Automata-based model checking for LTL

[Automated Reasoning, 2015/2016 1b — Lectures 4-5]

Doina Bucur

d.bucur@rug.nl

Dec 2015

In this lecture...

Omega-acceptance and Büchi automata

- Standard finite-state automata with finite runs
- Some intuition for what follows
- FSA with infinite accepting runs: Büchi automata

Automata-based model checking

- Unified formalism: ω-automata
- Model checking algorithm
- Graph searching algorithm

Historical and bibliographical notes

Assignment

In the following two lectures, you see one of the main model checking algorithms; this is how SPIN works internally. It translates both (i) the Kripke structure, and (ii) the LTL formula into an equivalent Finite-State Automaton (FSA) (where an accepting word expresses a violation of the original formula). Model-checking is then simply checking whether the **intersection** of these two automata is **empty**.

(Explicit-state) model checking

Specification language: a temporal logic (LTL) formula

Checking algorithm: **exhaustive search of the state space** of the finite-state system, to prove whether the specification is true or not

Advantages:

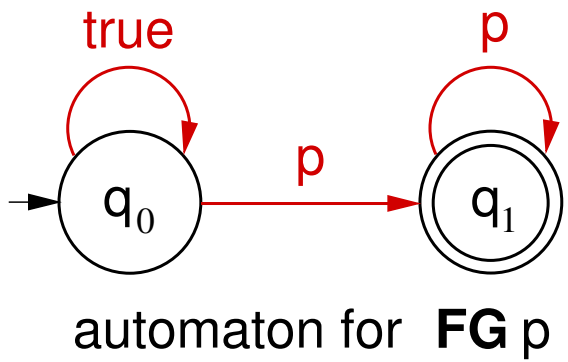
- The specification language can express complex properties.
- Many model checkers are implemented.
- The runtime is fast in many practical cases.
- Counterexamples are provided by many model checkers.

Disadvantages:

- The state-space explosion is a problem; it is caused by large number of threads, or large memory.
- This means that worst-case complexity of model checking will be high.

Automata can also write specifications

You can also express temporal specifications as automata[1].



However, the complexity of expressing a specification with automata is higher than with temporal logics. The same goes for operating on a specification (think how hard it is to complement an automaton). This is why temporal logics are more widely used as specification languages.

[1] Also, with other logics, and regular expressions.

Why automata as specification language?

Nevertheless, rewriting specifications as automata is central to this main checking algorithm for temporal-logic model checking. SPIN is an implementation of this algorithm.

Key idea: Express LTL (and also Kripke structures) into the same formalism: **finite-state automata**.

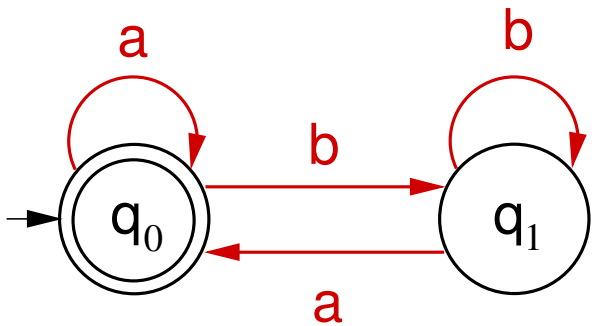
However, liveness properties cannot be expressed by standard finite-state automata and their finite accepted words. Thus, we first have to update the classical definition of finite-state automata.

Standard finite-state automata

Definition (Finite-state automaton, FSA)

A FSA $\mathcal{A}$ is a tuple $(\Sigma, Q, \Delta, Q_0, F)$, where

- The state space Q is finite.
- The initial states are $Q_0 \subseteq Q$, and the final states $F \subseteq Q$.
- Σ is a finite alphabet of symbols.
- Δ is the transition relation, $\Delta \subseteq Q \times \Sigma \times Q$.
  $\mathcal{A}$ is called deterministic iff $\forall q \in Q, \forall a \in \Sigma$,
  $((q, a, q') \in \Delta$ and $(q, a, q'') \in \Delta) \rightarrow q'$ is q''.



A classic finite–state automaton

Here, $\Sigma = \{a, b\}$.

Words and runs over FSA

Say v is a word (also: string or sequence) of length $|v|$ over the alphabet Σ in the automaton $\mathcal{A}$. The ith symbol in v is written $v[i]$.

A run of $\mathcal{A}$ over the word v is a sequence of states q_i starting in an initial state $q_0 \in Q_0$, so that a transition in $\mathcal{A}$ leads from state q_i to state q_{i+1} over symbol $v[i]$, i.e.

$$\forall 0 \leq i < |v| \quad (q_i, v[i], q_{i+1}) \in \Delta.$$

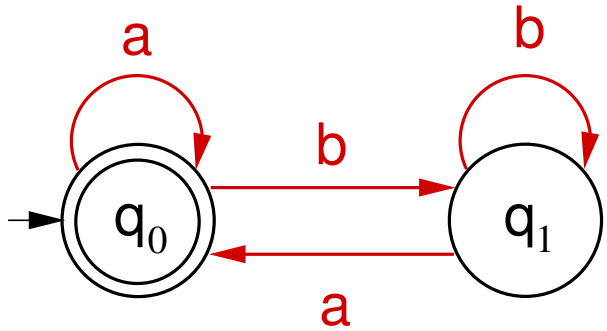
- We say that $\mathcal{A}$ reads word v if a run of $\mathcal{A}$ over v exists.
- In general, runs may be infinite.

Standard acceptance over FSA

Definition (Standard acceptance)

An accepting run of a FSA $\mathcal{A}$ is a finite run which ends in a final state from set F.

$\mathcal{A}$ accepts word v iff there exists an accepting run of $\mathcal{A}$ over v.



This FSA accepts the empty word, a, ba, aba, $aabba$, etc.

The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$, is the set of all words accepted by $\mathcal{A}$; any such language is **regular**.

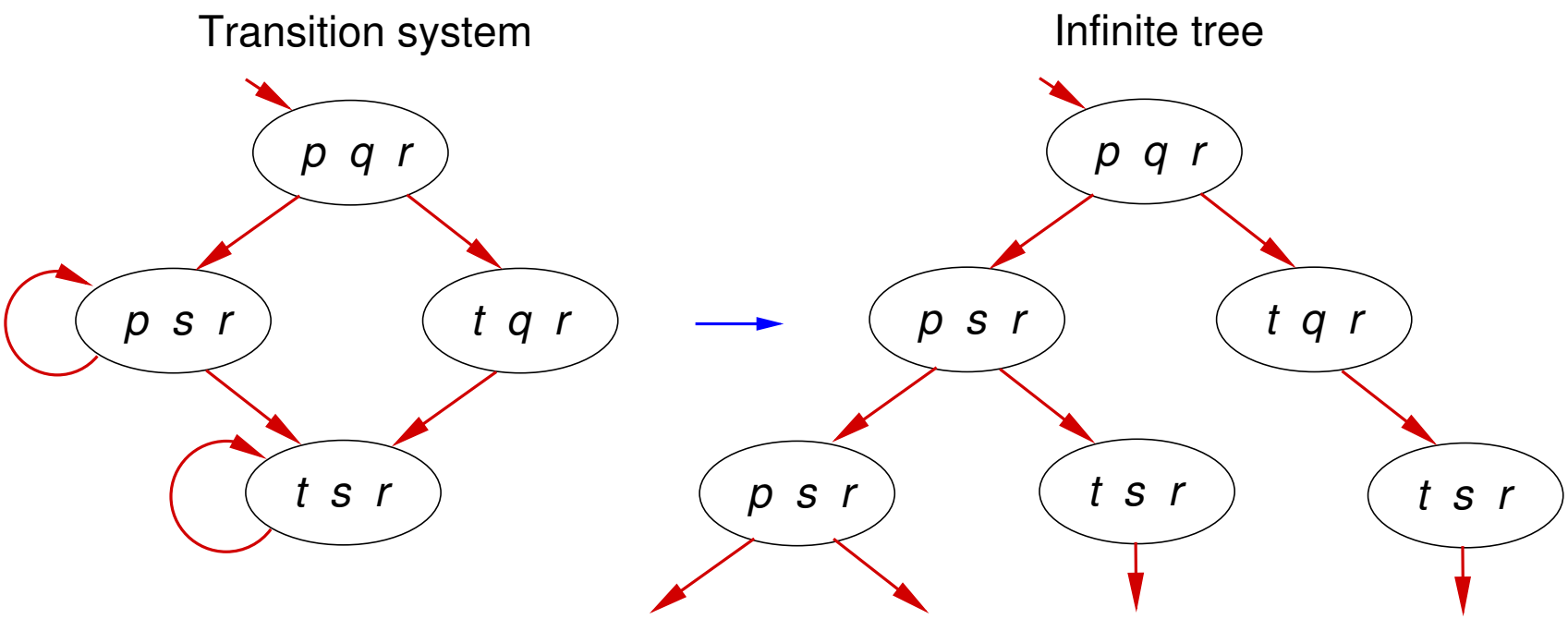
This $\mathcal{L}(\mathcal{A})$ is described by the **regular expression** $\epsilon \mid (a \mid b) * aa*$.[2]

[2] The $*$ operator is the *Kleene star*, a unary operator meaning "zero or any finite number of". $\mid$ is the binary operator indicating choice.

Some intuition (1)

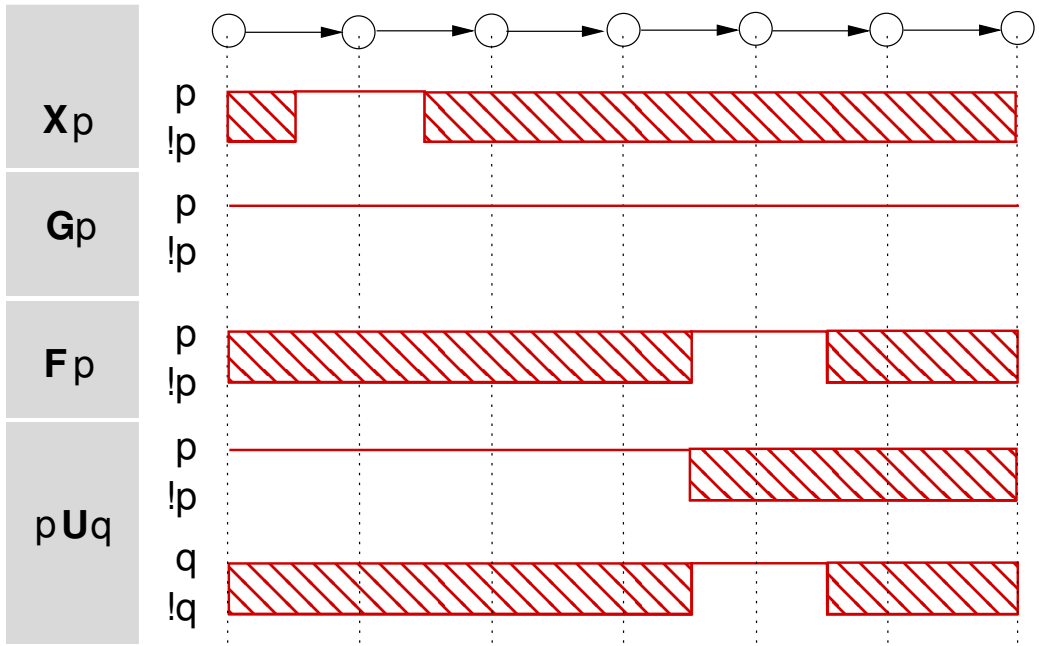
The FSA symbols $a \in \Sigma$ don't have particular meaning in the theory of FSA; what is their meaning for us?

Remember: LTL formulas describe properties of infinite execution paths in Kripke structures with state labels. A state label is a $p \in AP$ which holds in that state; label *true* will hold in any state, and label *false* will hold in no state.



Some intuition (2)

A formula is true or false depending on the sequence of state labels on that path:



Let's rewrite the same sequences
(assuming infinite execution paths, which is most general):

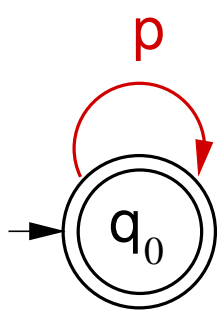
- true p true$^{\text{infinitely many}}$
- $p^{\text{infinitely many}}$
- true$^{\text{finitely many}}$ p true$^{\text{infinitely many}}$
- $p^{\text{finitely many}}$ q true$^{\text{infinitely many}}$

Some intuition (3)

These look exactly like infinite FSA runs, where the FSA symbols $a \in \Sigma$ are (boolean expressions over) the atomic propositions $p \in AP$.

We can write small FSA having such a run. We also try to add accepting states; the original formula holds if the run reaches an accepting state:

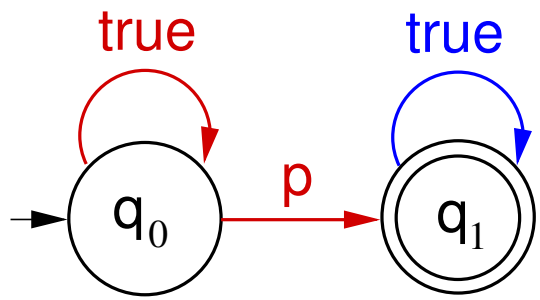
- $p^{\text{infinitely many}}$



automaton for **G** p

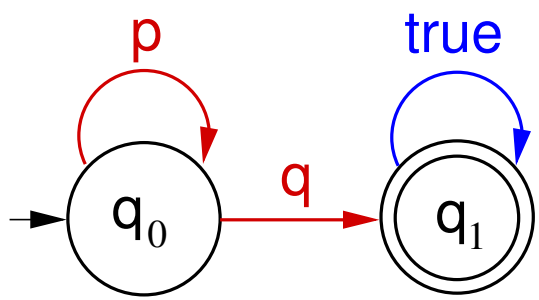
Some intuition (4)

- true$^{\text{finitely many}}$ p true$^{\text{infinitely many}}$



automaton for **F** p

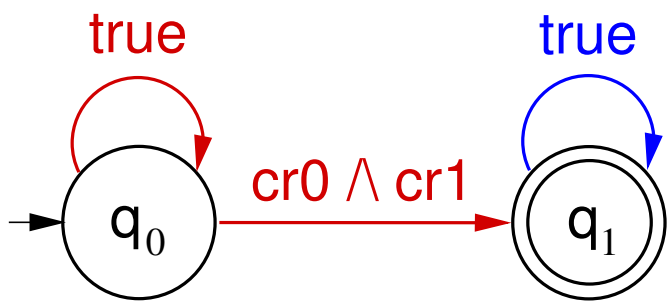
- $p^{\text{finitely many}}$ q true$^{\text{infinitely many}}$



automaton for p **U** q

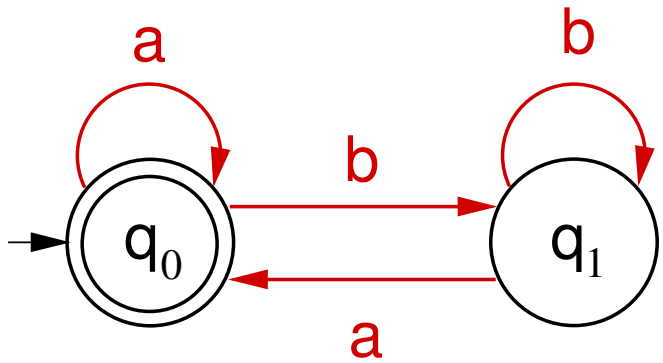
Some intuition (5)

- A negated mutual exclusion $\neg\mathbf{G}\neg(cr_0 \wedge cr_1)$, or $\mathbf{F}(cr_0 \wedge cr_1)$:



automaton for not mutual exclusion

Towards FSA with infinite accepting runs



Standard FSA can only have finite accepting runs. We are also interested in infinite accepting runs, such as *aaa*... ($a^{\text{infinitely many}}$).

An infinite run is called an omega-run, written ω-run. For some infinite run σ:

- σ^{ω} denotes the set of states which appear infinitely often;
- σ^{+} denotes the set of states which appear only finitely many times.

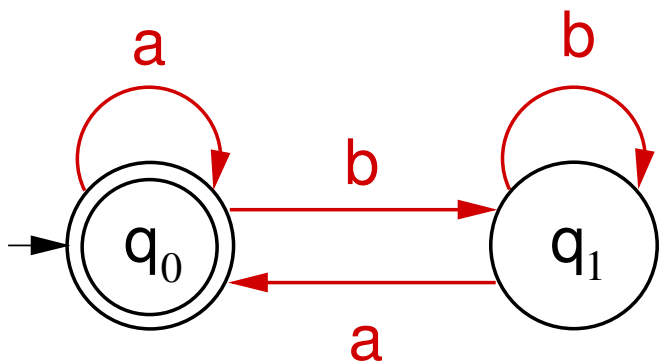
Büchi acceptance

Infinite runs over FSA need a new definition for acceptance; the most used one is Büchi acceptance.

Definition (Büchi acceptance)

An accepting ω-run of FSA $\mathcal{A}$ is any ω-run σ such that $\exists q_f.\ q_f \in F$ and $q_f \in \sigma^\omega$.

This means that an infinite run is accepted iff it visits a final state infinitely many times.



This FSA Büchi-accepts the ω-runs $a^{\text{infinitely many}}$, $(ba)^{\text{infinitely many}}$, $(bbba)^{\text{infinitely many}}$, etc.

Unifying standard and Büchi acceptance

For convenience, we need a way to also include standard acceptance into Büchi acceptance.

This is done by artificially extending finite runs into infinite runs. We extend the alphabet Σ with a new "null" symbol ϵ, to label a transition which can always execute.

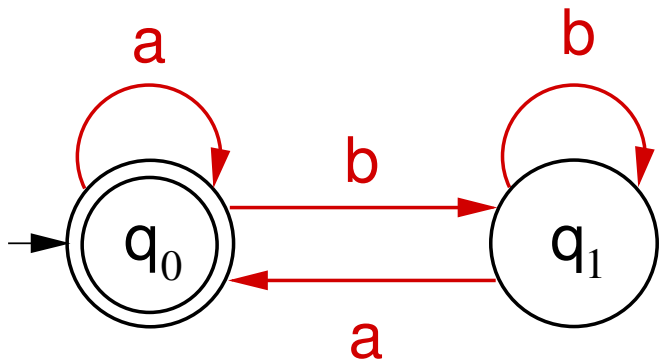
Definition (Stutter extension)

The stutter extension of a finite run σ with final state q_n is the ω-run $\sigma \cdot (q_n, \epsilon, q_n)^\omega$.[3]

This makes the final state of the finite run persist forever. It is easily proven that standard acceptance with stutter extension equals Büchi acceptance.

[3] A matter of notation: ω used as a superscript represents infinite repetition, and $\cdot$ represents concatenation.

Büchi automata



Accepting ω-runs can always provably be written as

- A finite prefix regular expression executed once, e.g. ba or $(ba)^3$, and
- A finite suffix regular expression repeated infinitely, e.g. a^ω or $(ba)^\omega$.

These expressions are called ω-regular expressions.
The class of properties they express is called ω-regular properties.
Automata with accepting ω-runs are called ω-automata.
Automata with Büchi acceptance conditions are called Büchi automata[4].

[4] There are other formalisms for ω-acceptance and ω-automata: Muller, Rabin, Streett acceptance and automata. They all express the same class of ω-regular properties as Büchi automata.

Properties of Büchi automata

It is proven that for every LTL formula there exists a Büchi automaton which accepts precisely the runs which satisfy the formula.

The upper bound of the Büchi state space is **exponential** in the size of the LTL formula (i.e., the number of atomic propositions in the formula)[5].

It has been shown that the following properties of Büchi automata are decidable:

- Language emptiness, i.e. whether the set of accepting runs of a given Büchi automaton is empty;
- Language intersection, i.e. generating a new Büchi automaton which accepts precisely those ω-runs accepted by all Büchi automata from a given set.

To obtain Büchi automata from LTL, use `spin -f '[]p'`, or the web engine `https://spot.lrde.epita.fr/trans.html`, where you choose the output Büchi Automaton, then "a state-based degeneralized Büchi automaton". These two algorithms are not identical, and may output different solutions.

[5] `http://en.wikipedia.org/wiki/Linear_temporal_logic_to_Buchi_automaton`

SPIN use

Key idea in what follows: The SPIN model checking problem is equivalent to the emptiness test for an intersection of two given Büchi automata. This emptiness test boils down to a depth-first search algorithm.

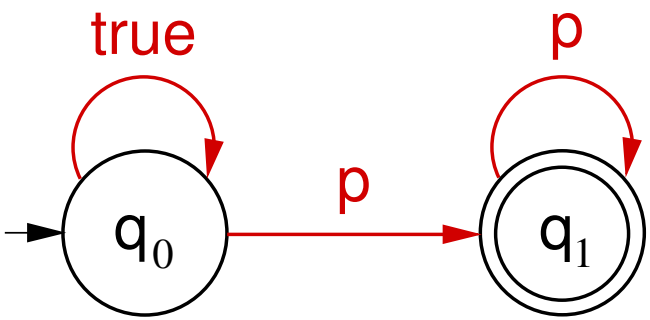
SPIN takes LTL formulas natively in PROMELA[6]; these LTL formulas are taken as **positive** properties that must be satisfied by the program.

SPIN includes a LTL-to-Büchi translation, see e.g. `spin -f '[]p'` (not guaranteed to be minimal)[7]. A given LTL formula is first **negated** internally, then translated into an automaton written as a `never` claim with `accept` state labels. The existence of an accepting run in this `never` claim is signalled as a violation of the original LTL formula.

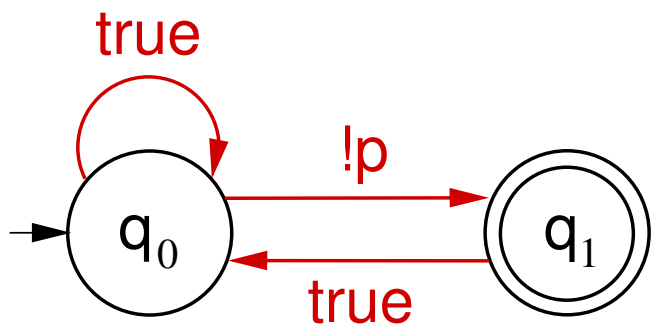
[6] Have a look at the relevant manual page, `http://spinroot.com/spin/Man/ltl.html`.

[7] The translation algorithm is quite complex; see Clarke's Model Checking book, p. 132.

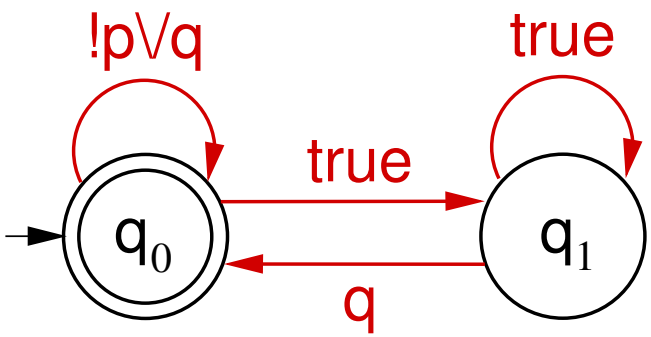
More examples



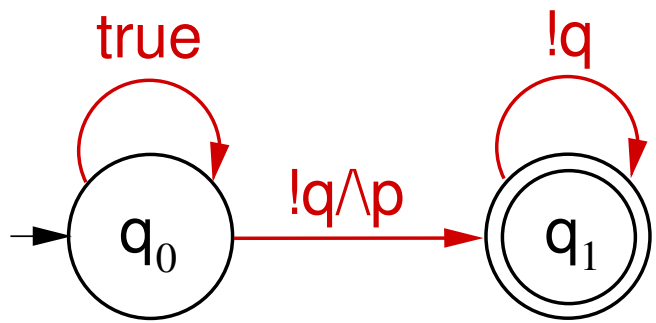
automaton for **FG** p



automaton for ! **FG** p



automaton for **G**(p –>**F** q)



automaton for **F**(p ∧ **G** !q)

Transition labels: some necessary clarifications

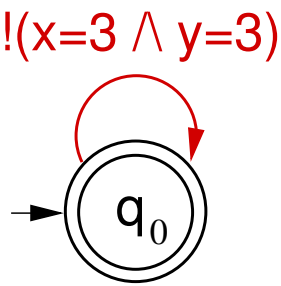
We wrote **transition labels** on FSA both as:

- subsets of AP, $\{x = 3, y = 3\}$ — this follows the definitions — and
- boolean formulas $x = 3 \wedge y = 3$.

Say that $AP = \{p, q, r, t\}$. When a transition is labelled with a **boolean formula**, this is in fact a short representation for a number of actual transitions:

- The label $p \vee q$ means any of the transitions labelled $\{p\}$ or $\{q\}$ or $\{p, q\}$ or $\{p, q, r\}$, etc.
- The label $p \wedge q$ means any of the transitions labelled $\{p, q\}$ or $\{p, q, r\}$ or $\{p, q, t\}$, etc.
- The label $\neg p$ means any of the transitions labelled $\{q\}$ or $\{q, r\}$ or $\{q, r, t\}$, etc.
- The label true means any transition.

LTL formulas to ω-automata (reminder)



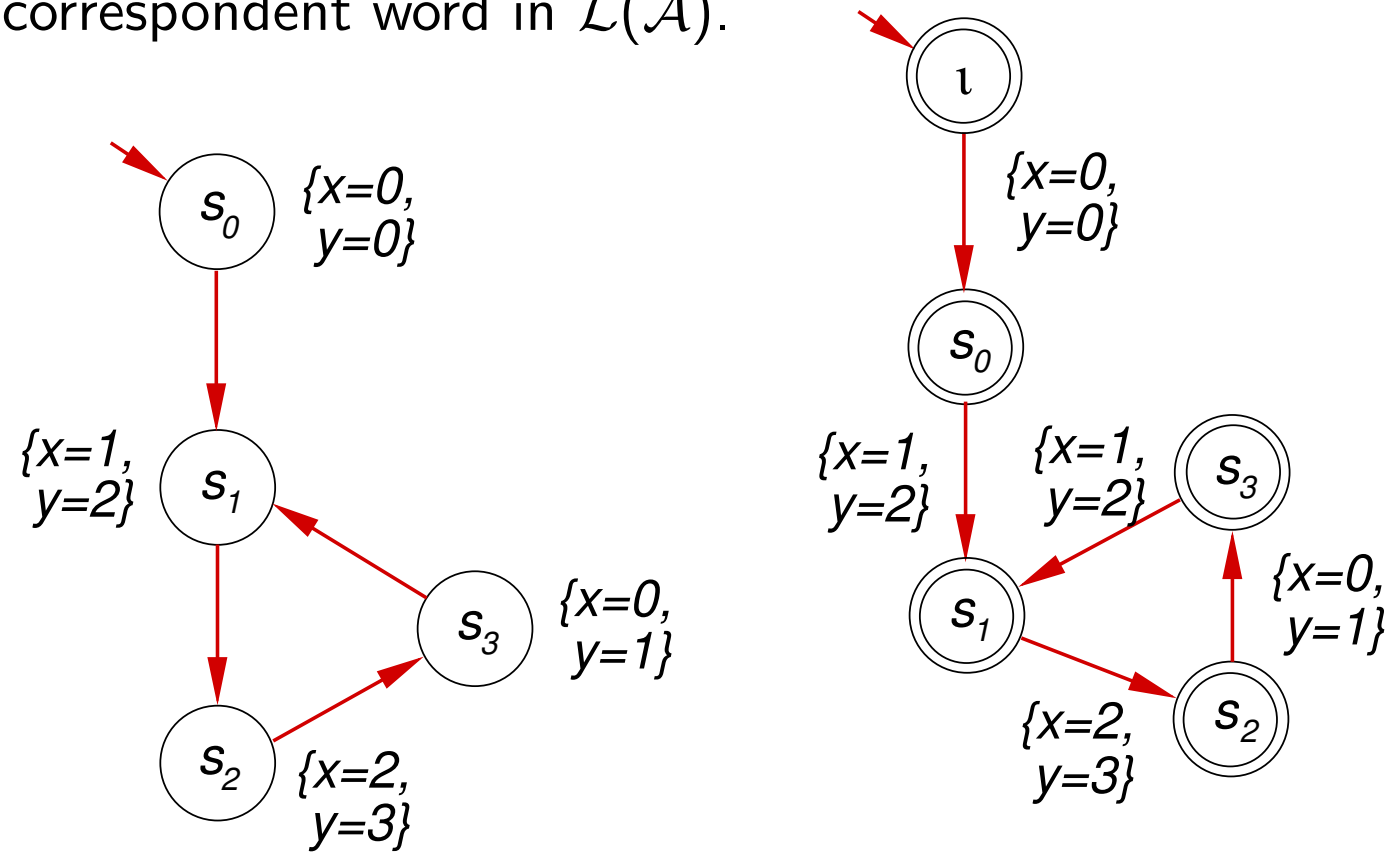
Specification ! **F** (x=3 /\ y=3)
as automaton

Also: The upper bound of the state space of the automata is **exponential** in the size of the LTL formula (i.e., the number of atomic propositions in the LTL formula).

Kripke structures to ω-automata (1)

Any Kripke structure M has an equivalent ω-automaton $\mathcal{A}$

M and $\mathcal{A}$ are equivalent iff the sequence of state labels on any path π in M has a correspondent word in $\mathcal{L}(\mathcal{A})$.



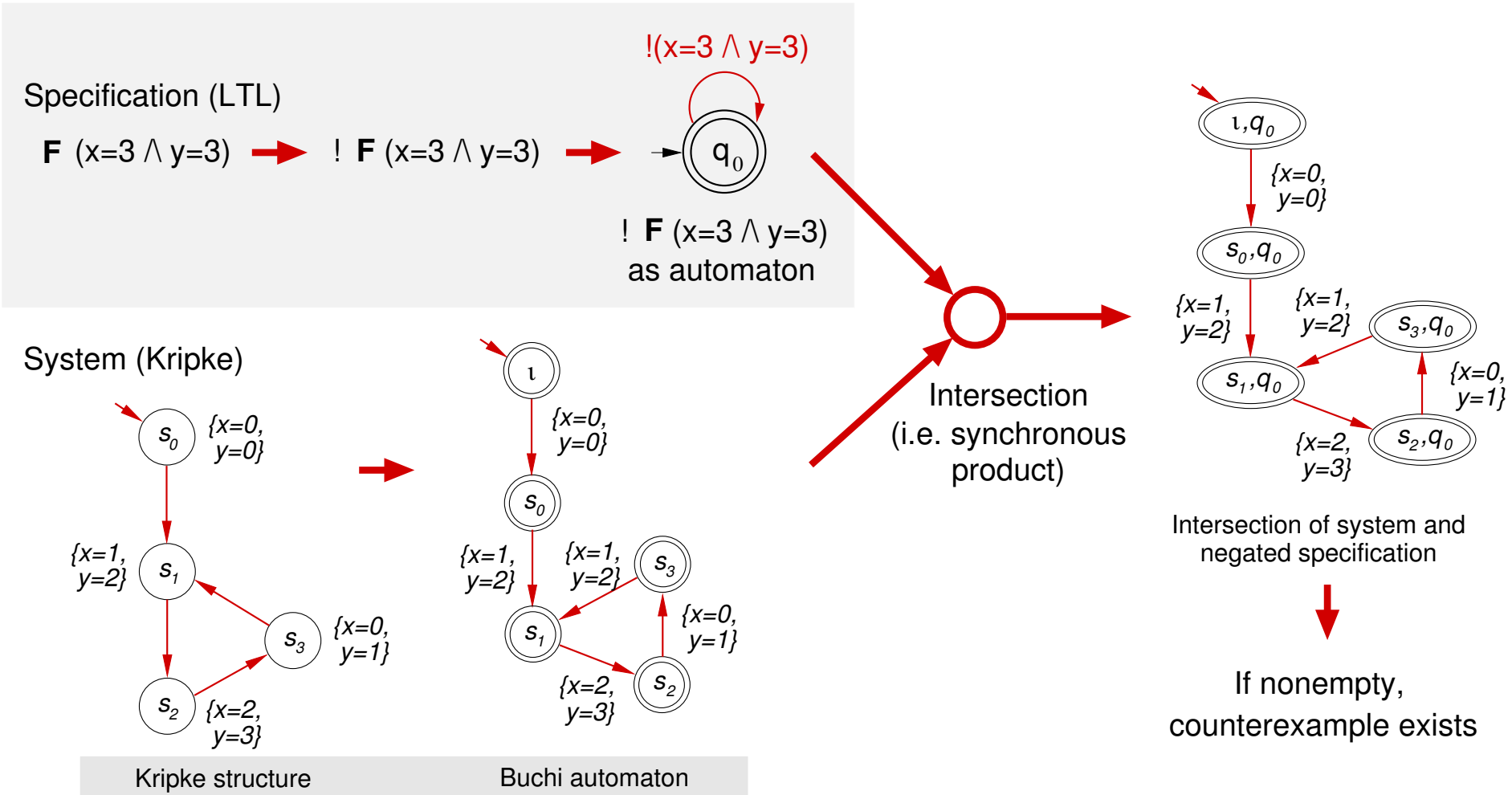
Kripke structure Buchi automaton

Kripke structures to ω-automata (2)

The translation algorithm is simple:

1. Take Kripke structure $M = (S, S_0, T, L)$ where $L : S \to \mathcal{P}(AP)$.
2. Write the FSA $\mathcal{A} = (\Sigma, Q, \Delta, Q_0, F)$, where:
   - The **alphabet** is the set of subsets of AP, i.e. $\Sigma := \mathcal{P}(AP)$.
   - We add an **initial state** ι, i.e. $Q_0 := \{\iota\}$, $Q := S \cup \{\iota\}$ and $F := S \cup \{\iota\}$.
   - The new **transitions** are: $(\iota, \alpha, s) \in \Delta$ iff $s \in S_0$ and $\alpha = L(s)$, and $(s, \alpha, s') \in \Delta$ iff $s, s' \in S$, $(s, s') \in T$ and $\alpha = L(s')$.

Roadmap for automata-based model checking



Now to automata-based model checking: Basic idea of the algorithm

- Any **temporal specification** can be represented as an ω-automaton $\mathcal{S}$ (as we have seen last time).
- The **system model**, also: a Kripke structure directly corresponds to an ω-automaton $\mathcal{A}$ (as we have just seen now).
- The system $\mathcal{A}$ satisfies the positive property $\mathcal{S}$ when

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$$

Satisfaction of a specification

The system $\mathcal{A}$ **satisfies** a positive property $\mathcal{S}$ (both as ω-automata) when

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$$

I.e., each system execution is among the behaviours accepted by the specification.

If we define the **complement** of language $\mathcal{L}(\mathcal{S})$ to be

$$\overline{\mathcal{L}(\mathcal{S})} := \Sigma^\omega \setminus \mathcal{L}(\mathcal{S})$$

we can write the satisfaction of a specification in terms of **intersection of languages**:

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$$

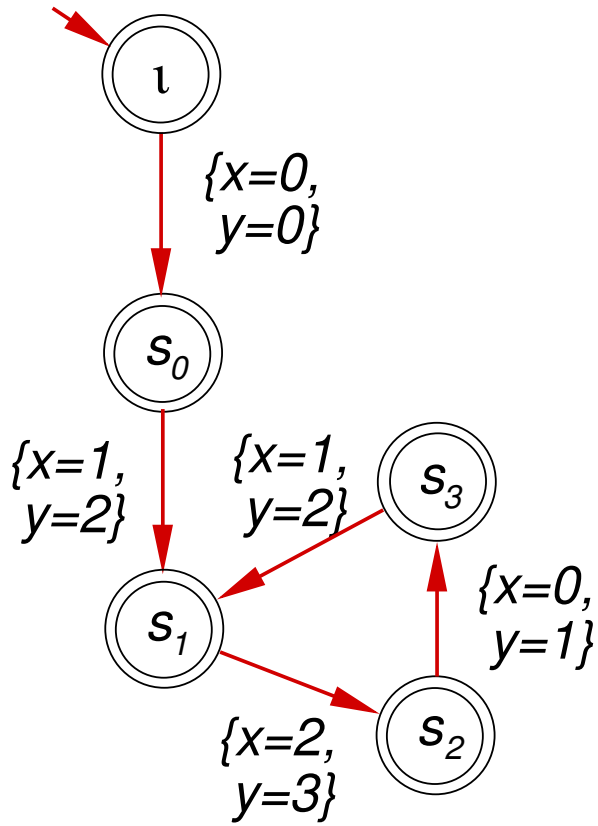
Model-checking procedure

As already suggested:

1. Take a specification as automaton $\mathcal{S}$. This describes a **positive** property which should not be violated.
   - (SPIN takes this either in positive LTL form, or directly in negative Büchi automaton form.)
2. **Complement** automaton $\mathcal{S}$, i.e. construct an automaton $\overline{\mathcal{S}}$ which accepts the language $\overline{\mathcal{L}(\mathcal{S})}$.
   - (SPIN calculates this complement automaton directly out of the positive LTL formula above, by first negating it, and then translating this negation into an automaton $\overline{\mathcal{S}}$.)
3. Construct the automaton which accepts the **intersection** of languages $\mathcal{L}(\mathcal{A})$ and $\overline{\mathcal{L}(\mathcal{S})}$.
4. If this intersection is empty, $\mathcal{S}$ **holds** for $\mathcal{A}$.
5. Otherwise, a **counterexample** is provided. An infinite counterexample (i.e. an ω-run) is expressed finitely as uv^ω, with u a finite prefix and v a finite suffix (as on slide 17).

Example (step 1)

System $\mathcal{A}$ (left) and a **positive** liveness specification $\mathcal{S}$ in LTL (right):

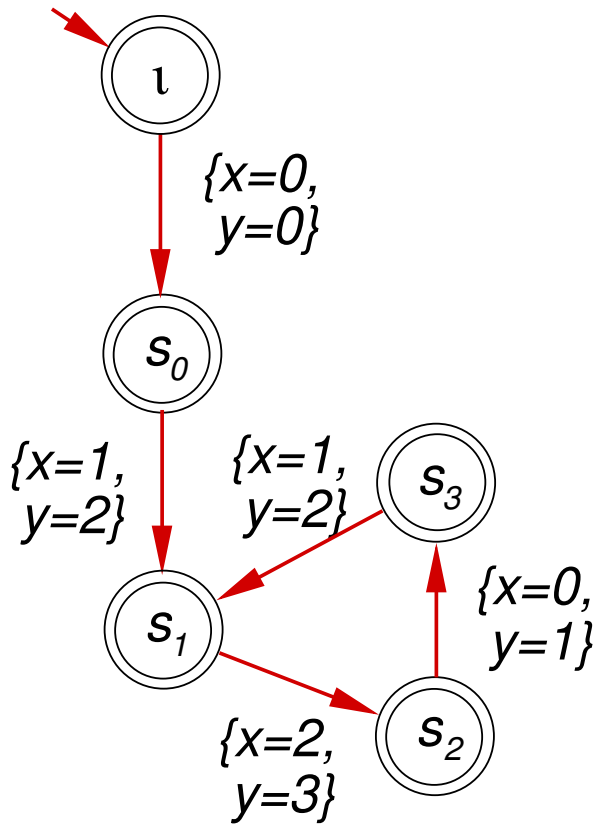


System as automaton

$$\mathbf{F}(x = 3 \wedge y = 3)$$

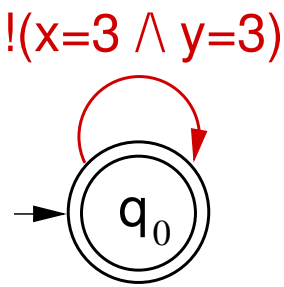
Example (step 2)

System $\mathcal{A}$ with **negative** specification $\overline{\mathcal{S}}$ as automaton:



System as automaton

$\mathbf{F}(x = 3 \wedge y = 3)$
is negated as
$\neg\mathbf{F}(x = 3 \wedge y = 3)$
which is translated as:



Specification ! **F** (x=3 /\ y=3)
as automaton

Calculating the intersection of two automata

Take two FSA,

- $\mathcal{B}_1 = (\Sigma, Q_1, \Delta_1, Q_1^0, F_1)$ and
- $\mathcal{B}_2 = (\Sigma, Q_2, \Delta_2, Q_2^0, F_2)$.

We want $\mathcal{B}_1 \cap \mathcal{B}_2$ which accepts $\mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$. We give an algorithm which is not general[8], but is sufficient when all states of one automaton are accepting (here, say the first automata $\mathcal{B}_1$ has $Q_1 = F_1$).

$\mathcal{B}_1 \cap \mathcal{B}_2 := (\Sigma, Q := Q_1 \times Q_2, \Delta, Q_0 := Q_1^0 \times Q_2^0, F := F_1 \times F_2)$,
where for any **symbol** $a \in \Sigma$,
an **edge** $((r_i, q_j), a, (r_m, q_n)) \in \Delta$ iff
$(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$.

[8] For the general algorithm, and information on complementing Büchi automata, see Ch. 9, *Model Checking and Automata Theory*, from *Model Checking*, E. M. Clarke, O. Grumberg, D. A. Peled.

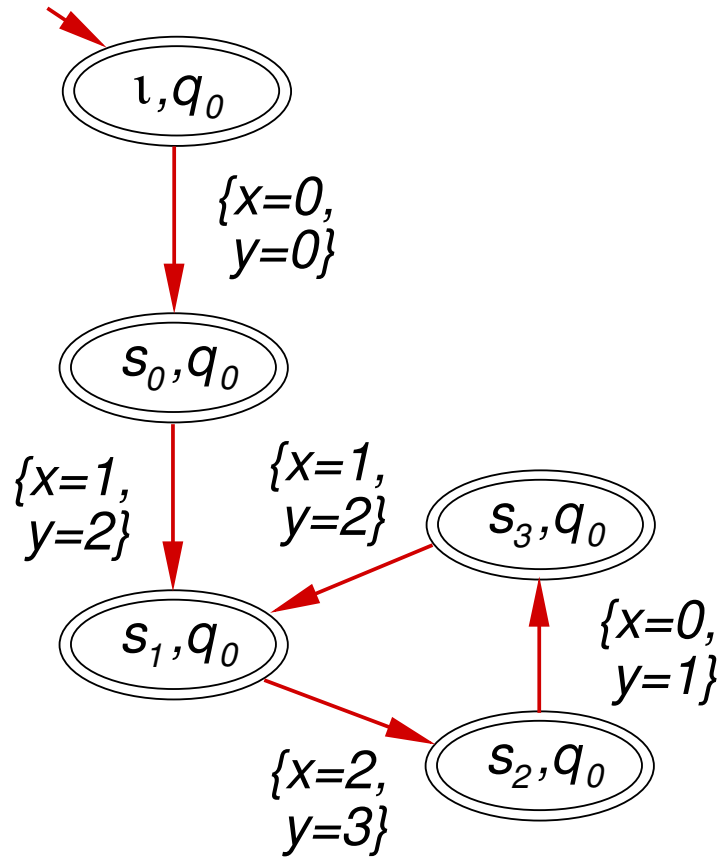
Example (step 3)

Some number-crunching for the calculation of the intersection in this example:

$\Sigma := \mathcal{P}\{x = 0, x = 1, x = 2, x = 3, y = 0, y = 1, y = 2, y = 3\}$
$Q := \{\iota, s_0, s_1, s_2, s_3\} \times \{q_0\}$
$Q_0 := \{\iota, q_0\}$
$F := Q$

Example (result from step 3)

And the intersection $\mathcal{A} \cap \overline{\mathcal{S}}$ is:



Intersection of system and negated specification

Emptiness of language

Reminder—steps 4 and 5:

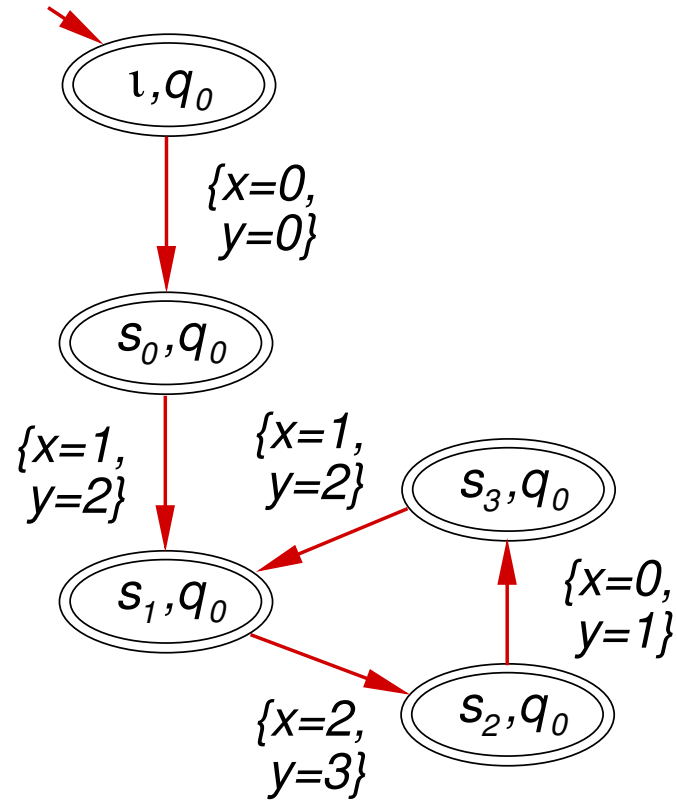
4. If this intersection is empty, $\mathcal{S}$ **holds** for $\mathcal{A}$.
5. Otherwise, a **counterexample** is provided. An infinite counterexample (i.e. an ω-run) is expressed finitely as uv^ω, with u a finite prefix and v a finite suffix.

A language $\mathcal{L}(\mathcal{B})$ is **empty** if $\mathcal{B}$ has no accepting ω-runs.

This means that:
A language $\mathcal{L}(\mathcal{B})$ is **nonempty** iff there exists a reachable accepting state with a cycle back to itself.

Example (step 4)

The intersection $\mathcal{A} \cap \overline{\mathcal{S}}$ is...



Intersection of system and negated specification

... **nonempty**. Three accepting states exist with a cycle back. This means that the original specification $\mathbf{F}(x = 3 \wedge y = 3)$ **does not hold**.

And finally, the counterexample?

This algorithm is constructed in this way so that the **counterexample** is exactly that run reaching an accepting state, then cycling back to it.

The nonemptiness test thus solves both steps 4 and 5.

This test can be implemented as a depth-first search algorithm (DFS), with two searches: one for any accepting state, and the other for a cycle back to that state. The resulting counterexample will not be the shortest.

Example (step 5)

An **accepting run** and **counterexample** is:

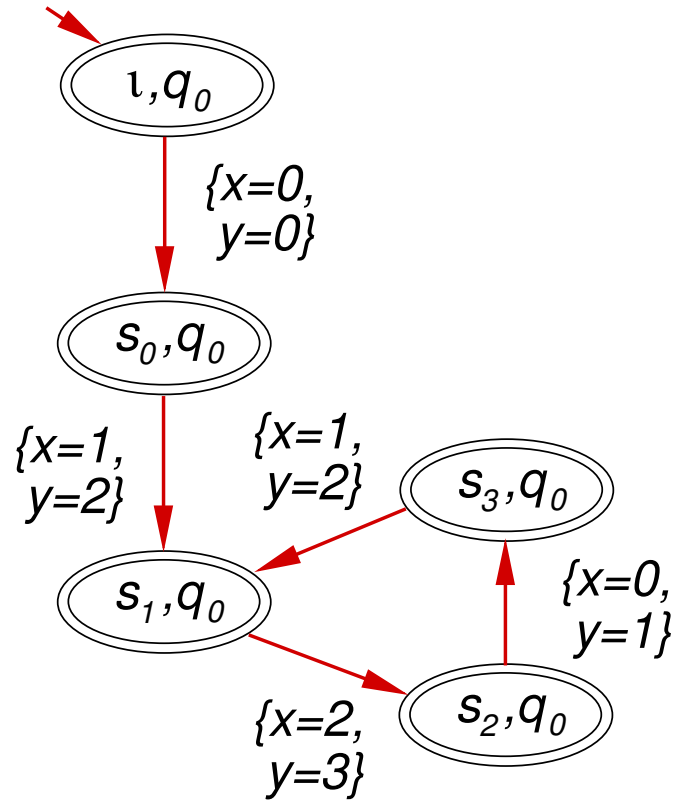
$$((\iota, q_0), \{x = 0, y = 0\}, (s_0, q_0)),$$

$$((s_0, q_0), \{x = 1, y = 2\}, (s_1, q_0)),$$

$$[((s_1, q_0), \{x = 2, y = 3\}, (s_2, q_0)),$$

$$((s_2, q_0), \{x = 0, y = 1\}, (s_3, q_0)),$$

$$((s_3, q_0), \{x = 1, y = 2\}, (s_1, q_0))]^{\omega}.$$



Intersection of system and negated specification

The double depth-first search algorithm, $\mathcal{O}(|Q| + |\Delta|)$

nonemptiness()
for all $q_0 \in Q_0$
    dfs1(q_0)
return False, i.e. no counterexample exists

dfs1(q)
    mark q
    for all successors q' of q
        if q' not marked **then** *dfs1*(q')
    if *accept*(q) **then** *dfs2*(q)

dfs2(q)
    mark q
    for all successors q' of q
        if q' marked **then return** True, i.e. counterexample found
        else if q' not marked **then** *dfs2*(q')

Model-checking "on-the-fly"

Note: This model checking procedure allows to skip on generating the entire, large system model $\mathcal{A}$ at each verification run.

We can instead generate the specification automaton $\mathcal{S}$, and compute the intersection while also doing the nonemptiness check. This may result in constructing only a small portion of the Kripke structure before a counterexample is found.

Note: this depth-first search (DFS) can also be implemented as a breadth-first search (BFS).

However, a DFS algorithm has the great advantage that the counterexample searched is exactly the two *stacks* of states visited by the two DFS rounds in the algorithm. (The stack maintenance was not shown in the algorithm, for simplicity.)

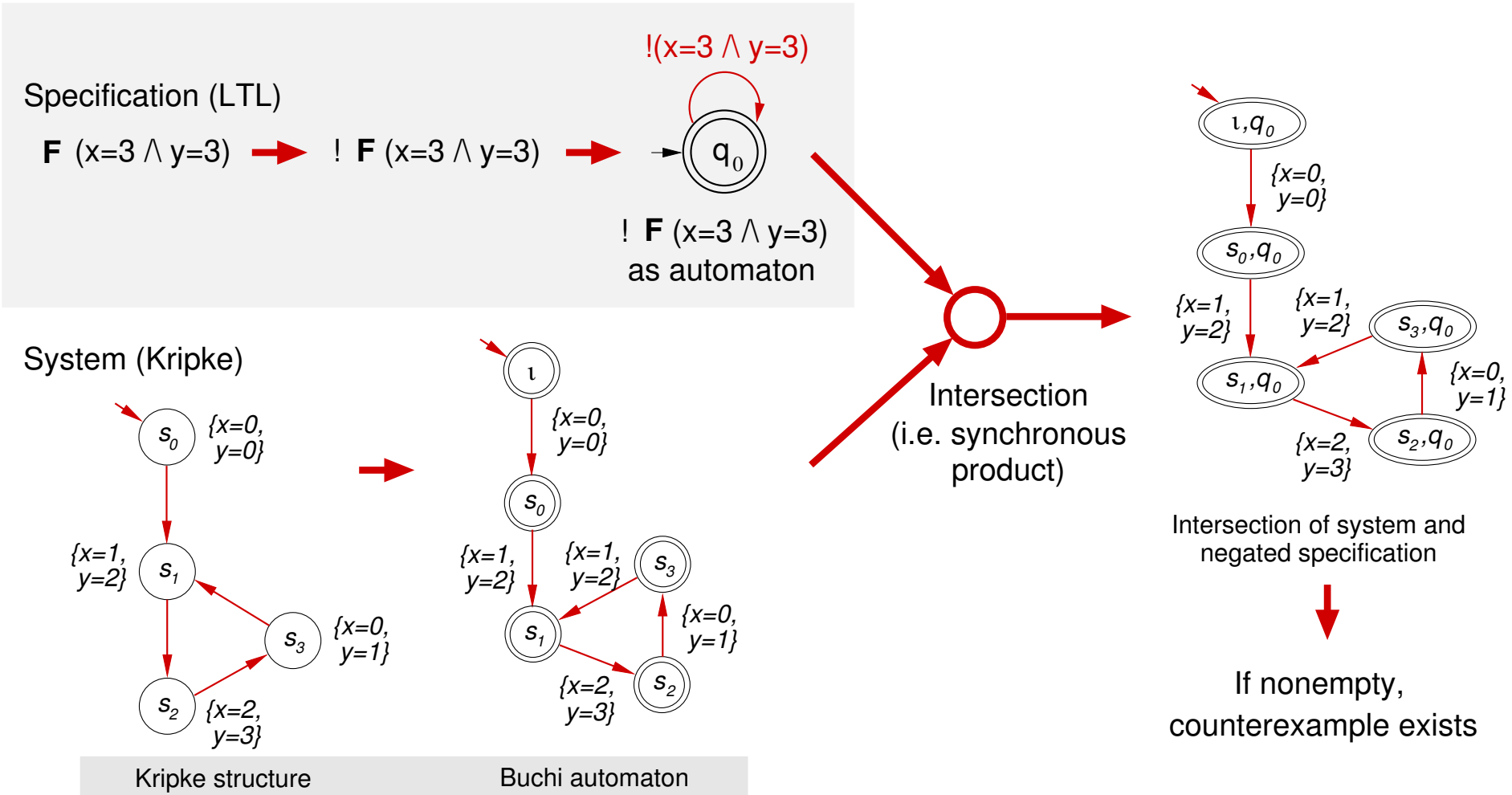
The SPIN implementation

- SPIN cuts corners compared to the algorithm above (which is as general as possible). When given a **safety specification** $\mathcal{S}$ (assertions, deadlock, invariants):

  SPIN safety checking is simply a standard DFS algorithm over the system model $\mathcal{A}$ itself, where at each state, the state labels are evaluated with regard to $\mathcal{S}$. This is a sufficient solution to locate an error and give a counterexample (not the shortest, as before).

- SPIN can also take a **depth bound** for the DFS search. All guarantees of finding a violation of the safety specification disappear now, even within the depth bound. To solve this, it also implements breadth-first search (BFS), which brings back the guarantees within the depth bound.

End of roadmap for automata-based model checking

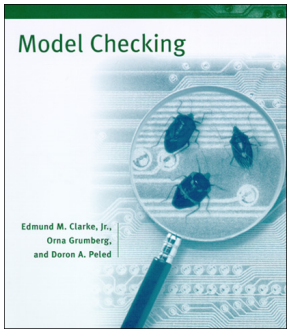


Historical notes

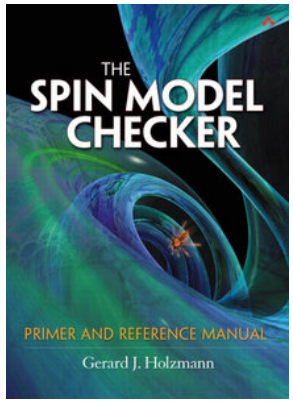
The basic theory of finite automata was developed in the 50s. The theory of ω-automata is almost as old: Büchi's work started in 1960. The correspondence between temporal logic and Büchi automata was first described in 1983.

The first DFS algorithm linear in the size of the graph is due to Tarjan (1972). The nested DFS algorithm dates from 1990.

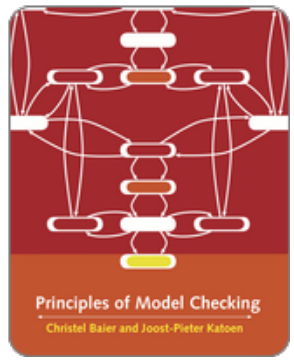
Bibliographical notes



The content in these lectures covers Ch. 9, *Model Checking and Automata Theory*, from *Model Checking*, E. M. Clarke, O. Grumberg, D. A. Peled.



This covers the automata-related text from Ch. 6, *Automata and Logic*, and Ch. 8, *Search Algorithms*, from *The* SPIN *Model Checker*, G. J. Holzmann.



This subject is described in Ch. 4, *Regular Properties*, and Section 5.2, *Automata-Based LTL Model Checking* from *Principles of Model Checking*, C. Baier, J.-P. Katoen.

Assignment (overview)

This assignment allows for some free choice.

You have a list of assignments on the following slides; they show practical uses of the Spin model checker, plus some recent scientific literature on model checking.

Solve any subset of these assignments. For a maximum grade, aim to reach 150  .

Assignment (Checking concurrent code)

(50) A piece of code (pseudocode below) needs to search for the bit value "0" in an unsorted array of bits a[] (which is to be initialized randomly). The code has two processes, P and Q. The code is considered correct if both processes terminate after one of them has found a "0". Show, using Spin, that the code is correct (or find a counterexample and explain the problem). You may need to add checks so the code doesn't go outside the bounds of a[].

```
bool found = false;     // global variables
bit a[8];               // add nondeterministic initialization for a[]

// Process P                    // Process Q
i = -1                          j = 8
while !found {                  while !found {
    i++                             j--
    if a[i] == 0                    if a[j] == 0
        found = true                    found = true
}                               }
```

Assignment (Checking a simple network protocol)

(75) The **Alternating-bit protocol (ABP)**[9] is a simple data-exchange network protocol between a sender A and a receiver B on a shared line, using retransmission of lost packets. See the existing model(s) of the protocol in the Spin sources, `abp.pml` or `Book_1991/p123.pml`; they implement communication channels with some **loss** of messages. Write yourselves also a second, simpler model, with **perfect** (lossless) communication channels. **Simulate** both.

Specify at least two behaviours for this protocol, for example:

- Every message sent by A is received error-free by B at least once;
- Every message sent by A is accepted at most once by B;
- Messages are not reordered;
- There are no non-progress cycles.

You can use assertions, LTL, progress labels, etc.[10]

Verify your properties on both types of channels and explain the results.

[9] `http://en.wikipedia.org/wiki/Alternating_Bit_Protocol`.

[10] For channel operations in PROMELA: `/Man/receive.html`, etc. Check progress with either LTL or `progress` labels, `Man/progress.html`. You may like Promela's *remote references* `.../remoterefs.html`.

Assignment (Searching for solutions to SAT instances)

(25) You can use a model checker to solve other types of problems.

Take the classic problem of searching for the solutions of a boolean equation (i.e., a SAT instance[11]). You have a small model of the problem in the Spin sources, `sat.pml`, where the boolean variables are initialized nondeterministically, and a simple `assert` is used to write the boolean equation.

Write yourselves a larger boolean equation (e.g., of 10 variables), and find **all** its solutions.

What will the size of the search space be (in theory, versus with Spin)? How long does the model checker take to do the search?

Compile with `gcc -DSAFETY -o pan pan.c` ([12]) to simplify the search to only safety properties.
Check with `time ./pan -e` to time the search and obtain **all** solutions as trails ([13]).
Simulate the Nth trail with `spin -tN -p -l sat.pml` ([14]).

[11] http://en.wikipedia.org/wiki/Boolean_satisfiability_problem
[12] http://spinroot.com/spin/Man/Pan.html#B
[13] http://spinroot.com/spin/Man/Pan.html#A
[14] http://spinroot.com/spin/Man/Spin.html

Assignment (Scientific literature on model checking)

(50) Read a recent publication in model checking. Choose any one from the list below. Write a half-page (critical) summary.

1. Ana Rosario Espada et al. **Runtime verification of expected Energy Consumption in Smartphone**. 2015 Int. SPIN Workshop on Model Checking of Software. Springer.
2. Moonzoo Kim et al. **Formal Verification of a Flash Memory Device Driver – An Experience Report**. 2008 Int. SPIN Workshop on Model Checking of Software. Springer.
3. Ezio Bartocci et al. **Towards a GPGPU-parallel SPIN model checker**. 2014 Int. SPIN Symposium on Model Checking of Software. ACM.
4. Oliver Sharma et al. **Towards Verifying Correctness of Wireless Sensor Network Applications Using Insense and Spin**. 2009 Int. SPIN Workshop on Model Checking of Software. Springer.
5. Eric Mercer and Michael Jones. **Model Checking Machine Code with the GNU Debugger**. 2005 Int. SPIN Workshop on Model Checking of Software. Springer.
6. Markus Weimann et al. **Model Checking Industrial Robot Systems**. 2011 Int. SPIN Workshop on Model Checking of Software. Springer.
7. Edmund M. Clarke. **The Birth of Model Checking**. 25 Years of Model Checking, Lecture Notes in Computer Science Volume 5000, 2008, Springer.

Assignment administration

Handing in:

- in lab session or electronically;
- either way, before the end of Wed Dec 16.

Grading:

- this assignment is expected to add up to 150  , or 150 grade centipoints, depending on your choices; any further work is also given credit.