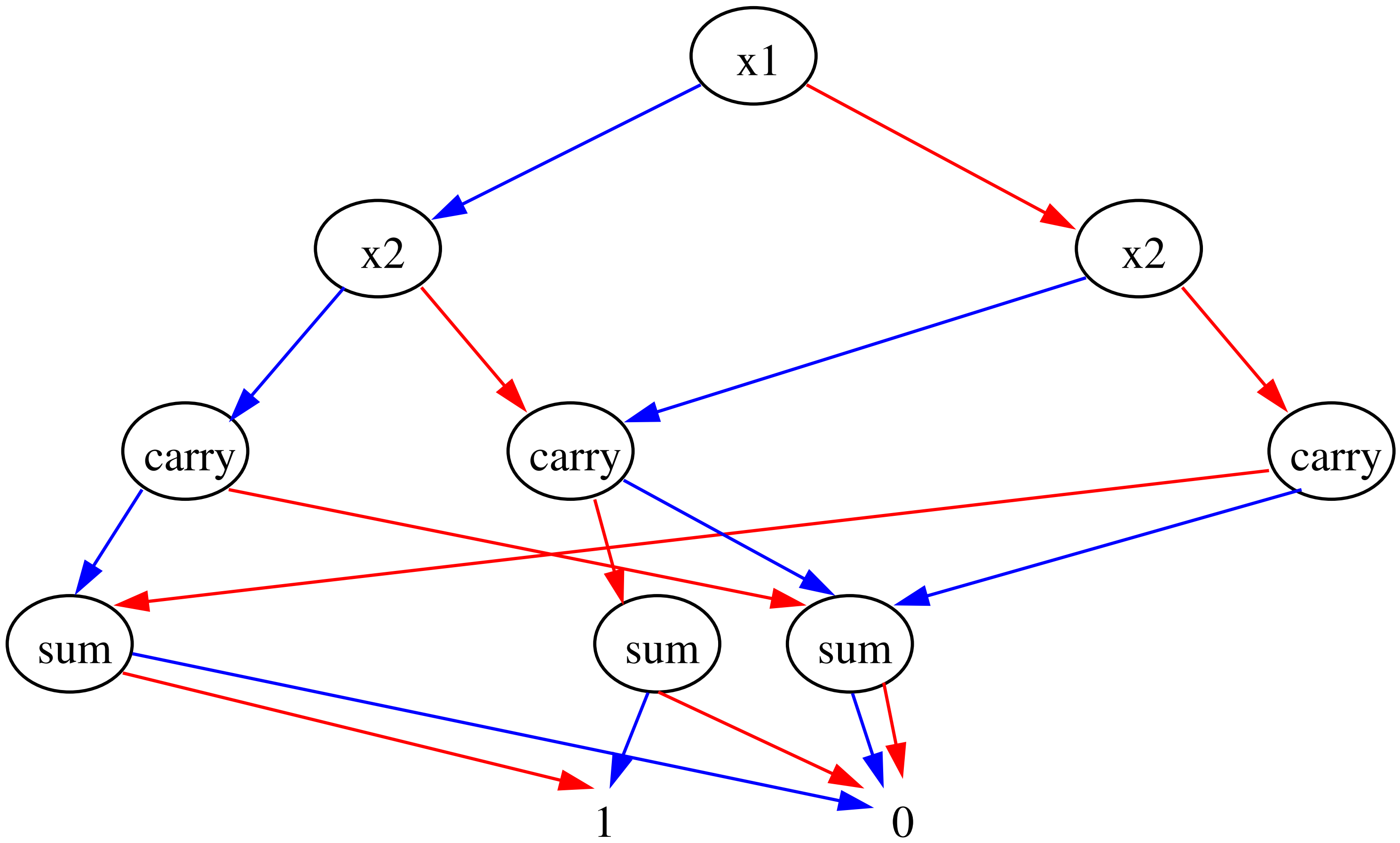
Merged the isomorphic sum-nodes.

Example 2: Eliminate isomorphic subgraphs (4/4)



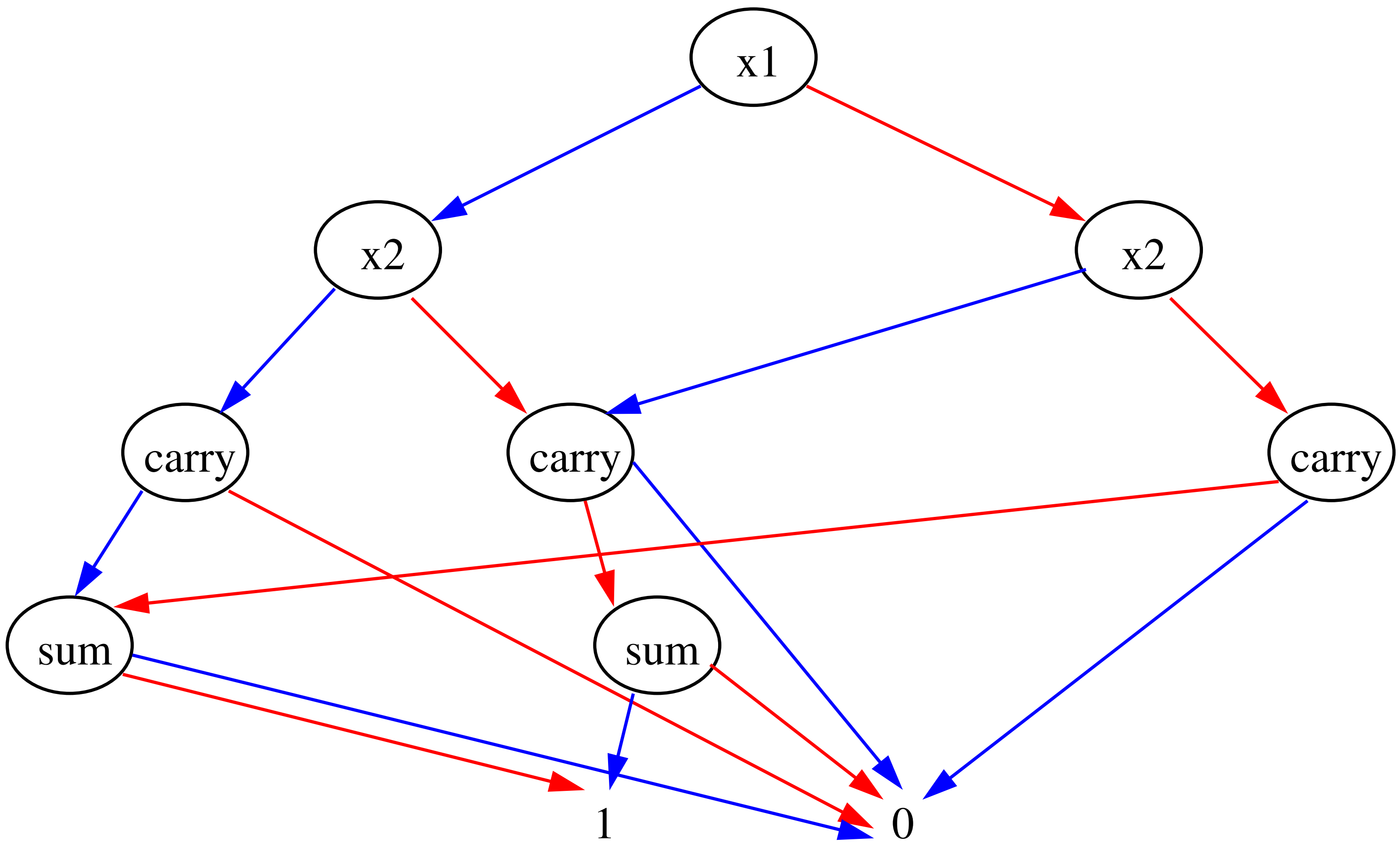
No isomorphic subgraphs are left $\rightarrow$ we are done.

Example 2: Remove redundant nodes (1/2)



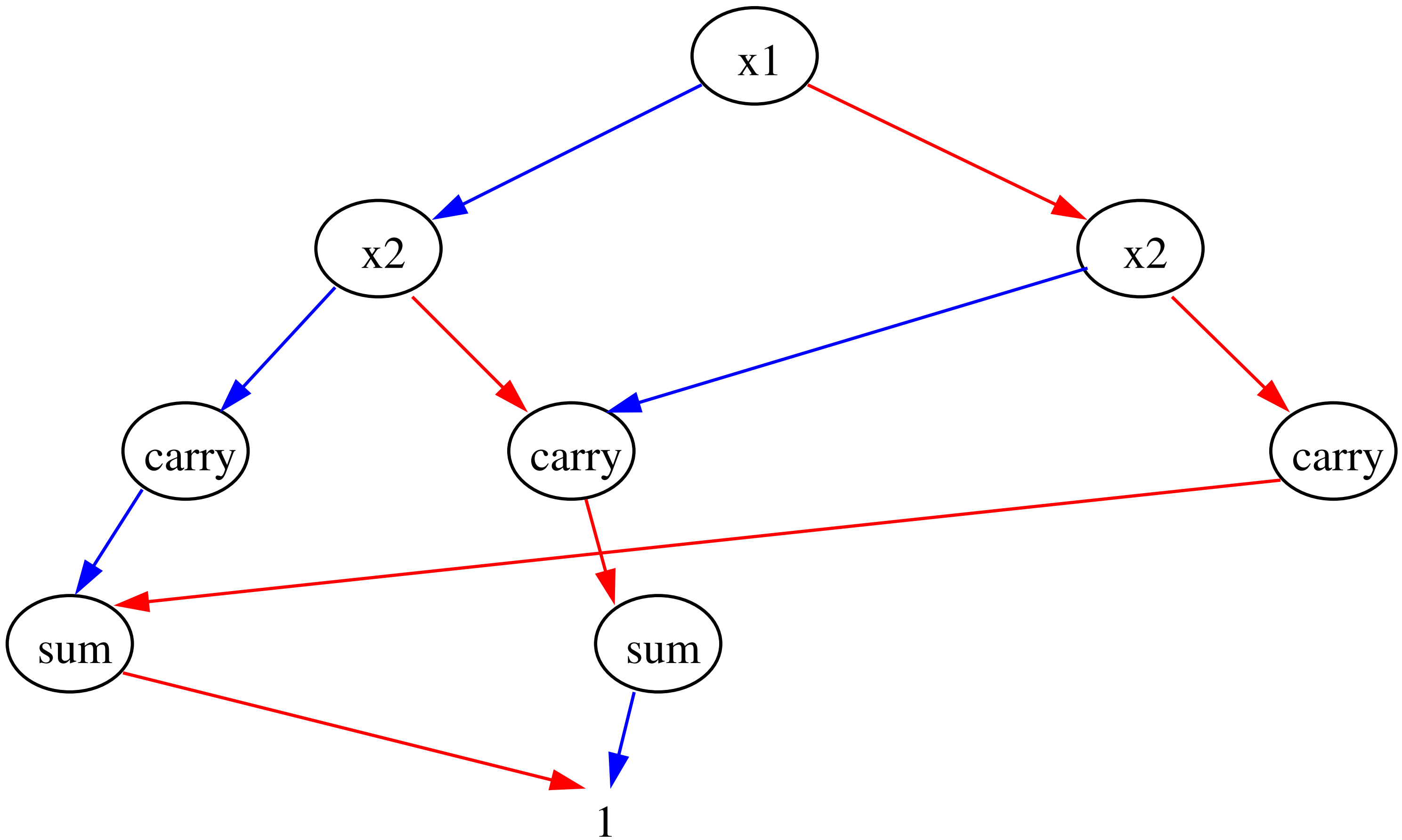
Both edges of the right *sum*-node point to 0.

Example 2: Remove redundant nodes (2/2)



No more redundant nodes → we are done.

Example 2: Omit 0-node



Optionally, we can remove the 0-node and edges leading to it, which makes the representation clearer.

Semantics of a BDD

Let B be a BDD with order $x_1 < \ldots < x_n$. Let P_B be the set of paths in B, leading from the root to the (unique) 1-node.

Let $p \in P_B$ be such a path, e.g. $x_{i_1} \xrightarrow{b_{i_1}} \ldots \xrightarrow{b_{i_m}} 1$. Then the "meaning" of p (denoted $[\![p]\!]$) is the conjunction of all x_{i_j} with $b_{i_j} = 1$ and all $\neg x_{i_j}$ with $b_{i_j} = 0$ (where $j = 1, \ldots, m$).

We then say that B represents the following PL formula:

$$F = \bigvee_{p \in P_B} [\![p]\!]$$

Preview

In the following, we shall investigate operations on BDDs that are needed for CTL model checking.

Construction of a BDD (from a PL formula)

Equivalence check

Intersection, complement, union

Relations, computing predecessors

Propositional logic with constants

In the following, we will consider formulae of propositional logic (PL), extended with the constants $\mathbf{0}$ and $\mathbf{1}$, where:

$\mathbf{0}$ is an unsatisfiable formula;

$\mathbf{1}$ is a tautology.

Substitution

Let F and G be formulae of PL (possibly with constants), and let x be an atomic proposition.

$F[x/G]$ denotes the PL formula obtained by replacing each occurrence of x in F by G.

In particular, we will consider formulae of the form $F[x/0]$ and $F[x/1]$.

Example: Let $F = x \wedge y$. Then $F[x/1] = 1 \wedge y \equiv y$ and $F[x/0] = 0 \wedge y \equiv 0$.

If-then-else

Let us introduce a new, ternary PL operator. We shall call it *ite* (if-then-else).

Note: *ite* does not extend the expressiveness of PL, it is simply a convenient shorthand notation.

Let F, G, H be PL formulae. We define

$$ite(F, G, H) := (F \wedge G) \vee (\neg F \wedge H).$$

The set of INF formulae (if-then-else normal form) is inductively defined as follows:

$\mathbf{0}$ and $\mathbf{1}$ are INF formulae;

if x is an atomic proposition and G, H are INF formulae, then $ite(x, G, H)$ is an INF formula.

Shannon partitioning

Let F be a PL formula and x an atomic proposition. We have:

$$F \equiv ite(x, F[x/1], F[x/0])$$

Proof: In the following, G denotes the right-hand side of the equivalence above. Let ν be a valuation s.t. $\nu \models F$. Either $\nu(x) = 1$, then ν is also a model of $F[x/1]$ and of x and therefore also of G. The case $\nu(x) = 0$ is analogous. For the other direction, suppose $\nu \models G$. Then either $\nu(x) = 1$ and the "rest" of ν is a model of $F[x/1]$. Then, however, ν will be a model for any formula in which some of the ones in $F[x/1]$ are replaced by x, in particular also for F. The case $\nu(x) = 0$ is again analogous.

Remark: G is called the Shannon partitioning of F.

Corollary: Every PL formula is equivalent to an INF formula.
(Proof: apply the equivalence above multiple times.)

Construction of BDDs

We can now solve our first BDD-related problem: Given a PL formula F and some ordering of variables $<$, construct a BDD w.r.t. $<$ that represents F.

If F does not contain any atomic propositions at all, then either $F \equiv \mathbf{0}$ or $F \equiv \mathbf{1}$, and the corresponding BDD is simply the corresponding leaf node.

Otherwise, let x be the smallest variable (w.r.t. $<$) occurring in F. Construct BDDs B_0 and B_1 for $F[x/1]$ and $F[x/0]$, respectively (these formulae have one variable less than F).

Because of the Shannon partitioning, F is representable by a binary decision *graph* whose root is labelled by x and whose subtrees are B_0 and B_1. To obtain a BDD, we check whether B_0 and B_1 are isomorpic; if yes, then F is represented by B_0. Otherwise we merge all isomorphic subtrees in B_0 and B_1.

Example: BDD construction

Consider again the formula from Example 2:

$$F \equiv \big(carry \leftrightarrow (x_1 \wedge x_2)\big) \wedge \big(sum \leftrightarrow (x_1 \vee x_2) \wedge \neg carry\big)$$

We have, e.g.:

$$F[x_1/0] \equiv \neg carry \wedge (sum \leftrightarrow x_2)$$

$$F[x_1/1] \equiv (carry \leftrightarrow x_2) \wedge (sum \leftrightarrow \neg carry)$$

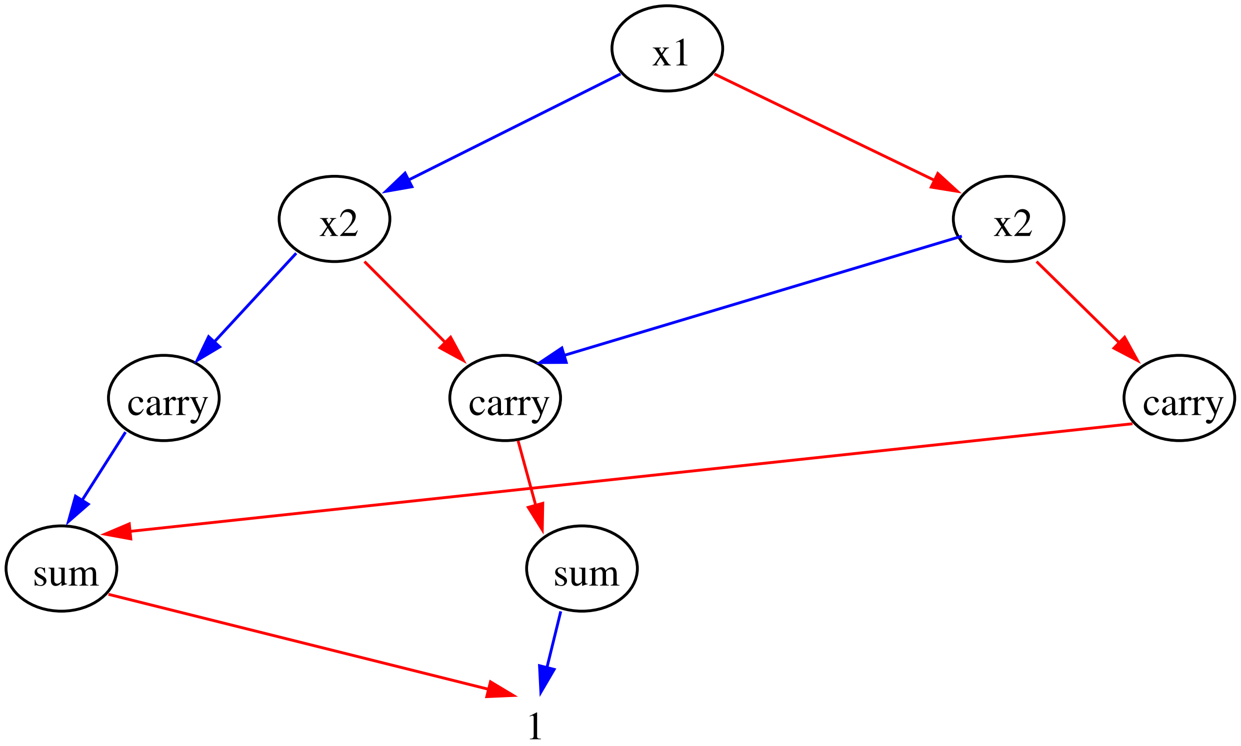
$$F[x_1/0][x_2/0] \equiv \neg carry \wedge \neg sum$$

$$F[x_1/0][x_2/1] \equiv F[x_1/1][x_2/0] \equiv \neg carry \wedge sum$$

$$F[x_1/1][x_2/1] \equiv carry \wedge \neg sum$$

Example: BDD construction

By applying the construction, we obtain the same BDD as before:



Remark: Obviously, we can also obtain an INF formula from each BDD.

BDDs are unique

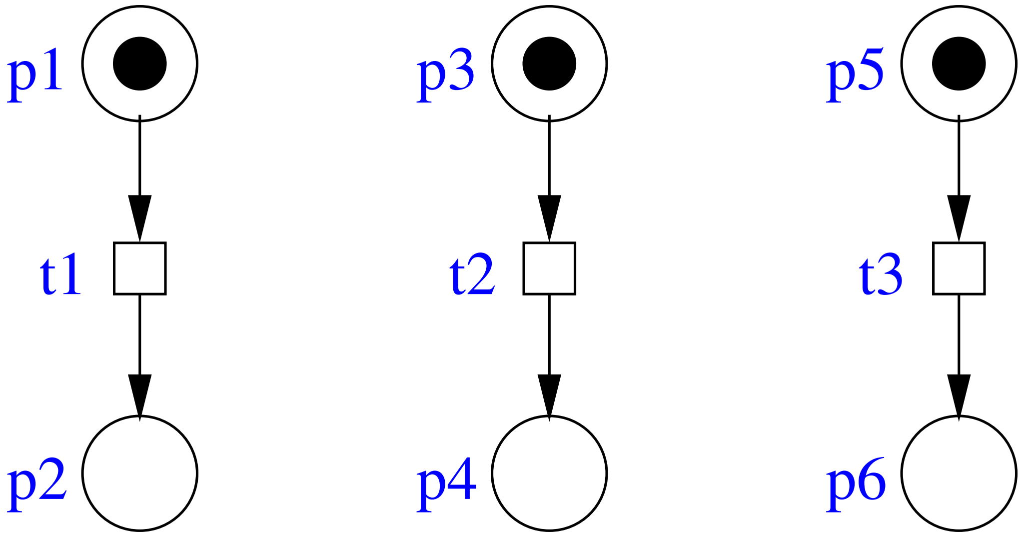
Remark: The result of the previously given construction is *unique* (up to isomorphism).

In other words, given F and $<$, there is (up to isomorphism) exactly one BDD that respects $<$ and represents F.

Remark: Different orderings still lead to different BDDs.
(possibly with vastly different sizes!)

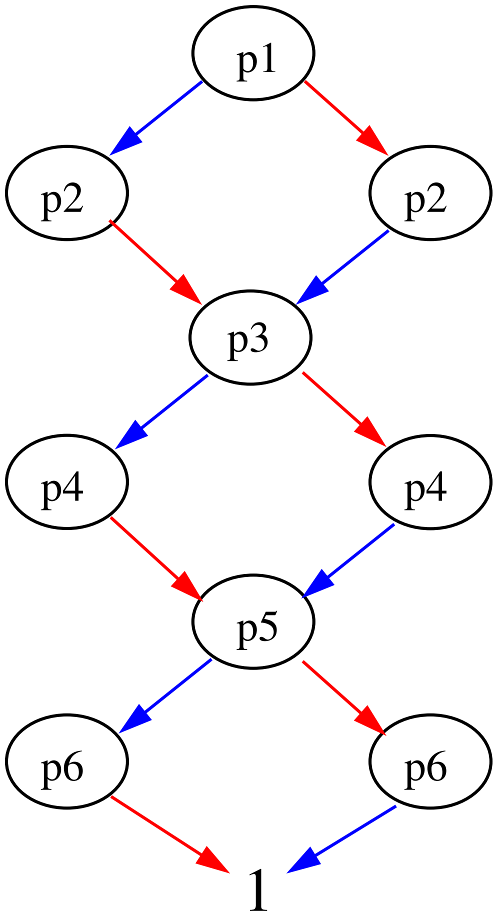
Example: Variable orderings

Recall Example 1 (the Petri net), and let us construct a BDD representing the reachable markings:



Remark: P_1 is marked iff P_2 is not, etc.

The corresponding BDD for the ordering $p_1 < p_2 < p_3 < p_4 < p_5 < p_6$:



Remarks:

If we increase the number of components from 3 to n (for some $n \geq 0$), the size of the corresponding BDD will be linear in n.

In other words, a BDD of size n can represent 2^n (or even more) valuations.

However, the size of a BDD strongly depends on the ordering!
Example: Repeat the previous construction for the ordering

$$p_1 < p_3 < p_5 < p_2 < p_4 < p_6.$$

Equivalence test

To implement CTL model checking, we need a test for equivalence between BDDs (e.g., to check the termination of a fixed-point computation).

Problem: Given BDDs B and C (w.r.t. the same ordering) do B and C represent equivalent formulae?

Solution: Test whether B and C are isomorphic.

Special cases:

Unsatisfiability test: Check if the BDD consists just of the $\mathbf{0}$ leaf.

Tautology test: Check if the BDD consists just of the $\mathbf{1}$ leaf.

Implementing BDDs with hash tables

Suppose we want to write an application in which we need to manipulate multiple BDDs.

Efficient BDDs implementations exploit the uniqueness property by storing all BDD nodes in a hash table. (Recall that each node is in fact the root of some BDD.)

Each BDD is then simply represented by a pointer to its root.

Initially, the hash table has only two unique entries, the leaves $\mathbf{0}$ and $\mathbf{1}$.

Every other node is uniquely identified by the triple (x, B_0, B_1), where x is the atomic proposition labelling that node and B_0, B_1 are the subtrees of that node, represented by pointers to their respective roots.

Usually, one implements a function $mk(x, B_0, B_1)$ that checks whether the hash table already contains such a node; if yes, then the pointer to that node is returned, otherwise a new node is created.

A multitude of BDDs is then stored as a “forest” (a DAG with multiple roots).

Problem: garbage collection (by reference counting)

Equivalence test II

Let us reconsider the equivalence-checking problem.
(Given two BDDs *B* and *C*, do *B* and *C* represent equivalent formulae?)

If *B* and *C* are stored in hash tables (as described previously), then *B* and *C* are representable by pointers to their roots.

Due to the uniqueness property, one then simply has to check whether the pointers are the same (a constant-time procedure).

Logical operations I: Complement

Let F be a PL formula and B a BDD representing F.

Problem: Compute a BDD for $\neg F$.

Solution: Exchange the two leaves of B.

(Caution: This is not quite so simple with the hash-table implementation.)

Logical operations II: Disjunction/union

Let F, G be PL formulae and B, C the corresponding BDDs (with the same ordering).

Problem: Compute a BDD for $F \vee G$ from B and C.

We have the following equivalence:

$$F \vee G \equiv ite(x, (F \vee G)[x/1], (F \vee G)[x/0]) \equiv ite(x, F[x/1] \vee G[x/1], F[x/0] \vee G[x/0])$$

If x is the smallest variable occurring in either F or G, then $F[x/1], F[x/0], G[x/1], G[x/0]$ are either the children of the roots of B and C (or the roots themselves).

We construct a BDD for disjunction according to the following, recursive strategy:

If B and C are equal, then return B.

If either B or C are the $\mathbf{1}$ leaf, then return $\mathbf{1}$.

If either B or C are the $\mathbf{0}$ leaf, then return the other BDD.

Otherwise, compare the two variables labelling the roots of B and C, and let x be the smaller among the two (or the one labelling both).

If the root of B is labelled by x, then let B_1, B_0 be the subtrees of B; otherwise, let $B_1, B_0 := B$. We define C_1, C_0 analogously.

Apply the strategy recursively to the pairs B_1, C_1 and B_0, C_0, yielding BDDs E and F. If $E = F$, return E, otherwise $mk(x, E, F)$.

Logical operations III: Intersection

Let F, G be PL formulae and B, C the corresponding BDDs (with the same ordering).

Problem: Compute a BDD for $F \wedge G$ from B and C.

Solution: Analogous to union, with the rules for $\mathbf{1}$ and $\mathbf{0}$ leaves adapted accordingly.

Complexity: With dynamic programming: $\mathcal{O}(|B| \cdot |C|)$ (every pair of nodes at most once).

Computing predecessors

In the following, we derive a strategy for computing the set

$$pre(M) = \{\, s \mid \exists s' \colon (s, s') \in \rightarrow \;\wedge\; s' \in M \,\}.$$

Note that the relation $\rightarrow$ is a subset of $S \times S$ whereas $M \subset S$.

We represent M by a BDD with variables $y_1, \ldots, y_m$.

$\rightarrow$ will be represented by a BDD with variables $x_1, \ldots, x_m$ and $y_1, \ldots, y_m$ (states "before" and "after").

Remark: Every BDD for M is at the same time a BDD for $S \times M$!

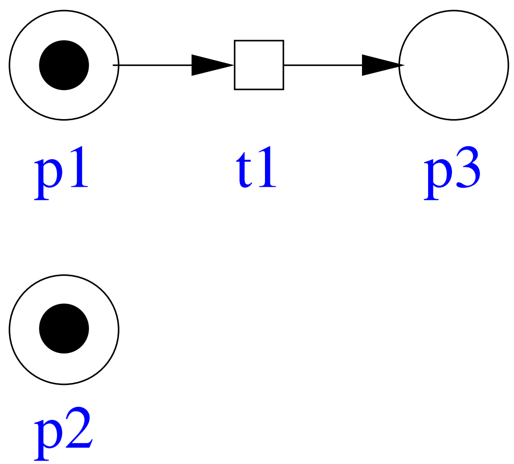
Thus, we can rewrite $pre(M)$ as follows:

$$\{ s \mid \exists s' : (s, s') \in \rightarrow \ \cap \ (S \times M) \}$$

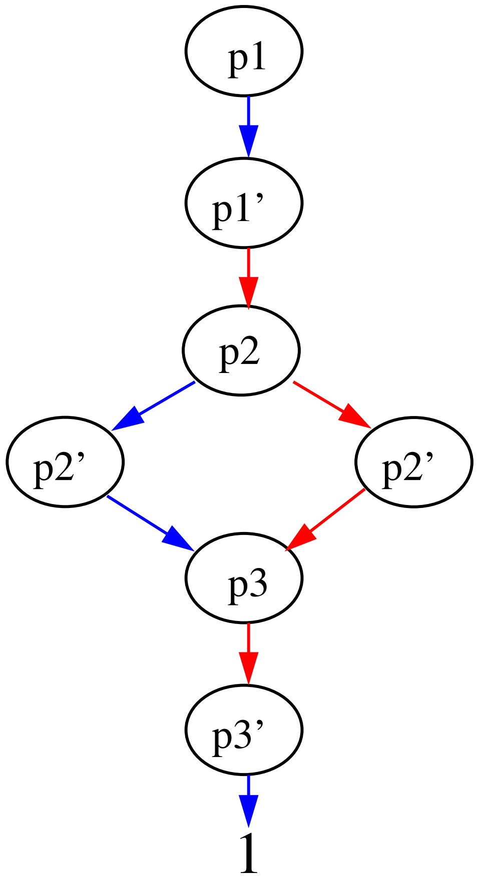
Then, pre reduces to the operations intersection and existential abstraction.

Example

Let us consider the following Petri net with just one transition:



The BDD F_{t_1} describes the effect of t_1, where p_1, p_2, p_3 describe the state *before* and p'_1, p'_2, p'_3 the state *after* firing t_1.



Existential abstraction

Existential abstraction w.r.t. an atomic proposition x is defined as follows:

$$\exists x : F \quad \equiv \quad F[x/0] \vee F[x/1]$$

I.e., $\exists x :\ F$ is true for those valuations that can be extended with a value for x in such a way that they become models for F.

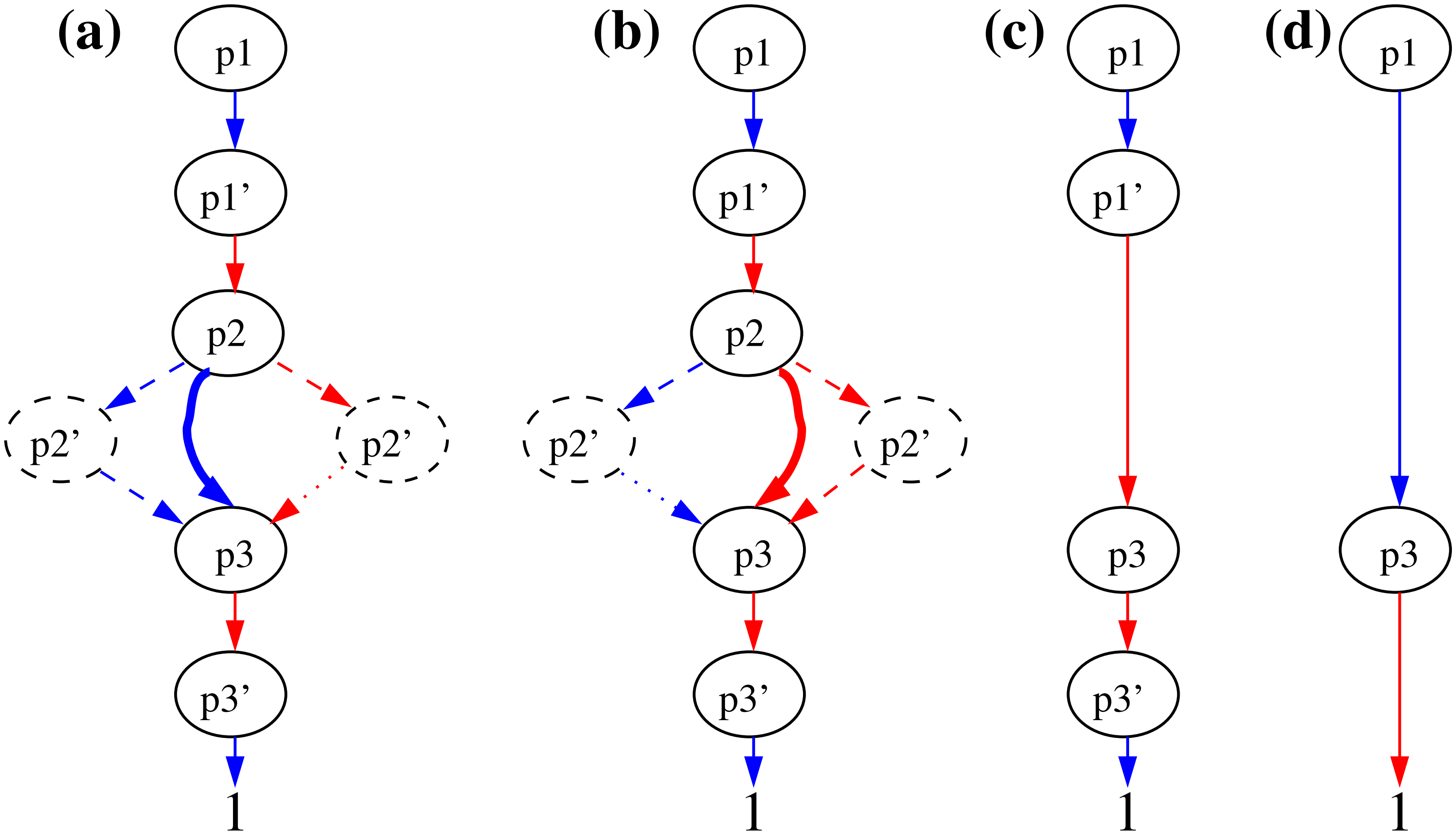
Example: Let $F \equiv (x_1 \wedge x_2) \vee x_3$. Then

$$\exists x_1 :\ F \equiv F[x_1/0] \vee F[x_1/1] \equiv (x_3) \vee (x_2 \vee x_3) \equiv x_2 \vee x_3$$

By extension, we can consider existential abstraction over sets of atomic propositions (abstract from each of them in turn).

Example: Existential abstraction

(a) $F_{t_1}[p'_2/1]$; (b) $F_{t_1}[p'_2/0]$; (c) $\exists p'_2 : F_{t_1}$; (d) $\exists p'_1, p'_2, p'_3 : F_{t_1}$



BDDs with complement arcs

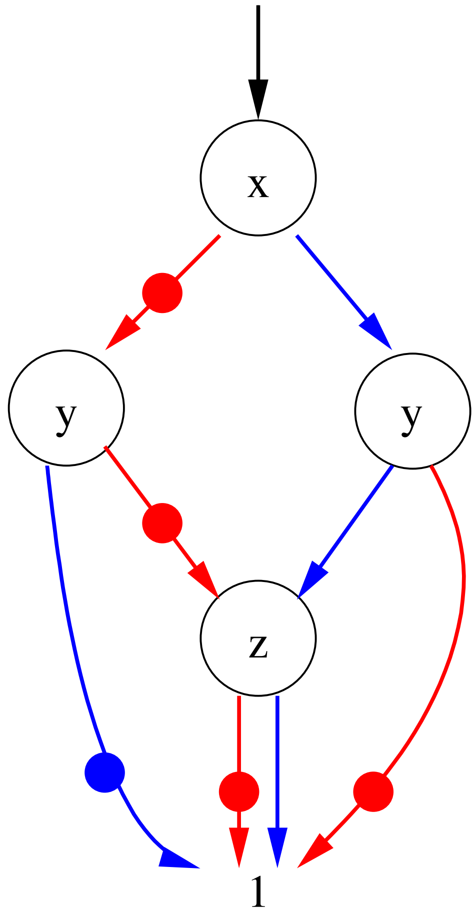
Implementation with hash tables makes negation a costly operation.

Therefore, BDD libraries often use a modification of BDDs, called BDDs with complement arcs (CBDDs).

In a CBDD, every edge is equipped with an additional bit. If the bit is true, then it means that the edge should really lead to the negation of its target.

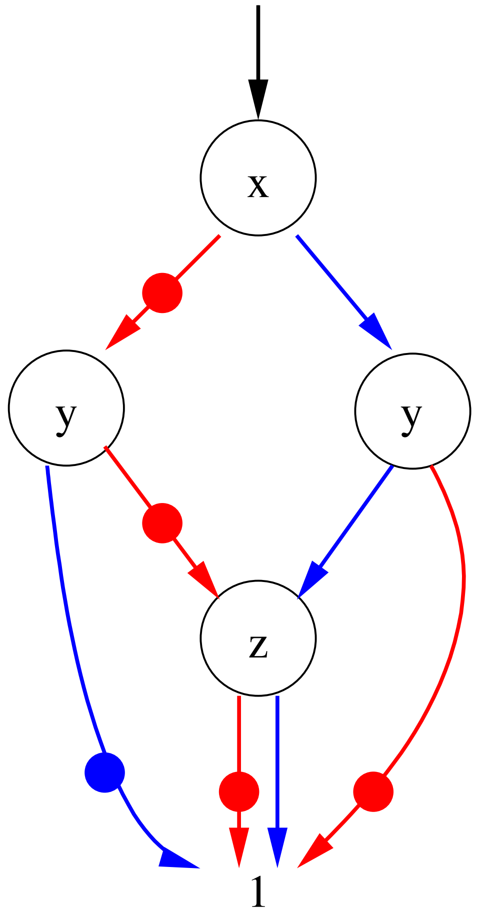
Representation: if the bit is set, we put a filled circle onto the edge.

CBDD: Example



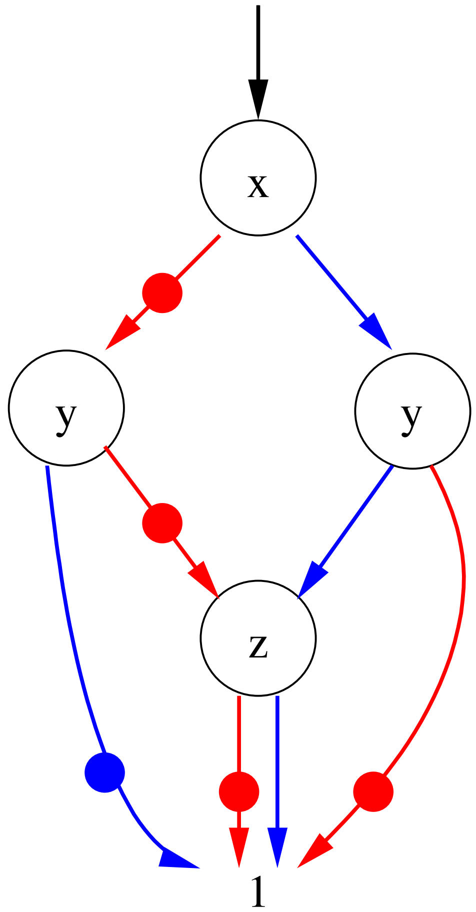
The red arc leaving the *z*-labelled node has its negation bit set, it therefore effectively leads to **0**.

CBDD: Example



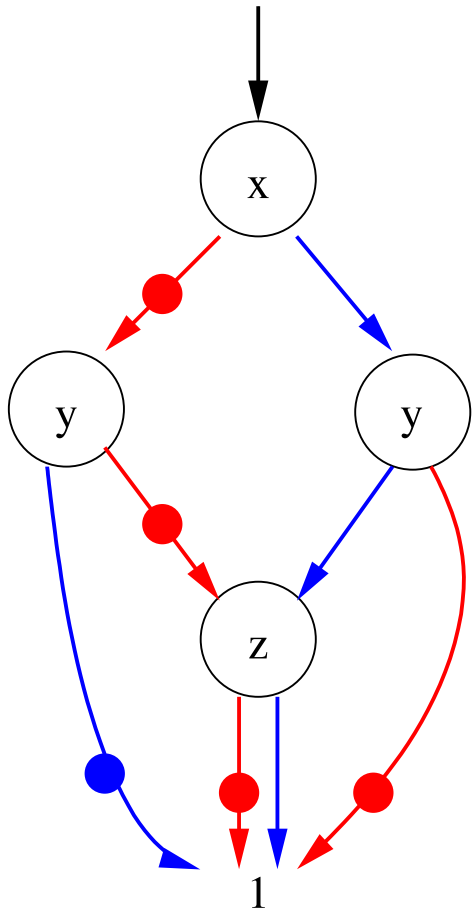
For this reason, the *z*-labelled node is *not redundant*.
The **0**-leaf can be omitted altogether.

CBDD: Example



The left *y*-labelled node represents the formula $\neg y \land \neg z$.

CBDD: Example



The pointer to the root is also equipped with a negation bit (false in this case).

Remarks

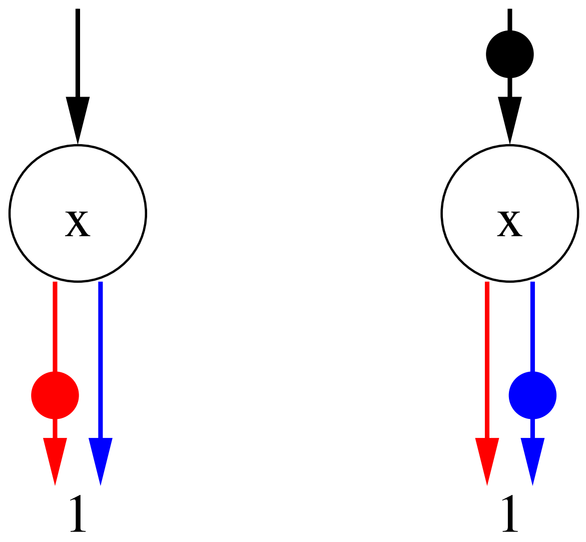
A valuation ν is a model of the formula represented by a CBDD iff the number of negations on the path corresponding to ν is *even* (including the pointer to the root).

Negation with CBDDs: trivial, invert the negation bit of the pointer to the root (constant-time operation).

Implementation (e.g., in the CUDD library): coded into the least significant bit of the pointer

Problem: CBDDs (as presented until now) are not unique!

CBDDs are not (yet) unique



Both of the CBDDs shown above represent the formula x.

Canonical form

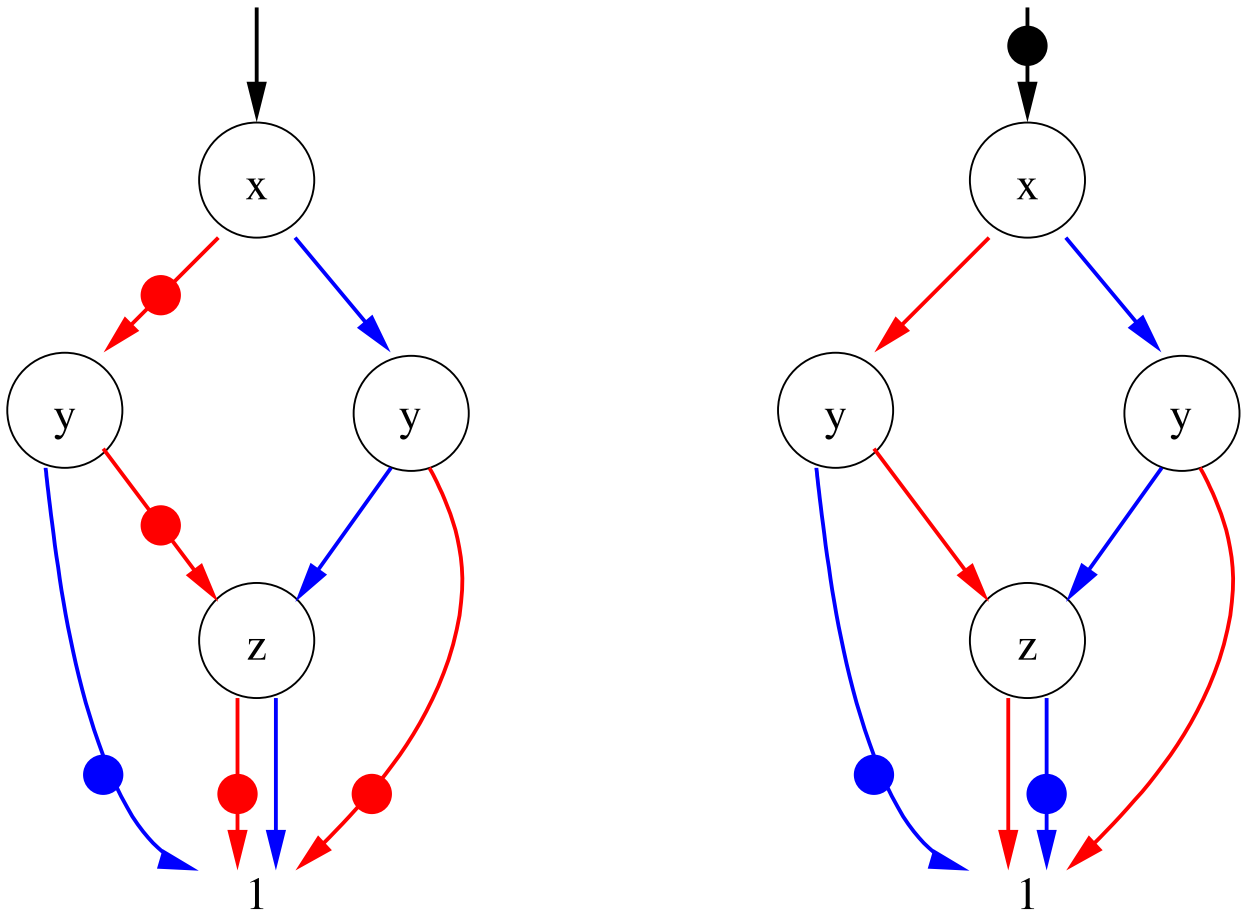
To ensure uniqueness, one can additionally prohibit negation bits on 0-labelled edges.

For this, we exploit the following equivalence:

$$ite(x, F, \neg G) \equiv \neg ite(x, \neg F, G)$$

Given any CBDD, one can eliminate negated 0-labelled edges by inverting all the negation bits on those edges that are incident with its source node (starting at the leaves, finishing with the root).

Canonical form



The CBDD shown on the right represents the same formula as before (on the lect) and does not have any negated 0-labelled edges.

LTL with BDDs

Question: Can one implement also LTL model checking using BDDs?

Answer: Yes and no (worst-case: quadratical, but works ok in practice).

Problems: BDD not compatible with depth-first search, combination with partial-order reduction difficult.

Symbolic algorithms for LTL

Idea: Find non-trivial SCCs with an accepting state, then search backwards for an initial state.

Algorithms: Emerson-Lei (EL), OWCTY

Emerson-Lei (1986)

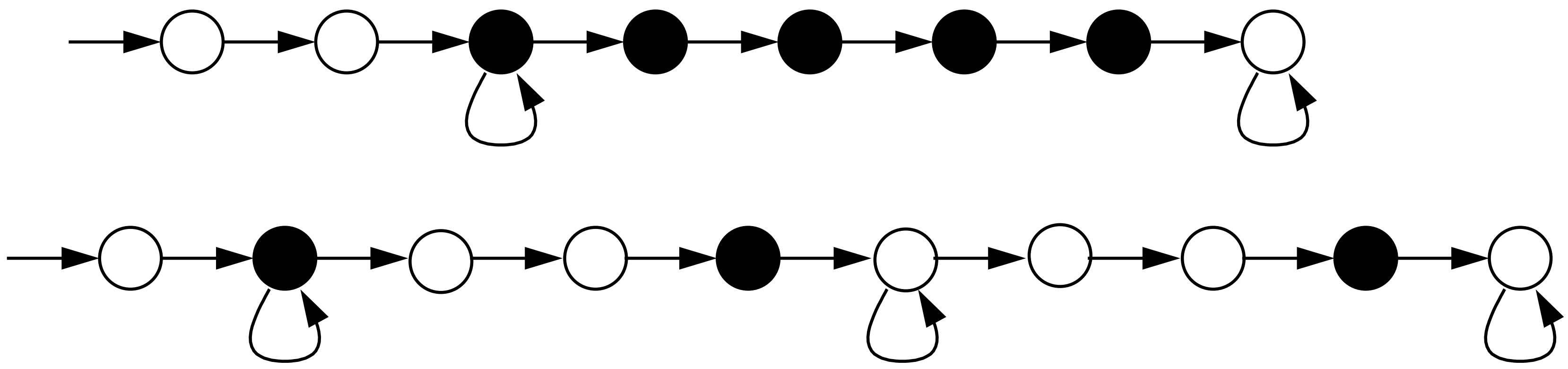
1. Assign to M the set of all states.
2. Let $B := M \cap F$.
3. Compute the set C of states that can reach elements of B.
4. Let $C := pre(C)$.
5. Let $M := C$.
6. If M has changed, then go to step 2, otherwise stop.

One-Way-Catch-Them-Young

(Hardin et al 1997, Fisler et al 2001)

Like EL, but step 4 is repeated until C does not change any more.

EL and OWCTY



In the upper case, OWCTY is superior, in the lower case EL is.

In practice, OWCTY appears to work better.

Part 13: Abstraction

Example 1 (loop)

Consider the following program with three numeric variables x, y, z.

ℓ_1: `y = x+1;`

ℓ_2: `z = 0;`

ℓ_3: `while (z < 100) z = z+1;`

ℓ_4: `if (y < x) error;`

Question: Is the error location reachable?

Example 2 (Sorting)

Another program with three numeric variables x, y, z.

ℓ_1: `if x > y then  swap x,y  else  skip;`

ℓ_2: `if y > z then  swap y,z  else  skip;`

ℓ_3: `if x > y then  swap x,y  else  skip;`

ℓ_4: `skip`

Assumption: initially, x, y, z are all different

Question: Are x, y, z sorted in ascending order when reaching ℓ_4?

Question 3 (Device driver)

C code for Windows device driver

Operations on a semaphor: lock, release

Lock and release must be used alternatingly

Abstraction

Idea: throw away (abstract from) “unimportant” information

Handling *infinite* state spaces

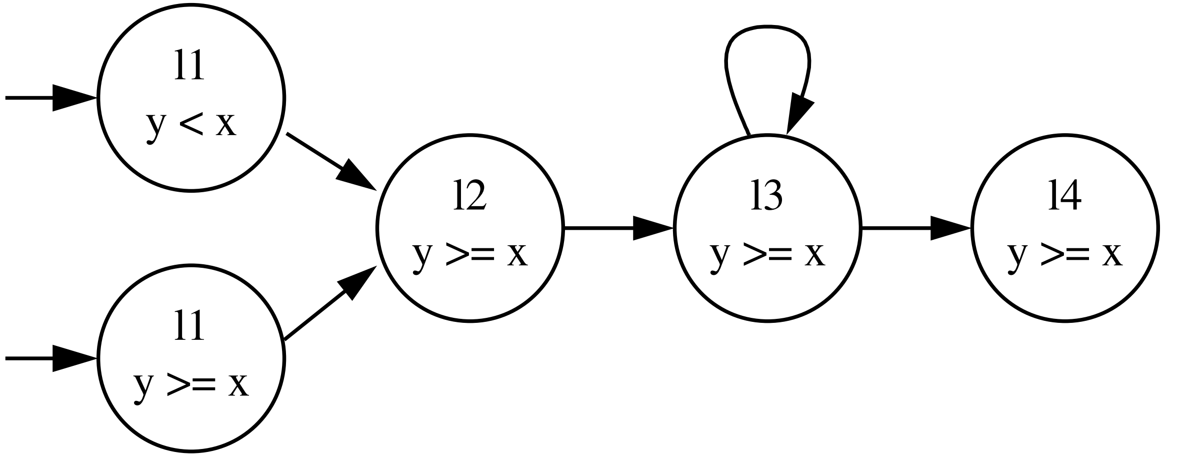
Reduce (large) finite problems to smaller ones

Alternative point of view: merge “equivalent” states

Example 1

Omit concrete values of `x,y,z`; retain only the following information: program counter, predicate $y < x$

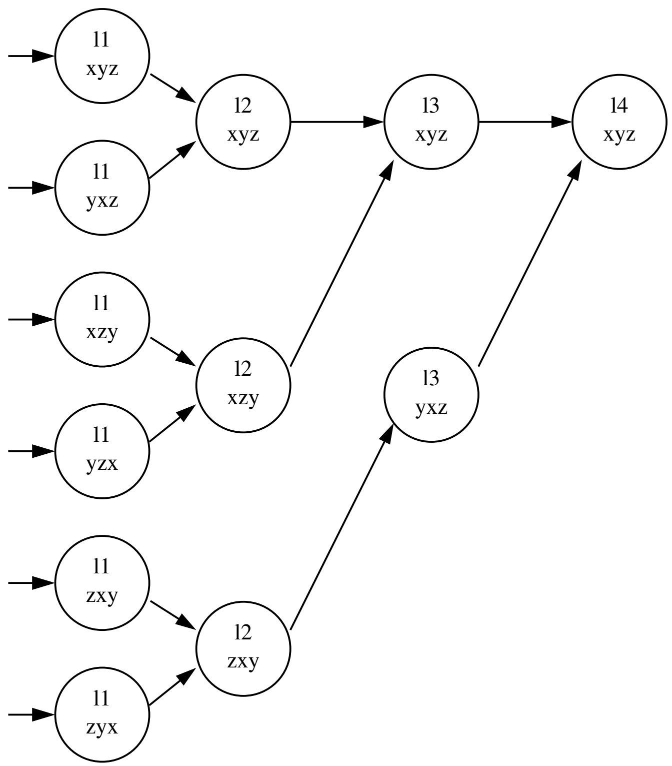
Resulting (abstract) Kripke structure:



Result: ℓ_4 is reachable only with $y \geq x$; the error will not happen.

Example 2

Omit concrete values of `x`, `y`, `z`; retain only program counter and permutation of x, y, z



Result: ℓ_4 is reachable only with xyz; no error.

Questions: What is the logical relation between the original programs and their abstract versions? What do the abstract versions really say about the original programs?

> In Example 1, the error is unreachable in both the original and the abstract version.
>
> However, in Example 1, the original structure terminates but the abstract version does not.

Which conditions must hold for the abstract structure in order to draw meaningful conclusions about the original structure?

Simulation

Let $\mathcal{K}_1 = (S, \rightarrow_1, s_0, AP, \nu)$ and $\mathcal{K}_2 = (T, \rightarrow_2, t_0, AP, \mu)$ be two Kripke structures (S, T are possibly *infinite*), and let $H \subseteq S \times T$ be a relation.

H is called a simulation from $\mathcal{K}_1$ to $\mathcal{K}_2$ iff

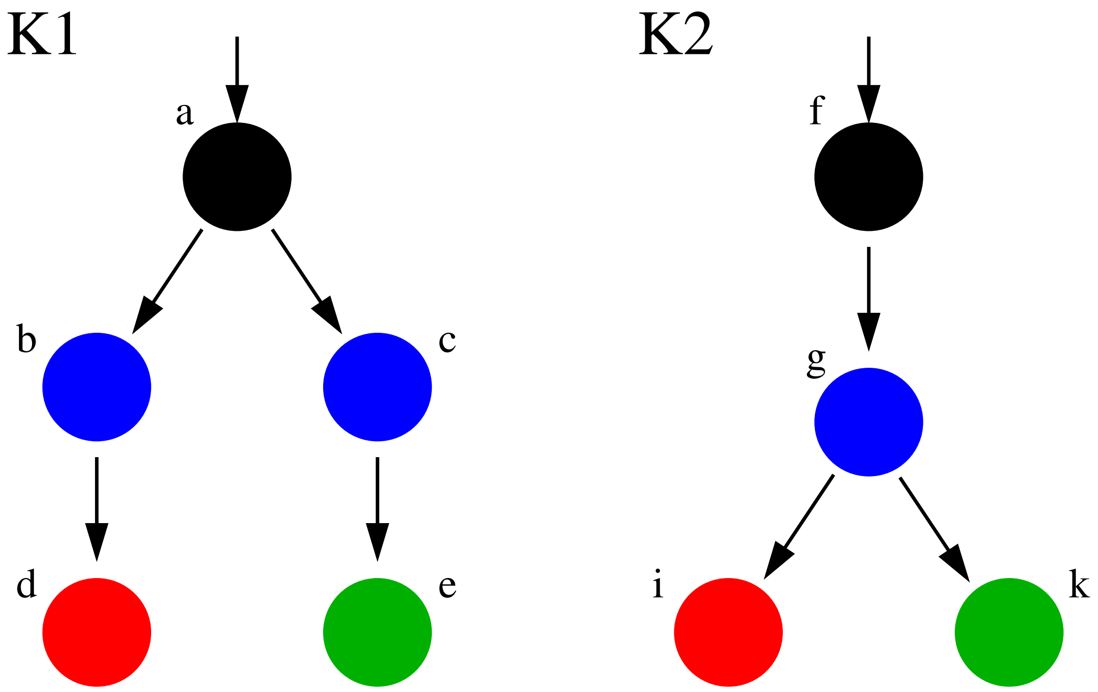
(i) $(s_0, t_0) \in H$;

(ii) for all $(s, t) \in H$ we have: $\nu(s) = \mu(t)$;

(iii) if $(s, t) \in H$ and $s \rightarrow_1 s'$, then there exists t' such that $t \rightarrow_2 t'$ and $(s', t') \in H$.

We say: $\mathcal{K}_2$ simulates $\mathcal{K}_1$ (written $\mathcal{K}_1 \leq \mathcal{K}_2$) if such a simulation H exists.

Intuition: $\mathcal{K}_2$ can do anything that is possible in $\mathcal{K}_1$.



$\mathcal{K}_2$ simulates $\mathcal{K}_1$ (with $H = \{(a, f), (b, g), (c, g), (d, i), (e, k)\}$).

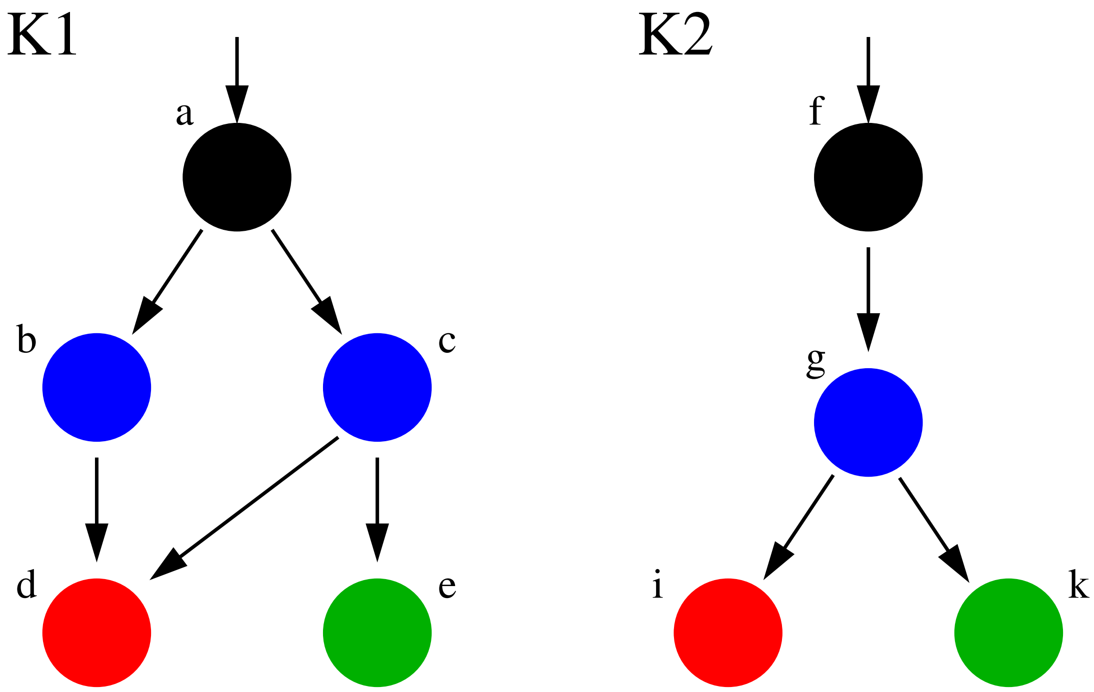
However,: $\mathcal{K}_1$ does *not* simulate $\mathcal{K}_2$!

Bisimulation

A relation H is called a bisimulation between $\mathcal{K}_1$ and $\mathcal{K}_2$ iff H is a simulation from $\mathcal{K}_1$ to $\mathcal{K}_2$ and $\{ (t, s) \mid (s, t) \in H \}$ is a simulation from $\mathcal{K}_2$ to $\mathcal{K}_1$.

We say: $\mathcal{K}_1$ and $\mathcal{K}_2$ are bisimilar (written $\mathcal{K}_1 \equiv \mathcal{K}_2$) iff such a relation H exists.

Careful: In general, $\mathcal{K}_1 \leq \mathcal{K}_2$ and $\mathcal{K}_2 \leq \mathcal{K}_1$ do *not* imply $\mathcal{K}_1 \equiv \mathcal{K}_2$!



(Bi-)Simulation and model checking

Let $\mathcal{K}_1 \leq \mathcal{K}_2$ and ϕ an LTL formula.

Then we have: $\mathcal{K}_2 \models \phi$ implies $\mathcal{K}_1 \models \phi$.

Let $\mathcal{K}_1 \equiv \mathcal{K}_2$ and ϕ a CTL or LTL formula.

Then we have: $\mathcal{K}_1 \models \phi$ iff $\mathcal{K}_2 \models \phi$.

Existential abstraction

Let $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ be a Kripke structure (concrete structure).

Let $\approx$ be an equivalence relation on S such that for all $s \approx t$ we have $\nu(s) = \nu(t)$ (we say: $\approx$ respects ν).

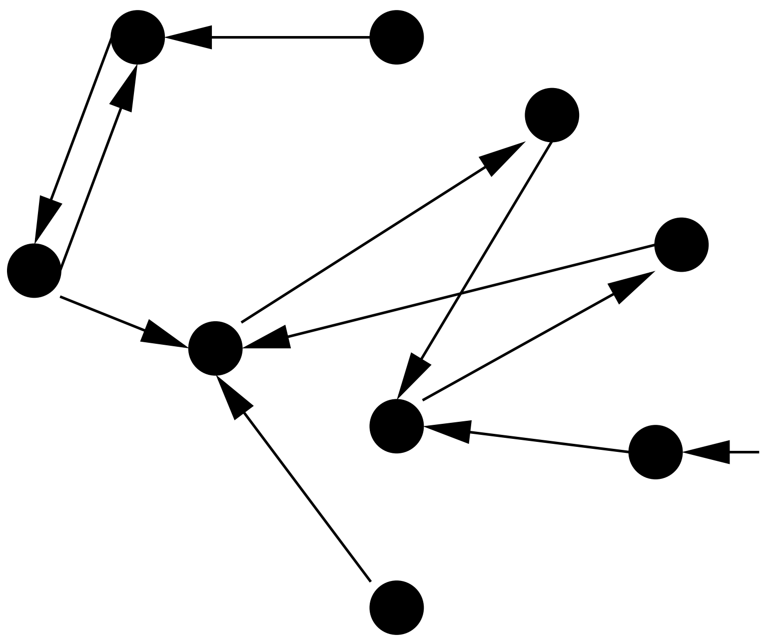
Let $[s] := \{\, t \mid s \approx t \,\}$ denote the equivalence class of s;
$[S]$ denotes the set of all equivalence classes.

The abstraction of S w.r.t. $\approx$ denotes the structure $\mathcal{K}' = ([S], \rightarrow', [r], AP, \nu')$, where

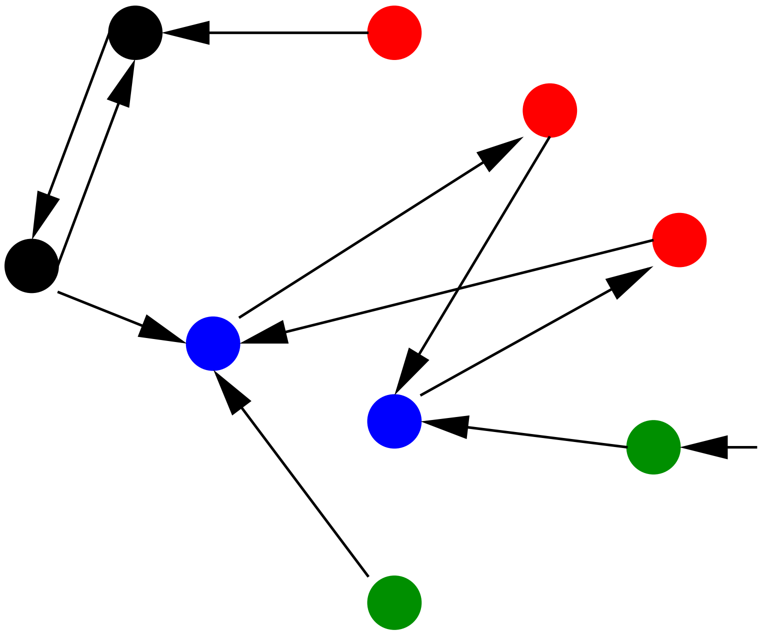
- $[s] \rightarrow' [t]$ for all $s \rightarrow t$;
- $\nu'([s]) = \nu(s)$ (this is well-defined!).

Example

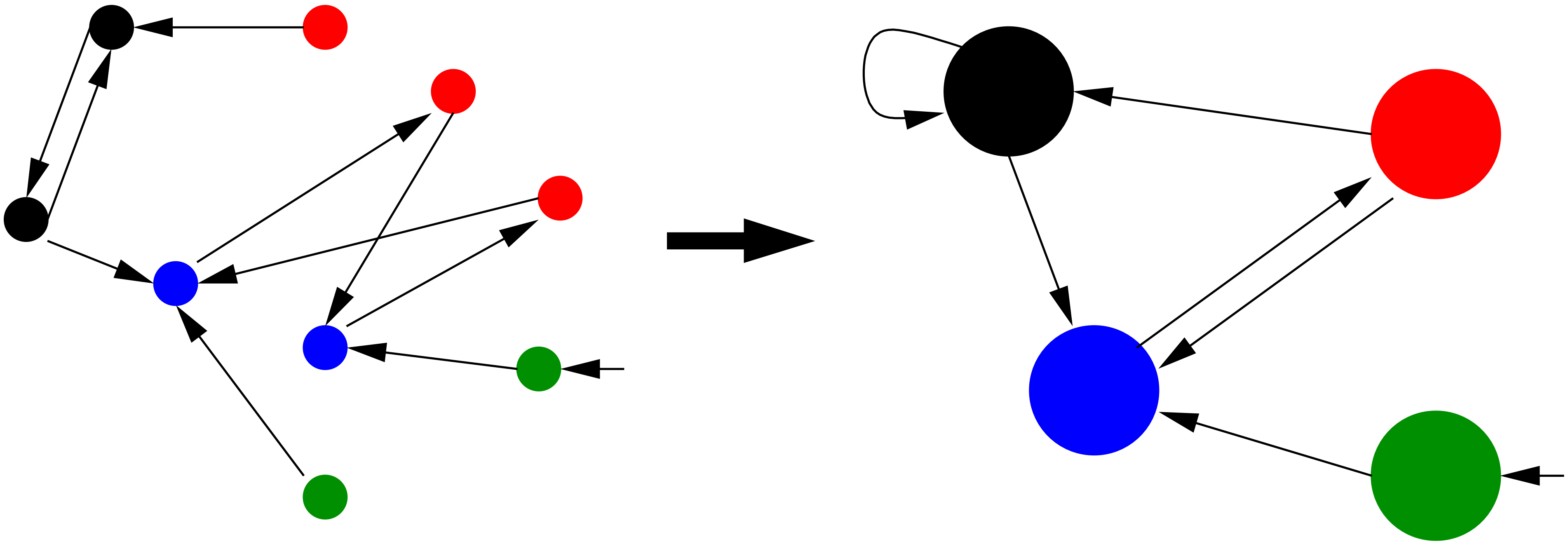
Consider the Kripke structure below:



States partitioned into equivalence classes:



Abstract structure obtained by quotienting:

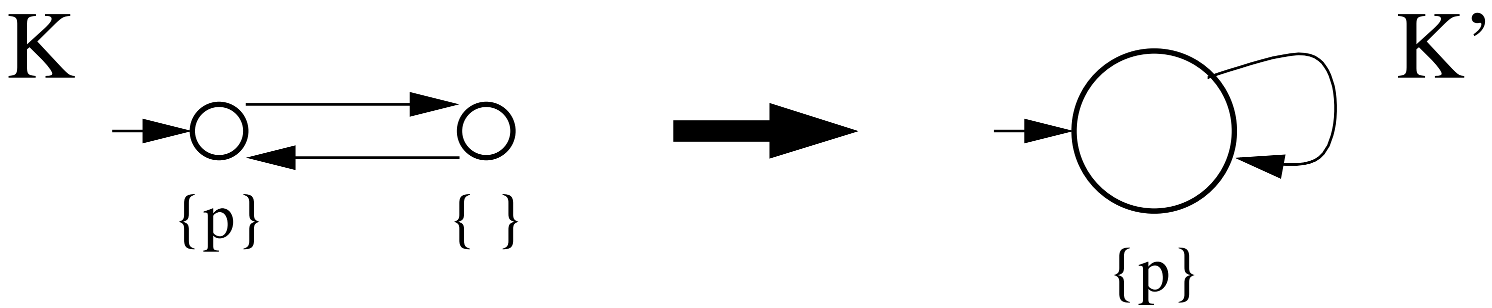


Let $\mathcal{K}'$ be a structure obtained by abstraction from $\mathcal{K}$.

Then $\mathcal{K} \leq \mathcal{K}'$ holds.

Thus, if $\mathcal{K}'$ satisfies some LTL formula, so does $\mathcal{K}$.

What happens if $\approx$ does *not* respect ν?



Then $\mathcal{K} \not\preceq \mathcal{K}'$ does not hold.

Example: The abstraction satisfies $\mathbf{G}\, p$, the concrete system does not.

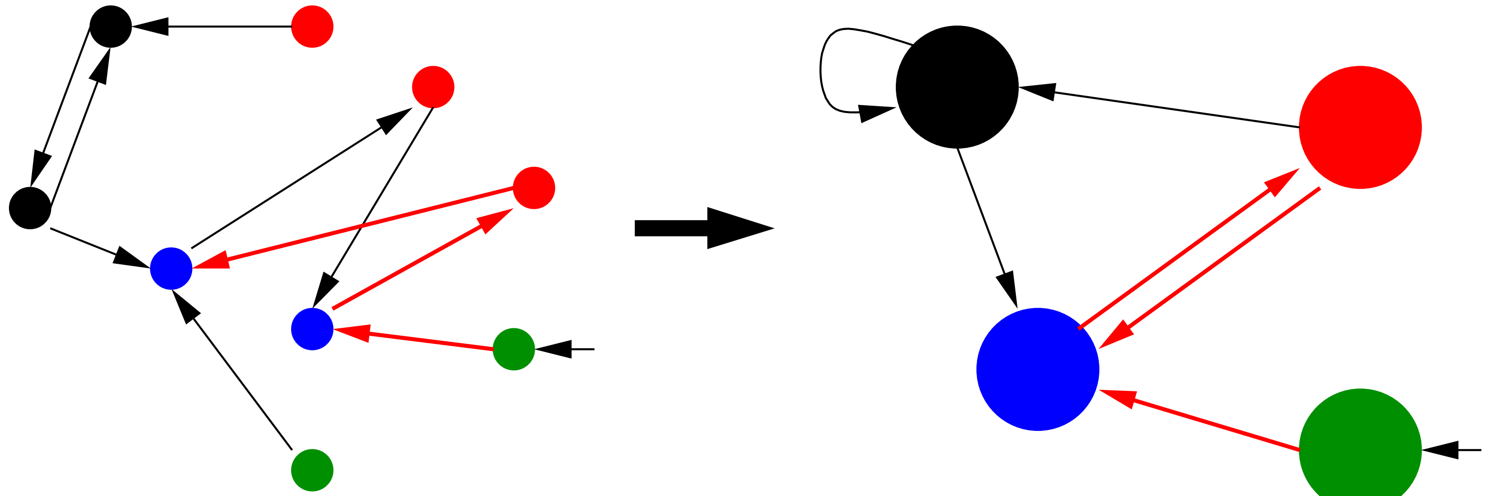
Let $\mathcal{K}'$ be a structure obtained by abstracting $\mathcal{K}$.

Then $\mathcal{K} \leq \mathcal{K}'$ holds; thus, if $\mathcal{K}'$ satisfies some LTL formula, then so does $\mathcal{K}$.

However, if $\mathcal{K}' \not\models \phi$, then $\mathcal{K} \models \phi$ may or may not hold!

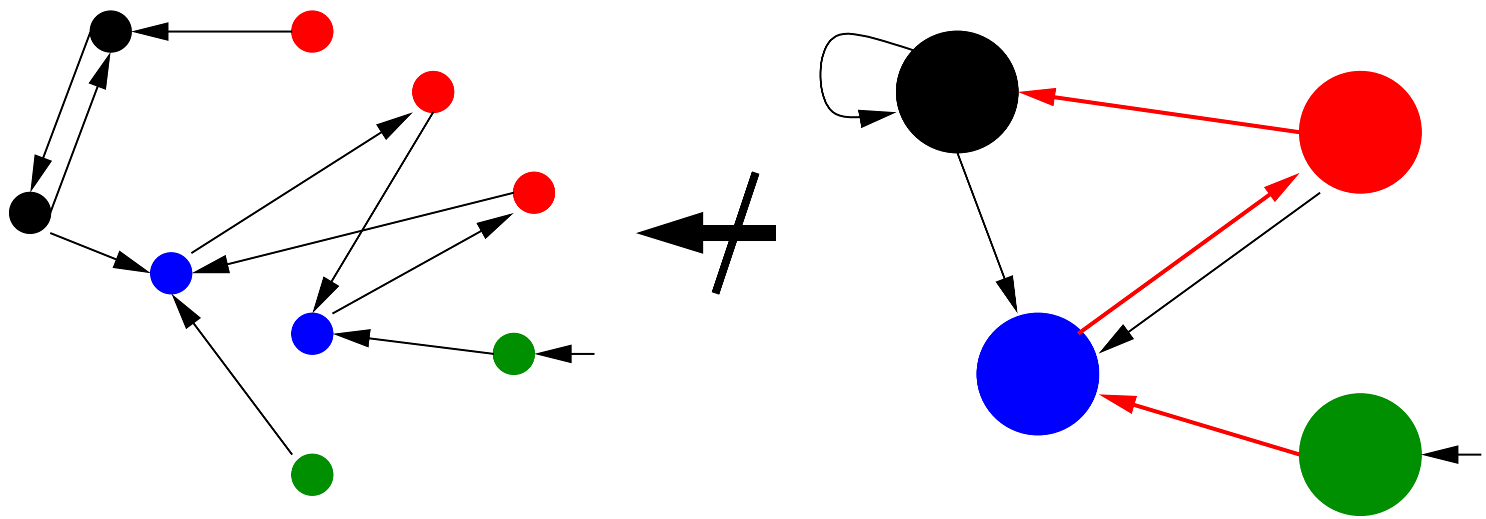
Abstraction gives rise to additional paths in the system:

Every concrete run has got a corresponding run in the abstraction ...



Abstraction gives rise to additional paths in the system:

. . . but not every abstract run has got a corresponding run in the concrete system.



Suppose that $\mathcal{K}' \not\models \phi$, where ρ is a counterexample.

Check whether there is a run in $\mathcal{K}$ that “corresponds” to ρ.

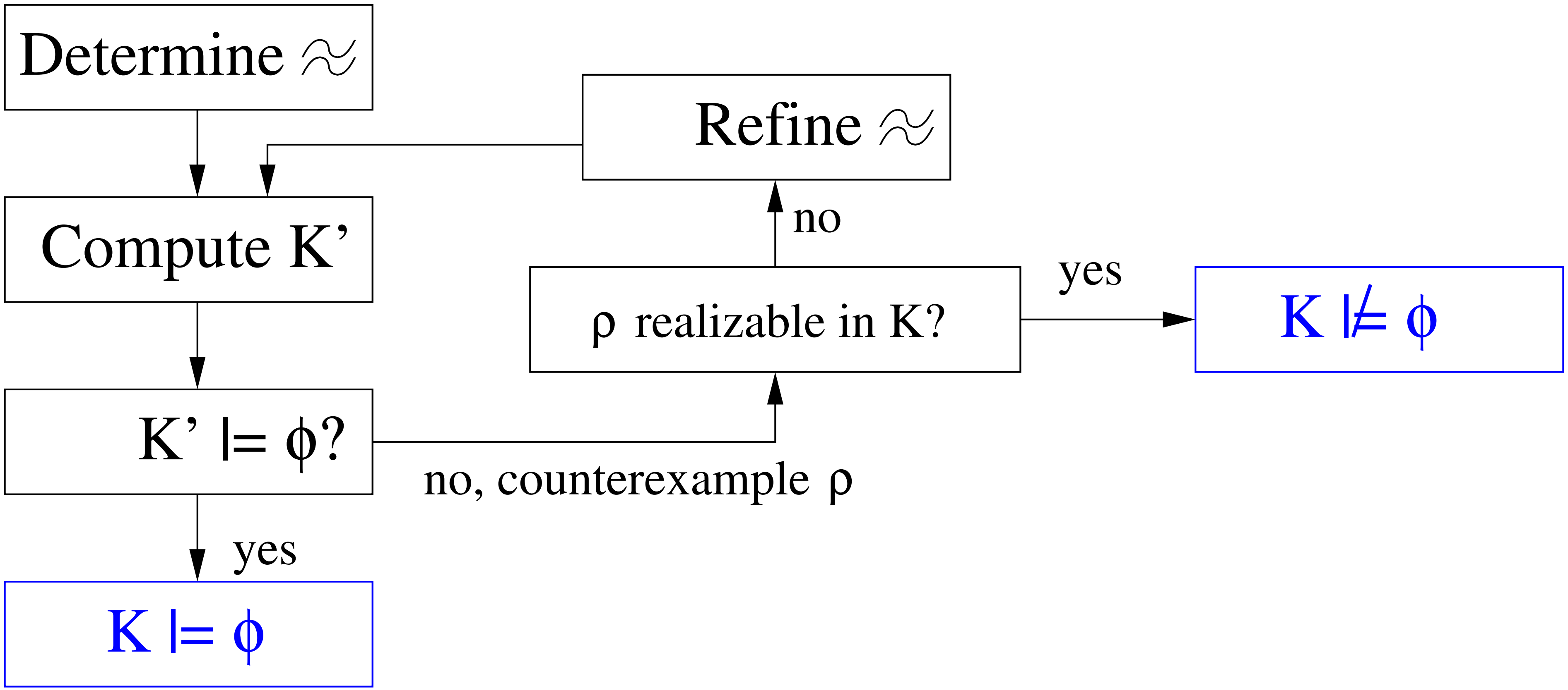
If yes, then $\mathcal{K} \not\models \phi$.

If no, then we can use ρ to refine the abstraction; i.e. we remove some equivalences from the relation H, introducing additional distinct states in $\mathcal{K}'$ so that ρ disappears.

The refinement can be repeated until a definite answer for $\mathcal{K} \models \phi$ (positive or negative) can be determined. This technique is called counterexample-guided abstraction refinement (CEGAR) [Clarke et al., 2000].

The abstraction-refinement cycle

Input: $\mathcal{K}, \phi$



Simulation of ρ

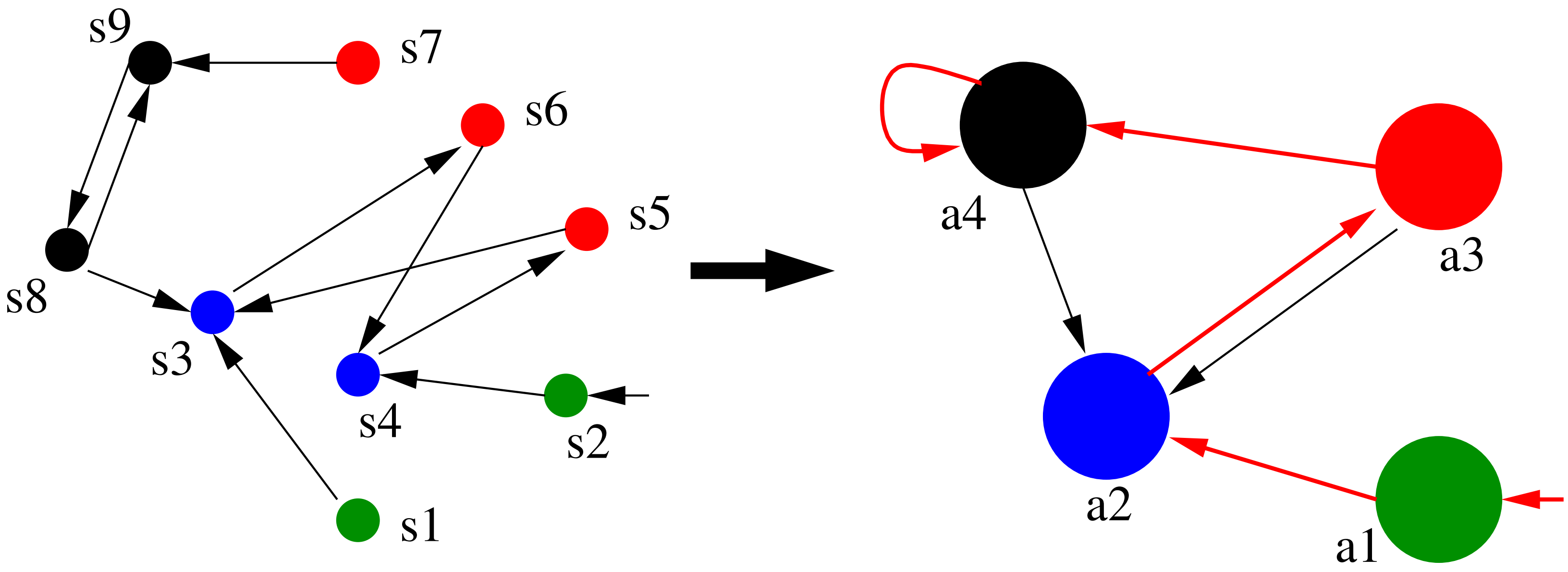
Problem: Given a counterexample ρ, is there a run corresponding to ρ in $\mathcal{K}$?

Solution: "Simulate" ρ on $\mathcal{K}$.

Remark: Any counterexample ρ can be partitioned into a finite stem and a finite loop, i.e. $\rho = w_S w_L^\omega$ for suitable w_S, w_L.

Case distinction: The simulation may fail in the stem or in the loop.

Example 1: $\mathbf{G}\,\neg$*black*



Abstraction yields a counterexample with stem $a_1 a_2 a_3 a_4$ and loop a_4.

Simulating the stem

Let $w_S = b_0 \cdots b_k$.

Start with $S_0 = \{r\}$. (We have $b_0 = [r]$.)

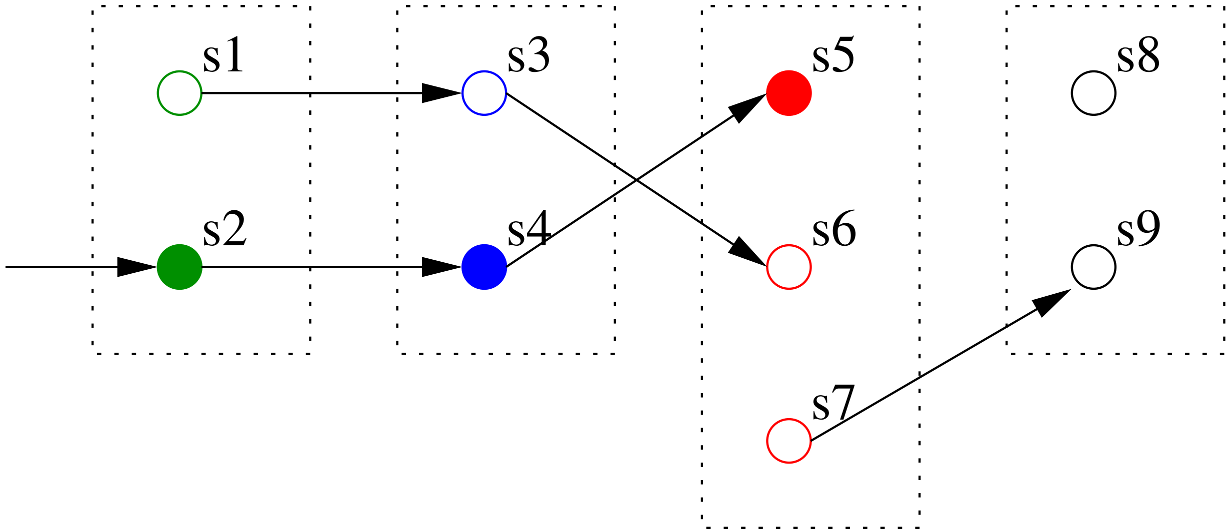
For $i = 1, \ldots, k$, compute $S_i = \{\, t \mid t \in b_i \wedge \exists s \in S_{i-1} \colon s \rightarrow t \,\}$.

If $S_k \neq \emptyset$, then there is a concrete correspondence for w_S.

If $S_k = \emptyset$: Find the smallest index ℓ with $S_\ell = \emptyset$: The refinement should distinguish the states in $S_{\ell-1}$ and those $b_{\ell-1}$-states that have immediate successors in b_ℓ.

Example: $w_S = a_1 a_2 a_3 a_4$

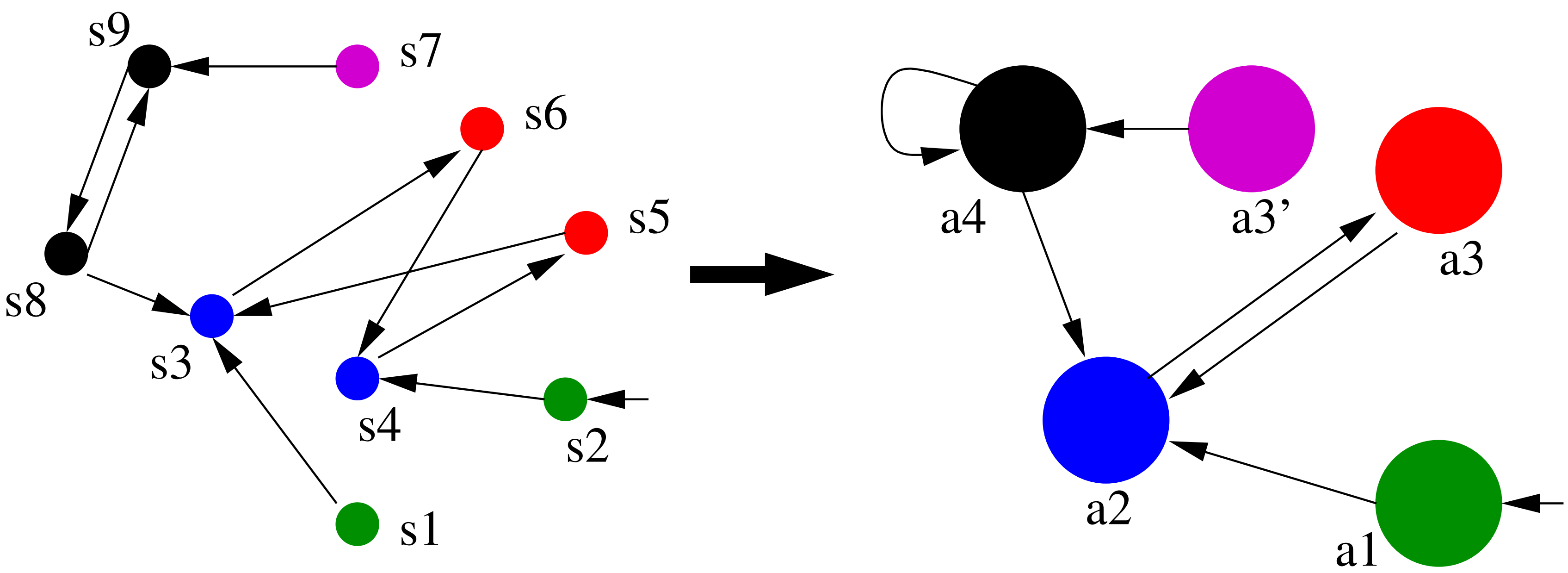
$S_0 = \{s_2\}, \qquad S_1 = \{s_4\}, \qquad S_2 = \{s_5\}, \qquad S_3 = \emptyset.$



In the next refinement, s_5 and s_7 must be distinguished.

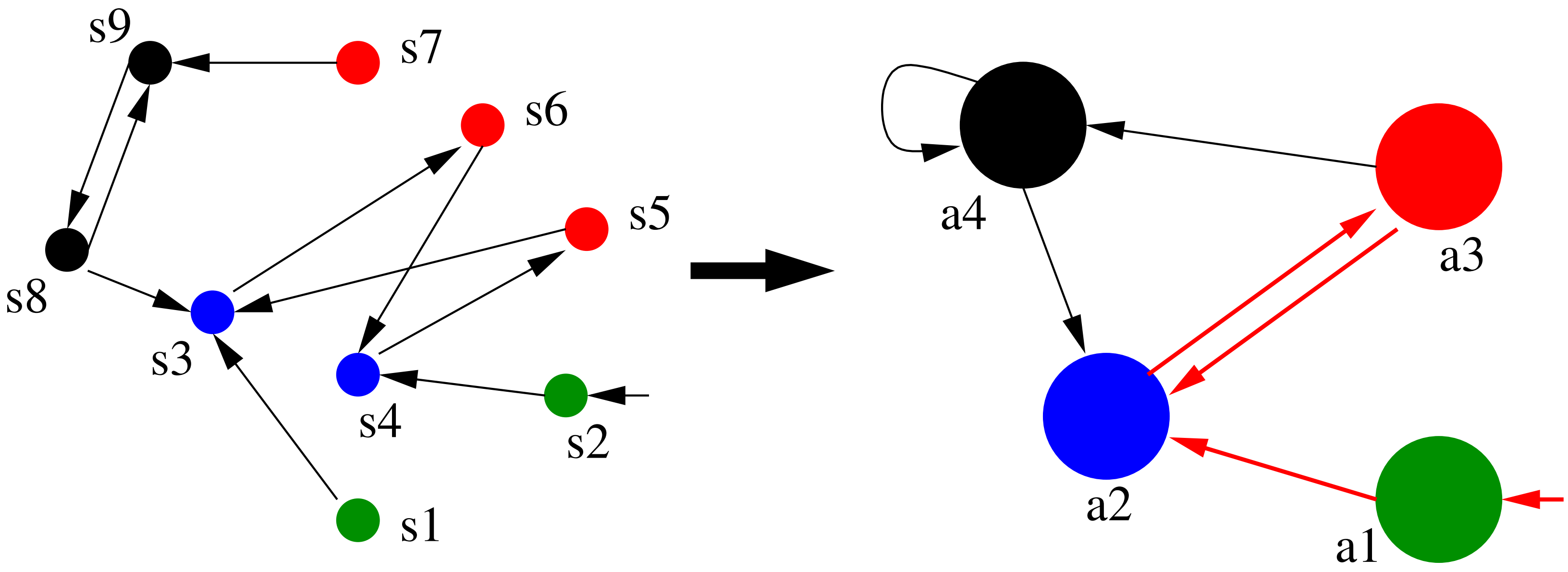
Possible new equivalence classes: $\{s_5, s_6\}, \{s_7\}$ or $\{s_5\}, \{s_6, s_7\}$.

Next try: $\mathbf{G}\ \neg$*black* with refinement



The new abstraction does not yield any counterexample; therefore, $\mathbf{G}\ \neg$*black* also holds in the concrete system.

Example 2: $\mathbf{F}\,\mathbf{G}$ *red*



The abstraction yields a counterexample with stem a_1a_2 and loop a_3a_2.

Simulating a loop

Assume $w_S = b_0 \cdots b_k, \quad w_L = c_1 \cdots c_\ell$

w_S is simulated as before, however w_L may have to be simulated multiple times.

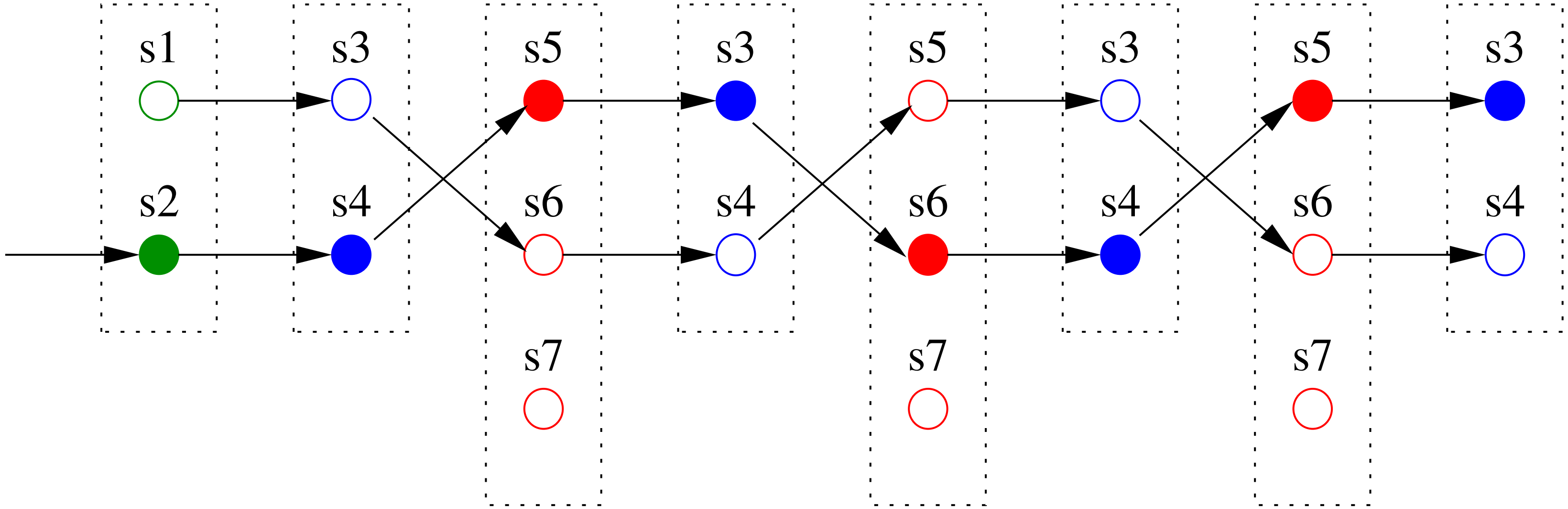
Let m be the size of the smallest equivalence class in w_L:

$$m = \min_{i=1,\ldots,\ell} |c_i|$$

Then we simulate the path $w_S w_L^{m+1}$; doing so, either the simulation will fail, or we will discover a real counterexample.

Refinement: same as before.

Example: $w_S = a_1 a_2$, $w_L = a_3 a_2$, $m = 2$



The simulation succeeds because there is a loop around s_4.

Thus, there is a real counterexample, so $\mathcal{K} \not\models \phi$.

Reasons for infinite state-spaces

So far, we have dealt with finite-state systems. However, many interesting real systems have infinitely many states.

Data: integers, reals, lists, trees, heap, ...

Control: procedures, dynamic thread creation, ...

Communication: unbounded message channels

Parameters: number of participants in a protocol, ...

Time: discrete or continuous clocks

Some (not all!) of these features (or combinations thereof) lead to Turing-powerful models of computation.

Part 14: Pushdown systems

Example 1

A small program (where $n \geq 1$):

```
bool g=true;
void main() {
    level_1();
    level_1();
    assume(g);
}
void level_n() {
    g:=not g;
}
```

```
void level_i() {
    for j:=1 to 8 do skip;
    level_{i+1}();
    level_{i+1}();
}
```

Question: Will `g` be true when the program terminates?

Example 1 has got *finitely* many states.
(The call stack is bounded by n.)

Can be treated by “inlining” (replace procedure calls by a copy of the callee).

Inlining causes an exponential state-space explosion.

Inlining is inefficient: every copy of each procedure will be investigated separately.

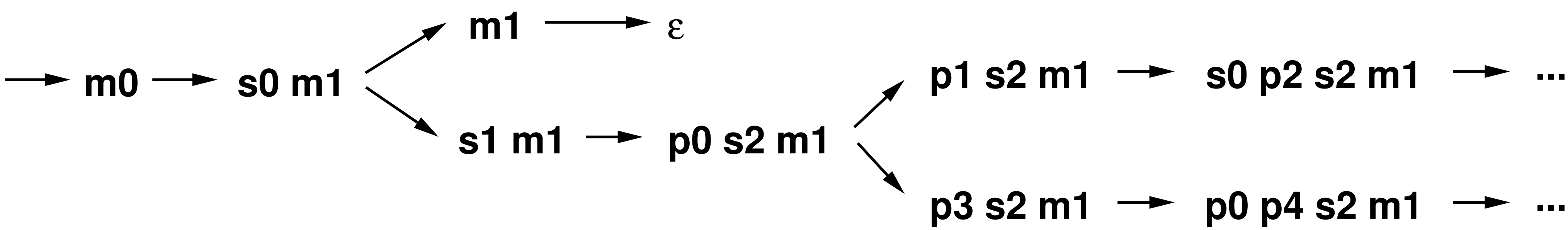
Inlining not applicable for recursive procedure calls.

Example 2: Recursive program (plotter)

```
procedure p;
p0:  if ? then
p1:        call s;
p2:        if ? then call p; end if;
     else
p3:        call p;
     end if
p4:  return
```

```
procedure s;
s0:  if ? then return; end if;
s1:  call p;
s2:  return;

procedure main;
m0:  call s;
m1:  return;
```

$S = \{p_0, \ldots, p_4, s_0, \ldots, s_2, m_0, m_1\}^*$,   initial state m_0



Example 2 has got infinitely many states.

Inlining not applicable!

Cannot be analyzed by naïvely searching all reachable states.

We shall require a *finite* representation of infinitely many states.

Example 3: Quicksort

```
void quicksort (int left, int right) {
   int lo,hi,piv;
   if (left >= right) return;
   piv = a[right]; lo = left; hi = right;
   while (lo <= hi) {
     if (a[hi]>piv) {
       hi = hi - 1;
     } else {
       swap a[lo],a[hi];
       lo = lo + 1;
     }
   }
   quicksort(left,hi);
   quicksort(lo,right);
}
```

Question: Does Example 3 sort correctly? Is termination guaranteed?

The mere structure of Example 3 does not tell us whether there are infinitely many reachable states:

- *finitely* many if the program terminates
- *infinitely* many if it fails to terminate

Termination can only be checked by directly dealing with infinite state sets.

A computation model for procedural programs

Control flow:

- sequential program (no multithreading)
- procedures
- mutual procedure calls (possibly recursive)

Data:

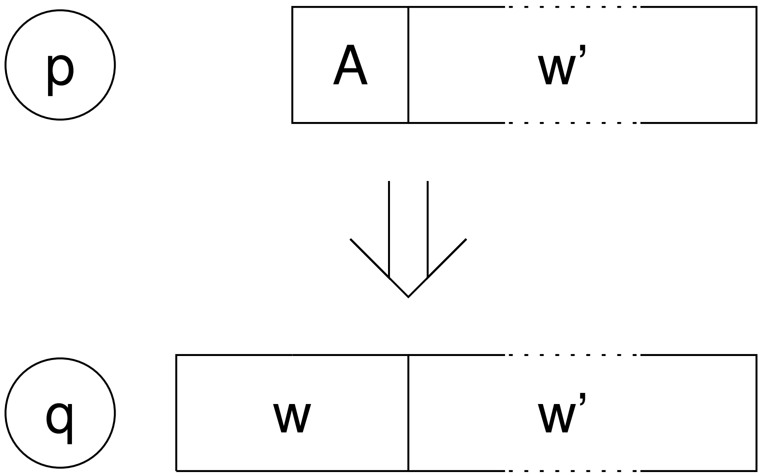
- global variables (restriction: only finite memory)
- local variables in each procedure (one copy per call)

Pushdown systems

A pushdown system (PDS) is a triple (P, Γ, Δ), where

- P is a finite set of control states;
- Γ is a finite stack alphabet;
- Δ is a finite set of rules.

Rules have the form $pA \hookrightarrow qw$, where $p, q \in P$, $A \in \Gamma$, $w \in \Gamma^*$.



Like acceptors for context-free language, but without any input!

Behaviour of a PDS

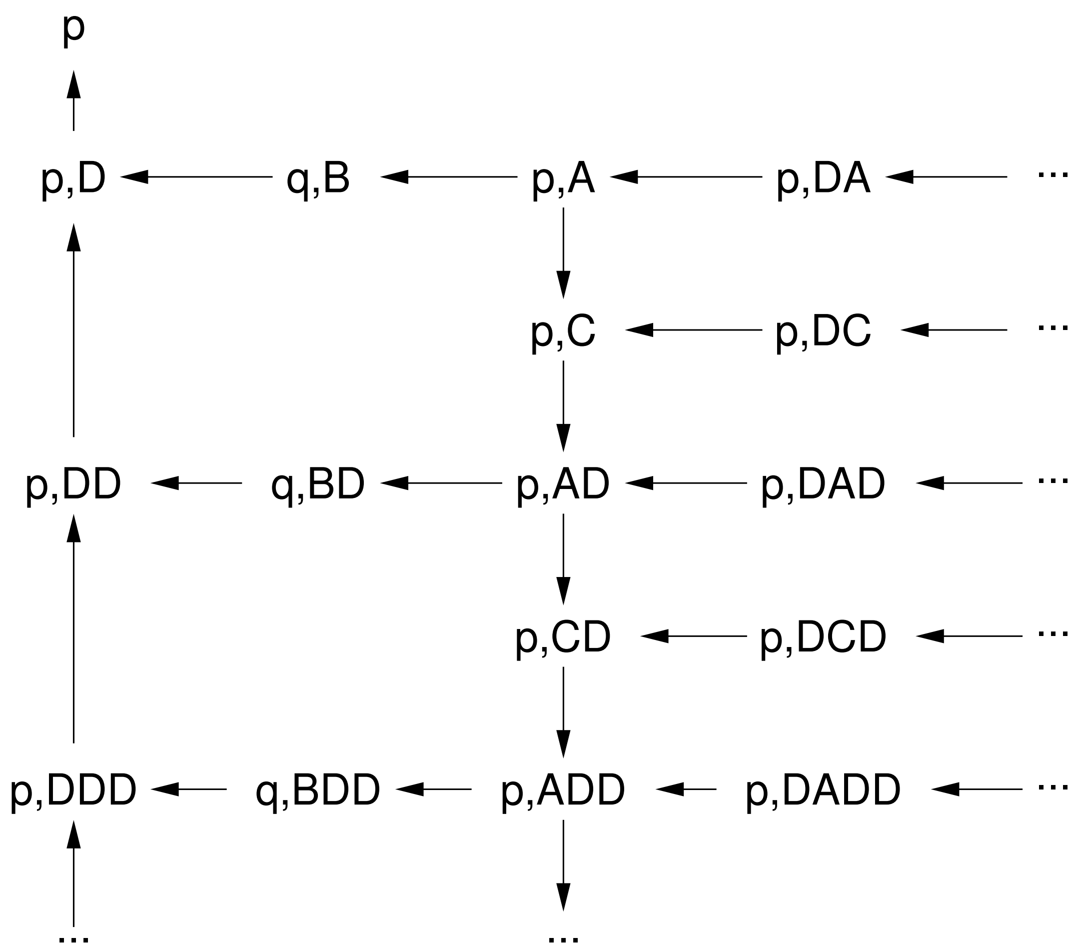
Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS and $c_0 \in P \times \Gamma^*$.

With $\mathcal{P}$ we associate a transition system $\mathcal{T}_{\mathcal{P}} = (S, \rightarrow, r)$ as follows:

- $S = P \times \Gamma^*$ are the states (which we call configurations);
- we have $pAw' \rightarrow qww'$ for all $w' \in \Gamma^*$ iff $pA \hookrightarrow qw \in \Delta$;
- $r = c_0$ is the initial configuration.

Transition system of a PDS

$pA \hookrightarrow qB$
$pA \hookrightarrow pC$
$qB \hookrightarrow pD$
$pC \hookrightarrow pAD$
$pD \hookrightarrow p\varepsilon$



Procedural programs and PDSs

P may represent the valuations of global variables.

Γ may contain tuples of the form (*program counter*, *local valuations*)

Interpretation of a configuration pAw:

global values in p, current procedure with local variables in A

"suspended" procedures in w

Rules:

$pA \hookrightarrow qB \mathrel{\widehat{=}}$ statement within a procedure

$pA \hookrightarrow qBC \mathrel{\widehat{=}}$ procedure call

$pA \hookrightarrow q\varepsilon \mathrel{\widehat{=}}$ return from a procedure

Reachability in PDS

Let $\mathcal{P}$ be a PDS and c, c' two of its configurations.

Problem: Does $c \rightarrow^* c'$ hold in $\mathcal{T}_\mathcal{P}$?

Note: $\mathcal{T}_\mathcal{P}$ has got infinitely many (reachable) states.

Nonetheless, the problem is decidable!

Finite automata

To represent (infinite) sets of configurations, we shall employ finite automata.

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS. We call $\mathcal{A} = (\Gamma, Q, P, \delta, F)$ a $\mathcal{P}$-automaton.

The alphabet of $\mathcal{A}$ is the stack alphabet Γ.

The initial states of $\mathcal{A}$ are the control states P.

We say that $\mathcal{A}$ accepts the configuration pw if $\mathcal{A}$ has got a path labelled by input w starting at p and ending at some final state.

Let $\mathcal{L}(\mathcal{A})$ be the set of configurations accepted by $\mathcal{A}$.

A set C of configurations is called regular iff there is some $\mathcal{P}$-automaton $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = C$.

An automaton is normalized if there are no transitions leading into initial states.

Remark: In the following, we shall use the following notation:

$pw \Rightarrow p'w'$ (in the PDS $\mathcal{P}$) and $p \xrightarrow{w} q$ (in $\mathcal{P}$-automata)

Reachability in PDS

Let $pre^*(C) = \{ c' \mid \exists c \in C \colon c' \Rightarrow c \}$ denote the predecessors of C,
and let $post^*(C) = \{ c' \mid \exists c \in C \colon c \Rightarrow c' \}$ the successors.

The following result is due to Büchi (1964):

Let C be a regular set and $\mathcal{A}$ be a (normalized) $\mathcal{P}$-automaton accepting C.

If C is regular, then so are $pre^*(C)$ and $post^*(C)$.

Moreover, $\mathcal{A}$ can be transformed into an automaton accepting $pre^*(C)$ resp. $post^*(C)$.

The basic idea (for pre)

Saturation rule: Add new transitions to $\mathcal{A}$ as follows:

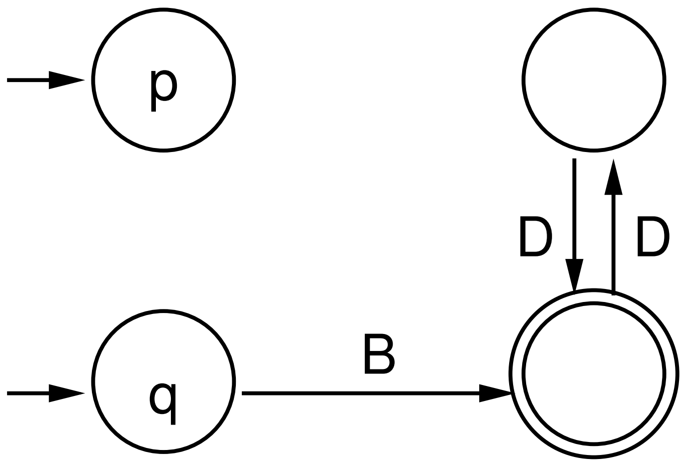
> If $q \xrightarrow{w} r$ currently holds in $\mathcal{A}$ and $pA \hookrightarrow qw$ is a rule, then add the transition (p, A, r) to $\mathcal{A}$.

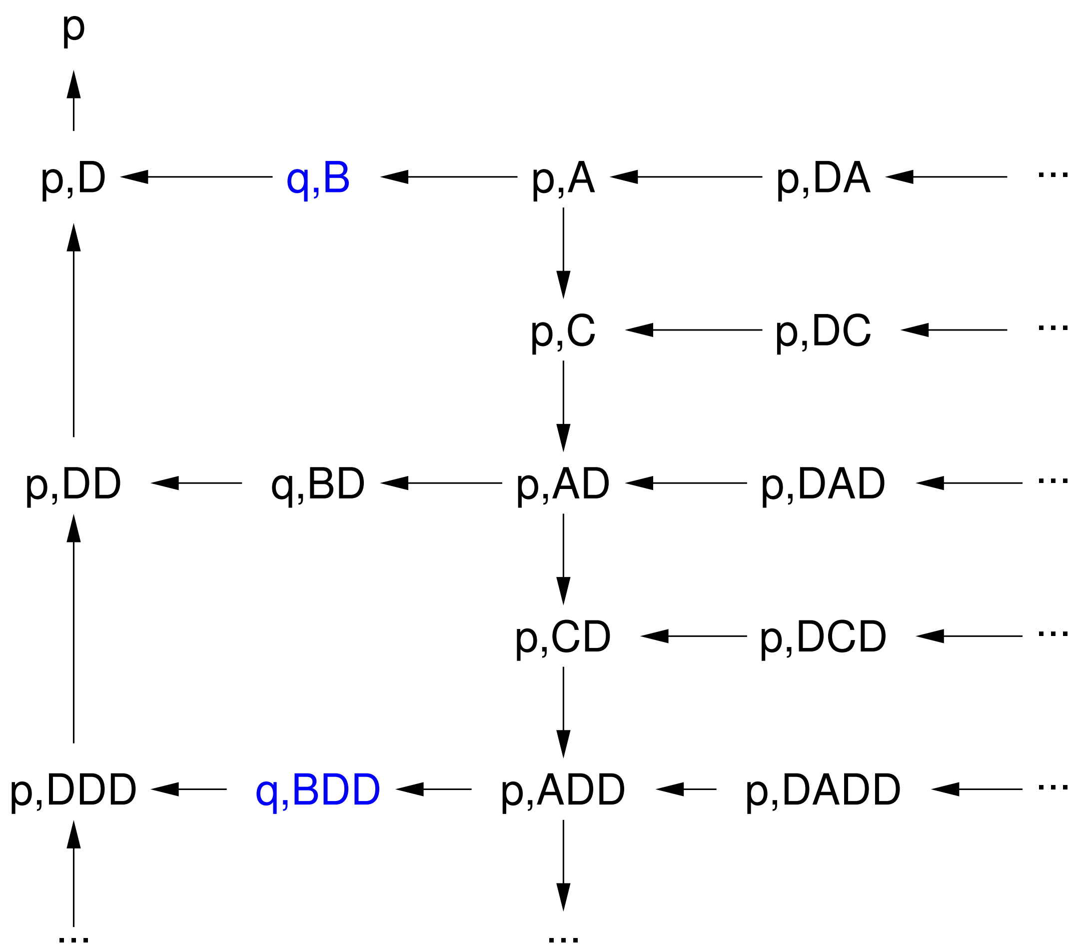
Repeat this until no other transition can be added.

At the end, the resulting automaton accepts $pre^*(C)$.

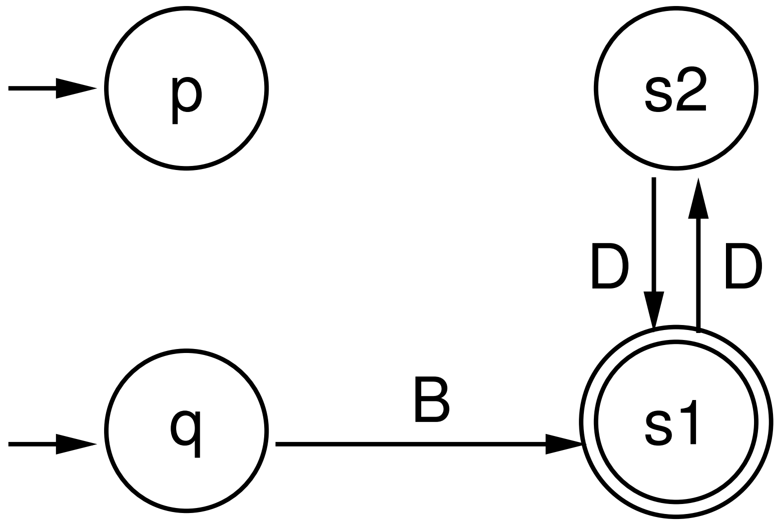
For $post^*(C)$: similar procedure.

Automaton $\mathcal{A}$ for C



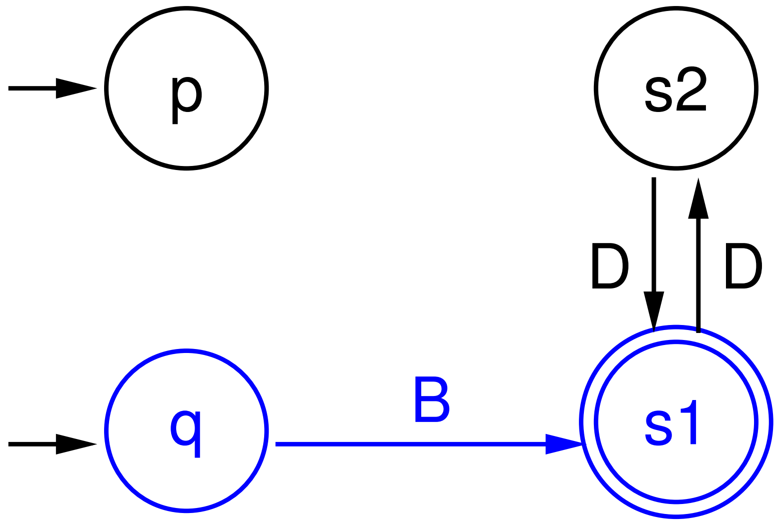


Extending $\mathcal{A}$



Extending $\mathcal{A}$

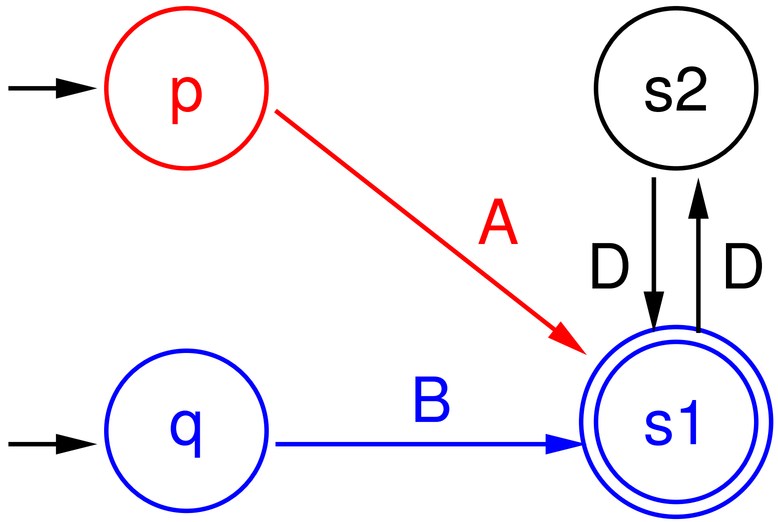
If the right-hand side of a rule can be read,



Rule: $pA \hookrightarrow qB$ Path: $q \xrightarrow{B} s_1$

Extending $\mathcal{A}$

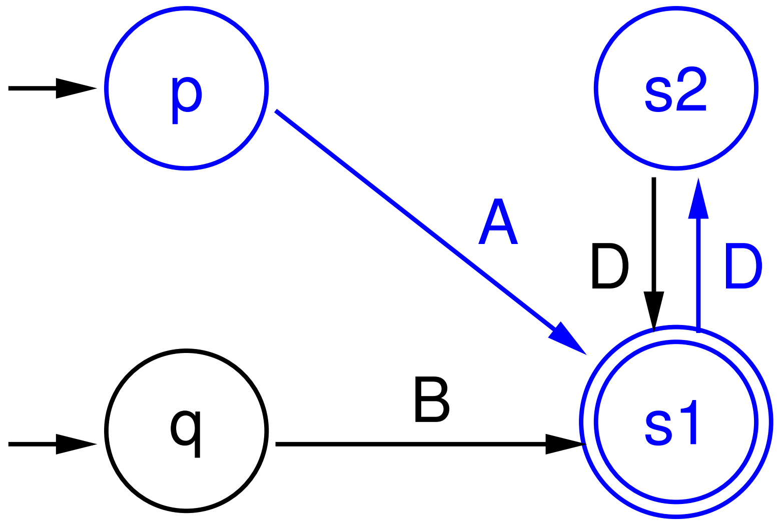
If the right-hand side of a rule can be read, add the left-hand side.



Rule: $pA \hookrightarrow qB$ Path: $q \xrightarrow{B} s_1$ New path: $p \xrightarrow{A} s_1$

Extending $\mathcal{A}$

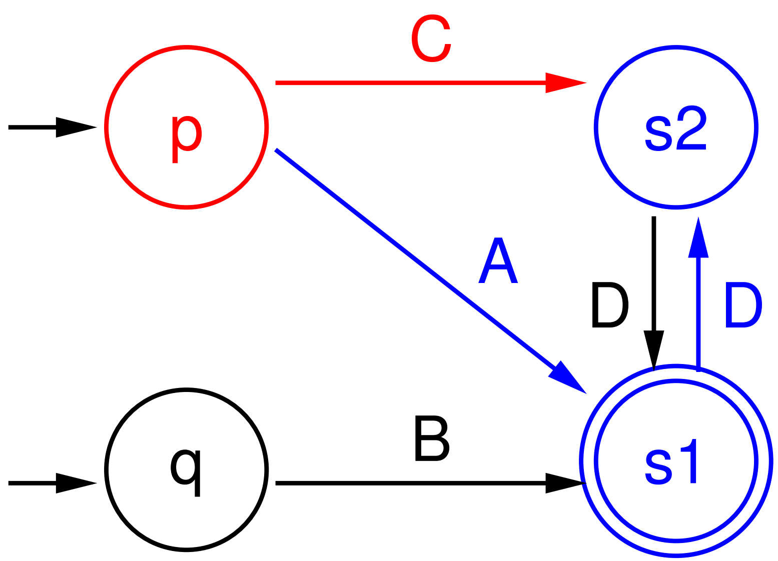
If the right-hand side of a rule can be read,



Rule: $pC \hookrightarrow pAD$ Path: $p \xrightarrow{A} s_1 \xrightarrow{D} s_2$

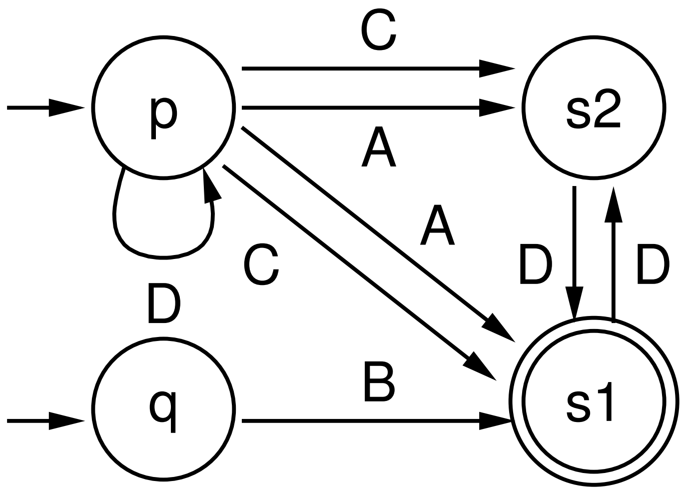
Extending $\mathcal{A}$

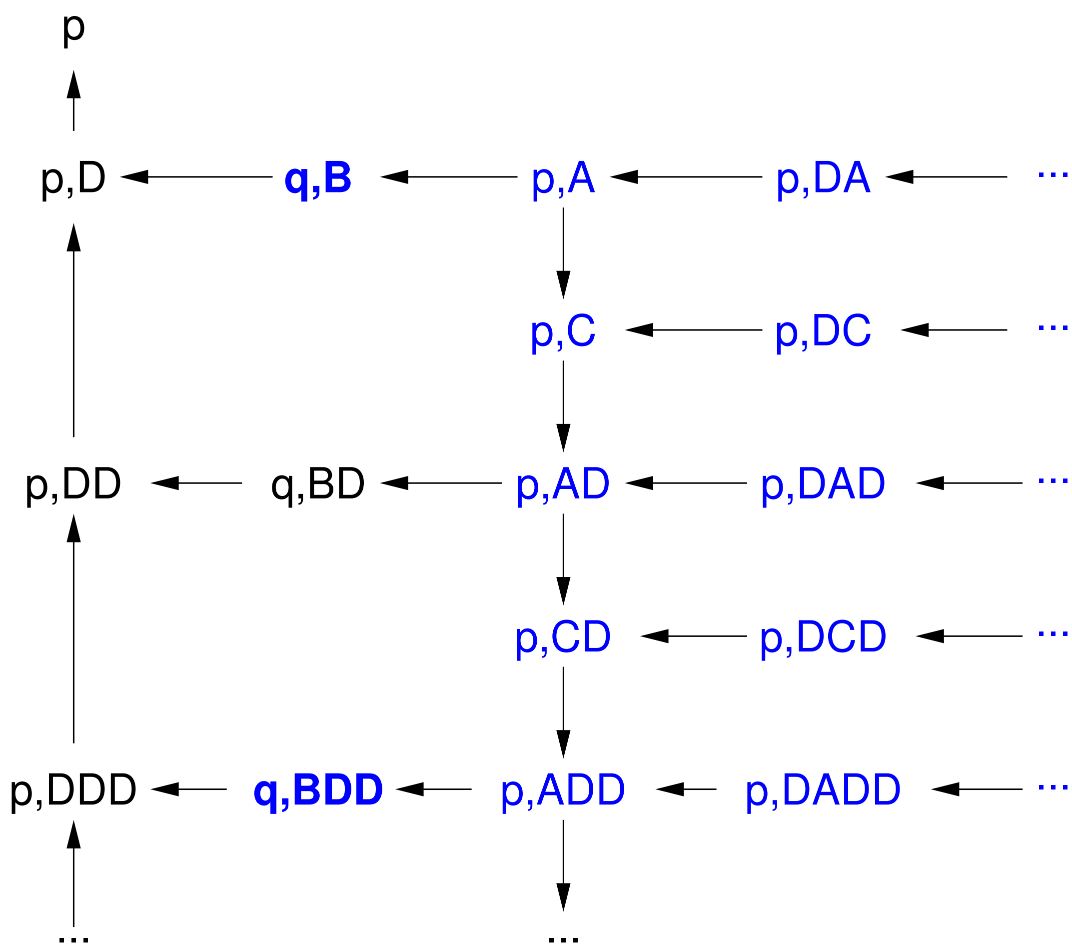
If the right-hand side of a rule can be read, add the left-hand side.



Rule: $pC \hookrightarrow pAD$ Path: $p \xrightarrow{A} s_1 \xrightarrow{D} s_2$ New path: $p \xrightarrow{C} s_2$

Final result





Complexity:
$\mathcal{O}(|Q|^2 \cdot |\Delta|)$ time.

Proof of correctness

We shall show:

Let $\mathcal{B}$ be the $\mathcal{P}$-automaton arising from $\mathcal{A}$ by applying the saturation rule. Then $\mathcal{L}(\mathcal{B}) = pre^*(C)$.

Part 1: Termination

The saturation rule can only be applied finitely many times because no states are added and there are only finitely many possible transitions.

Part 2: $pre^*(C) \subseteq \mathcal{L}(\mathcal{B})$

Let $c \in pre^*(C)$ and $c' \in C$ such that c' is reachable from c in k steps. We proceed by induction on k (simple).

Part 3: $\mathcal{L}(\mathcal{B}) \subseteq pre^*(C)$

Let $\underset{i}{\rightarrow}$ denote the transition relation of the automaton after the saturation rule has been applied i times.

We show the following, more general property: If $p \overset{w}{\underset{i}{\rightarrow}} q$, then there exist $p'w'$ with $p' \overset{w'}{\underset{0}{\rightarrow}} q$ and $pw \Rightarrow p'w'$; if $q \in P$, then additionally $w' = \varepsilon$.

Proof by induction over i: The base case $i = 0$ is trivial.

Induction step: Let $t = (p_1, A, q')$ be the transition added in the i-th application and j the number of times t occurs in the path $p \overset{w}{\underset{i}{\rightarrow}} q$.

Induction over j: Trivial for $j = 0$. So let $j > 0$.

There exist p_2, p', u, v, w', w_2 with the following properties:

(1) $p \xrightarrow[i-1]{u} p_1 \xrightarrow[i]{A} q' \xrightarrow[i]{v} q$ (splitting the path $p \xrightarrow[i]{w} q$)

(2) $p_1 A \hookrightarrow p_2 w_2$ (pre-condition for saturation rule)

(3) $p_2 \xrightarrow[i-1]{w_2} q'$ (pre-condition for saturation rule)

(4) $pu \Rightarrow p_1 \varepsilon$ (ind.hyp. on i)

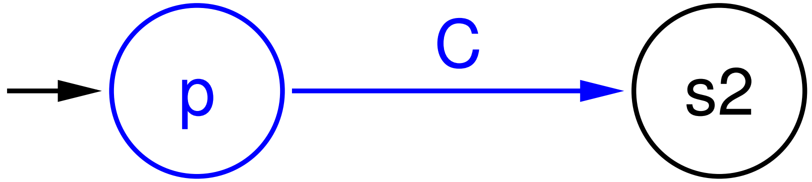
(5) $p_2 w_2 v \Rightarrow p' w'$ (ind.hyp. on j)

(6) $p' \xrightarrow[0]{w'} q$ (ind.hyp. on j)

The desired proof follows from (1), (4), (2), and (5).
If $q \in P$, then the second part follows from (6) and the fact that $\mathcal{A}$ is normalized.

Example: *post** (without proof)

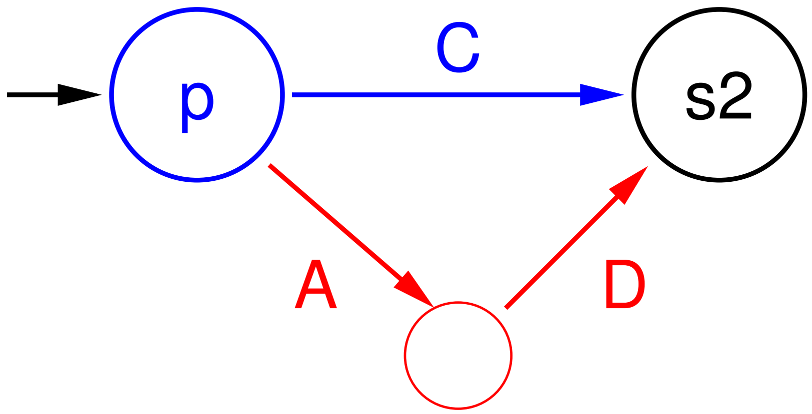
If the *left-hand side* of a rule can be read, .



Rule: $pC \hookrightarrow pAD$ Path: $p \xrightarrow{C} s_2$

Example: *post** (without proof)

If the *left-hand side* of a rule can be read, add the *right-hand side*.



Rule: $pC \hookrightarrow pAD$ Path: $p \xrightarrow{C} s_2$ New Path: $p \xrightarrow{AD} s_2$

LTL and Pushdown Systems

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS with initial configuration c_0, let $\mathcal{T}_\mathcal{P}$ denote the corresponding transition system, AP a set of atomic propositions, and $\nu \colon P \times \Gamma^* \to 2^{AP}$. a valuation function.

$\mathcal{T}_\mathcal{P}$, AP, and ν form a Kripke structure $\mathcal{K}$; let ϕ be an LTL formula (over AP).

Problem: Does $\mathcal{K} \models \phi$?

Undecidable for arbitrary valuation functions!
(could encode undecidable decision problems in ν . . .)

However, LTL model checking *is* decidable for certain “reasonable” restrictions of ν.

In the following, we consider "simple" valuation functions satisfying the following restriction:

$\nu(pAw) = \nu(pA)$, for all $p \in P$, $A \in \Gamma$, and $w \in \Gamma^*$.

In other words, the "head" of a configuration holds all information about atomic propositions.

LTL model checking is decidable for such "simple" valuations.

Approach

Same principle as for finite Kripke structures:

Translate $\neg\phi$ into a Büchi automaton $\mathcal{B}$.

Build the cross product of $\mathcal{K}$ and $\mathcal{B}$.

Test the cross product for emptiness.

Note that the cross product is not a Büchi automaton in this case (e.g., it is not finite).

Büchi PDS

The cross product is a new pushdown system $\mathcal{Q}$, as follows:

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a PDS, $p_0 w_0$ the initial configuration, and AP, ν as usual.

Let $\mathcal{B} = (2^{AP}, Q, q_0, \delta, F)$ be the Büchi automaton for $\neg\phi$.

Construction of $\mathcal{Q}$:

$\mathcal{Q} = (P \times Q, \Gamma, \Delta', P \times F)$, where

$(p, q)A \hookrightarrow (p', q')w \in \Delta'$ iff

– $pA \hookrightarrow p'w \in \Delta$ and

– $(q, L, q') \in \delta$ such that $\nu(pA) = L$.

Initial configuration: $(p_0, q_0)w_0$

Let ρ be a run of $\mathcal{Q}$ with $\rho(i) = (p_i, q_i)w_i$.

We call ρ accepting if $q_i \in F$ for infinitely many values of i.

The following is easy to see:

> $\mathcal{P}$ does *not* satisfy ϕ iff there exists an accepting run in $\mathcal{Q}$.

Characterization of accepting runs

Question: If there an accepting run starting at $(p_0, q_0)w_0$?

In the following, we shall consider the following, more general global model-checking problem:

> Compute *all* configurations c such that there exists an accepting run starting at c.

Lemma: There is an accepting run starting at c iff there exists $(p, q) \in P \times Q$, $A \in \Gamma$ with the following properties:

(1) $c \Rightarrow (p, q)Aw$ for some $w \in \Gamma^*$

(2) $(p, q)A \Rightarrow (p, q)Aw'$ for some $w' \in \Gamma^*$, where

the path from $(p, q)A$ to $(p, q)Aw'$ contains at least one step;

the path contains at least one accepting Büchi state.

Repeating heads

We call $(p, q)A$ a repeating head if $(p, q)A$ satisfies properties (1) and (2).

Strategy:

1. Compute all repeating heads.
(naïvely: check for each pair $(p, q)A$ whether
$(p, q)A \in pre^*(\{ (p, q)Aw \mid w \in \Gamma^* \})$. Visiting an accepting state can be encoded into the control state, see next slide.

2. Compute the set $pre^*(\{ (p, q)Aw \mid (p, q)A \text{ is a repeating head}, w \in \Gamma^* \})$

Implementing step 1

First, we transform $\mathcal{Q}$ into a modified PDS $\mathcal{Q}'$.

For each pair $(p, q) \in P \times Q$, $\mathcal{Q}'$ has got two control states: $(p_0, q), (p_1, q)$.

For each rule $(p, q)A \hookrightarrow (p', q')w$ in $\mathcal{Q}$, $\mathcal{Q}'$ has got rules:
$(p_b, q)A \hookrightarrow (p'_1, q)w$ for $b = 0, 1$ if $q \in F$;
$(p_b, q)A \hookrightarrow (p'_b, q)w$ for $b = 0, 1$ if $q \notin F$.

Then, in $\mathcal{Q}'$, we compute whether $(p_0, q)A \in pre^*(\{ (p_1, q)Aw \mid w \in \Gamma^* \})$, for each head $(p, q)A$ of $\mathcal{Q}$.

CTL* and PDS

The model-checking approach for PDS can even be lifted to CTL*.

The approach is analogous to finite-state systems (sketch only):

Solve the innermost $\mathbf{E}$-subformulae using the global LTL model-checking algorithm we have developed.

The result is a finite-state automaton $\mathcal{A}$ giving all the configurations satisfying that subformula.

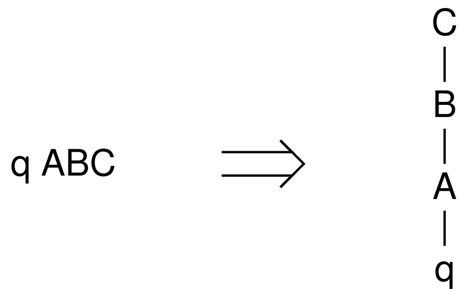
Modify the PDS: Encode the states of $\mathcal{A}$ into the stack alphabet and synchronize the push/pop operations with the actions of $\mathcal{A}$ such that the top-of-stack symbol shows the final state of $\mathcal{A}$ iff the current stack content is accepted by $\mathcal{A}$.

Replace the subformula by a fresh atomic proposition (which holds if the top-of-stack symbol shows the final state of $\mathcal{A}$), and continue as in the finite-state case.

Part 15: Tree-Rewriting Systems

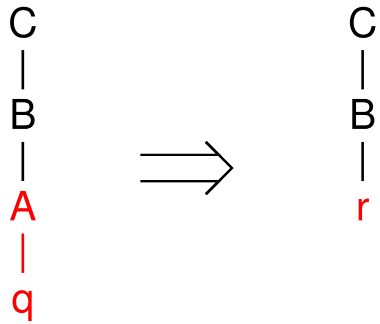
From sequences to trees

Pushdown configurations are *words* (i.e., sequences of symbols).



Alternative view: a sequence is a (degenerated) tree.

PDS rules replace one (degenerated) subtree by another, e.g. for $qA \hookrightarrow r\varepsilon$:

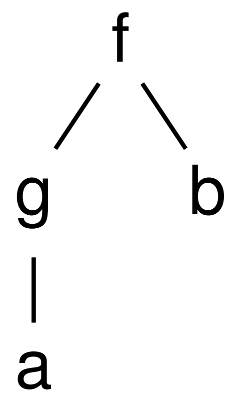


Let us consider the case where configurations are general trees, and where rules replace one subtree by another.

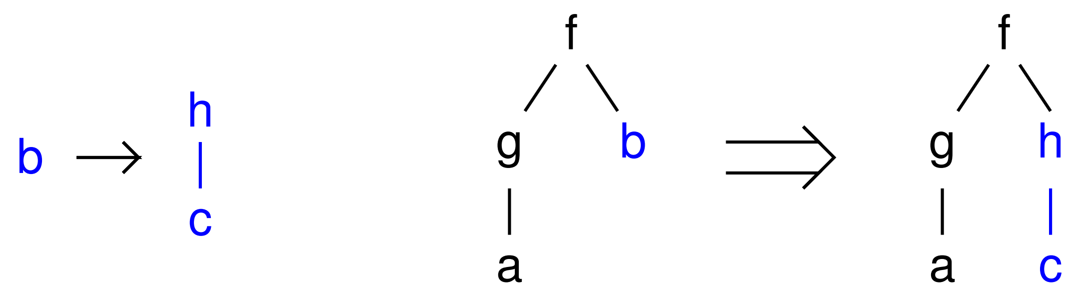
Motivation:

Systems with procedures and threads

Functional languages, e.g., the term $f(g(a), b)$ is a tree:



An equality $b = h(c)$ corresponds to the rewrite rule:



Notice that if a node has more than one child, their order matters (i.e., there is a designated *first*, *second* etc. child); we speak of ordered trees.

In the trees we are interested in, each node is labelled by some symbol.
If the symbols of a tree are all taken from an alphabet Σ, we speak of Σ-labelled trees.

Ground Tree Rewrite Systems

A Ground Tree Rewrite System (GTRS) is a tuple (Σ, Δ), where:

- Σ is a finite alphabet;
- Δ is a finite set of rewrite rules;
  each rewrite rule is a pair of ordered Σ-labelled trees.

A configuration of a GTRS $\mathcal{G} = (\Sigma, \Delta)$ is a Σ-labelled tree.

If t is a configuration of $\mathcal{G}$, and t' a subtree of t such that $(t', u') \in \Delta$, then t can be rewritten to u, where u is the tree obtained from t by replacing t' with u' (see previous example).

GTRS and PDS

It is easy to see that GTRS are a generalization of PDS; i.e. each PDS can be expressed as a GTRS (but not necessarily vice versa).

Using the rewriting relation of GTRS, we can define reachability between configurations, i.e., $t \Rightarrow u$ if u can be obtained from t by zero or more rewritings; pre^* and $post^*$ are then defined as in PDS.

We shall see the following:

- Like for PDS, there is a representation for ("regular") sets of trees.
- Like for PDS, regularity is closed under reachability.
- Unlike for PDS, LTL model checking becomes undecidable!

Representing sets of trees

Like PDS, GTRS are infinite-state systems (i.e., any GTRS may have infinitely many different configurations).

Therefore, like for PDS, we need a representation for infinite sets of configurations.

For PDS, we used *finite automata*. We shall generalize these to *tree automata*.

A run of a finite automaton, given input *abcde*, may have the following form (corresponding automaton not shown):

$$q0 \xrightarrow{a} q1 \xrightarrow{b} q2 \xrightarrow{c} q2 \xrightarrow{d} q1 \xrightarrow{e} q3$$

A run is considered *accepting* if at the end of the input, we reach a final state.

Alternative view: A word is a (degenerated) tree, a run labels each node with a state:

```
e   q3
|
d   q1
|
c   q2
|
b   q2
|
a   q1
```

In this view, a run is *accepting* if the root is labelled with a final state.

In a tree automaton, the input is a tree, and a run labels each node with a state. Thus a *sequence of states* transitions under a letter into the next state. A run is considered accepting if the root is labelled by a final state.

Rules of the form $q' \stackrel{a}{\to} q$ specify how to label nodes with one child;
rules of the form $(q', q'') \stackrel{a}{\to} q$ are for nodes with two children etc.;
rules of the form $() \stackrel{a}{\to} q$ specify how to label *leaves* (corresponding to q being initial)

The next slides contain the formal definitions.

Tree automata

A tree automaton is a tuple $\mathcal{T} = (\Sigma, Q, F, \delta)$, where:

- Σ is a finite input alphabet;
- Q is a finite set of states;
- $F \subset Q$ are the final states;
- $\delta \subseteq Q^* \times \Sigma \times Q$ is a finite set of transitions.

$\mathcal{T}$ is called deterministic if for each pair $(\vec{q}, a) \in Q^* \times \Sigma$ there is at most one $r \in Q$ such that $(\vec{q}, a, r) \in \delta$.

Remark: We denote transition $(q_1 \cdots q_n, a, q)$ as $(q_1, \ldots, q_n) \xrightarrow{a} q$.

Regular tree languages

Let $\mathcal{T} = (\Sigma, Q, F, \delta)$ be a tree automaton and t a Σ-labelled tree.

Suppose that n is a node of t labelled by $a \in \Sigma$.
We say that n can be labelled by $q \in Q$ if

either n is a leaf and δ contains $\stackrel{a}{\to} q$,

or n has children that can be labelled by $q_1, \ldots, q_n$ (in that order), and δ contains $(q_1, \ldots, q_n) \stackrel{a}{\to} q$.

We say that t is accepted by $\mathcal{T}$ if its root can be labelled by some final state of $\mathcal{T}$.

The set of trees accepted by $\mathcal{T}$ is called the language of $\mathcal{T}$, denoted $\mathcal{L}(\mathcal{T})$.

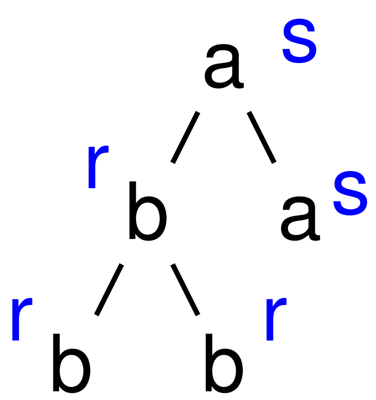
A set of trees is called regular if there is a tree automaton accepting it.

Example

Let $\mathcal{T} = (\Sigma, Q, F, \delta)$, where $\Sigma = \{a, b\}$, $Q = \{r, s\}$, $F = \{s\}$,
and δ contains the following rules:

$$\xrightarrow{a} s, \quad (r, s) \xrightarrow{a} s, \quad \xrightarrow{b} r, \quad (r, r) \xrightarrow{b} r.$$

In the tree shown below, the possible labels are shown in blue.



Since the root is labelled by final state s, the tree is accepted.

Tree automata with empty moves

Like in finite automata, it is sometimes convenient to consider tree automata with "empty" moves.

An "empty" move is denoted by a rule $q \to q'$, meaning that any state that can be labelled with q can also be labelled with q'.

Like for finite automata, empty moves can only occur in nondeterministic automata; they can be eliminated by adding, for each pair of rules $\vec{q} \stackrel{a}{\to} q$ and $q \to q'$, another rule $\vec{q} \stackrel{a}{\to} q'$.

Thus, like for finite automata, empty moves do not increase the expressive power of tree automata.

Notes on tree automata

The tree automata considered here are also called bottom-up tree automata.

Like regular word languages, regular tree languages are closed under union, intersection, and negation.

Like regular word languages, deterministic tree automata are equally powerful as non-deterministic ones.

The corresponding operations on tree automata can be adapted from those on finite automata.

Reachability on GTRS

Let $\mathcal{G} = (\Sigma, \Delta)$ be a GTRS and $\mathcal{L}$ be a set of Σ-labelled trees.

We make the following claims (analogous to PDS):

If $\mathcal{L}$ is regular, then so are $pre^*(\mathcal{L})$ and $post^*(\mathcal{L})$.

If $\mathcal{T}$ is a tree automaton accepts $\mathcal{L}$, then $\mathcal{T}$ can be effectively transformed into an automaton accepting $pre^*(\mathcal{L})$ (or $post^*(\mathcal{L})$, respectively).

In the following, we shall transform $\mathcal{T}$ into $\mathcal{T}'$ such that $\mathcal{L}(\mathcal{T}') = pre^*(\mathcal{L}(\mathcal{T}))$.

1. Set $\mathcal{T}' := \mathcal{T}$.

2. For each pair $(t, u) \in \Delta$, build a tree automaton $\mathcal{T}_t$ that accepts just t. Assume that $\mathcal{T}_t$ has one single final state q_t. Add the states and transitions of $\mathcal{T}_t$ to $\mathcal{T}'$, however q_t will not be accepting in $\mathcal{T}'$.

3. If $(t, u) \in \Delta$ and u can be labelled by some state q of $\mathcal{T}'$, then add an empty move $q_t \to q$ to $\mathcal{T}'$.

4. Repeat step 3 until no more additions are possible.

Note: For $post^*$, just switch the left/right-hand sides of the rules.

LTL on GTRS

We now show that the following problem is undecidable.

Let $\mathcal{G}$ be a GTRS and ϕ an LTL formula, does $\mathcal{G} \models \phi$?

Proof idea:

Reduction from the halting problem of Turing machines on empty tape.

Given a TM $\mathcal{M}$ we construct $\mathcal{G}$ and ϕ, such that $\mathcal{G} \models \phi$ iff $\mathcal{M}$ stops.

Recap: Turing machines

Let $(\Sigma, Q, q_0, q_f, \#, \delta)$ be a Turing machine with

tape alphabet Σ, control states Q, initial state q_0,

accepting state q_f, empty tape symbol $\#$, transitions δ.

W.l.o.g., we assume that $\mathcal{M}$ is deterministic.

Moreover, $\# \notin \Sigma$ and we define $\Sigma' := \Sigma \cup \{\#\}$.

The transitions in δ are of the form (p, a, X, q, b) with $p, q \in Q$, $a, b \in \Sigma$, $X \in \{L, N, R\}$.

Construction of the GTRS

Our GTRS has the following alphabet:

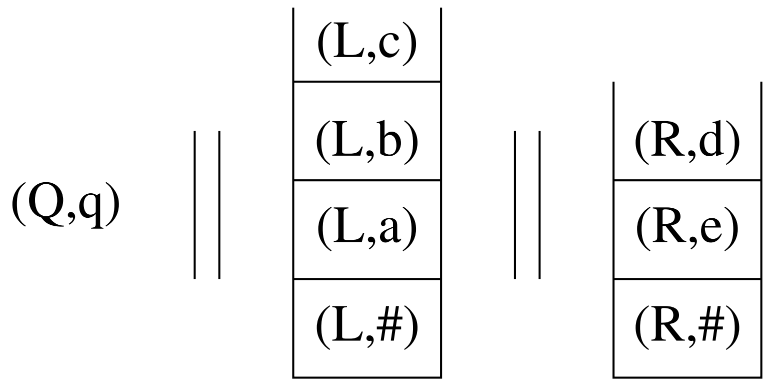
$$\{S\} \cup \{\, (Q, q) \mid q \in Q \,\} \cup \{\, (X, a) \mid X \in \{L, R\},\ a \in \Sigma \,\}$$

The intial configuration is the tree with a root and three children labelled (Q, q_0), $(L, \#)$, and $(R, \#)$, respectively. (In the following, these are called the Q/L/R-components.)

All other rules modify one of these three components.

Reachable trees of the GTRS

Each reachable tree will have three components below the root, one consisting of a single node, and two stacks:



Each such configuration can be interpreted as a configuration of $\mathcal{M}$ (for instance, the tape content $\#abcde\#$ with control state q, where the reading head is on c).

Rules of the GTRS

The other rules simulate the usual operations on the stack, where the actions "record" what happens.

Operations on the Q-component:

$$(Q, q) \stackrel{(Q,q,q')}{\longrightarrow} (Q, q') \text{ for all } q, q' \in Q \text{ and } q \neq q_f \text{ and } (Q, q_f) \stackrel{Halt}{\longrightarrow} \varepsilon$$

Operations on the L/R-components:

$$(X, a) \stackrel{(X,a,b)}{\longrightarrow} (X, b) \quad \text{for all } X \in \{L, R\}, a, b \in \Sigma'$$

$$(X, a) \stackrel{(X,a,\varepsilon)}{\longrightarrow} \varepsilon \quad \text{for all } X \in \{L, R\}, a \in \Sigma$$

$$(X, \#) \stackrel{(X,\#,\varepsilon)}{\longrightarrow} (X, \#) \quad \text{for all } X \in \{L, R\}$$

$$(X, a) \stackrel{(X,a,bc)}{\longrightarrow} (X, c) \,.\, (X, b) \text{ for all } X \in \{L, R\}, a, b, c \in \Sigma'$$

Behaviour of the GTRS

Note: $\mathcal{G}$ can emulate the (single) behaviour of $\mathcal{M}$, but additionally it can do completely different things.

Our LTL formula ϕ will have the following meaning: if $\mathcal{G}$ behaves like $\mathcal{M}$, then it reaches the accepting state, i.e., ϕ has the form

$$\phi_{\mathcal{M}} \rightarrow \mathbf{F}\, \textit{Halt}$$

$\phi_{\mathcal{M}}$ specifies how $\mathcal{G}$ should behave in order to emulate $\mathcal{M}$.

Construction of $\phi_{\mathcal{M}}$

Each step of $\mathcal{M}$ modifies the three components; we demand that $\mathcal{G}$ works on them in turns.

$$\psi := Q \;\wedge\; \mathbf{G}(Q \to (\mathbf{X}\, L \;\wedge\; \mathbf{X}\,\mathbf{X}\, R \;\wedge\; \mathbf{X}\,\mathbf{X}\,\mathbf{X}(Q \vee \mathit{Halt})))$$

Here, Q, L, R are abbreviations for the disjunction of all actions starting with Q, L, R, respectively.

For each transition $t = (p, a, X, q, b) \in \delta$, we define a formula that is true iff a sequence of three steps corresponds to an execution of t.

for $X = N$: $\phi_t := (Q, p, q) \wedge \mathbf{X}(L, a, b) \wedge \mathbf{X}\,\mathbf{X} \bigvee_{c \in \Sigma'} (R, c, c)$

for $X = L$: $\phi_t := (Q, p, q) \wedge \mathbf{X}(L, a, \varepsilon) \wedge \mathbf{X}\,\mathbf{X} \bigvee_{c \in \Sigma'} (R, c, cb)$

for $X = R$: $\phi_t := (Q, p, q) \wedge \bigvee_{c \in \Sigma'} \Big(\mathbf{X}(L, a, bc) \wedge \mathbf{X}\,\mathbf{X}(R, c, \varepsilon)\Big)$

We now define

$$\phi_{\mathcal{M}} := \psi \ \wedge \ \mathbf{G}(Q \to \bigvee_{t \in \delta} \phi_t)$$

Thus, ϕ says that each correct run (there is only one!) reaches the accepting state. This is the case iff $\mathcal{M}$, starting on the empty tape, halts, which is undecidable.

Part 16: Outlook

SAT-based techniques

Bounded model checking (BMC)

Unwind the transition system with the property for k of steps

Obtain a formula that is satisfiable iff there is a counterexample for the property up to length k

Utilize modern satisfiability (SAT) solvers

Very relevant to industrial practice for *refutation*

Safety

LTL property $\mathbf{G}\, p$

Counterexample: a finite path that ends with a state s that satisfies $\neg p$

$$\exists s_0, \ldots, s_k. \quad I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg p(s_k)$$

where I denotes the initial state predicate, and T the transition relation

Creates k replicas of the transition relation

Only one level of (existential) quantification, hence a propositional satisfiability problem

SAT solvers often require conjunctive normal form (CNF); use *Tseitin transformation* in linear-time, resulting in an equi-satisfiable formula in CNF

Liveness

LTL property $\mathbf{F}\, p$

Counterexample: a finite (possibly empty) prefix followed by a finite loop, all states on the path satisfy $\neg p$

$$\exists s_0, \ldots, s_k. \quad I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k-1} \neg p(s_i) \wedge \bigvee_{i=0}^{k-1} s_k = s_i$$

LTL

Syntactic translations follow the syntactic structure of the LTL property

"Semantic" translations are based on the Büchi automaton $\mathcal{A}_{\neg\varphi}$

Completeness

Generally incomplete, but e.g.

Programs with worst-case execution time (WCET)

Other uses of SAT: Inductive techniques, proof generalization using interpolation, ...

Inductive techniques

A state property P is an inductive invariant if

1. P holds in the initial state, i.e.,

$$I \Longrightarrow P$$

   and

2. P holds in all states reachable from states that satisfy P, i.e.,

$$(P(s) \wedge T(s, s')) \implies P(s').$$

Often need be strengthened

k-Induction

An automated strengthening technique

Increase the depth of the unwinding (forms a formula similar to a BMC instance)

1. Check that there is no counterexample of length k or less
2. Check that no state reachable from a sequence of k-states that satisfy P violates P

Further streams

Combine model checking with

Data-flow analysis

Testing: e.g. white-box fuzzing

Deduction

Model checking for

Parametrized systems

Security

Reactive synthesis (see the course *Games on graphs*)

Quantitative systems (see the course *Quantitative verification*), cyber-physical systems

- Real-time systems
- Probabilistic systems
- Hybrid systems