CS 267: Automated Verification

Lecture 8: Automata Theoretic Model Checking

Instructor: Tevfik Bultan

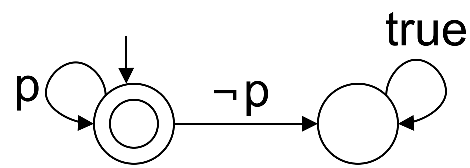
LTL Properties ≡ Büchi automata

[Vardi and Wolper LICS 86]

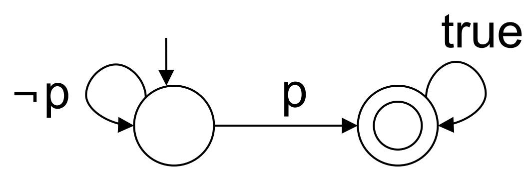
- Büchi automata: Finite state automata that accept ***infinite*** strings
  - The better known variant of finite state automata accept finite strings (used in lexical analysis for example)
- A Büchi automaton ***accepts*** a string when the corresponding run visits an accepting state ***infinitely often***
  - Note that an infinite run never ends, so we cannot say that an accepting run ends at an accepting state
- LTL properties can be translated to Büchi automata
  - The automaton accepts a path if and only if the path satisfies the corresponding LTL property

LTL Properties ≡ Büchi automata

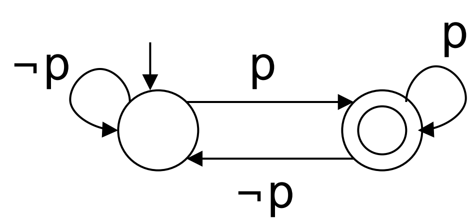
G p



F p



G (F p)



The size of the property automaton can be exponential in the size of the LTL formula (recall the complexity of LTL model checking)

Büchi Automata: Language Emptiness Check

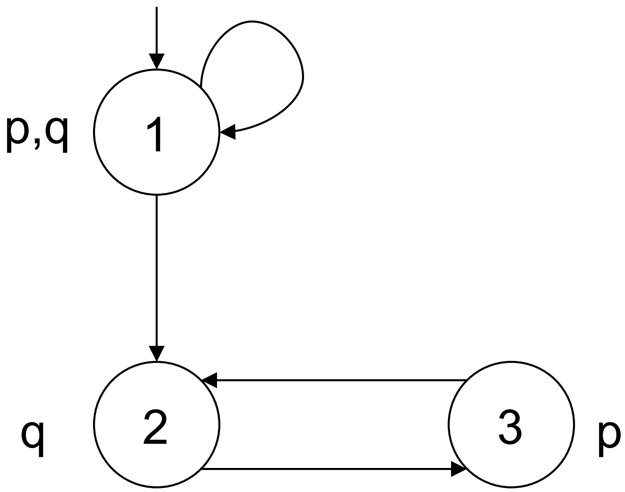
- Given a Buchi automaton, one interesting question is:
  - Is the language accepted by the automaton empty?
    - i.e., does it accept any string?
- A Büchi automaton ***accepts*** a string when the corresponding run visits an accepting state infinitely often
- To check ***emptiness***:
  - Look for a cycle which contains an accepting state and is reachable from the initial state
    - Find a strongly connected component that contains an accepting state, and is reachable from the initial state
  - If no such cycle can be found the language accepted by the automaton is empty

LTL Model Checking

- Generate the property automaton from the negated LTL property
- Generate the product of the property automaton and the transition system
- Show that there is no accepting cycle in the product automaton (check language emptiness)
  - i.e., show that the intersection of the paths generated by the transition system and the paths accepted by the (negated) property automaton is empty
- If there is a cycle, it corresponds to a counterexample behavior that demonstrates the bug

LTL Model Checking Example

Example transition system



Each state is labeled with the propositions that hold in that state

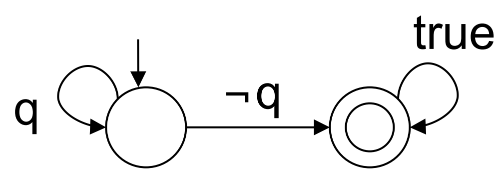
Property to be verified

$G\ q$

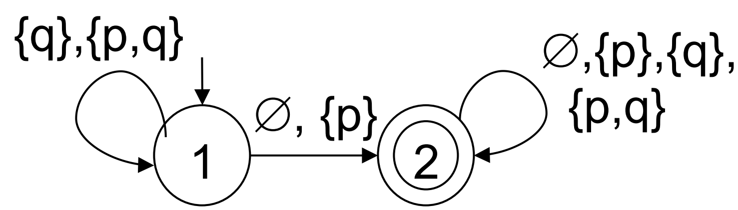
Negation of the property

$\neg\ G\ q \equiv F\ \neg q$

Property automaton for the negated property

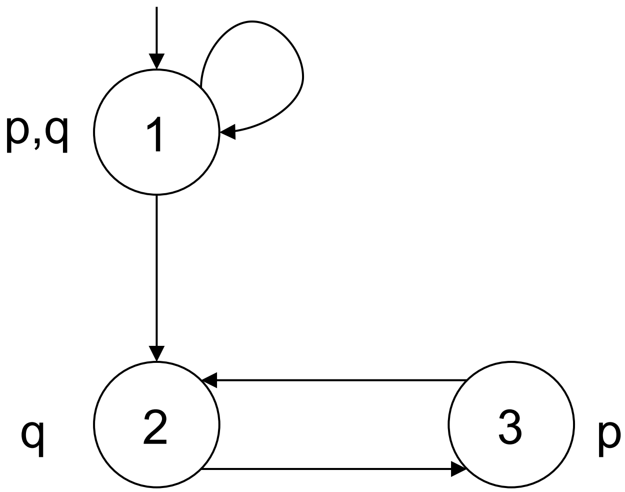


Equivalently



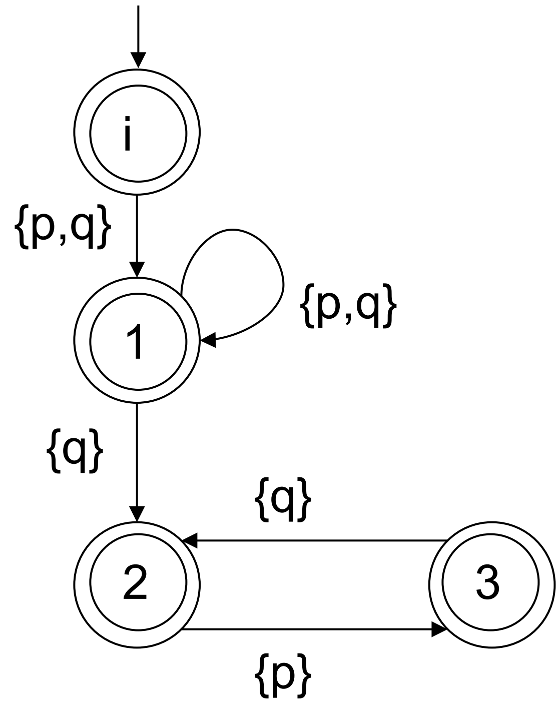
Transition System to Buchi Automaton Translation

Example transition system

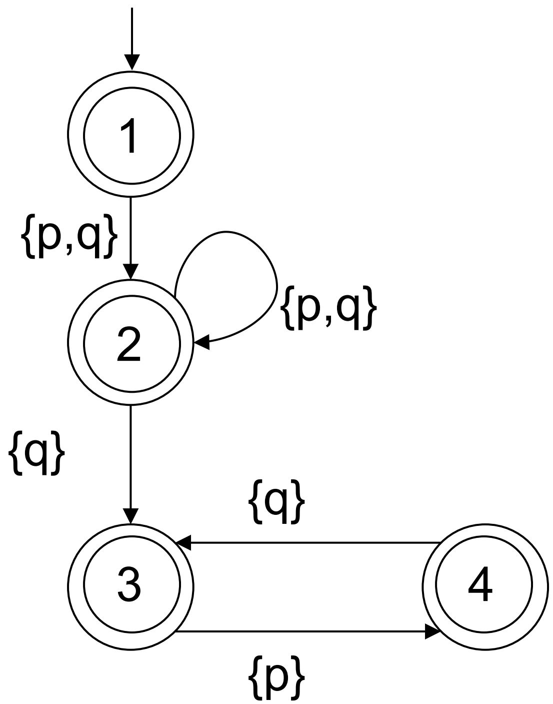


Each state is labeled with the propositions that hold in that state

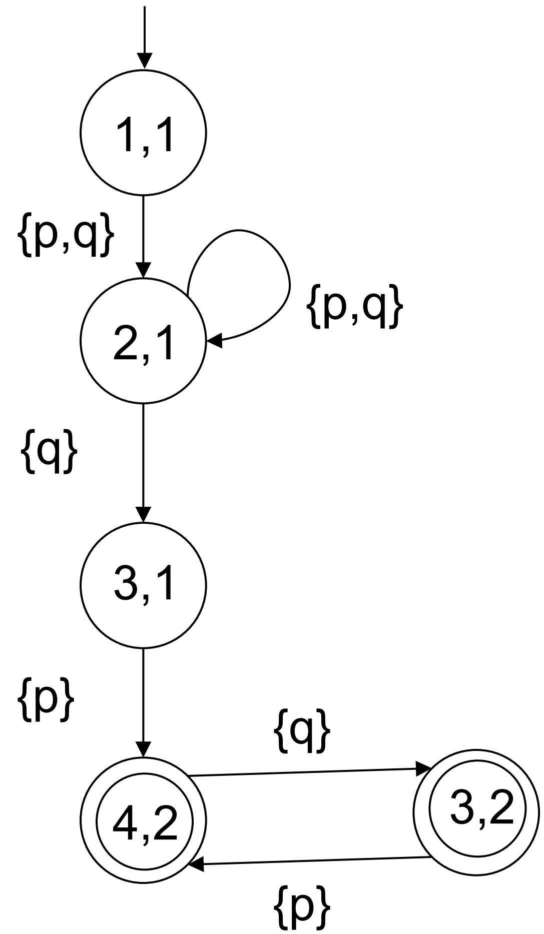
Corresponding Buchi automaton



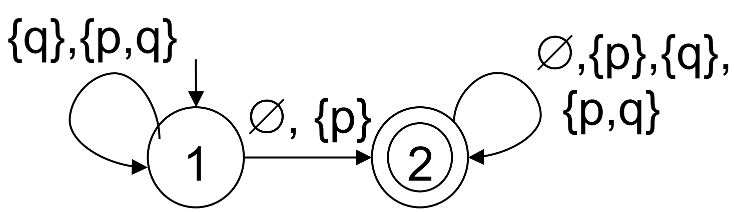
Buchi automaton for the transition system (every state is accepting)



Product automaton



Property Automaton



Accepting cycle:
(1,1), (2,1), (3,1), $((4,2), (3,2))^{\omega}$
Corresponds to a counter-example path for the property G q

SPIN [Holzmann 91, TSE 97]

- Explicit state model checker
- Finite state
- Temporal logic: LTL
- Input language: PROMELA
  - Asynchronous processes
  - Shared variables
  - Message passing through (bounded) communication channels
  - Variables: boolean, char, integer (bounded), arrays (fixed size)
  - Structured data types

SPIN

Verification in SPIN

- Uses the LTL model checking approach
- Constructs the product automaton on-the-fly
  - It is possible to find an accepting cycle (i.e. a counter-example) without constructing the whole state space
- Uses a nested depth-first search algorithm to look for an accepting cycle
- Uses various heuristics to improve the efficiency of the nested depth first search:
  - partial order reduction
  - state compression

Example Mutual Exclusion Protocol

Two concurrently executing processes are trying to enter a critical section without violating mutual exclusion

```
Process 1:
while (true) {
   out:  a := true; turn := true;
   wait: await (b = false or turn = false);
   cs:   a := false;
}
||
Process 2:
while (true) {
   out:  b := true; turn := false;
   wait: await (a = false or turn);
   cs:   b := false;
}
```

Example Mutual Exclusion Protocol in Promela

```
#define cs1 process1@cs
#define cs2 process2@cs
#define wait1 process1@wait
#define wait2 process2@wait
#define true        1
#define false       0
bool a;
bool b;
bool turn;
proctype process1()
{
out:    a = true; turn = true;
wait:   (b == false || turn == false);
cs:     a = false; goto out;
}
proctype process2()
{
out:    b = true; turn = false;
wait:   (a == false || turn == true);
cs:     b = false; goto out;
}
init {
  run process1(); run process2()
}
```

Property automaton generation

```
% spin -f "! [] (! (cs1 && cs2))"

never {    /* ! [] (! (cs1 && cs2)) */
T0_init:
        if
        :: ((cs1) && (cs2)) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}

% spin -f "!([](wait1 -> <>(cs1)))"

never {    /* !([](wait1 -> <>(cs1))) */
T0_init:
        if
        :: ( !((cs1)) && (wait1) ) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (! ((cs1))) -> goto accept_S4
        fi;
}
```

- Input formula
  "[]" means G
  "<>" means F
- "spin –f" option generates a Buchi automaton for the input LTL formula

Concatanate the generated never claims to the end of the specification file

SPIN

- "spin –a mutex.spin" generates a C program "pan.c" from the specification file
  - This C program implements the on-the-fly nested-depth first search algorithm
  - You compile "pan.c" and run it to the model checking

- Spin generates a counter-example trace if it finds out that a property is violated

```
%mutex -a
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.6 -- 27 October 2005)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 28 byte, depth reached 33, errors: 0
      22 states, stored
      15 states, matched
      37 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)

unreached in proctype process1
        line 18, state 6, "-end-"
        (1 of 6 states)
unreached in proctype process2
        line 27, state 6, "-end-"
        (1 of 6 states)
unreached in proctype :init:
        (0 of 3 states)
```

Automata Theoretic LTL Model Checking

Input: A transition system T and an LTL property f

- Translate the transition system T to a Buchi automaton A_T
- Negate the LTL property and translate the negated property $\neg f$ to a Buchi automaton $A_{\neg f}$
- Check if the intersection of the languages accepted by A_T and $A_{\neg f}$ is empty
  - Is $L(A_T) \cap L(A_{\neg f}) = \varnothing$?
  - If $L(A_T) \cap L(A_{\neg f}) \neq \varnothing$, then the transition system T violates the property f

Automata Theoretic LTL Model Checking

- Note that
  - $L(A_T) \cap L(A_{\neg f}) = \emptyset$ if and only if $L(A_T) \subseteq L(A_f)$
- By negating the property f we are converting language subsumption check to language intersection followed by language emptiness check
- Given the Buchi automata A_T and $A_{\neg f}$ we will construct a product automaton $A_T \times A_{\neg f}$ such that
  - $L(A_T \times A_{\neg f}) = L(A_T) \cap L(A_{\neg f})$
- So all we have to do is to check if the language accepted by the Buchi automaton $A_T \times A_{\neg f}$ is empty

Buchi Automata

A Buchi automaton is a tuple $A = (\Sigma, Q, \Delta, Q_0, F)$ where

Σ is a finite alphabet

Q is a finite set of states

$\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation

$Q_0 \subseteq Q$ is the set of initial states

$F \subseteq Q$ is the set of accepting states

- A Buchi automaton A recognizes a language which consists of infinite words over the alphabet Σ

  $L(A) \subseteq \Sigma^\omega$

Σ^ω denotes the set of infinite words over the alphabet Σ

Buchi Automaton

- Given an infinite word $w \in \Sigma^\omega$ where $w = a_0, a_1, a_2, \ldots$

  a run r of the automaton A over w is an infinite sequence of automaton states $r = q_0, q_1, q_2, \ldots$ where $q_0 \in Q_0$ and for all $i \geq 0$, $(q_i, a_i, q_{i+1}) \in \Delta$

- Given a run r, let $inf(r) \subseteq Q$ be the set of automata states that appear in r infinitely many times

- A run r is an accepting run if and only if $inf(r) \cap F \neq \emptyset$

  i.e., a run is an accepting run if some accepting states appear in r infinitely many times

Transition System to Buchi Automaton Translation

Given a transition system $T = (S, I, R)$
a set of atomic propositions AP and
a labeling function $L : S \times AP \rightarrow \{true, false\}$

the corresponding Buchi automaton $A_T = (\Sigma_T, Q_T, \Delta_T, Q_{0T}, F_T)$

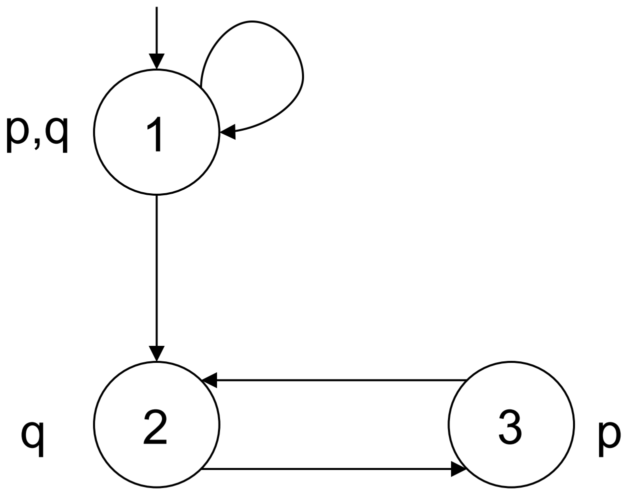
$\Sigma_T = 2^{AP}$	an alphabet symbol corresponds to a set of atomic propositions
$Q_T = S \cup \{i\}$	i is a new state which is not in S
$Q_{oT} = \{i\}$	i is the only initial state
$F_T = S \cup \{i\}$	all states of A_T are accepting states

Δ_T is defined as follows:

$(s,a,s') \in \Delta$ iff either $(s,s') \in R$ and $p \in a$ iff $L(s',p) = true$
or $s=i$ and $s' \in I$ and $p \in a$ iff $L(s',p) = true$

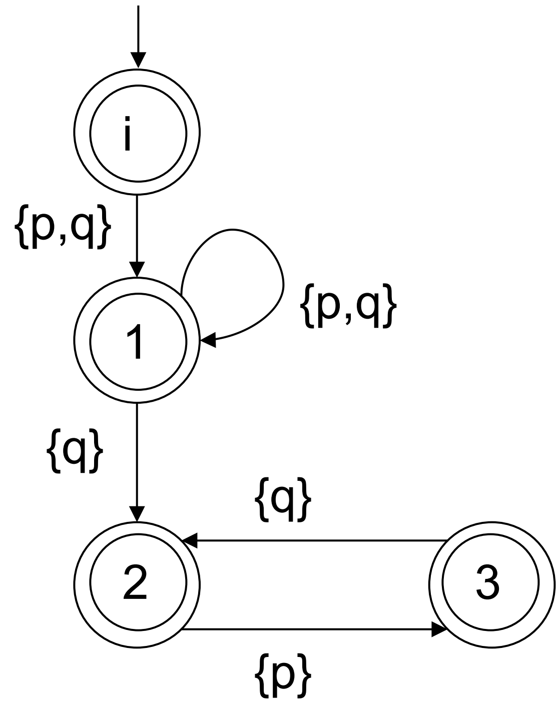
Transition System to Buchi Automaton Translation

Example transition system



Each state is labeled with the propositions that hold in that state

Corresponding Buchi automaton



Generalized Buchi Automaton

A generalized Buchi automaton is a tuple $A = (\Sigma, Q, \Delta, Q_0, F)$ where

Σ is a finite alphabet

Q is a finite set of states

$\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation

$Q_0 \subseteq Q$ is the set of initial states

$F \subseteq 2^Q$ is sets of accepting states

i.e., $F = \{F_1, F_2, \ldots, F_k\}$ where $F_i \subseteq Q$ for $1 \leq i \leq k$

This is different than the standard definition

- Given a generalized Buchi automaton A, a run r is an accepting run if and only if
  - for all $1 \leq i \leq k$, $inf(r) \cap F_i \neq \emptyset$

Buchi Automata Product

Given $A_1 = (\Sigma, Q_1, \Delta_1, Q_{01}, F_1)$ and $A_2 = (\Sigma, Q_2, \Delta_2, Q_{02}, F_2)$ the product automaton $A_1 \times A_2 = (\Sigma, Q, \Delta, Q_0, F)$ is defined as:

$Q = Q_1 \times Q_2$

$Q_0 = Q_{01} \times Q_{02}$

$F = \{F_1 \times Q_2, Q_1 \times F_2\}$ (a generalized Buchi automaton)

Δ is defined as follows:

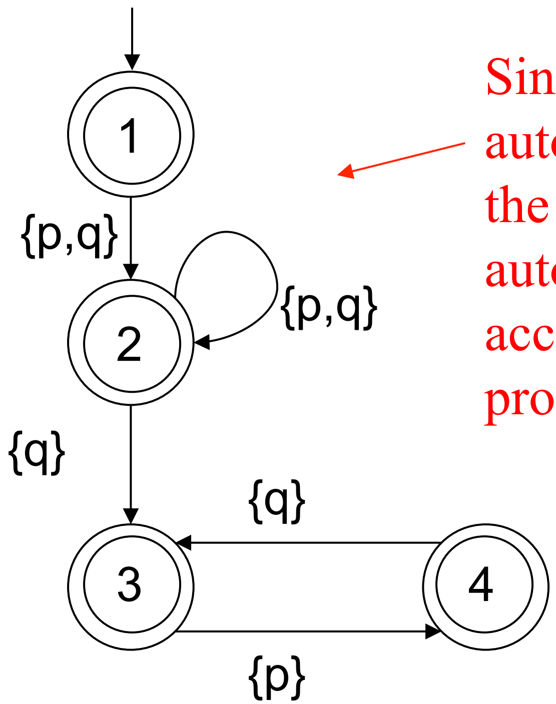
$((q_1,q_2),a,(q_1',q_2')) \in \Delta$ iff $(q_1,a,q_1') \in \Delta_1$ and $(q_2,a,q_2') \in \Delta_2$

Based on the above construction, we get

$L(A_1 \times A_2) = L(A_1) \cap L(A_2)$

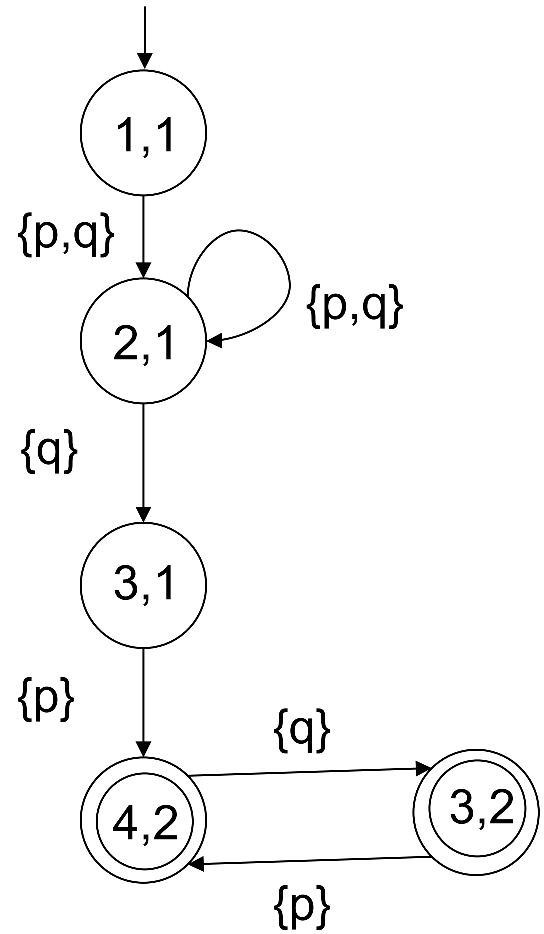
Example from the Last Lecture is a Special Case

Buchi automaton 1

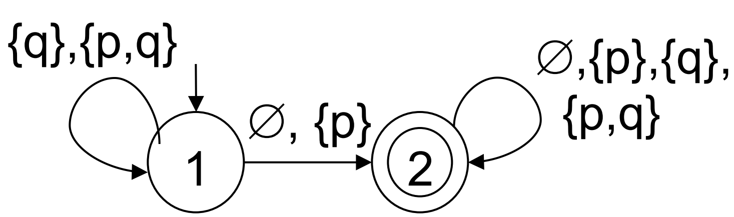


Since all the states in the automaton 1 is accepting, only the accepting states of automaton 2 decide the accepting states of the product automaton

Product automaton

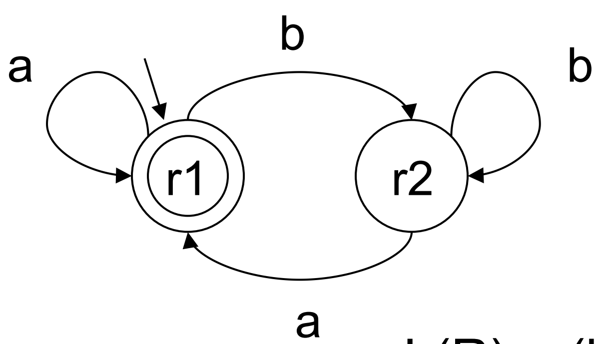


Buchi automaton 2



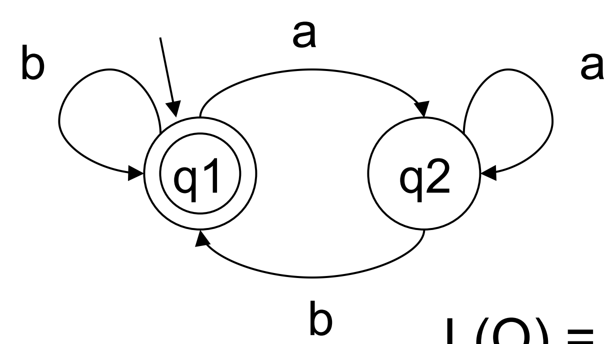
Buchi Automata Product Example

Automaton R



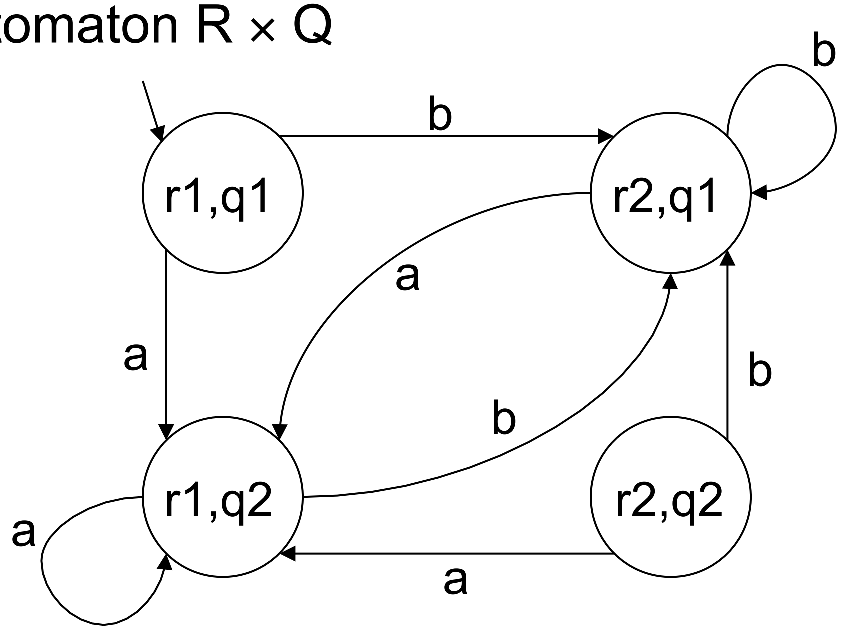
$L(R) = (b^*a)^\omega$

Automaton Q



$L(Q) = (a^*b)^\omega$

Automaton R × Q



$L(R \times Q) = L(R) \cap L(Q)$

F = {
{(r1,q1), (r1,q2)},
{(r1,q1), (r2,q1)}
}

Generalized to Standard Buchi Automata Conversion

Given a generalized Buchi automaton $A = (\Sigma, Q, \Delta, Q_0, F)$
where $F = \{F_1, F_2, \ldots, F_k\}$
it is equivalent to standard Buchi automaton
$A' = (\Sigma, Q', \Delta', Q_0', F')$ where
$Q' = Q \times \{1, 2, \ldots, k\}$
$Q_0' = Q_0 \times \{1\}$
$F' = F_1 \times \{1\}$

Keep a counter. When the counter is i look only for the accepting states in F_i. When you see a state from F_i, increment the counter (mod k). When the counter makes one round, you have seen an accepting state from all F_is.

Δ' is defined as follows:

$((q_1, i), a, (q_2, j)) \in \Delta'$ iff $(q_1,a,q_2) \in \Delta$ and

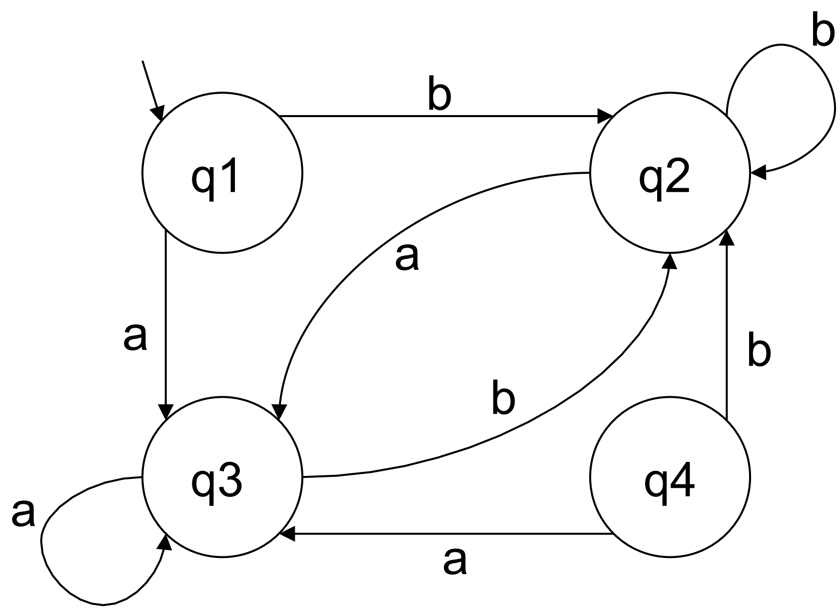
$j=i$	if $q_1 \notin F_i$
$j=(i \bmod k) + 1$	if $q_1 \in F_i$

Based on the above construction we have $L(A') = L(A)$

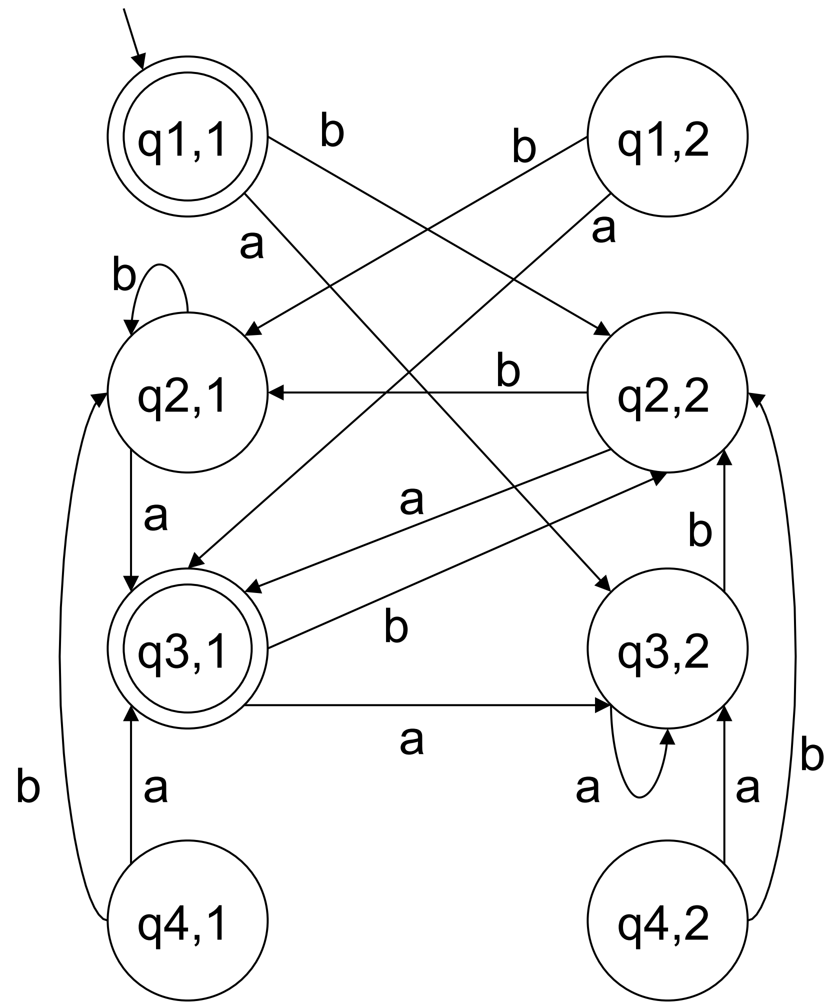
Example (Cont' d)

A standard Buchi automaton S where L(S) = L(G)

A generalized Buchi automaton G



F = { {q1, q3}, {q1, q2} }



F = { (q1,1), (q3,1)}