Computational Linguistics 2014-2015

- **Walter Daelemans** (walter.daelemans@uantwerpen.be)
- **Guy De Pauw** (guy.depauw@uantwerpen.be)
- **Mike Kestemont** (mike.kestemont@uantwerpen.be)

http://www.clips.uantwerpen.be/cl1415

Practical

Location	P0.11 (Scribanihuis)
Reading material	• D. Jurafsky & J.H. Martin (2009) Speech and Language Processing - An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition (2nd ed). Pearson Education, USA. • Natural Language Processing with Python
Software	Python 3.4 and NLTK: Installation Instructions
Evaluation	Take-home assignments and oral examination
Lecturers	Walter Daelemans: walter.daelemans@uantwerpen.be Mike Kestemont: mike.kestemont@uantwerpen.be Guy De Pauw: guy.depauw@uantwerpen.be

Program

Session	Day	Date	Chapter	Topic	Reading Assignment	Slides	Take-home Assignment
1	Monday	29/9/2014	**Python**	Session 1 - Variables	See Github		
2	Thursday	2/10/2014	**Python**	Session 2 - Collections			
3	Monday	6/10/2014	**Python**	Session 3 - Conditions (and an introduction to loops)			
4	Thursday	9/10/2014	**Python**	Session 4 - Loops			
5	Monday	13/10/2014	**Python**	Session 5 - Reading and writing to files			
6	Thursday	16/10/2014	**Python**	Session 6 - Writing your own Functions and importing packages			
7	Monday	20/10/2014	**Python**	Session 7 - Regular Expressions in Python			
8	Thursday	23/10/2014	**Python**	Session 8 - Advanced looping in Python and list comprehensions			
9	Monday	27/10/2014	**Theory**	Introduction to Computational Linguistics	Jurafsky & Martin: Chapter 1		
10	Monday	3/11/2014	**Theory**	Regular Expressions and Finite State Automata & Transducers	Jurafsky & Martin: Chapter 2; Chapter 3		
	Monday	10/11/2014	**Remembrance day: no session**				
11	Monday	17/11/2014	**Theory**	Part-of-Speech Tagging	Jurafsky & Martin: Chapter 5 (not 5.5, 5.8 and 5.9)		
12	Monday	24/11/2014	**Theory**	Syntactic Analysis & Parsing	Jurafsky & Martin: Chapter 12 (not 12.7.2, 12.8); Chapter 13 (not 13.4.1, 13.4.2, 13.5.1)		
13	Monday	1/12/2014	**Theory**	Probabilistic Methods	Jurafsky & Martin: Chapter 4.1, 4.2 and 4.3; Chapter 5.5 and 5.9; Chapter 14.1, 14.3 and 14.4		
14	Monday	8/12/2014	**Theory**	Word Sense Disambiguation	Jurafsky & Martin: Chapter 19.1, 19.2, 19.3, Chapter 20 (20.1->20.5)		
15	Monday	15/12/2014	**Theory**	Sentence semantics and discourse; Information extraction	Jurafsky & Martin: Chapter 21; Chapter 22		

Regular Expressions & Finite State Automata

Universiteit Antwerpen

Introduction

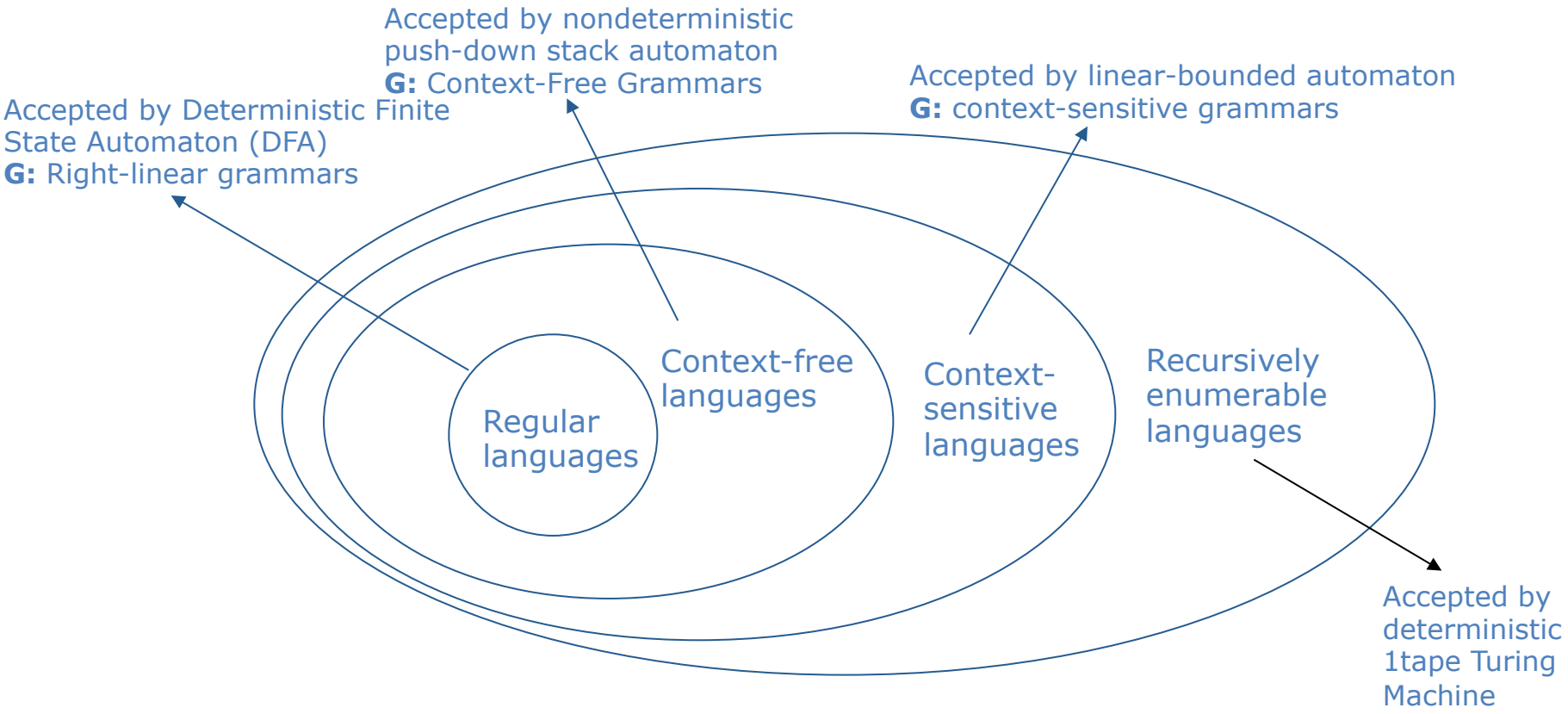
- Turing Machines, Formal Language Theory
  http://www.youtube.com/watch?v=E3keLeMwfHY

- A lot of work in computational linguistics assumes that natural languages can be treated like **formal languages**.
  - A formal language is a set of *strings* – finite sequences of tokens (cf. words, morphemes, phonemes, …). A set of strings is called an *alphabet* or *vocabulary*.

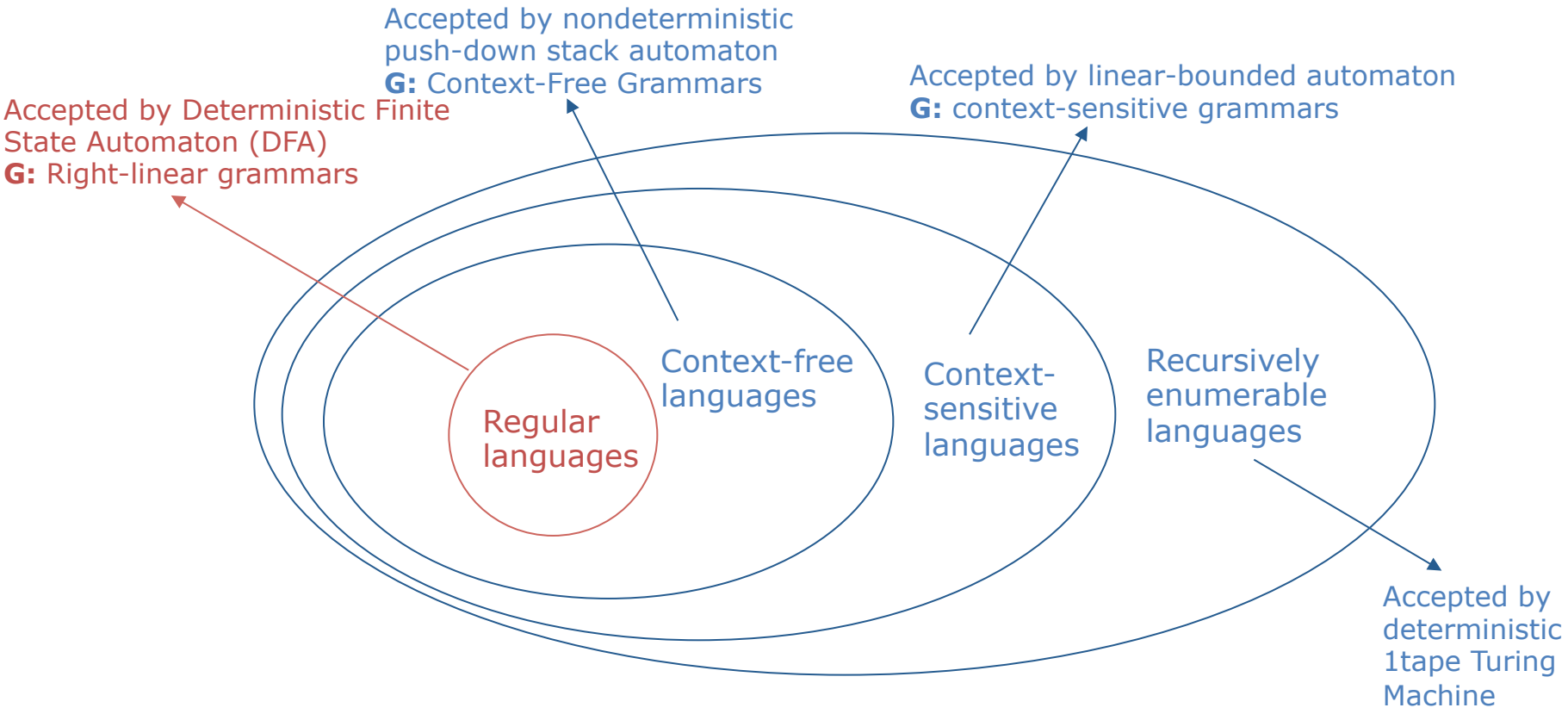
- Some classes of formal languages can be generated by a *grammar*. To determine which strings belong to such a language, we need a control mechanism.

Generative Grammars

The Chomsky Hierarchy



The Chomsky Hierarchy



Formal Languages & Regular Expressions

- Given a finite set A, we define a string over A as the finite sequence of occurrences of elements of A. The set of all strings over A is represented as A*.
- Regular expressions are a concise way to denote patterns in text. They are typical components of word processors, scripting languages such as *AWK, Perl* and *Python* and Unix utilities like *grep*.

Regular Expressions

Regular Expressions

The simplest regular expressions are just some string between two slashes

/cat/

The slashes are not part of the regular expression itself. They indicate that the string has to be interpreted as a regular expression.

Regular Expressions

/woodchucks/	"interesting links to woodchucks and lemurs"
/a/	"Mary Ann stopped by Mona's"
/Claire says,/	"Dagmar, my gift please," Claire says,"
/song/	"all our pretty songs"
/!/	"You've left the burglar behind again!" said Nori

Meta-characters

We can also use special **operators**, the so-called meta-characters. There are a few classes of operators.

The first class allows us to specify **alternatives**:

.	any character (wildcard)
[]	a subset, can also be abbreviated [A-Za-z]
[^]	complement of a subset [^A-Z]
()	groups a number of characters (ab)c
|	specifies alternatives a|b

Examples

/.at/	matches cat,bat,3at,…	"I was at the door"
/[wW]oodchuck/	matches Woodchuck,woodchuck	"Woodchucks woodchuck"
/[abc]/	matches 'a', 'b', *or* 'c'	"In uomini, in soldati"
/[^abc]/	matches anything but 'a', 'b', *or* 'c'	"In uomini, in soldati"

/cat|bat/	matches cat or bat	"a sweet cat doesn't scratch"
/(cat)|(bat)/	matches cat or bat	"a new category"
		"like a bat out of hell"
		"rechargeable batteries"

/[A-Z]/	matches any uppercase letter	"we should call it 'Drenched Blossoms'"
/[a-z]/	matches any lowercase letter	"SP.a & N-VA"
/[0-9]/	matches any digit	"Chapter 1: Down the Rabbit Hole"

Meta characters

A second class of operators refers to boundaries. These operators mark the start or end of a line, word, …

\b	word boundary
^	start of a line
$	end of a line

Examples

/\bcat\b/	matches the word cat	"The cat was there" "A new category"
/^[CB]ats are /	matches "Cats are" and "Bats are"	at the beginning of a line "Cats are mammals" "Bats are mammals" "You know: Cats are…"
/^Cats are nice$/	matches the entire sequence on 1 line	"Cats are nice" "Cats are nice pets"

Meta characters

The last class of operators indicates repetition:

*	indicates 0 or more occurrences of some expression
+	indicates 1 or more occurrences of some expression
?	Indicates one or no occurrence of some expression
{n}	indicates n occurrences of some expression
{n,m}	indicates at least n and up to m occurrences of some expression

Examples

/woodchucks*/ matches {woodchuck,woodchucks,woodchuckss,…}
/woodchucks+/ matches {woodchucks,woodchuckss,…}
/woodchucks?/ matches {woodchuck,woodchucks}

/colou?r/ matches {color,colour}

/woodchucks{3}/ matches {woodchucksss}
/(woodchucks){2}/ matches {woodchuckswoodchucks}
/(woodchucks){1,2}/ matches {woodchucks,woodchuckswoodchucks}

Examples

The language of sheep

baa!
baaa!
baaaa!
baaaaa!
…

Regular expression: /baa+!/ or /baaa*!/

Hierarchy

1. Parenthesis ()
2. Counters * + ? {}
3. Sequences/anchors the ^my end$
4. Disjunction

/woodchucks{3}/ matches {woodchucksss}
/(woodchucks){2}/ matches {woodchuckswoodchucks}

/the*/ matches {*the*,*theeeeee*,…} not *thethe*
/the|any/ matches {*the*,*any*} not *theny*

Examples

/\b[(in)|(de)][(duce)|(duction)]\b/

→ {induce,deduce,induction,deduction}

/[(work)|(part)|(reap)]ed\b/

→ {worked,parted,reaped,reworked,…}

/(de).*(ed)/

→ {departed,deducted,deformed,detect weed,…}

Exercise

- *a** 0 or more a's
- () groups characters
- ¦ or

What language does /((ab)*)|((ba)*)/ denote? Give a few examples

Exercise

- *a** 0 or more a's
- () groups characters
- ¦ or

What language does /((ab)*)|((ba)*)/ denote? Give a few examples

The language containing all and only even-length words consisting of alternating a's and b's {ε,ab,ba,abab,baba,…}

Exercise

Go to http://regexpal.com

- write the following words in the bottom box

cat cats bat bats dog dogs sheep child children

- Write a regular expression that can generate/ recognize these words

- be as concise as possible without overgenerating

Exercise

Go to http://regexpal.com

- write the following words in the bottom box

cat cats bat bats dog dogs sheep child children

- Write a regular expression that can generate/ recognize these words

- be as concise as possible without overgenerating

/\b[[cb]at(s)?|dog(s)?|sheep|child(ren)?]\b/

or /\b[[([cb]at)|(dog)](s)?|sheep|child(ren)?]\b/

correct too: /\bcat|cats|bat|bats|dog|dogs|sheep|child|children\b/

Substitution

Regular expressions are often used to perform substitutions (cf. search and replace (also see transducers (next week))):

s/python/PYTHON/

this substitution will replace all occurrences of the string *perl* with its uppercase equivalent

You can also perform back-reference by using brackets

s/\b([part|duct|form])\b/de\1ed/

- part → departed
- duct → deducted
- form → deformed

ELIZA (Weizenbaum 1964)

- Simulates Rogerian therapist

  http://www.manifestation.com/neurotoys/eliza.php3
- Based on simple pattern recognition

  … father … *tell me more about your family*

  … X Y my Z … *Why do you think X Y your Z?*

- People believed it was real
- Even therapists thought it would be a useful tool
- Popular press thought computers could understand language

ELIZA

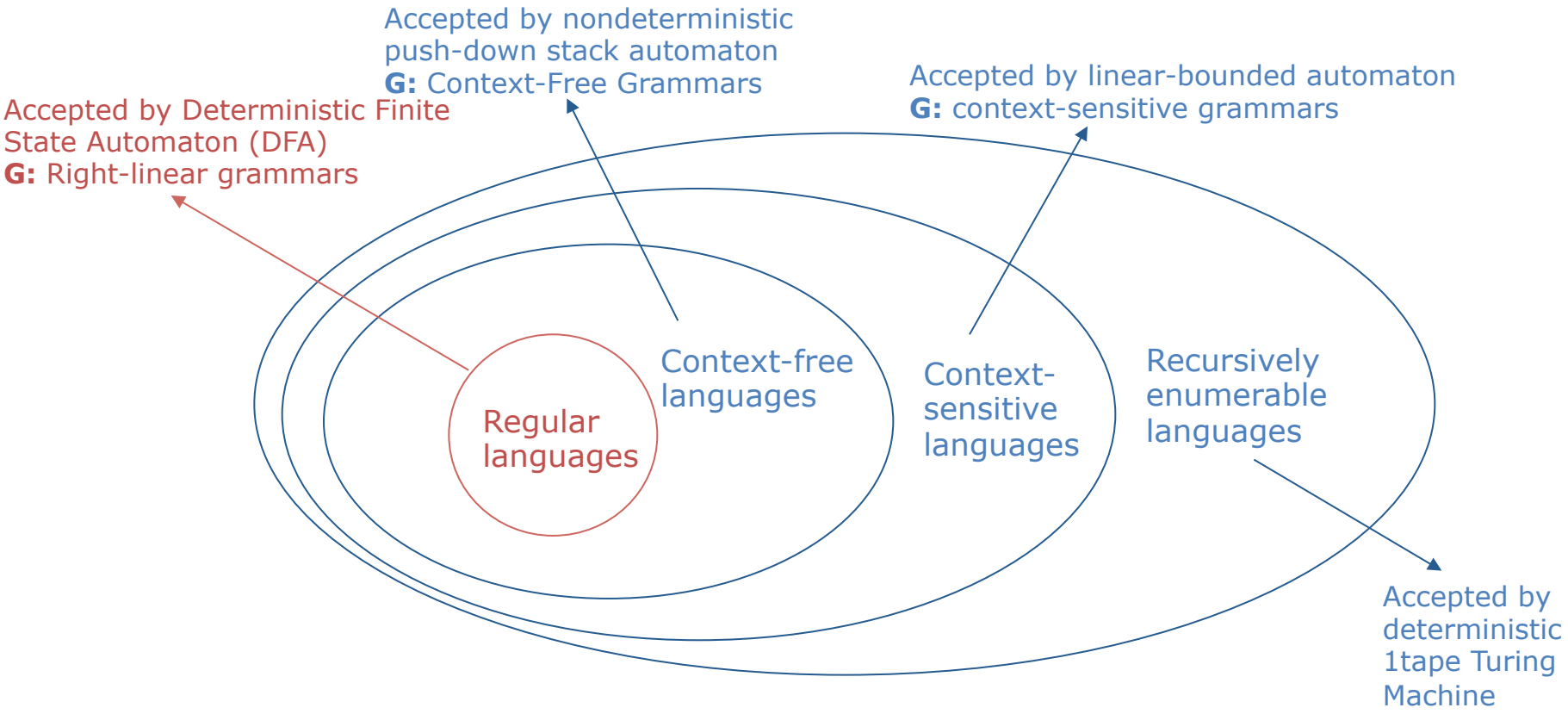
s/.* I AM (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/.* I AM NEVER (depressed|sad) .*/WHY DO YOU THINK YOU ARE NEVER \1/

s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/

better chatbot: www.cleverbot.com

Finite-state automata

The Chomsky Hierarchy



Finite-state automata

A finite state automata is a system $<K,\Sigma,\delta,q_0,F>$ with:

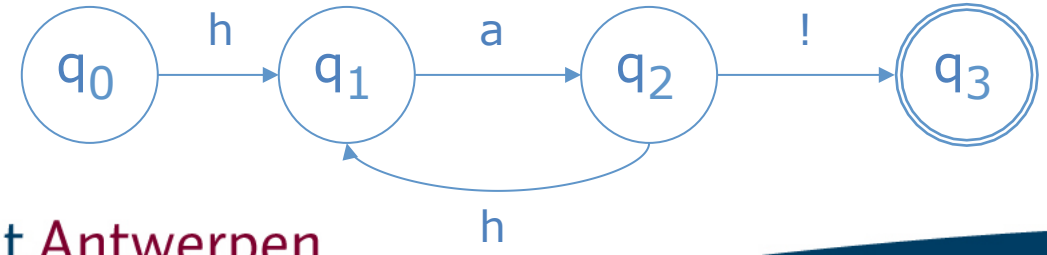
- K: a finite, non-empty set of states
- Σ: a finite, non-empty vocabulary
- q_0: a start state
- $F \subset K$: a set of terminal states (indicated with double line in state diagram)
- δ: a transition function that maps $K \times \Sigma$ in $K \cup \varnothing$

Finite-state automata

A FA can be represented using a state diagram.
For example, automaton M with

$K=\{q_0,q_1,q_2,q_3\}$
$\Sigma=\{h,a,!\}$ /(ha)+!/
and $\delta(q_0,h) = q_1$
$\delta(q_1,a) = q_2$
$\delta(q_2,h) = q_1$
$\delta(q_2,!) = q_3$

The state diagram can be represented as follows:

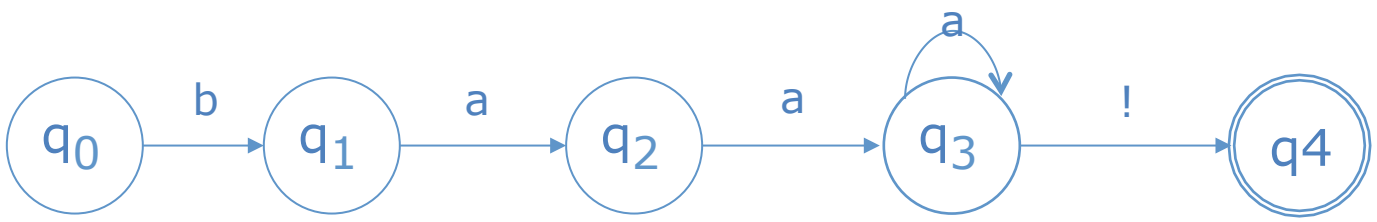


Finite-state automata

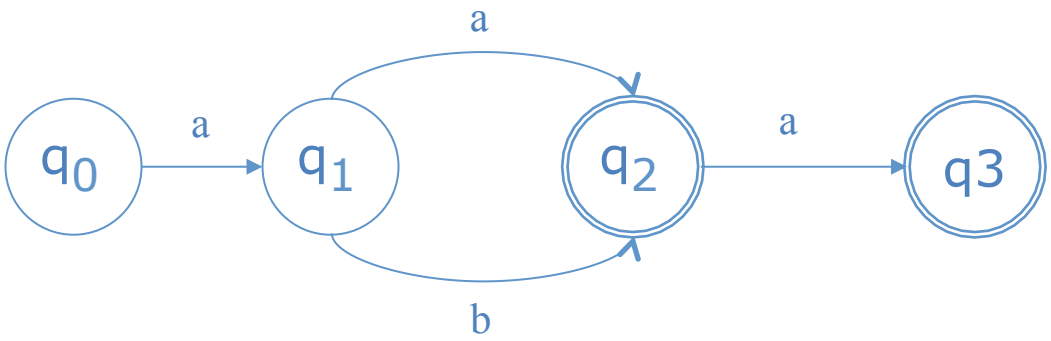
A FA can be represented using a state diagram.
For example, automaton M with

$K=\{q_0,q_1,q_2,q_3,q_4\}$
$\Sigma=\{b,a,!\}$ /baaa*!/
and $\delta(q_0,b) = q_1$
$\delta(q_1,a) = q_2$
$\delta(q_2,a) = q_3$
$\delta(q_3,!) = q_4$ $\delta(q_3,a) = q_3$

The state diagram can be represented as follows:

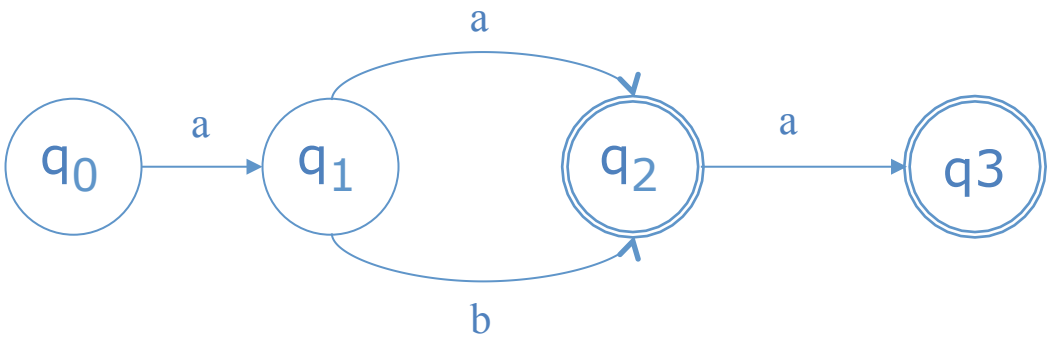


Exercise



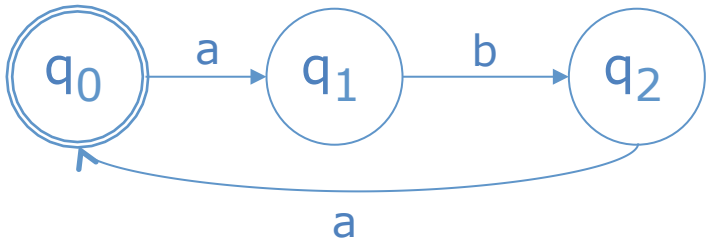
- Which regular expression is equivalent to this FSA?
- What are the terminal states?
- Is ε an accepted word?
- Is the language finite?

Exercise



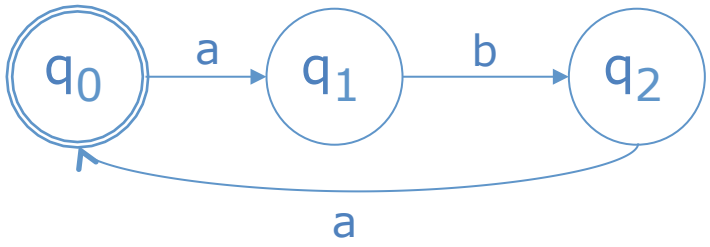
- Which regular expression is equivalent to this FSA? /a(a|b)a?/
- What are the terminal states? q2 and q3
- Is ε an accepted word? no
- Is the language finite? yes

Exercise



- Which regular expression is equivalent to this FSA?
- What are the terminal states?
- Is ε an accepted word?
- Is the language finite?

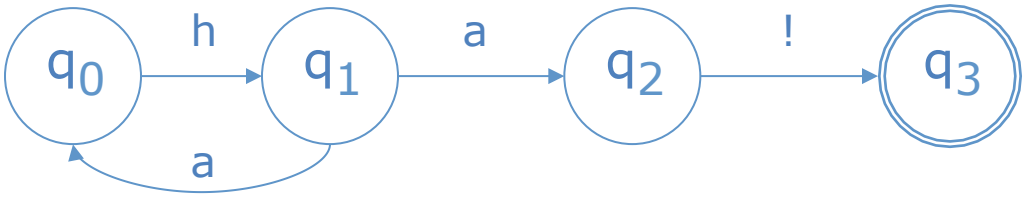
Exercise



- Which regular expression is equivalent to this FSA? /(aba)*/
- What are the terminal states? q0
- Is ε an accepted word? yes
- Is the language finite? no

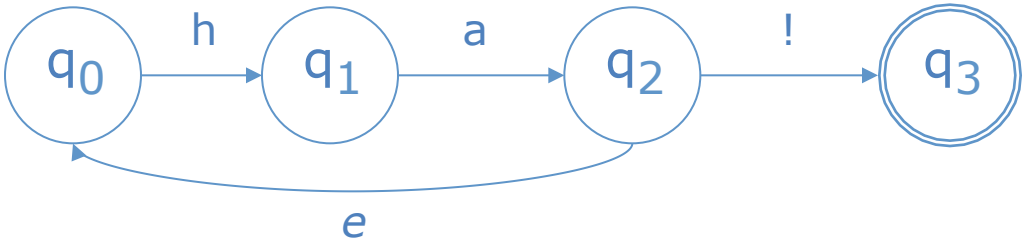
Non-deterministic finite-state automata (NFA)

Non-deterministic automata can have two transitions for the same state/symbol pair



Non-deterministic finite-state automata (NFA)

Another extension are empty transitions



Advantages: more intuitive design
Disadvantage: search problem

Acceptance with a NFA

Since there are more than two possible transitions for $\{q_1,a\}$, we have to use some kind of search strategy.

There are different paths through the diagram. If one of them reaches the terminal state, the string is accepted, i.e. "grammatical".

To guarantee that we can track all paths through the diagram, we use a *search algorithm*, a procedure with a finite number of steps.

More on search algorithms in Session on Parsing

Transforming a NFA to a DFA

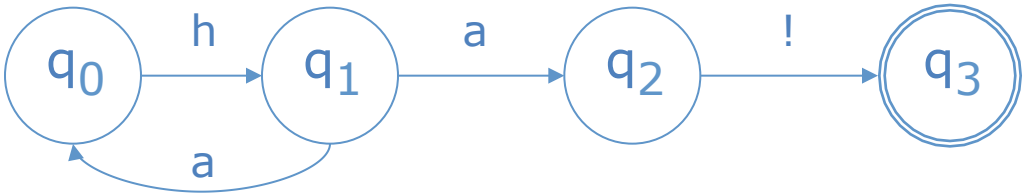
- Any NFA can be converted into a DFA
- General strategy:
  - Create new FA with powerset of states of the NFA
  - Draw transitions between states that are connected in the NFA
  - Remove states that do not have a path leading to them from the start state(s)

Power sets

- P(S): power of set S
  - Set of all subsets of S
  - $P(S) = \{A | A \subseteq S\}$
  - e.g. S = {1,2,3}

    $P(S) = \{\varnothing,\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$
- Unary set-forming operation
- For any finite set S, we have $card(P(S)) = 2^{card(S)}$

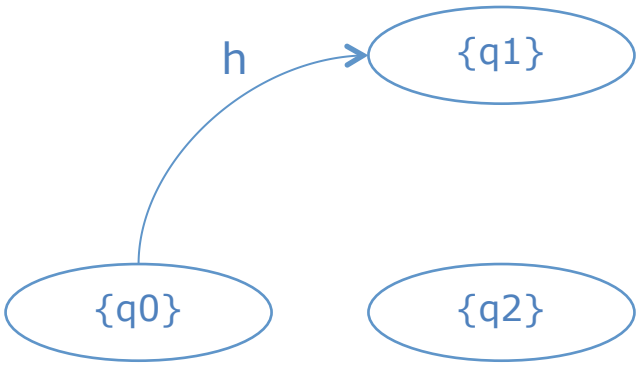
Example



- *M* has 4 states: $\{q_0,q_1,q_2,q_3\}$
- So *M'* will have 2^4 - 1 state(-sets), with some terminal state sets

{q0}	{q0,q2}	{q0,q1,q2}
{q1}	{q0,q3}	{q0,q1,q3}
{q2}	{q1,q2}	{q0,q2,q3}
{q3}	{q1,q3}	{q1,q2,q3}
{q0,q1}	{q2,q3}	{q0,q1,q2,q3}


{q0,q1}
{q0,q1,q2}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
q0
h
q1
a
q2
!
q3
a



{q0,q1}

{q0,q2}

{q0,q3}

{q1,q2}

{q1,q3}

{q2,q3}

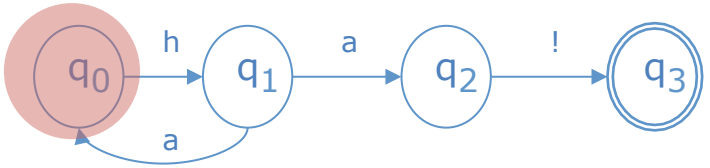
{q0,q1,q2}

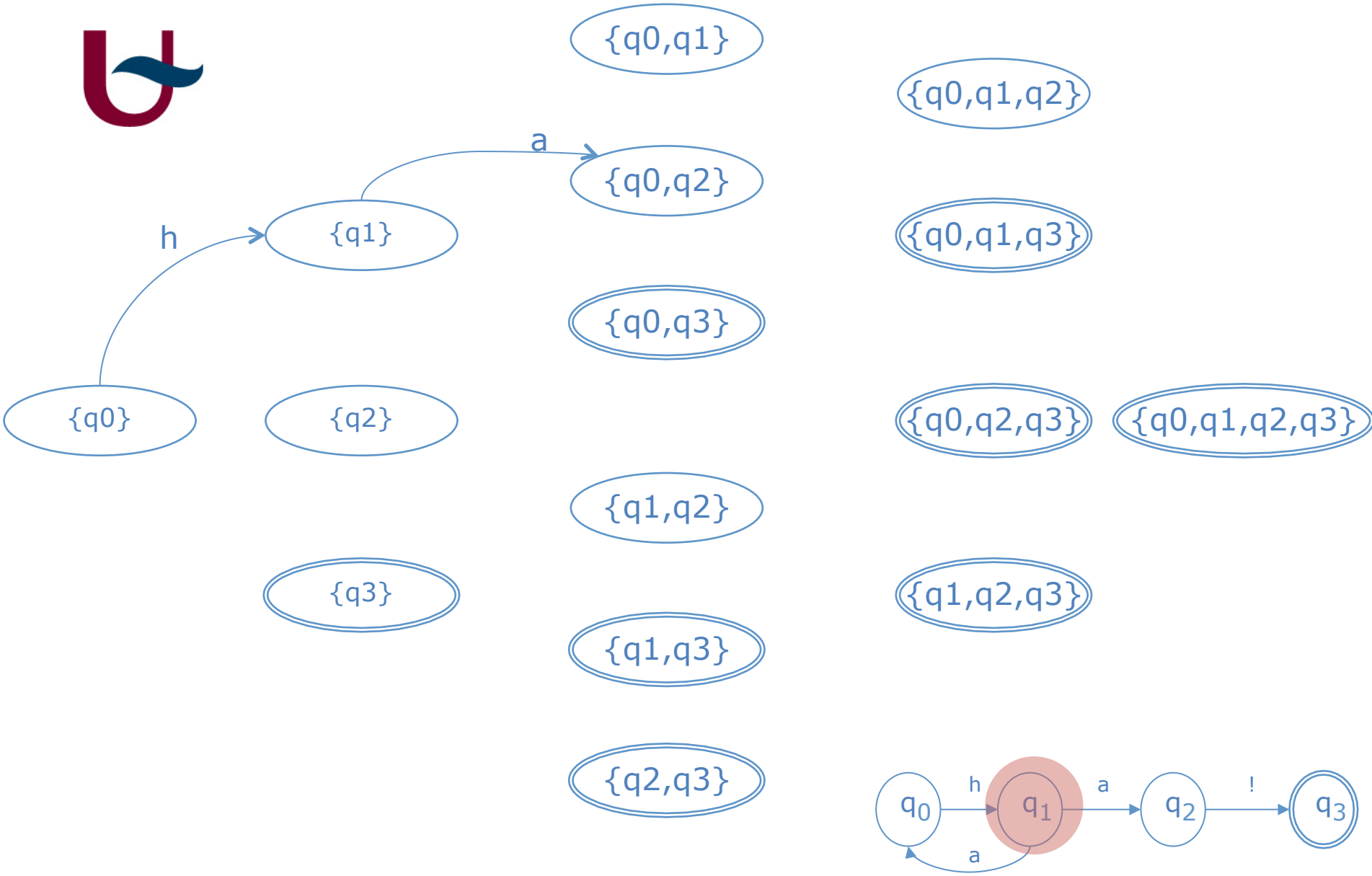
{q0,q1,q3}

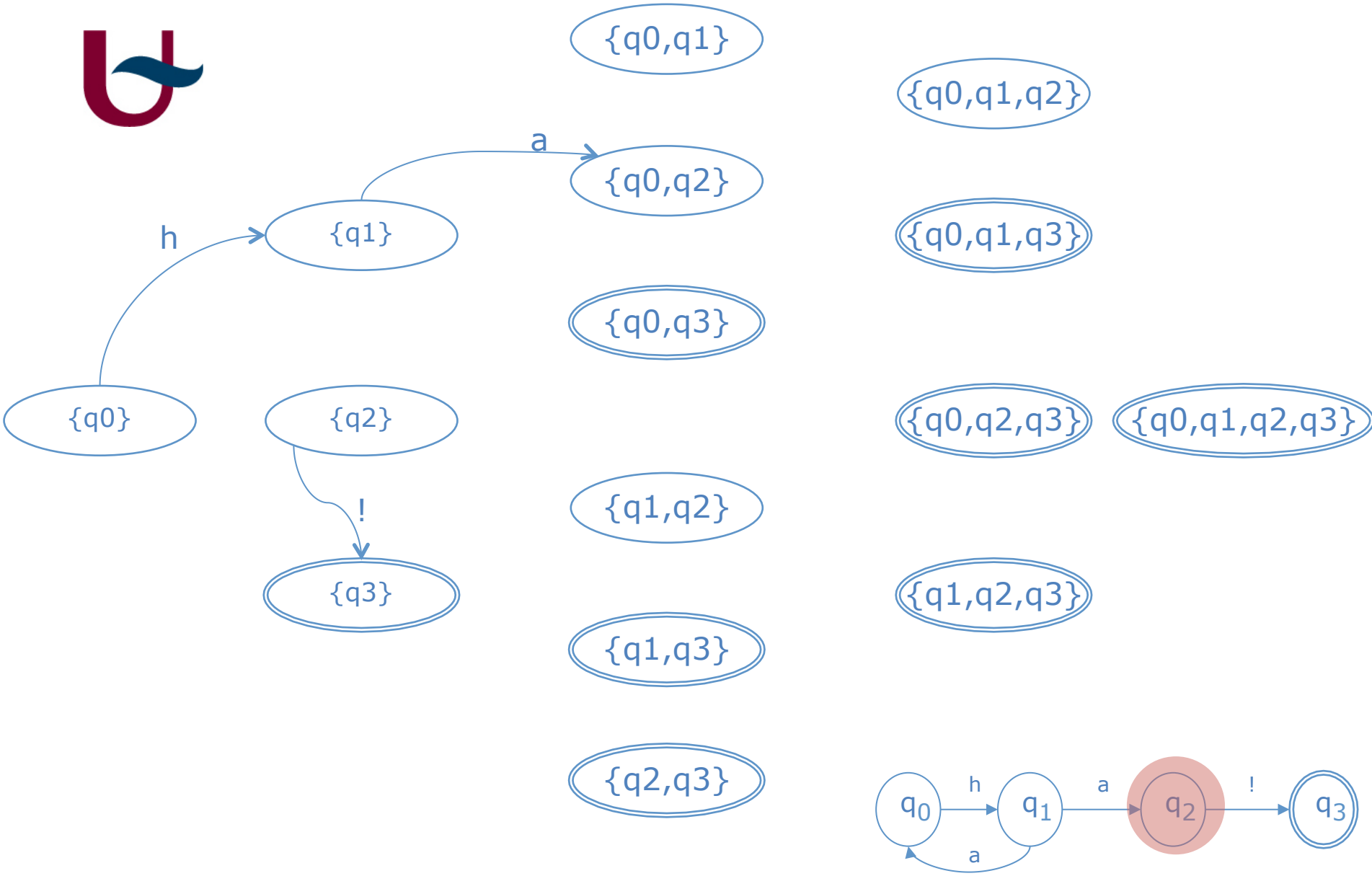
{q0,q2,q3}

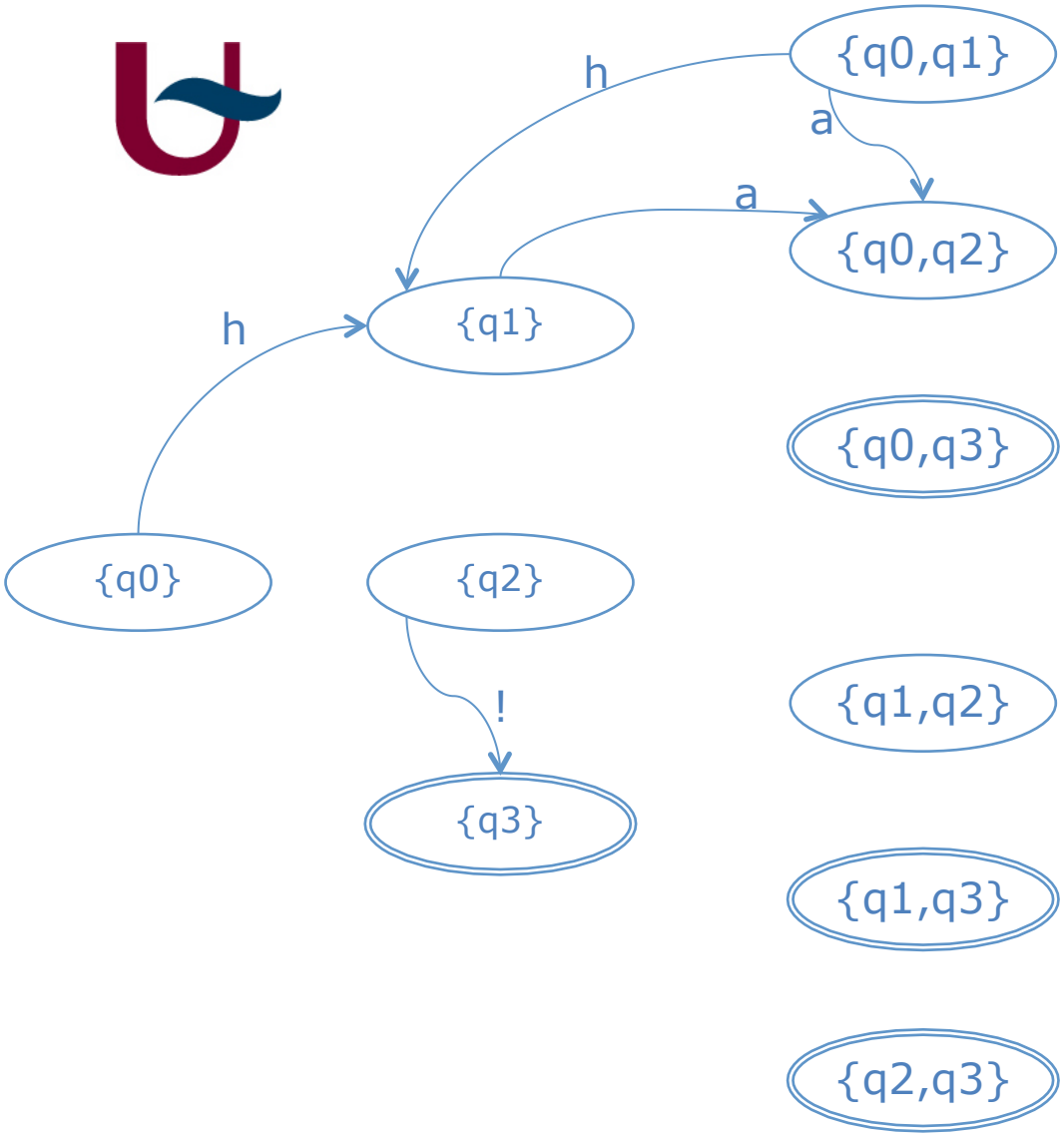
{q0,q1,q2,q3}

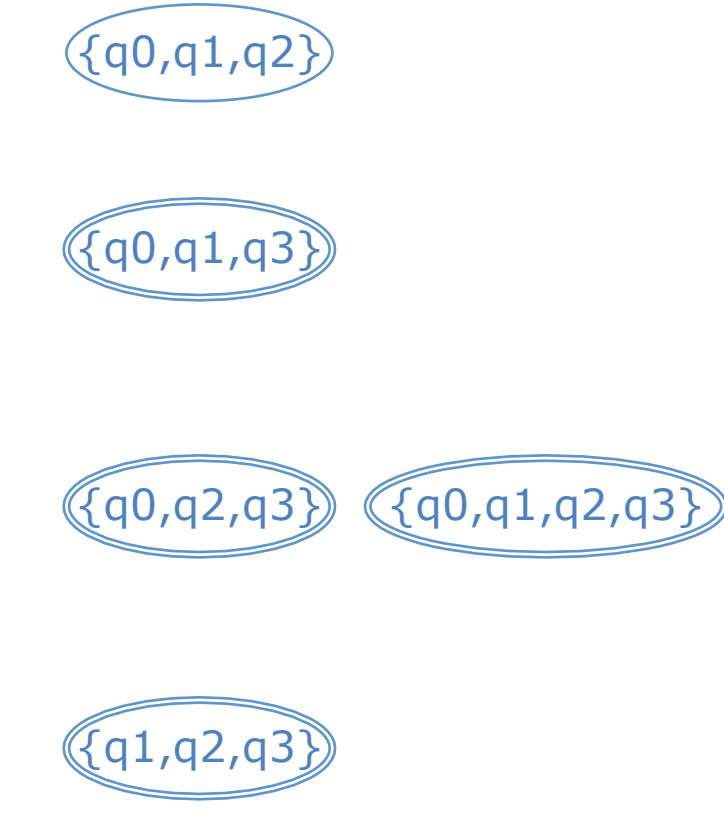
{q1,q2,q3}

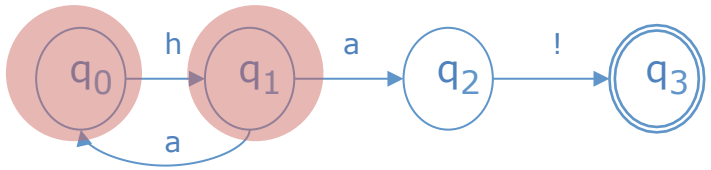


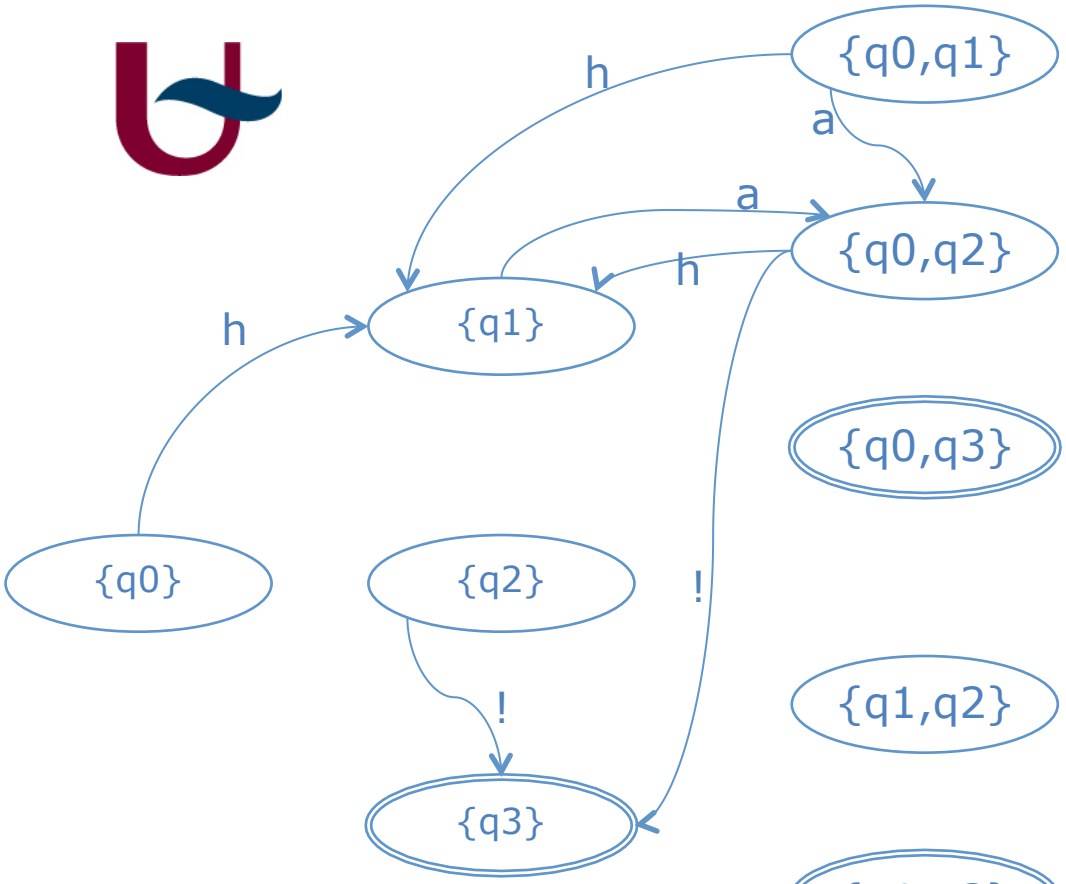

{q0,q1}
{q0,q1,q2}
a
{q0,q2}
h
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
q0
h
q1
a
q2
!
q3
a


{q0,q1}
{q0,q1,q2}
a
{q0,q2}
h
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
!
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
q0
h
q1
a
q2
!
q3
a


{q0,q1}
h
a
a
{q0,q2}
h
{q1}
{q0}
{q2}
!
{q3}
{q0,q3}
{q1,q2}
{q1,q3}
{q2,q3}


{q0,q1,q2}
{q0,q1,q3}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2,q3}


q0
h
q1
a
q2
!
q3
a



{q0,q1,q2}

{q0,q1,q3}

{q0,q3}

{q0,q2,q3}

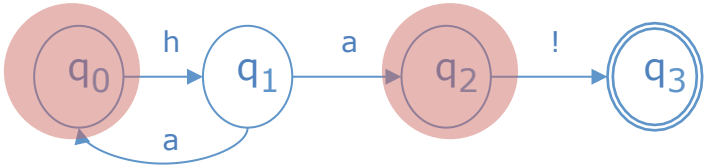
{q0,q1,q2,q3}

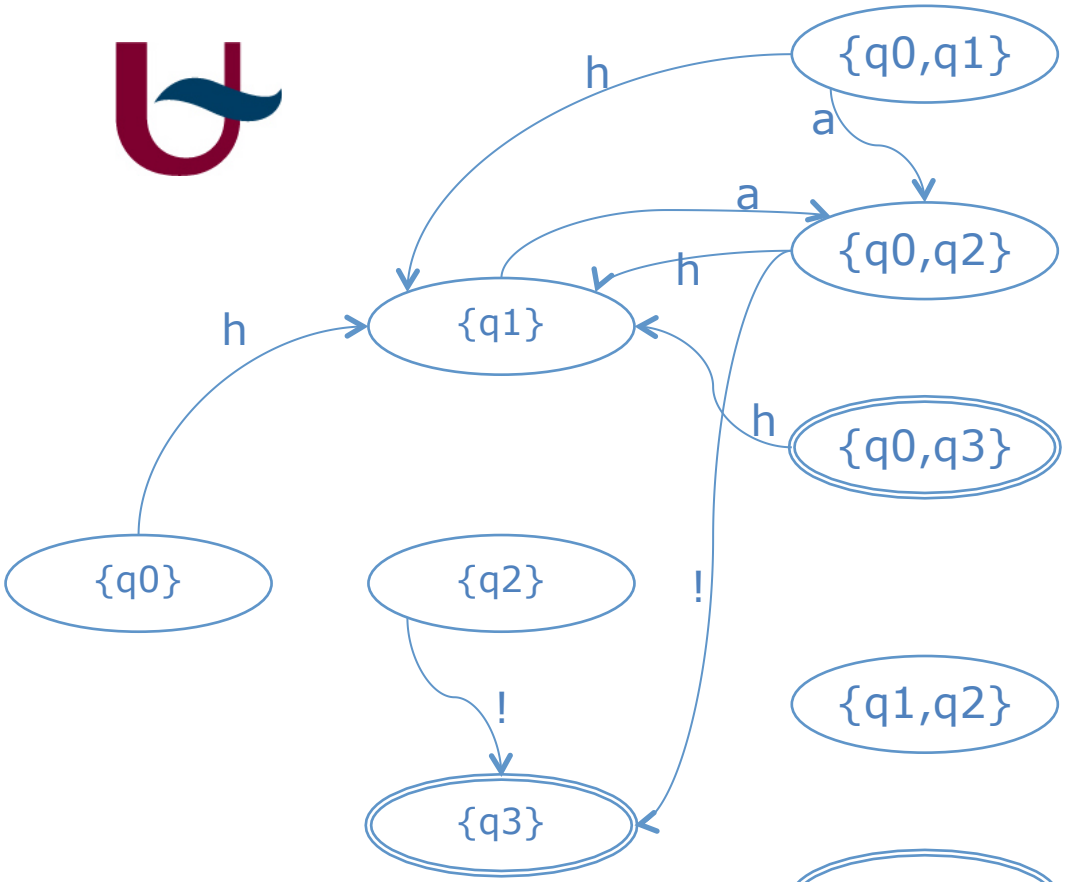
{q1,q2}

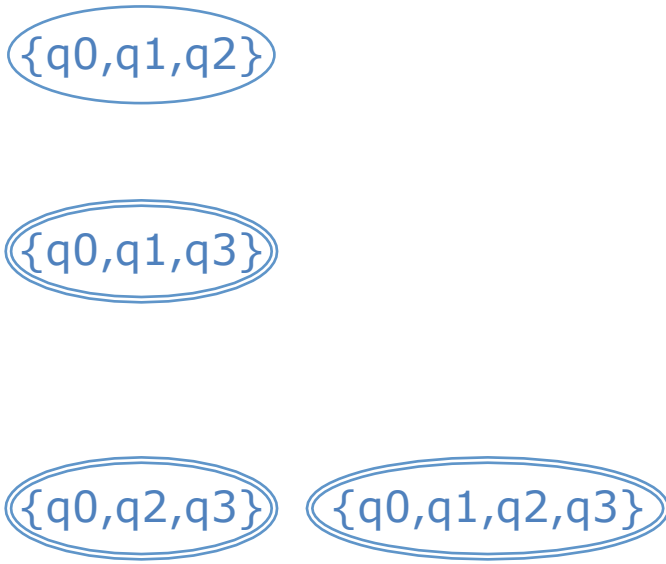
{q1,q2,q3}

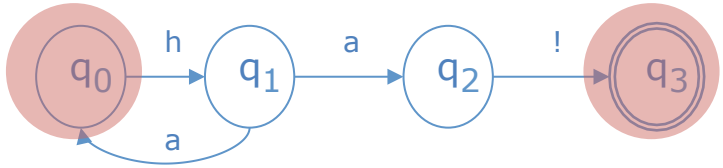
{q1,q3}

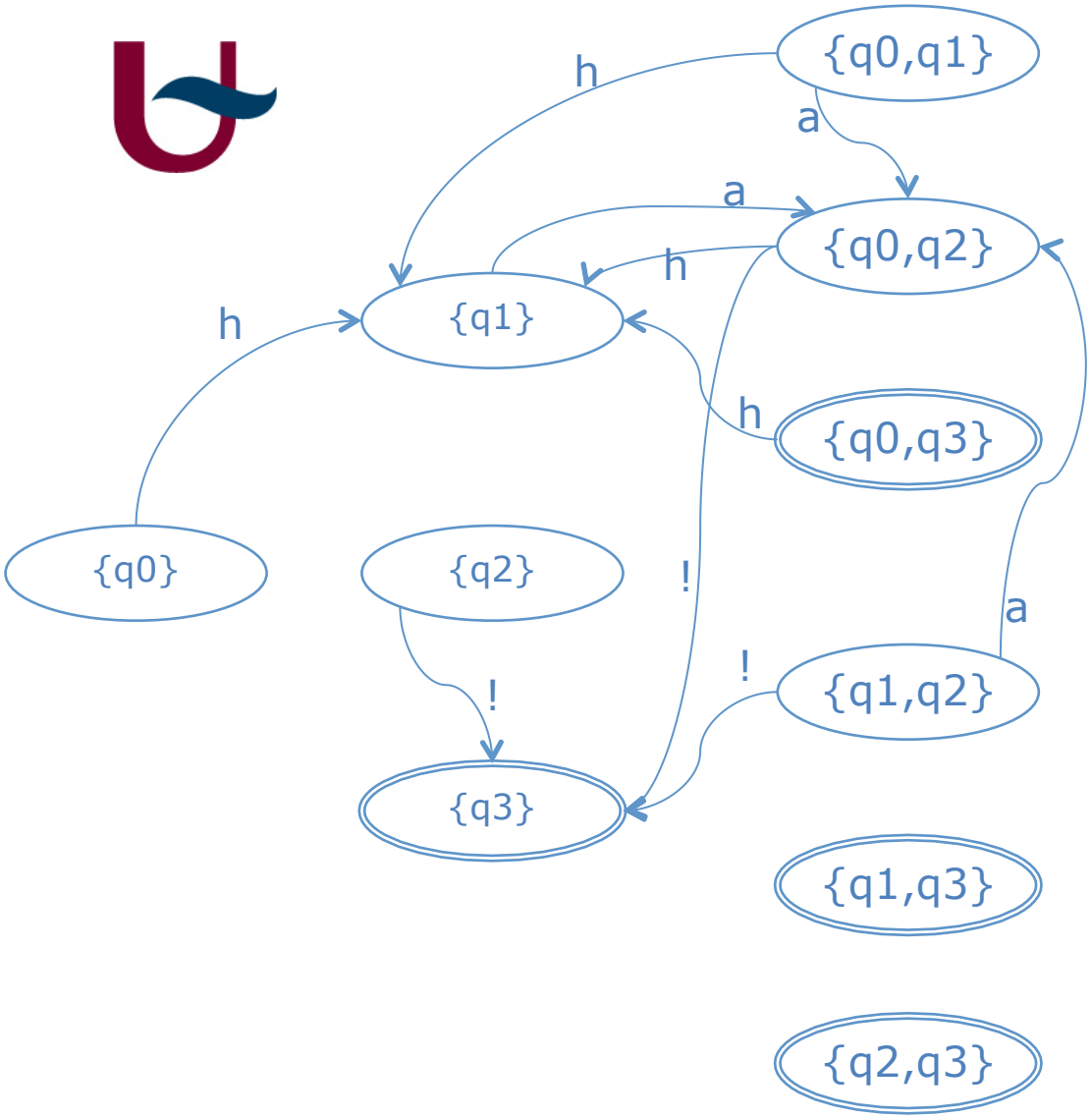
{q2,q3}

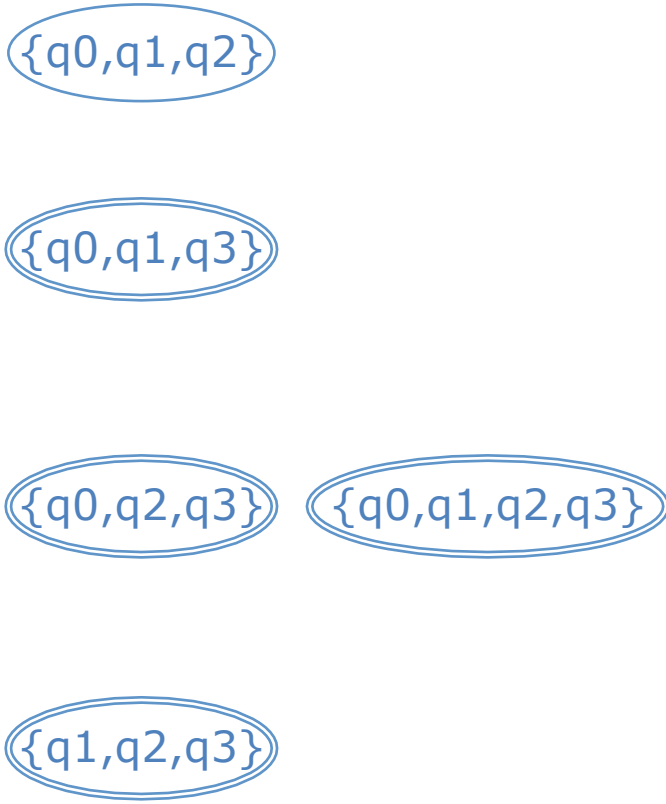


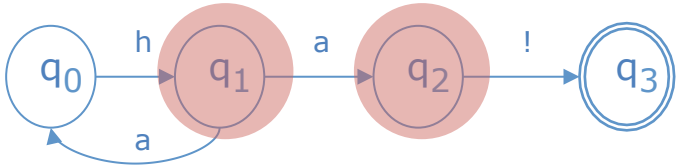

{q0,q1}
h
a
a
{q0,q2}
h
h
{q1}
h
{q0,q3}
{q0}
{q2}
!
!
{q3}
{q1,q2}
{q1,q3}
{q2,q3}

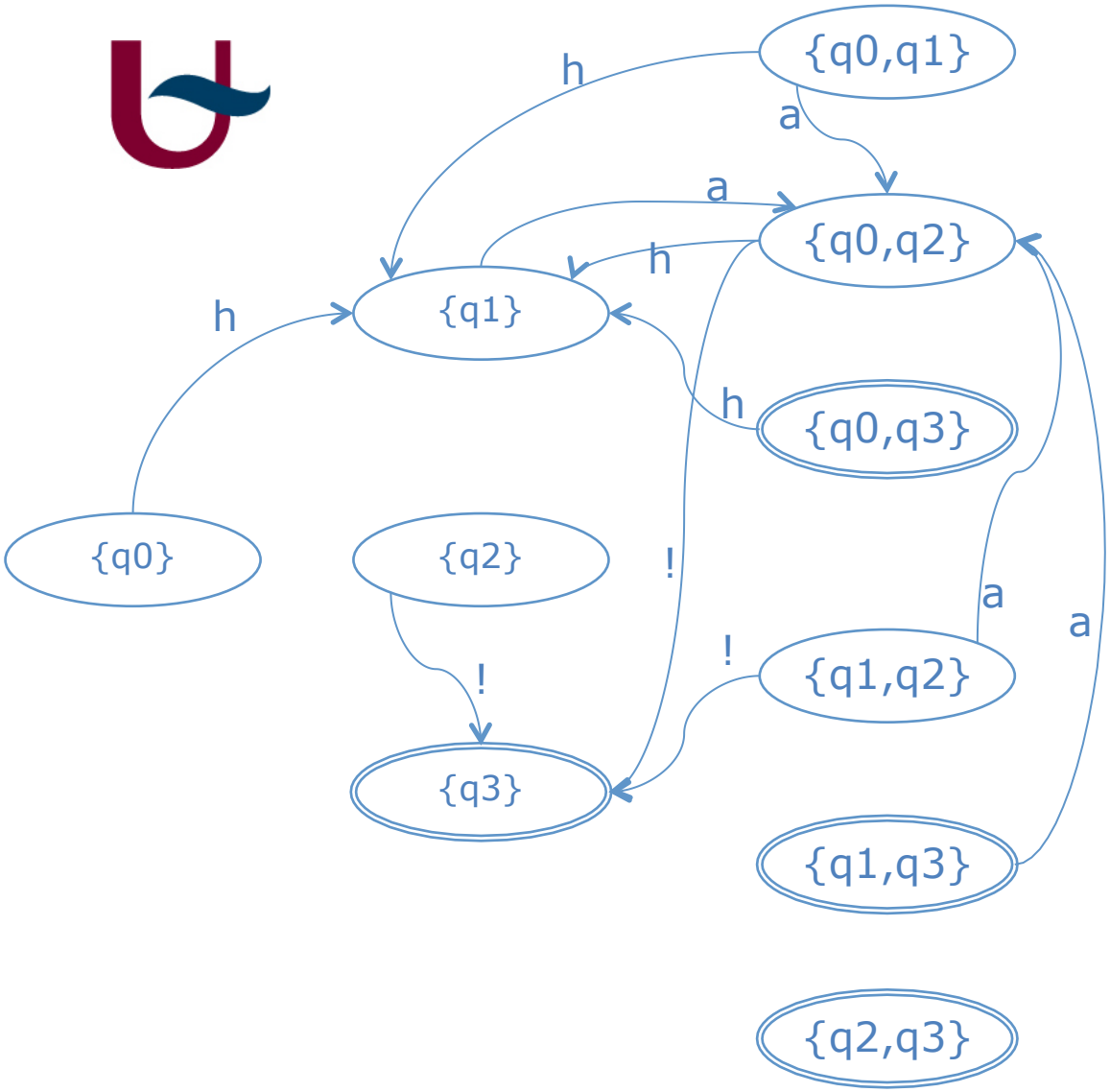

{q0,q1,q2}
{q0,q1,q3}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2,q3}

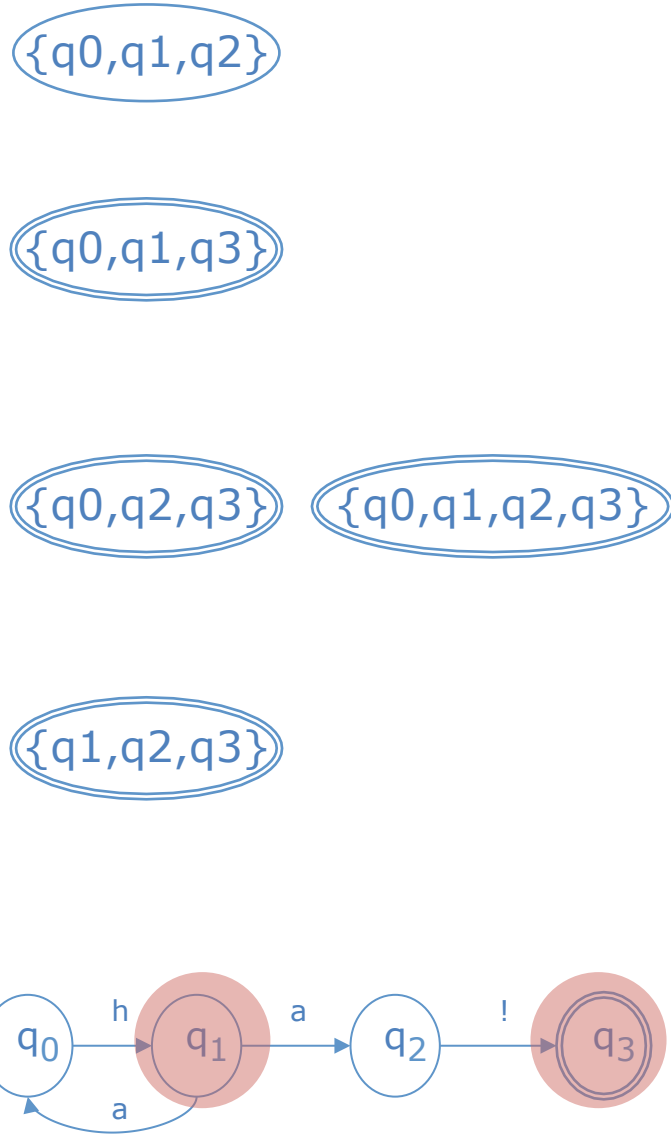

q0
h
q1
a
q2
!
q3
a


{q0,q1}
h
a
a
{q0,q2}
h
h
{q1}
h
{q0,q3}
{q0}
{q2}
!
a
!
{q1,q2}
!
{q3}
{q1,q3}
{q2,q3}

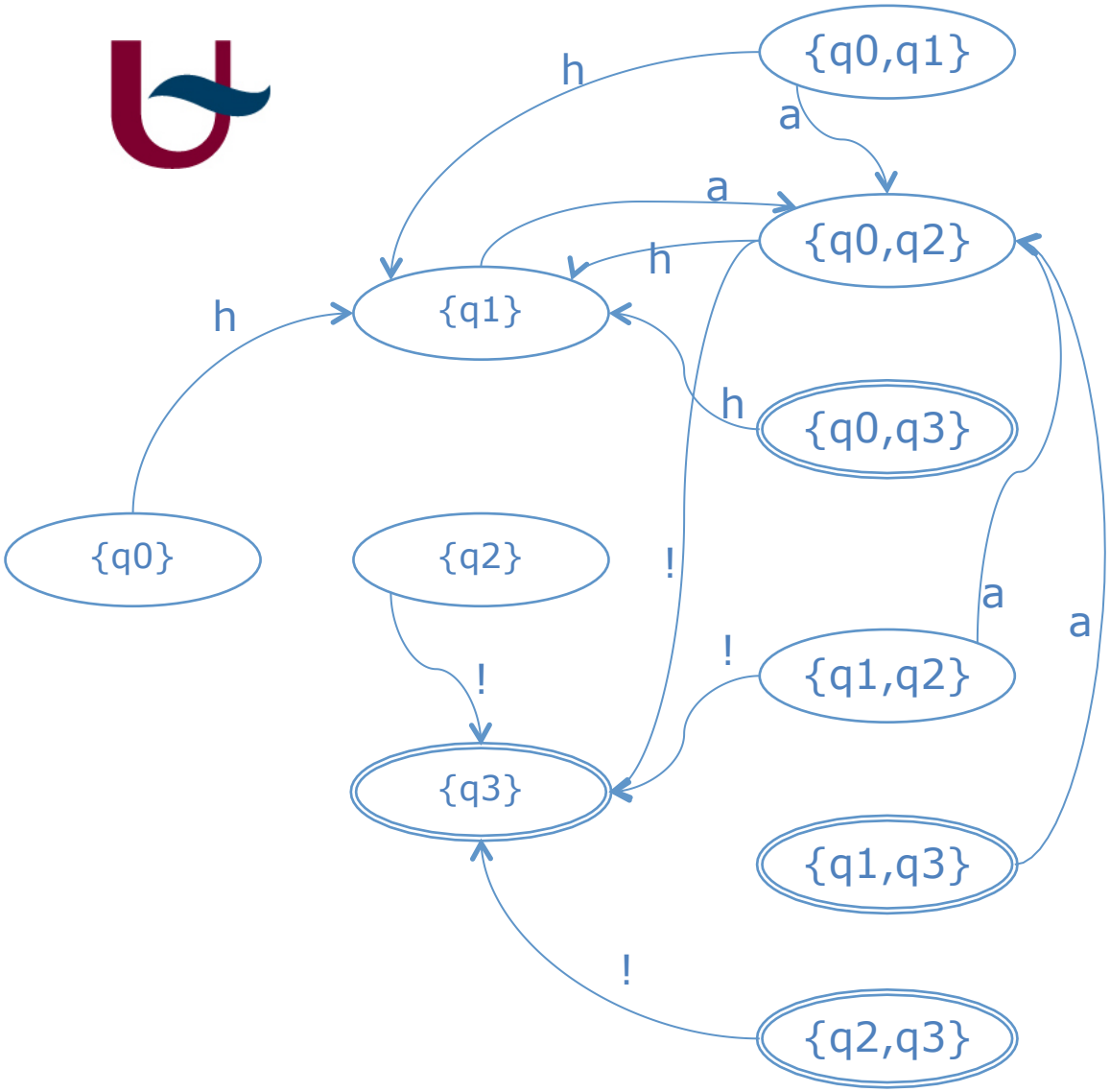

{q0,q1,q2}
{q0,q1,q3}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2,q3}

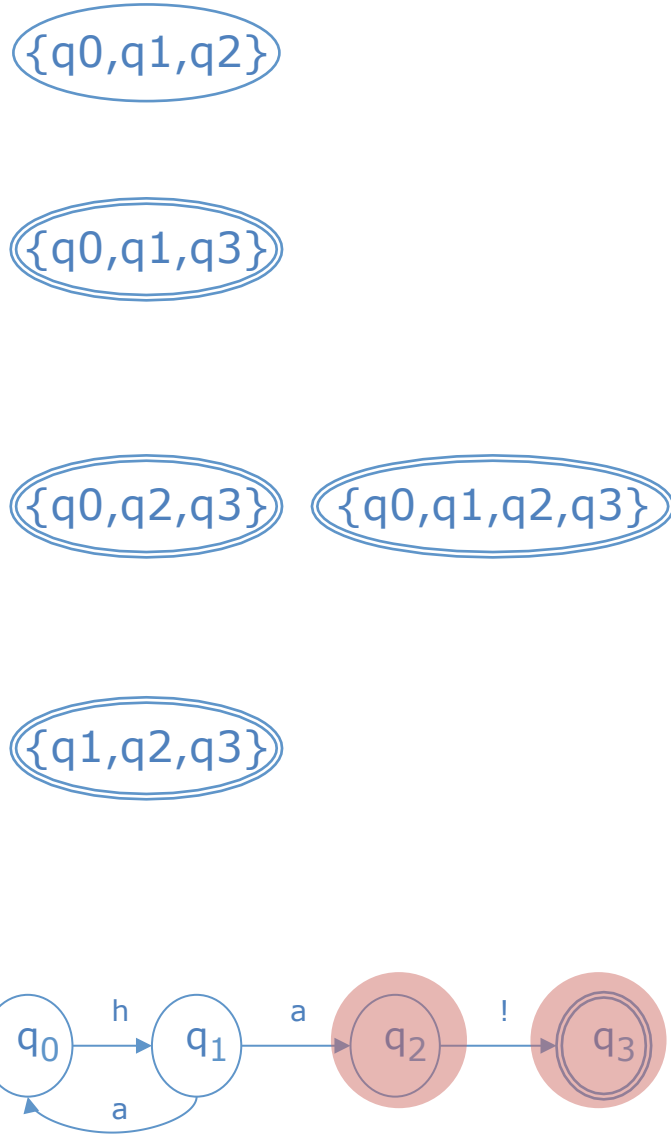

q0
h
q1
a
q2
!
q3
a


{q0,q1}
h
a
a
{q0,q2}
h
h
{q1}
h
{q0,q3}
{q0}
{q2}
!
a
a
!
{q1,q2}
!
{q3}
{q1,q3}
{q2,q3}

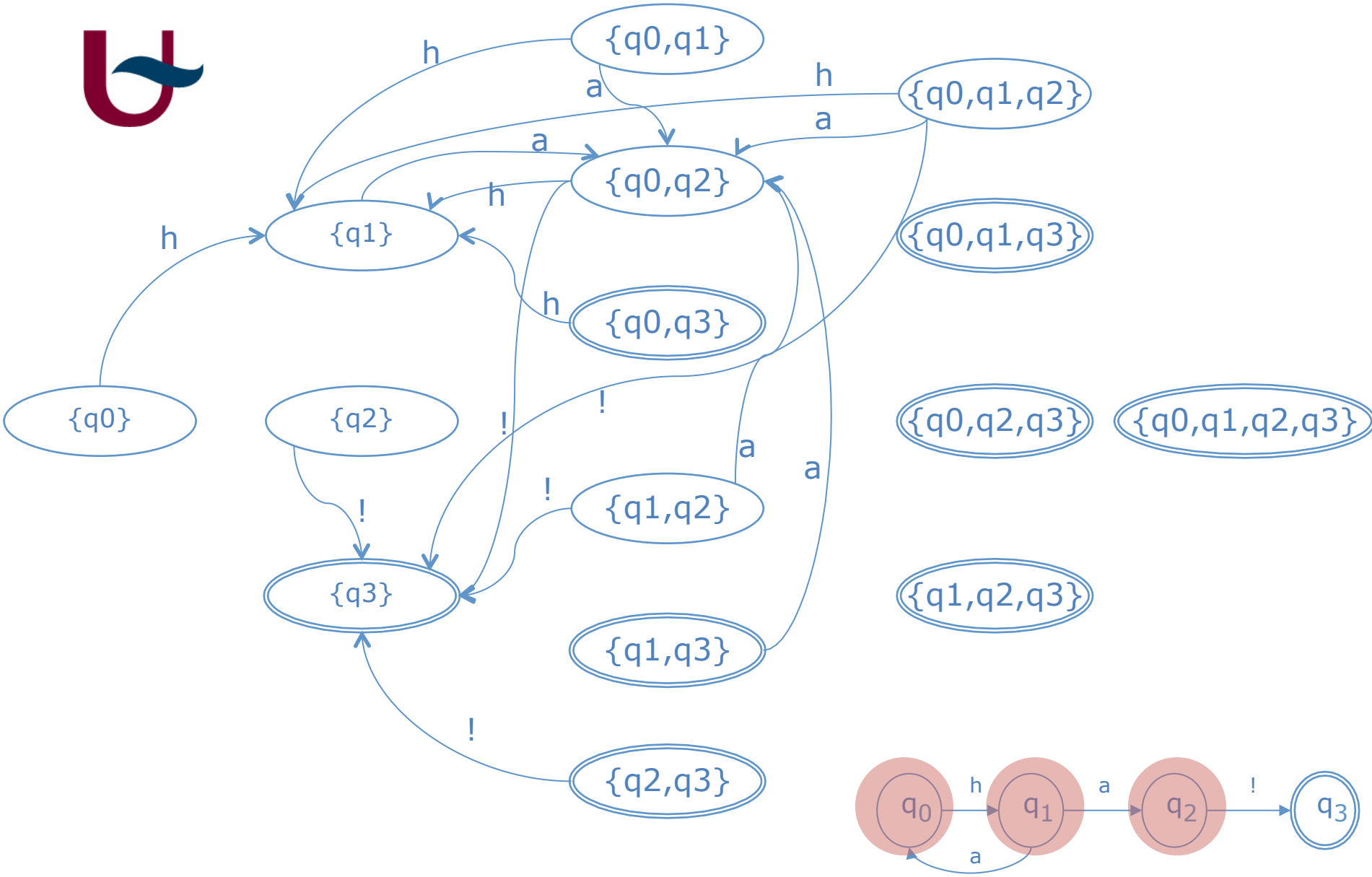

{q0,q1,q2}
{q0,q1,q3}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2,q3}

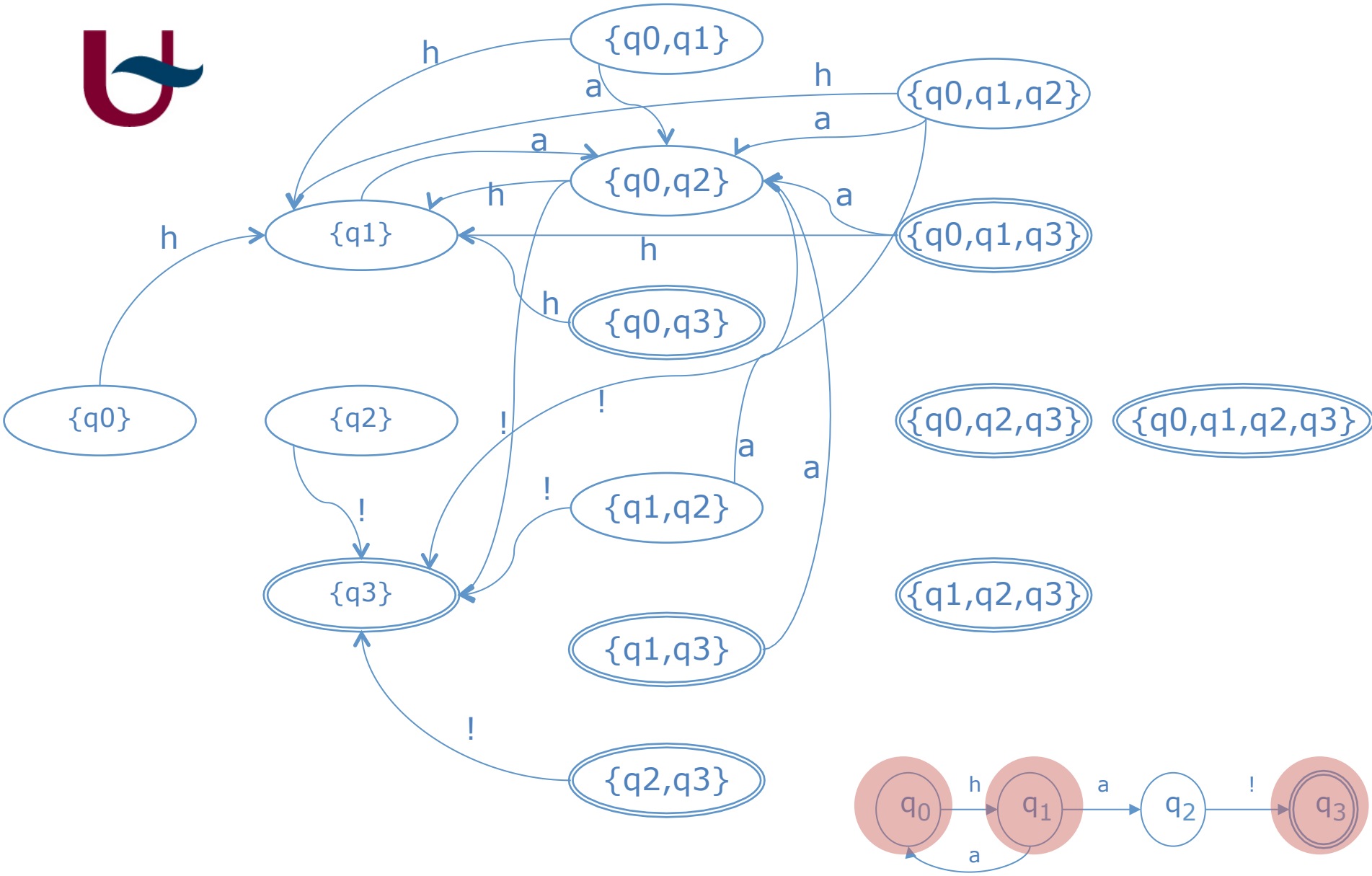
q0
h
q1
a
q2
!
q3
a

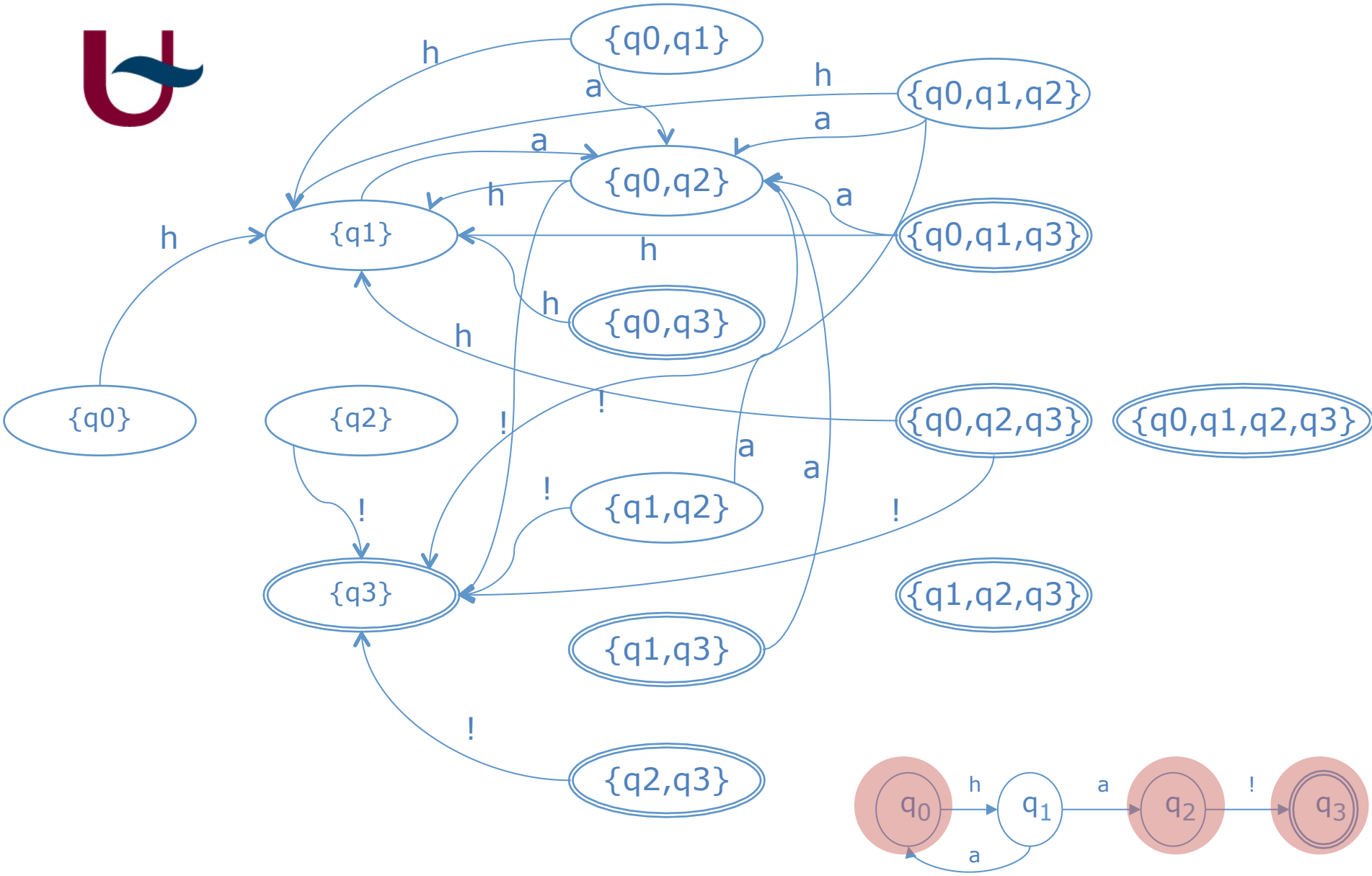

{q0,q1}
h
a
a
{q0,q2}
h
h
{q1}
h
{q0,q3}
{q0}
{q2}
!
a
a
!
{q1,q2}
!
{q3}
{q1,q3}
!
{q2,q3}

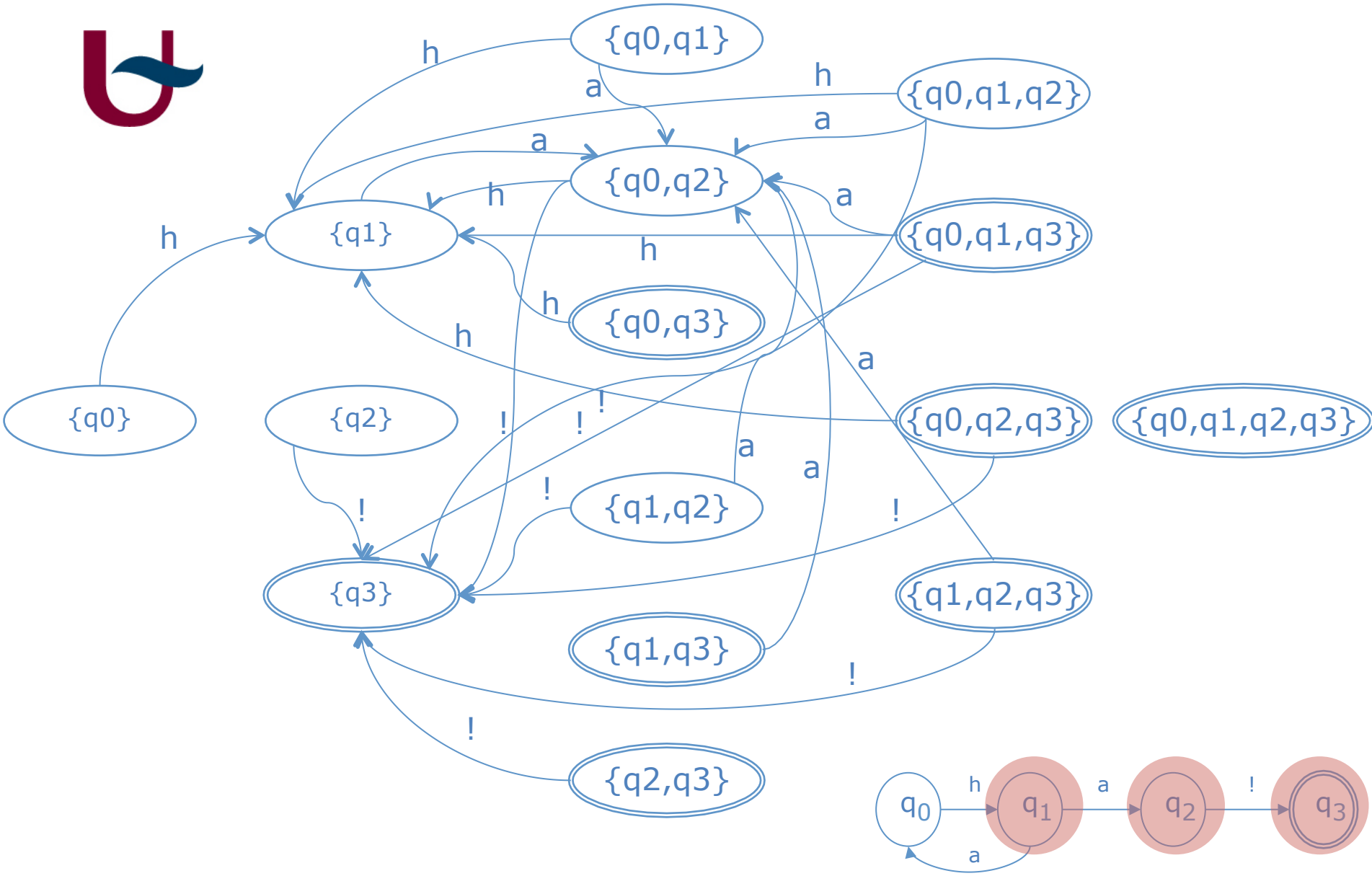

{q0,q1,q2}
{q0,q1,q3}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2,q3}

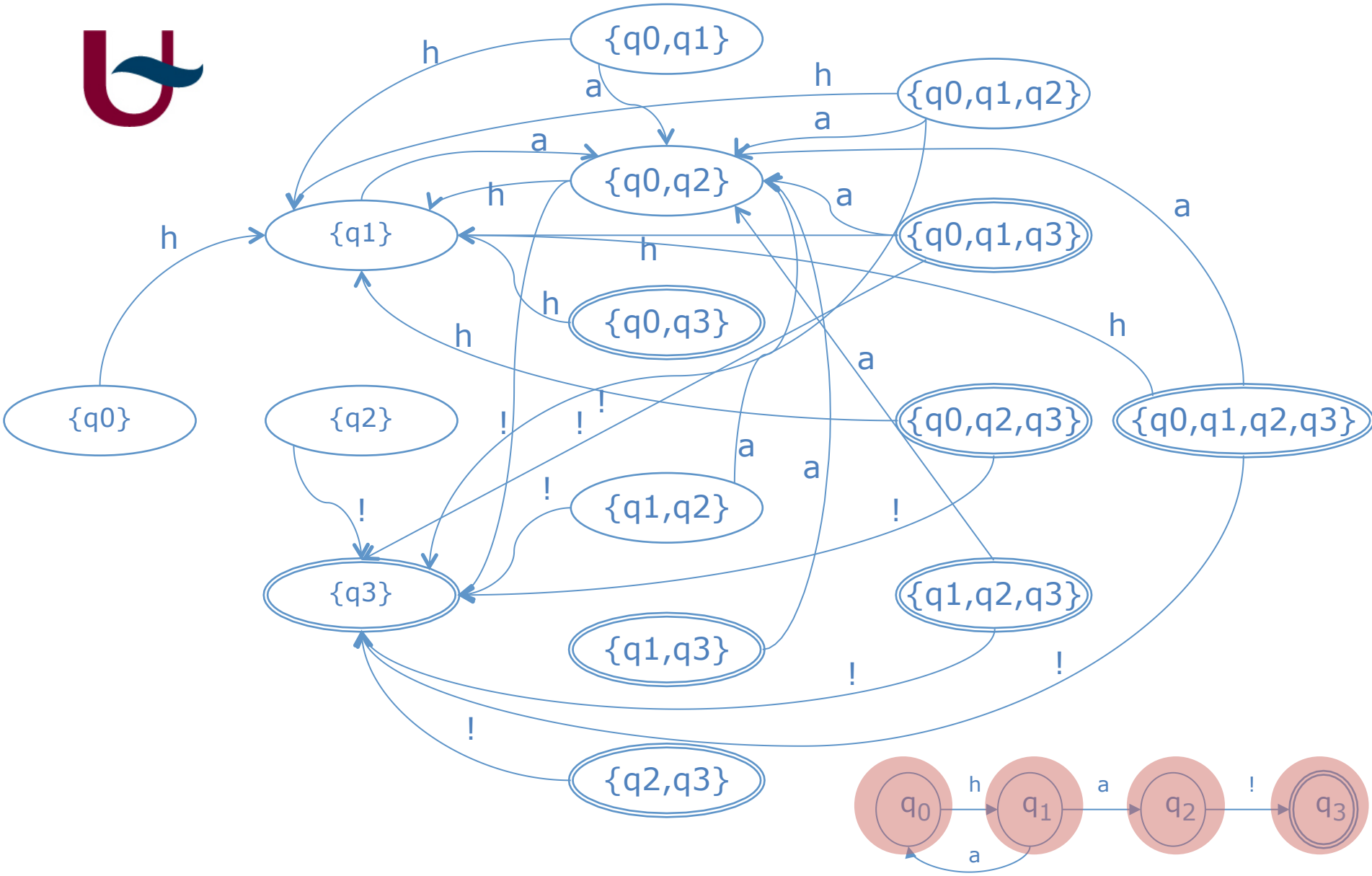
q0
h
q1
a
q2
!
q3
a

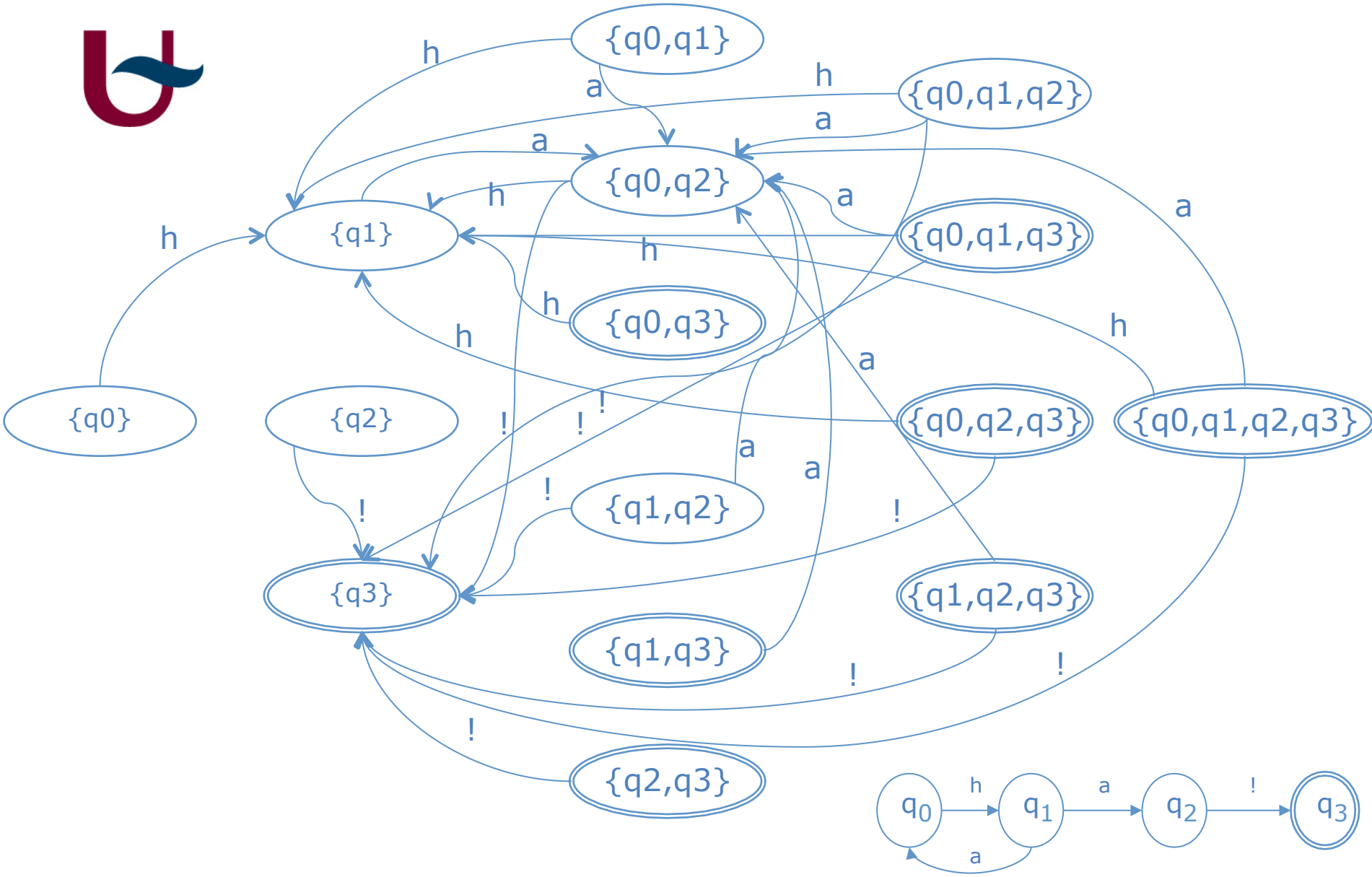

{q0,q1}
{q0,q1,q2}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
h
a
!
q0
q1
q2
q3

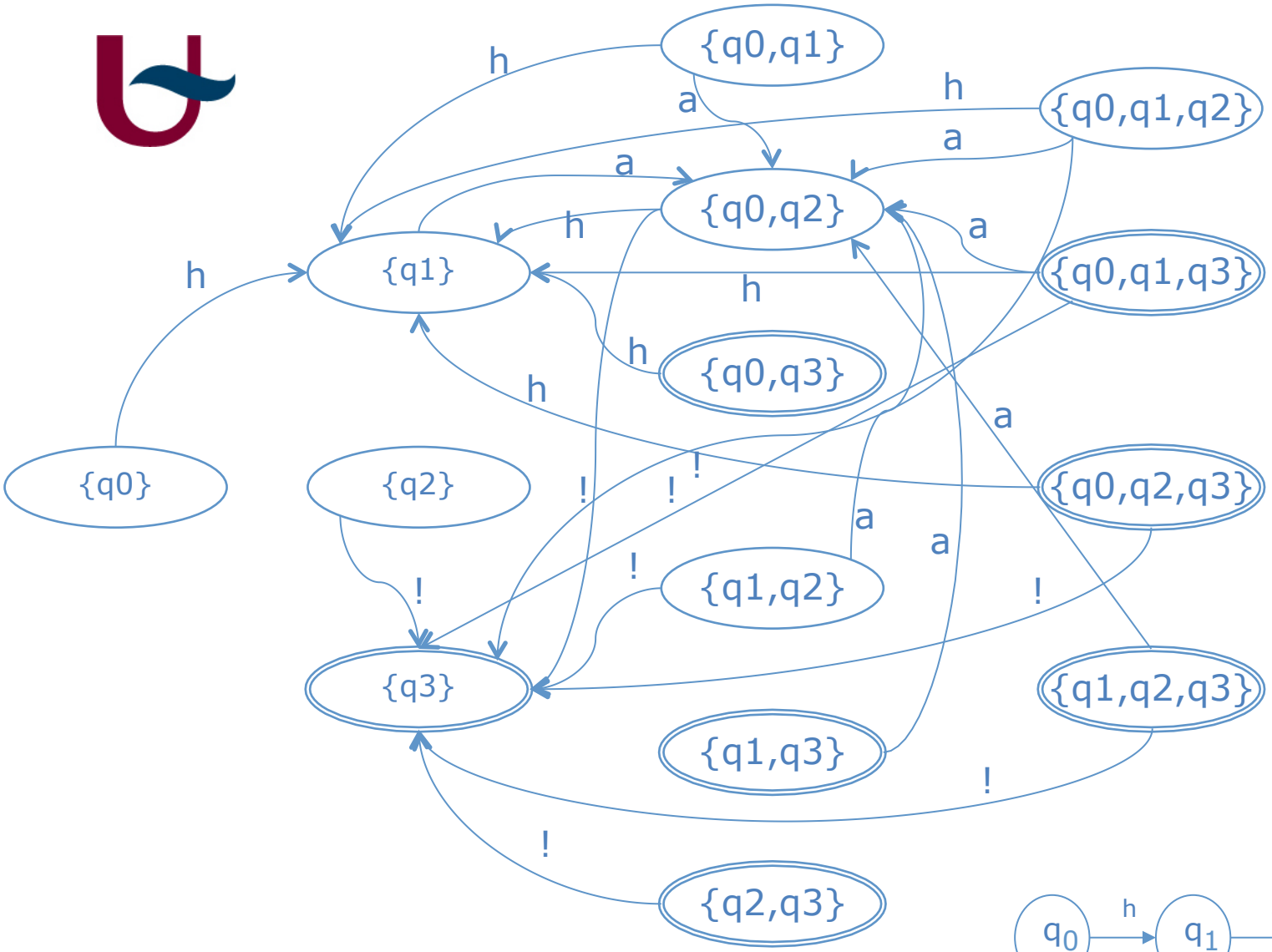

{q0,q1}
{q0,q1,q2}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
h
a
!
q0
q1
q2
q3

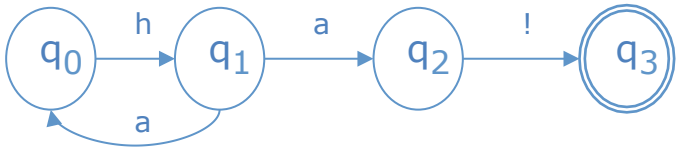

{q0,q1}
{q0,q1,q2}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
h
a
!
q0
q1
q2
q3

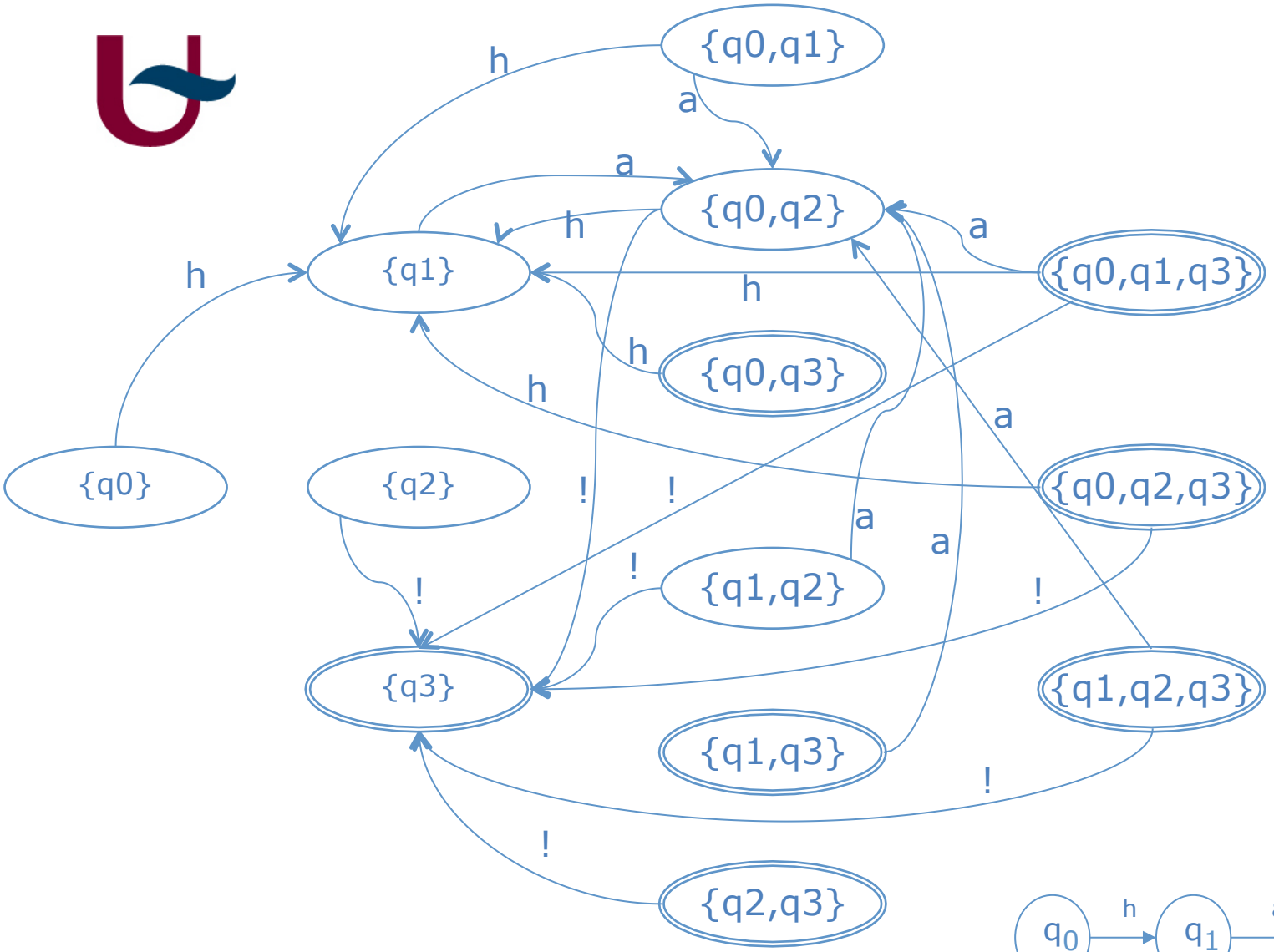

{q0,q1}
{q0,q1,q2}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
h
a
!
q0
q1
q2
q3

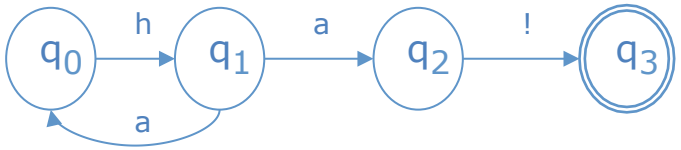

{q0,q1}
{q0,q1,q2}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
h
a
!
q0
q1
q2
q3

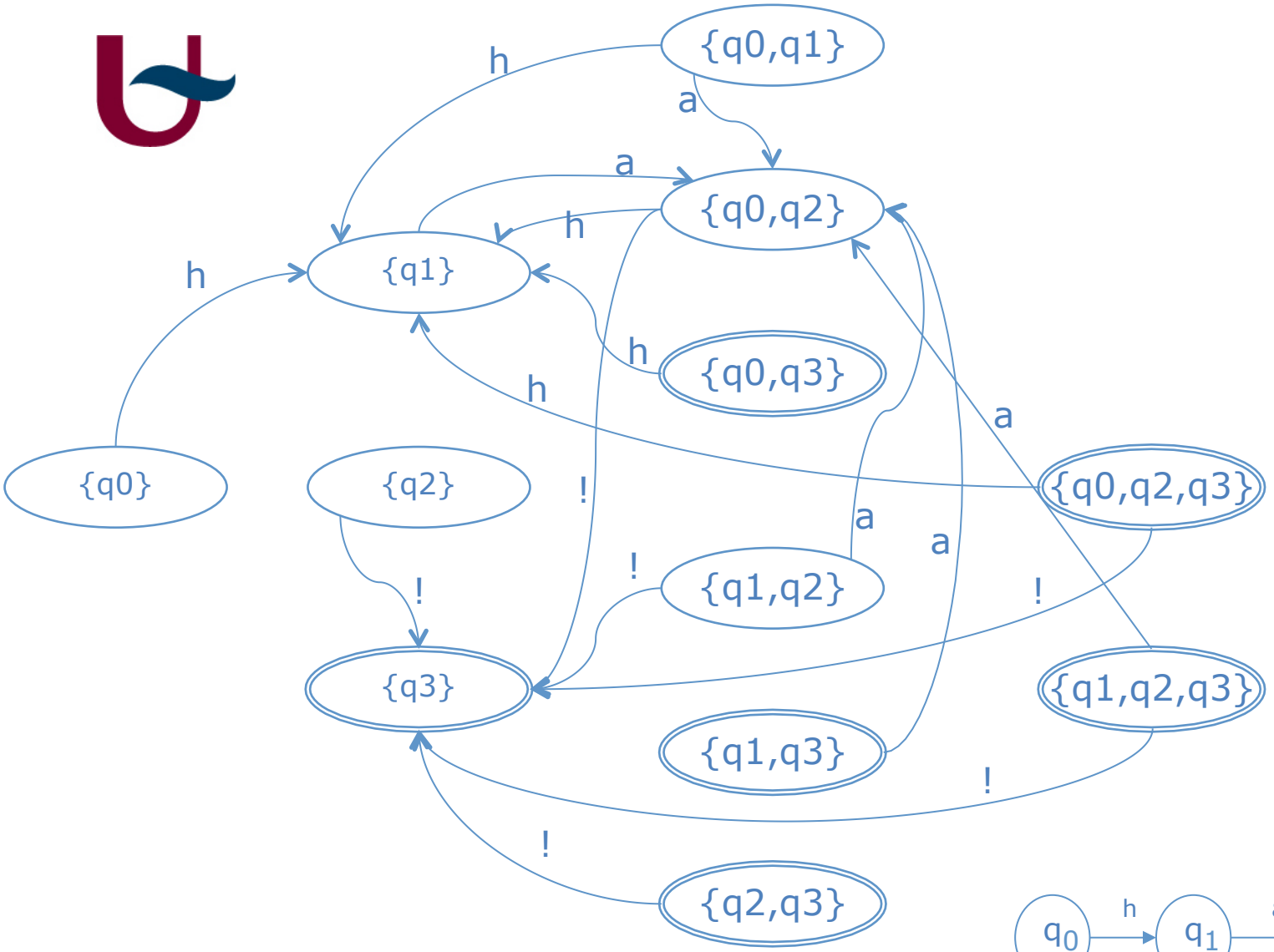

{q0,q1}
{q0,q1,q2}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q0,q1,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
h
a
!
q0
q1
q2
q3

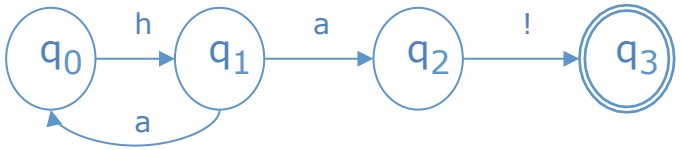

{q0,q1}
{q0,q1,q2}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
h
a
!

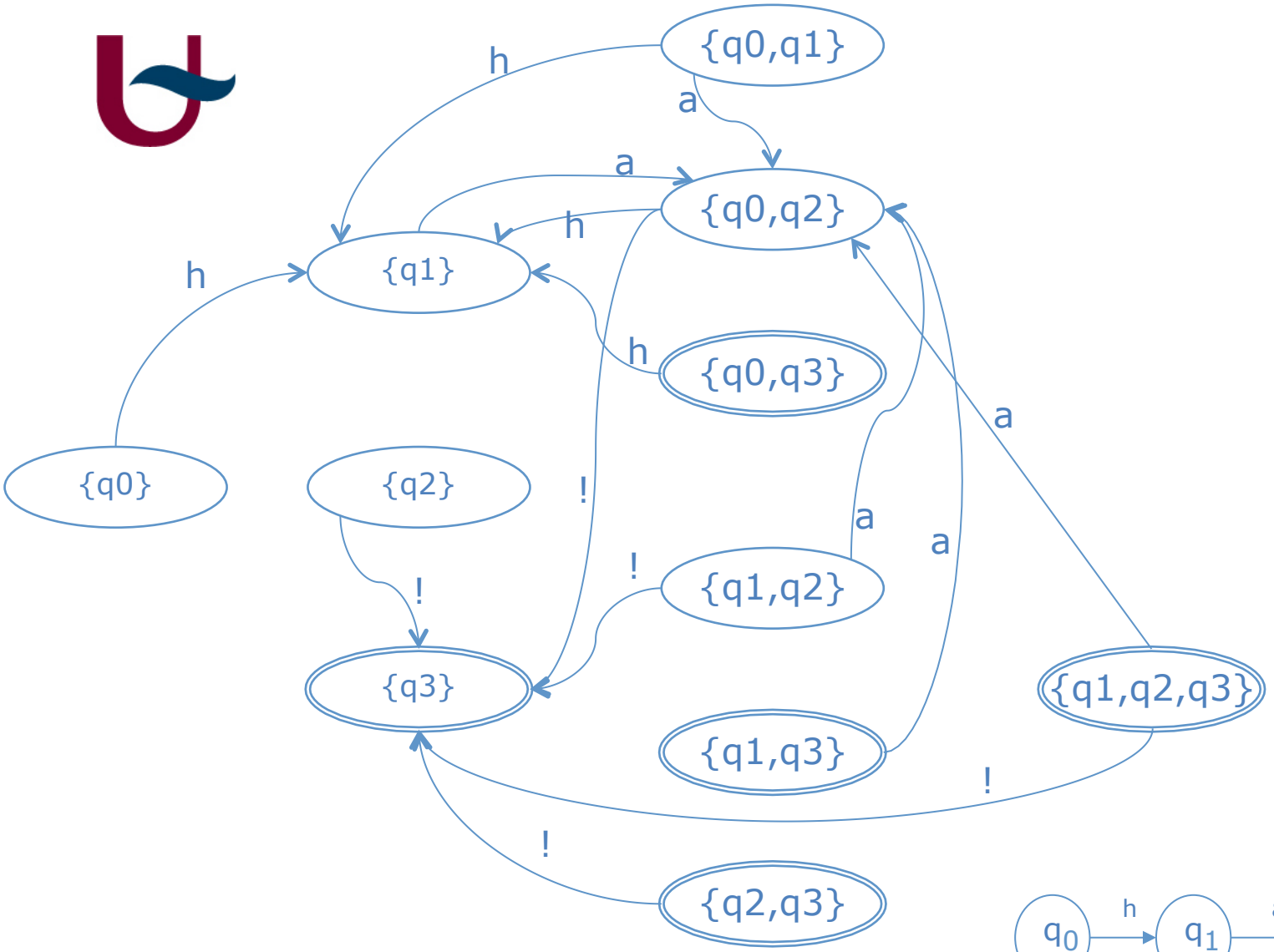

q0
q1
q2
q3
h
a
!
a

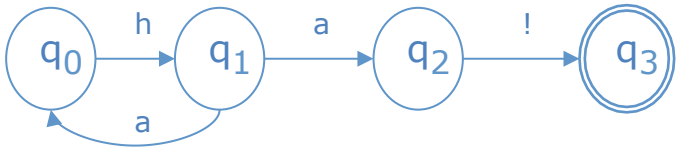

{q0,q1}
{q0,q2}
{q1}
{q0,q1,q3}
{q0,q3}
{q0}
{q2}
{q0,q2,q3}
{q1,q2}
{q3}
{q1,q2,q3}
{q1,q3}
{q2,q3}
h
a
!

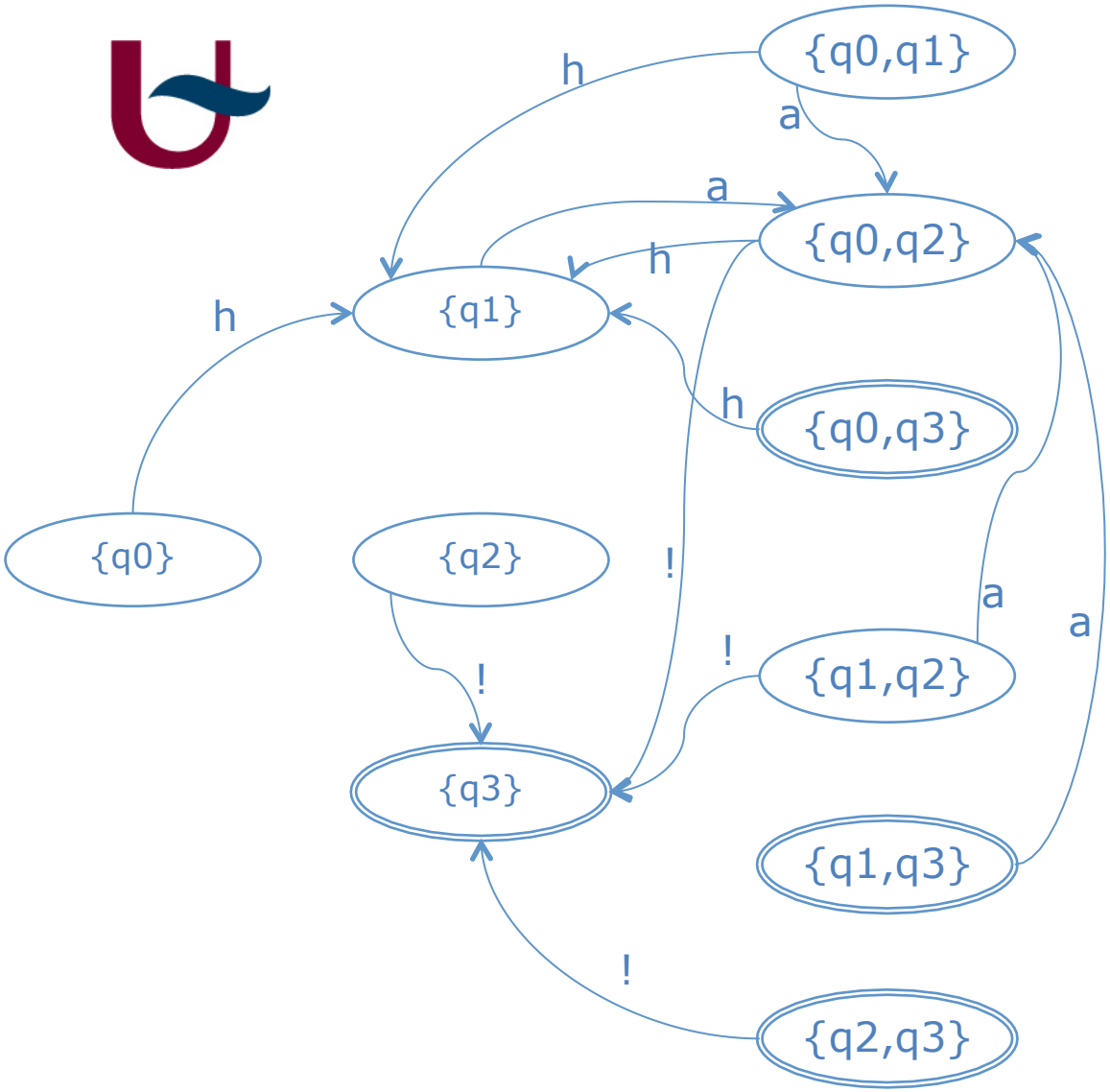

q0
q1
q2
q3
h
a
!
a

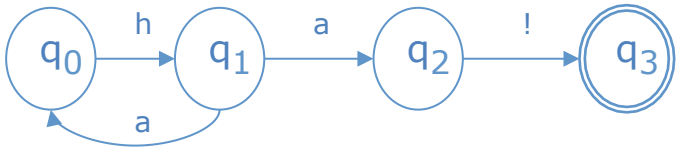

{q0,q1}
h
a
a
{q0,q2}
h
h
{q1}
h
h
{q0,q3}
a
{q0}
{q2}
!
{q0,q2,q3}
a
a
!
{q1,q2}
!
!
{q3}
{q1,q2,q3}
{q1,q3}
!
!
{q2,q3}

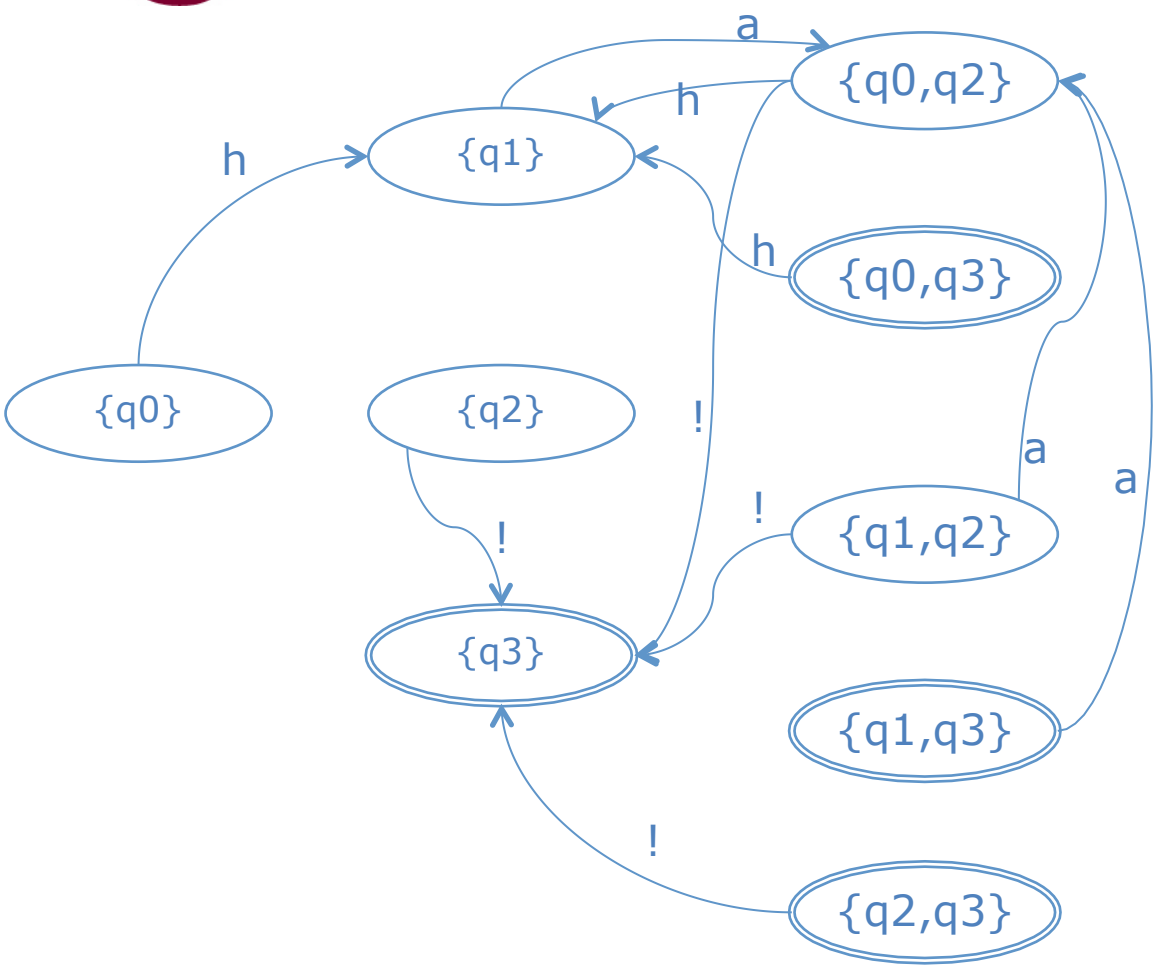

q0
h
q1
a
q2
!
q3
a

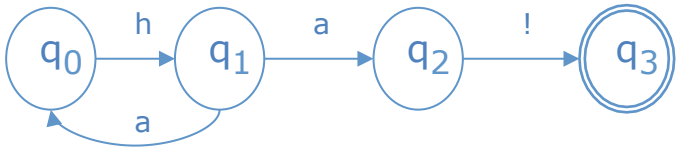

{q0,q1}
h
a
a
{q0,q2}
h
h
{q1}
h
{q0,q3}
a
{q0}
{q2}
!
a
a
!
{q1,q2}
!
{q3}
{q1,q2,q3}
{q1,q3}
!
!
{q2,q3}

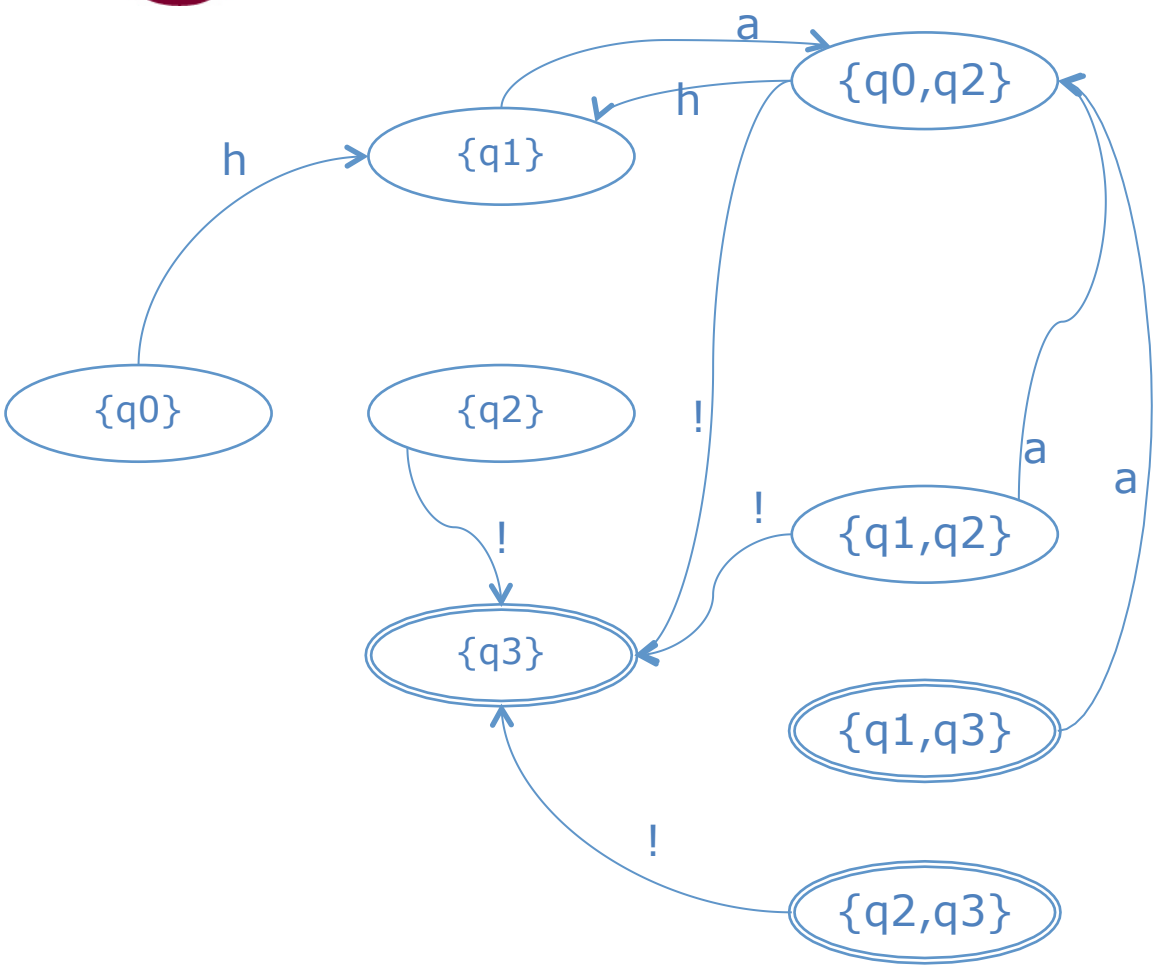

q0
h
q1
a
q2
!
q3
a

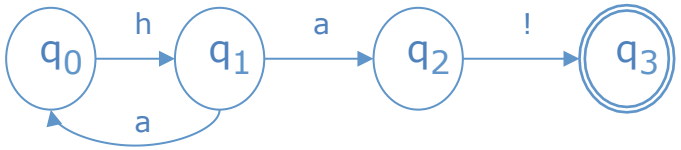

{q0,q1}
h
a
a
{q0,q2}
h
h
{q1}
h
{q0,q3}
{q0}
{q2}
!
a
a
!
{q1,q2}
!
{q3}
{q1,q3}
!
{q2,q3}

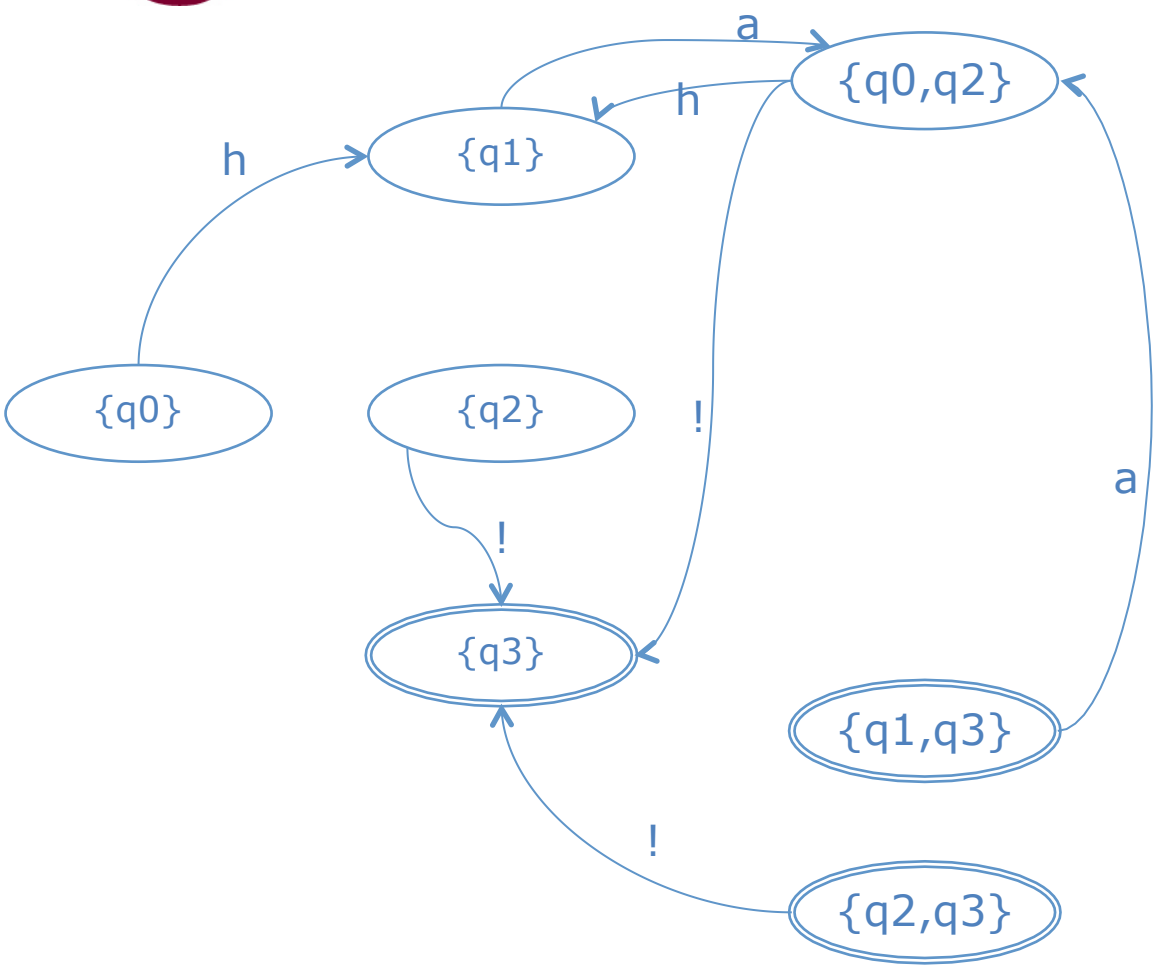

q0
h
q1
a
q2
!
q3
a

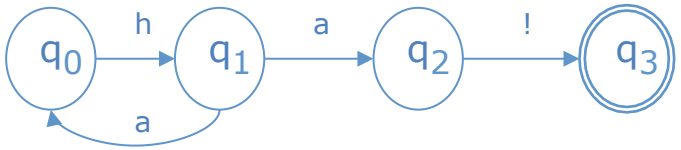

a
{q0,q2}
h
{q1}
h
h
{q0,q3}
{q0}
{q2}
!
a
a
!
{q1,q2}
!
{q3}
{q1,q3}
!
{q2,q3}

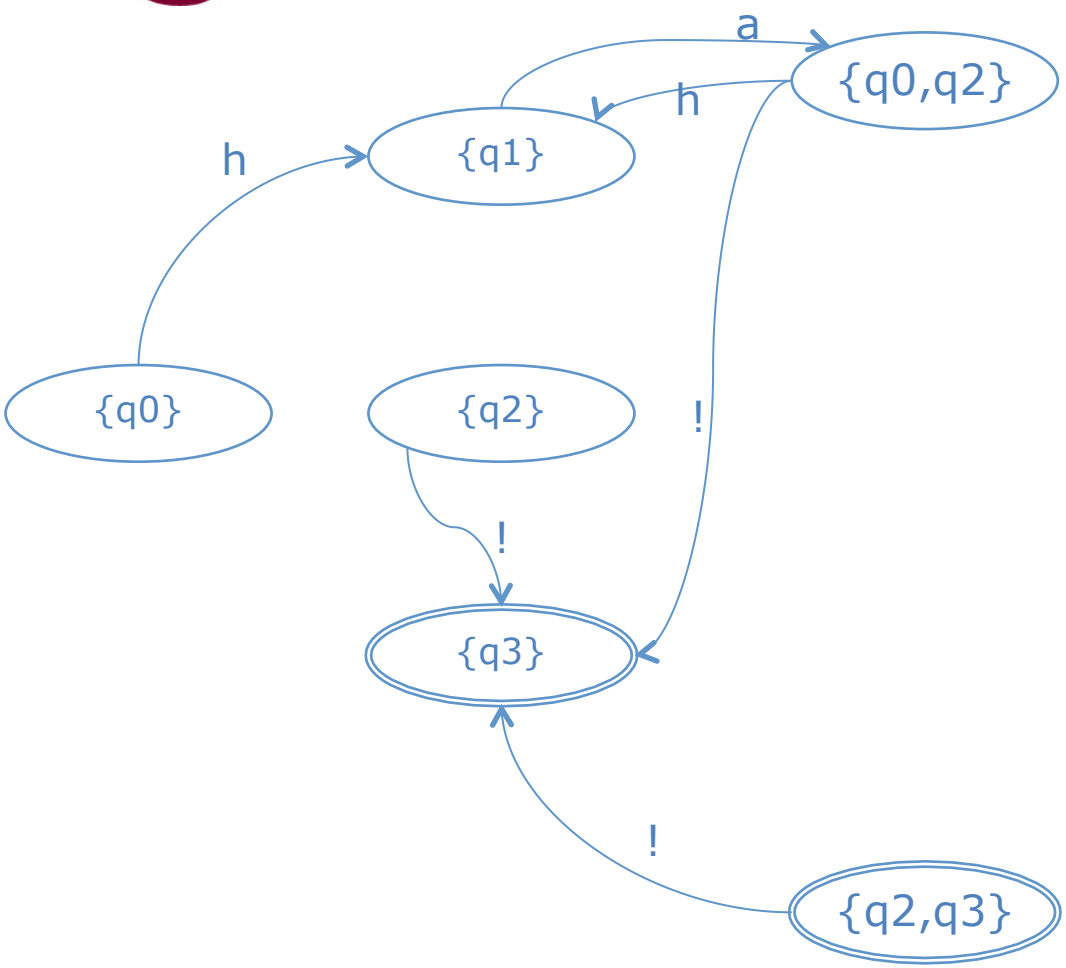

q0
h
q1
a
q2
!
q3
a

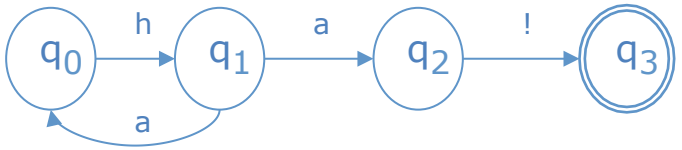

a
{q0,q2}
h
{q1}
h
{q0}
{q2}
!
a
a
!
{q1,q2}
!
{q3}
{q1,q3}
!
{q2,q3}

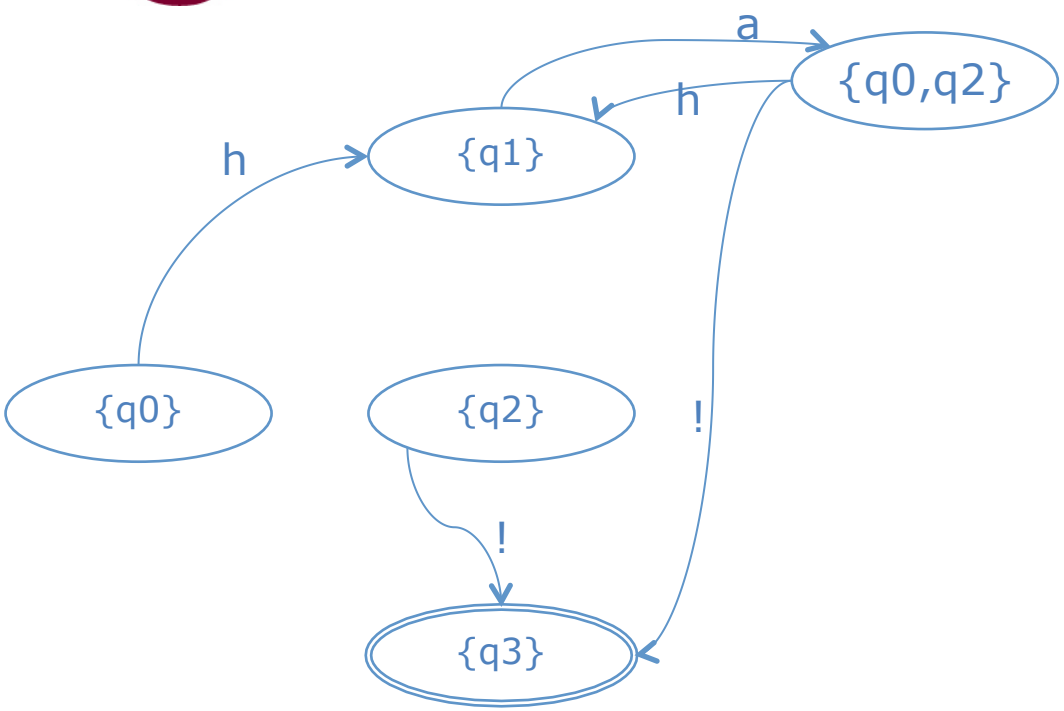

q0
h
q1
a
q2
!
q3
a

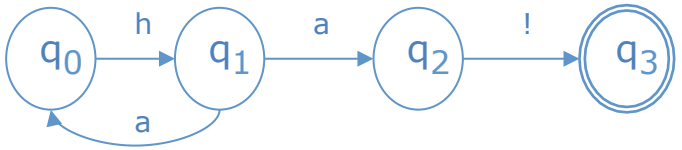

a
{q0,q2}
h
{q1}
h
{q0}
{q2}
!
a
!
{q3}
{q1,q3}
!
{q2,q3}

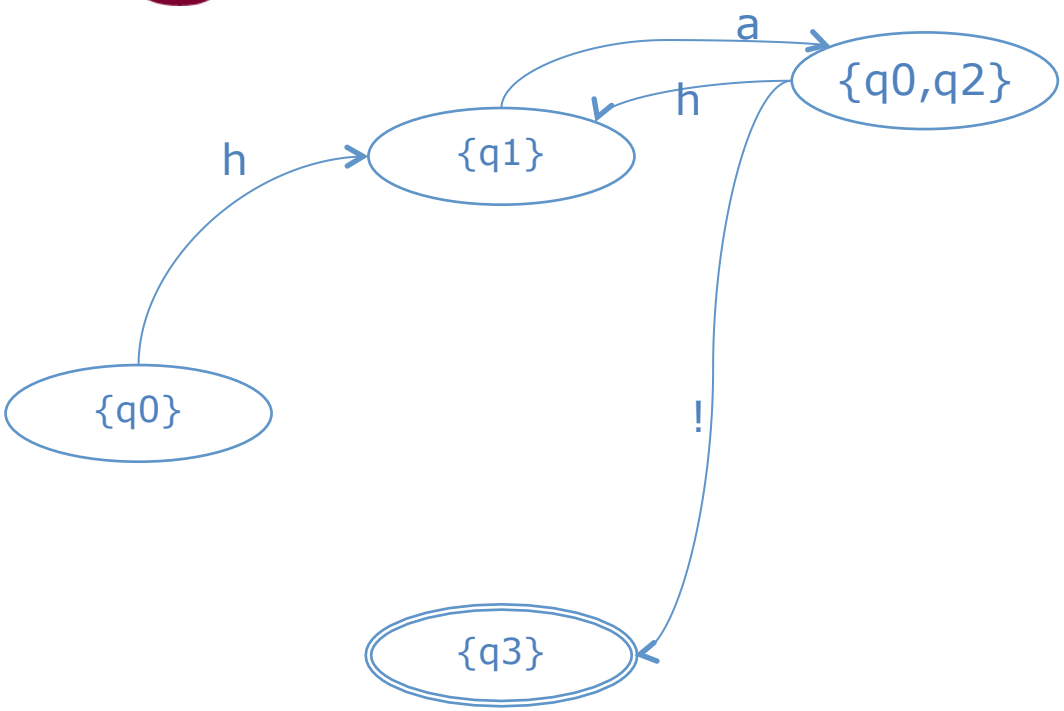

q0
h
q1
a
q2
!
q3
a

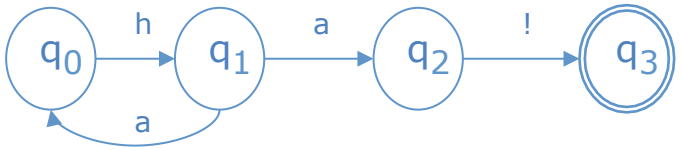

a
{q0,q2}
h
h
{q1}
{q0}
{q2}
!
!
{q3}
!
{q2,q3}

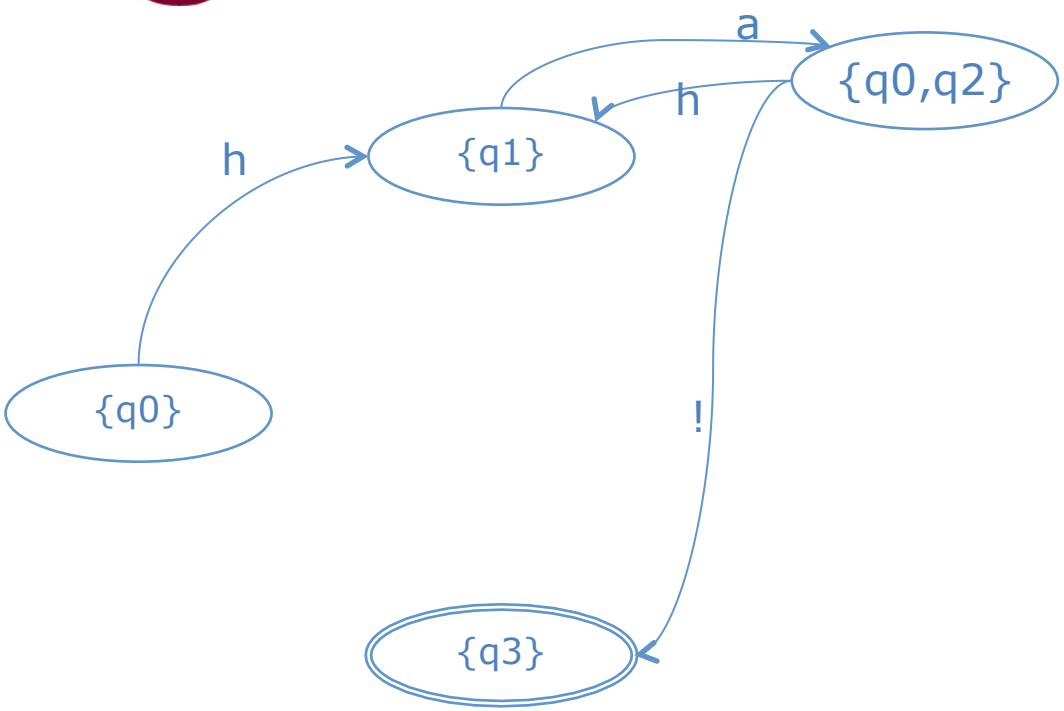

q0
h
q1
a
q2
!
q3
a

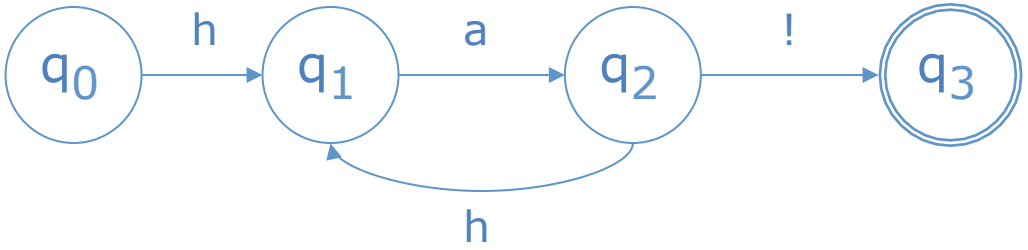

a
{q0,q2}
h
h
{q1}
{q0}
{q2}
!
!
{q3}


q0
h
q1
a
q2
!
q3
a


a
{q0,q2}
h
h
{q1}
{q0}
!
{q3}

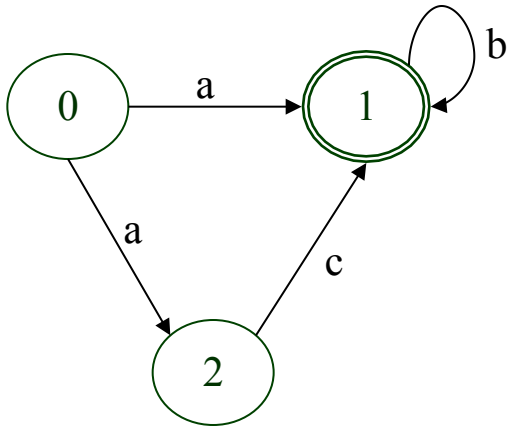

q0
h
q1
a
q2
!
q3
a

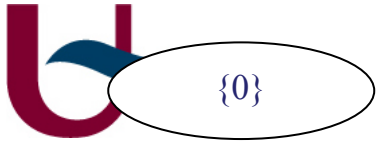

a
{q0,q2}
h
h
{q1}
{q0}
!
{q3}


q0
h
q1
a
q2
!
q3
h

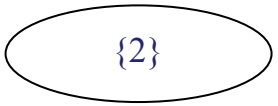
(Exercise)

- Turn this nondeterministic FSA for /ac?b*/ in a deterministic FSA




{0}


{1}

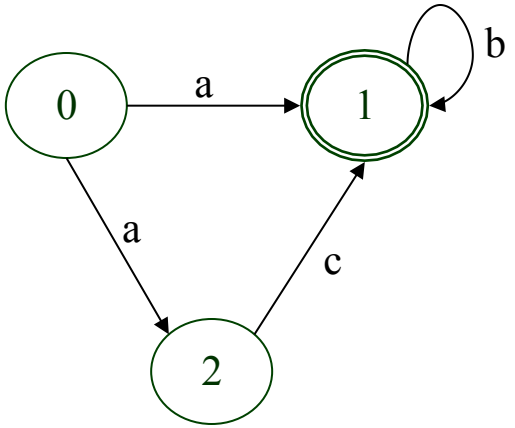

{2}

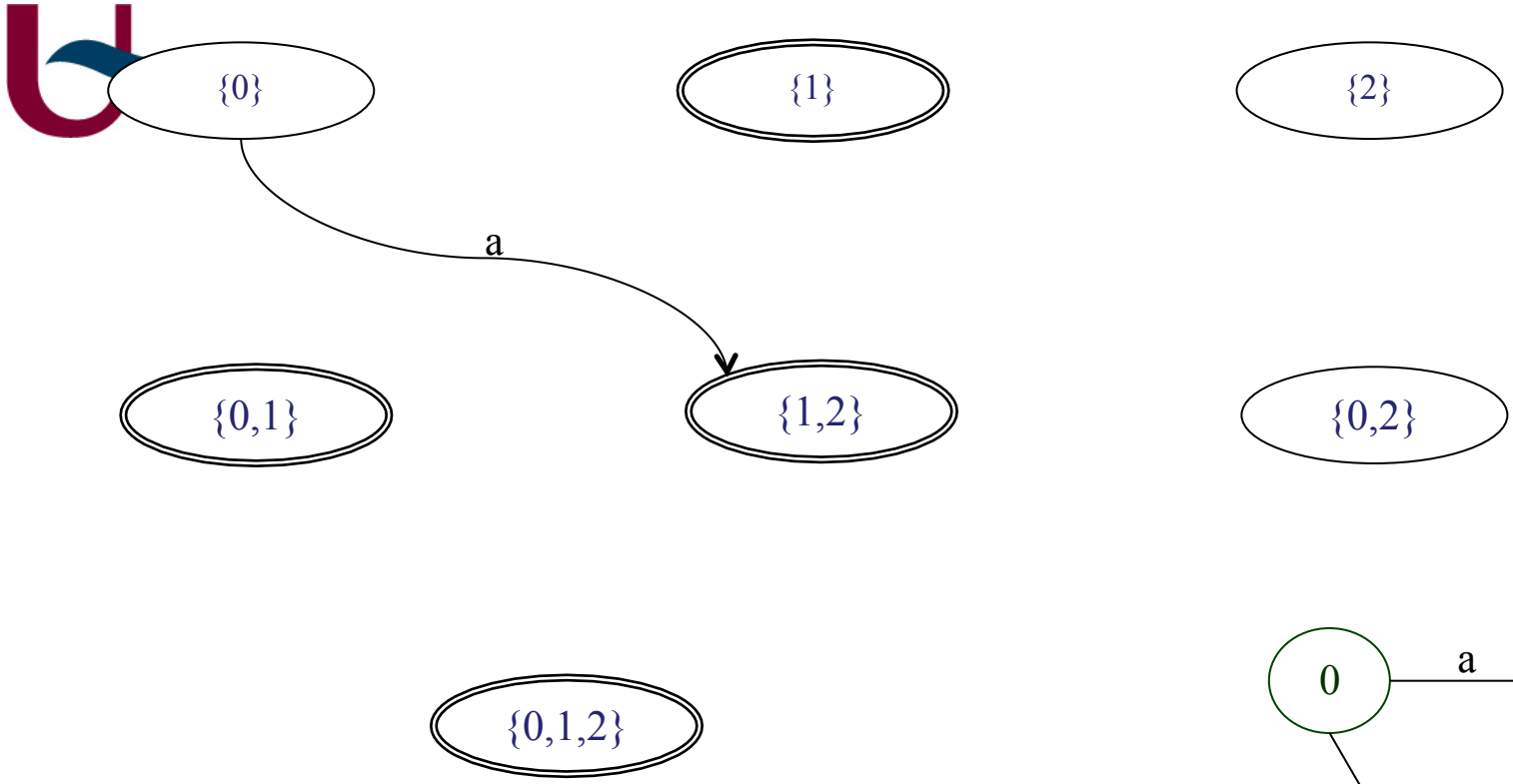

{0,1}

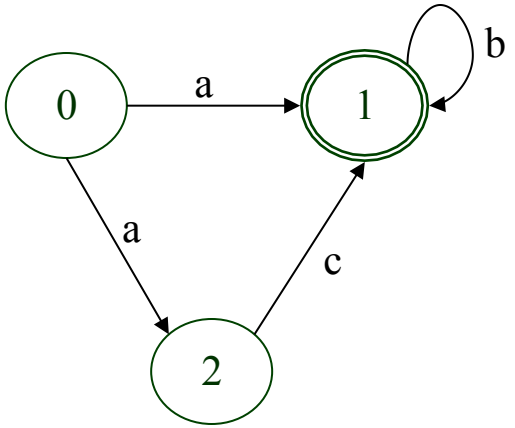

{1,2}

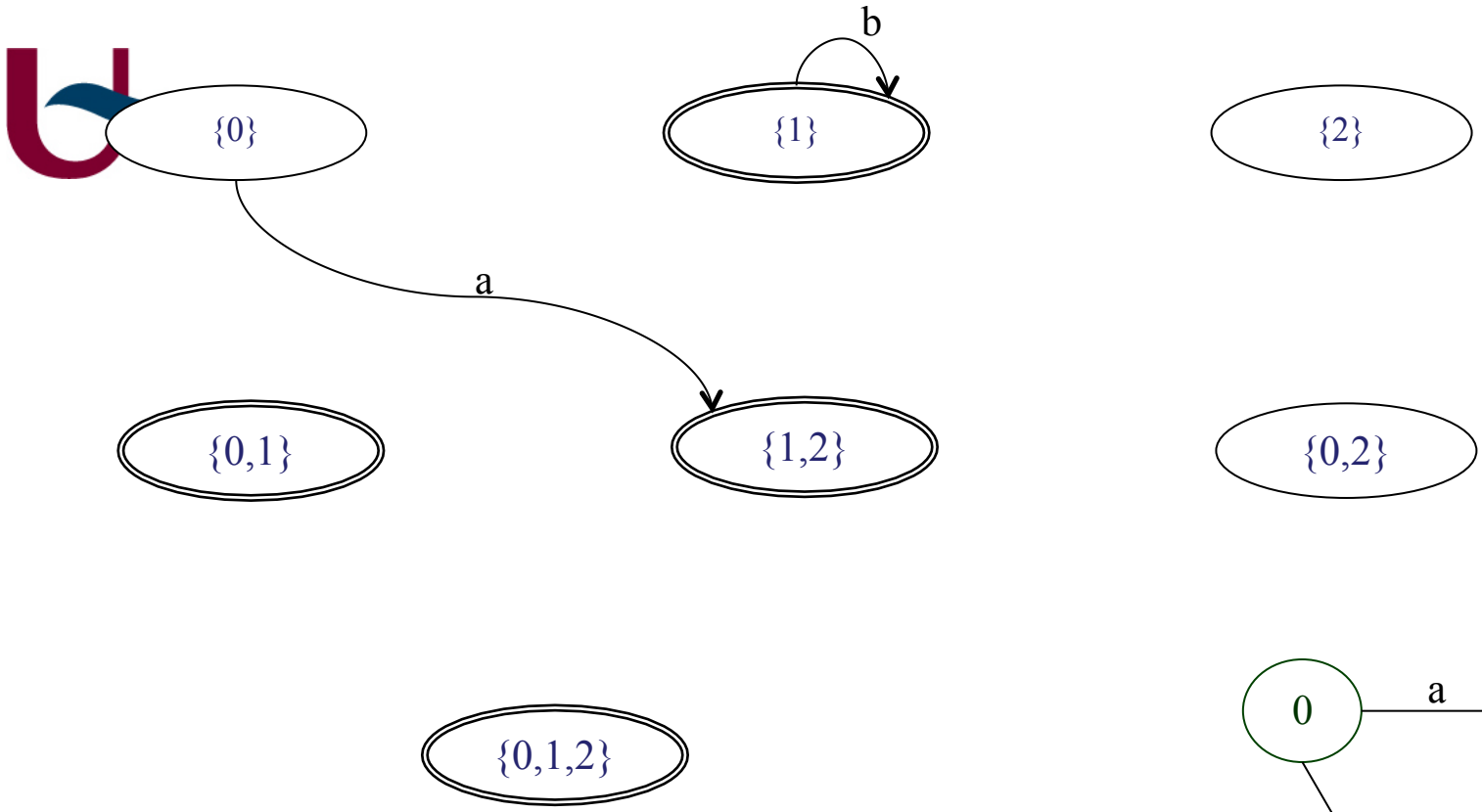

{0,2}

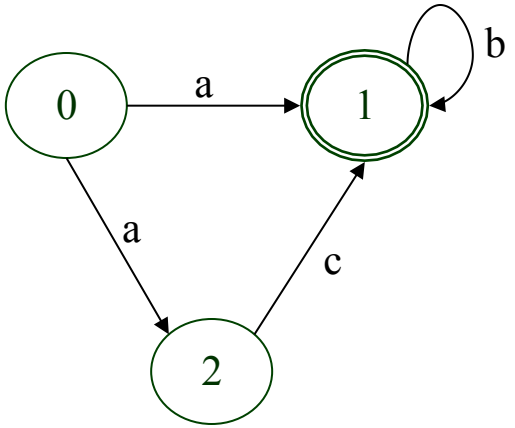

{0,1,2}

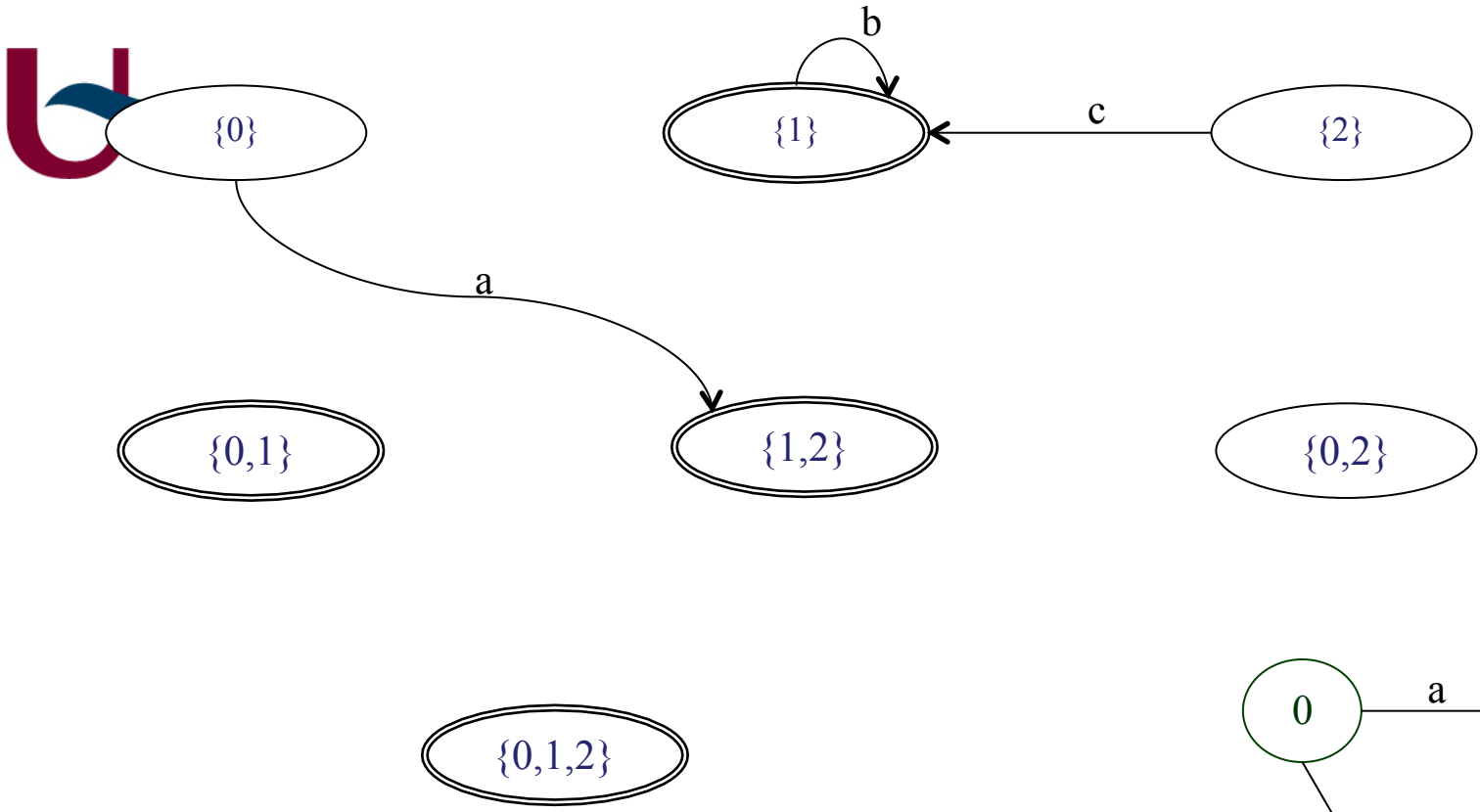

0
a
1
b
a
c
2

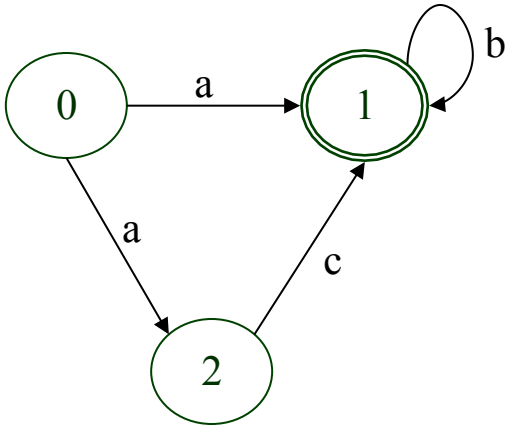

{0}
{1}
{2}
a
{0,1}
{1,2}
{0,2}
{0,1,2}

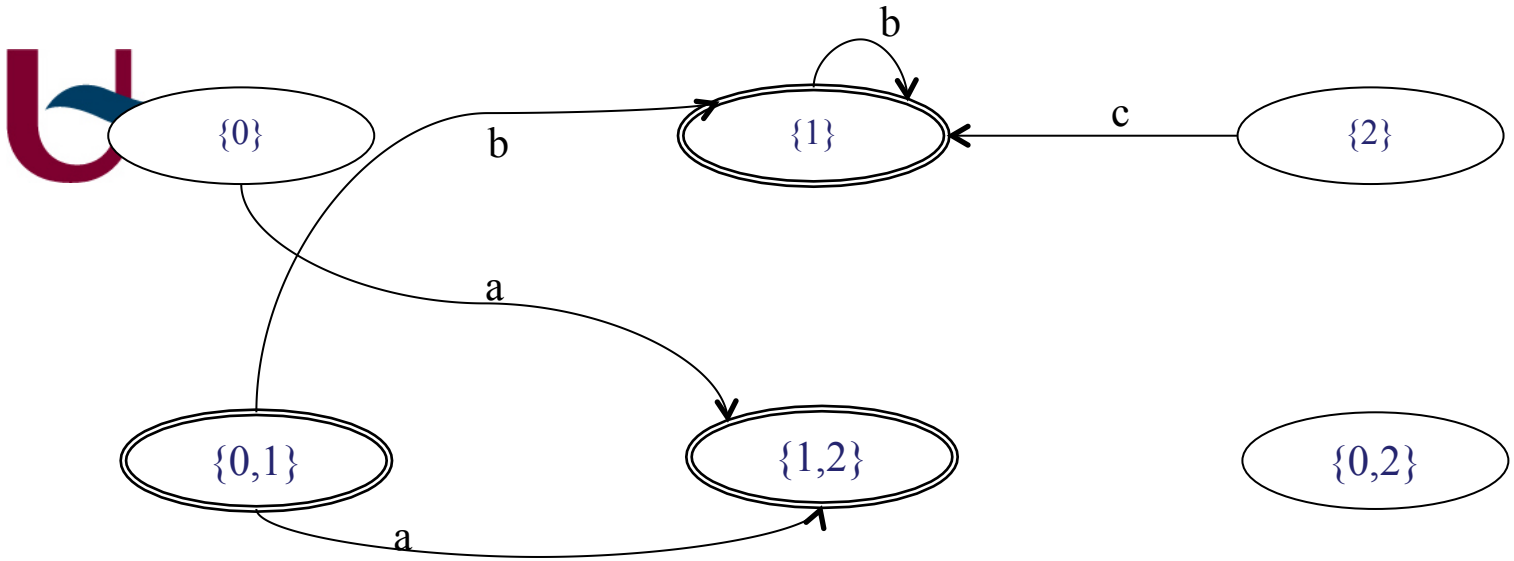

0
a
1
b
a
c
2

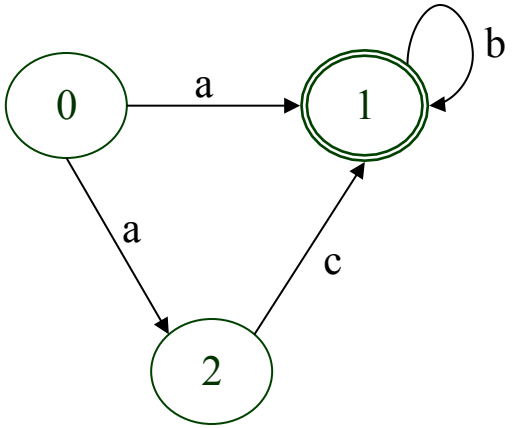

b
{0}
{1}
{2}
a
{0,1}
{1,2}
{0,2}
{0,1,2}

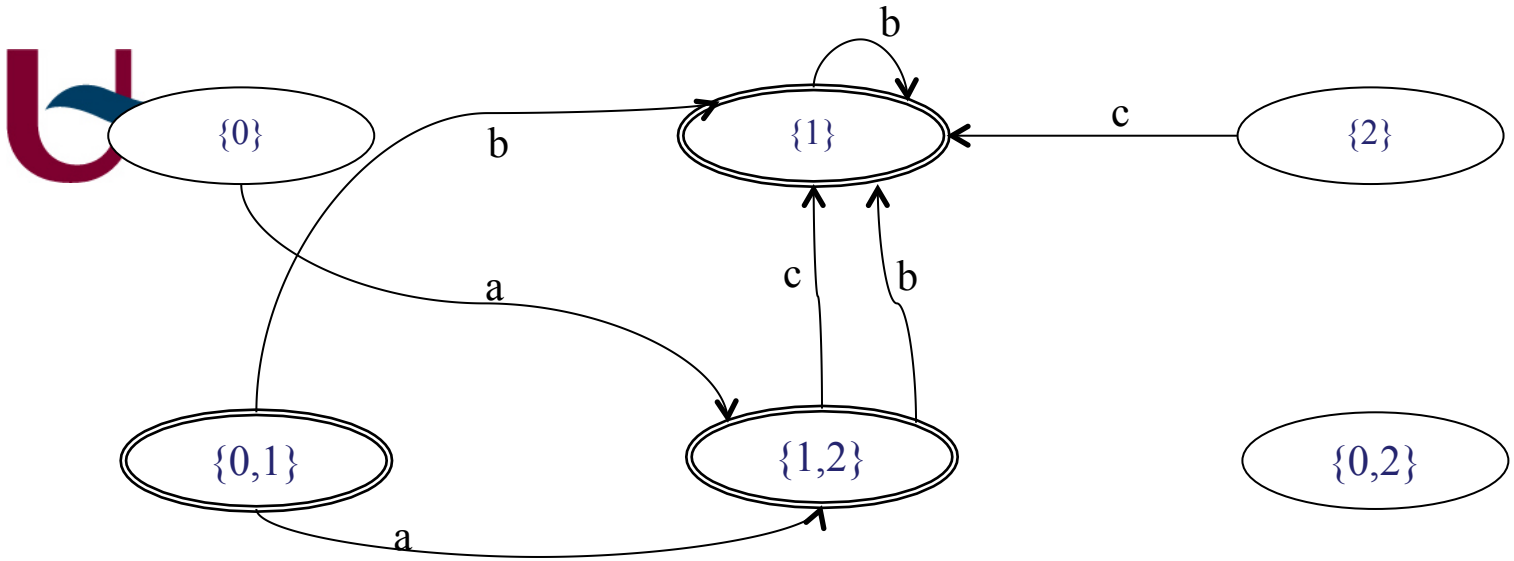

b
0
a
1
a
c
2

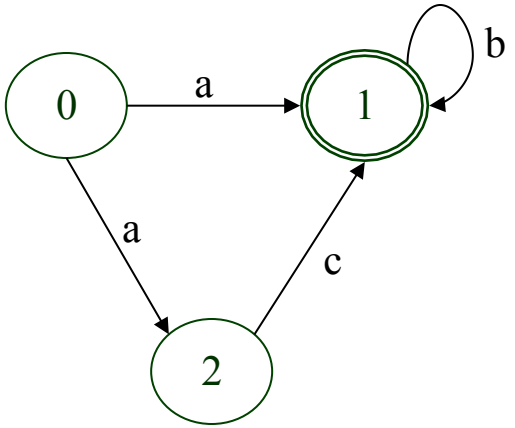

b
{0}
{1}
c
{2}
a
{0,1}
{1,2}
{0,2}
{0,1,2}

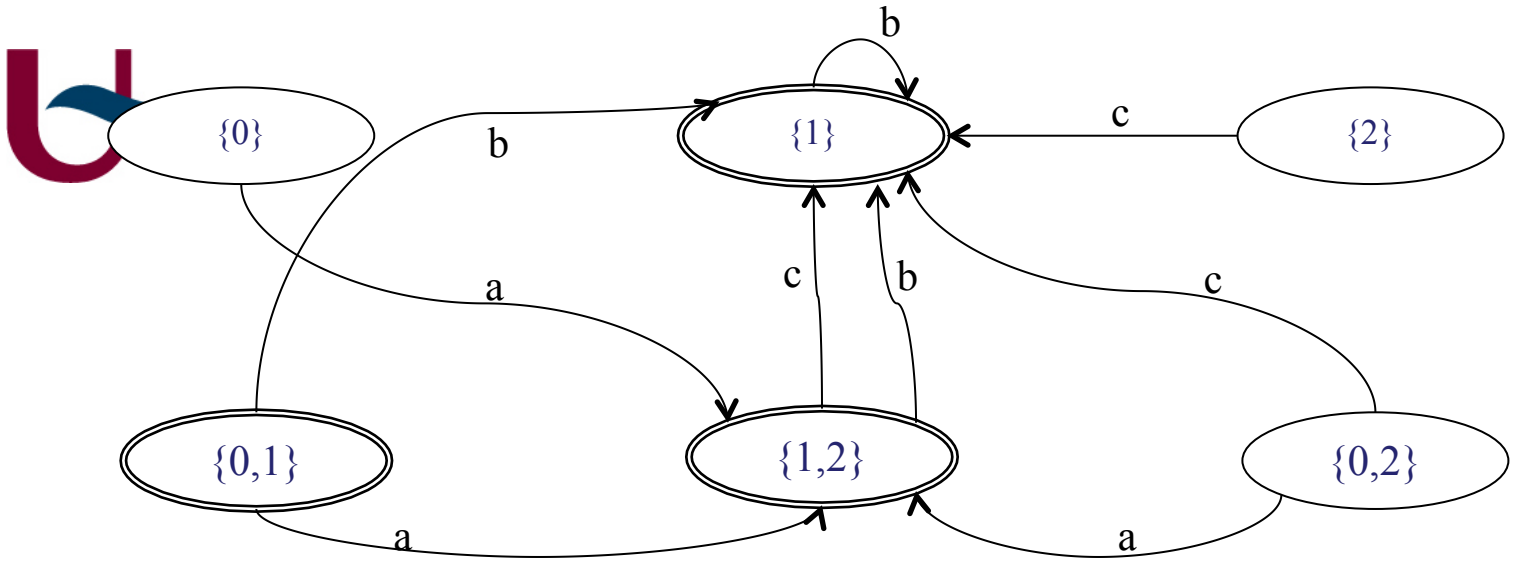

b
0
a
1
a
c
2

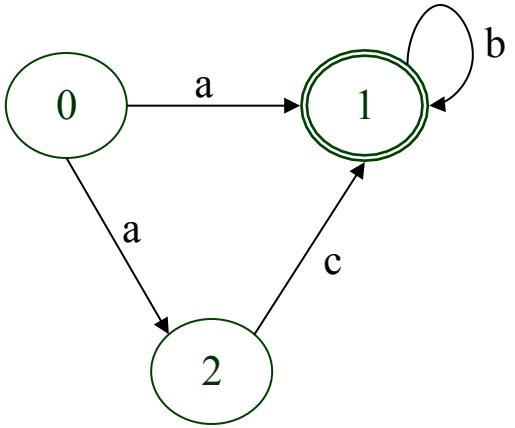

b
{0}
b
{1}
c
{2}
a
{0,1}
{1,2}
{0,2}
a
{0,1,2}

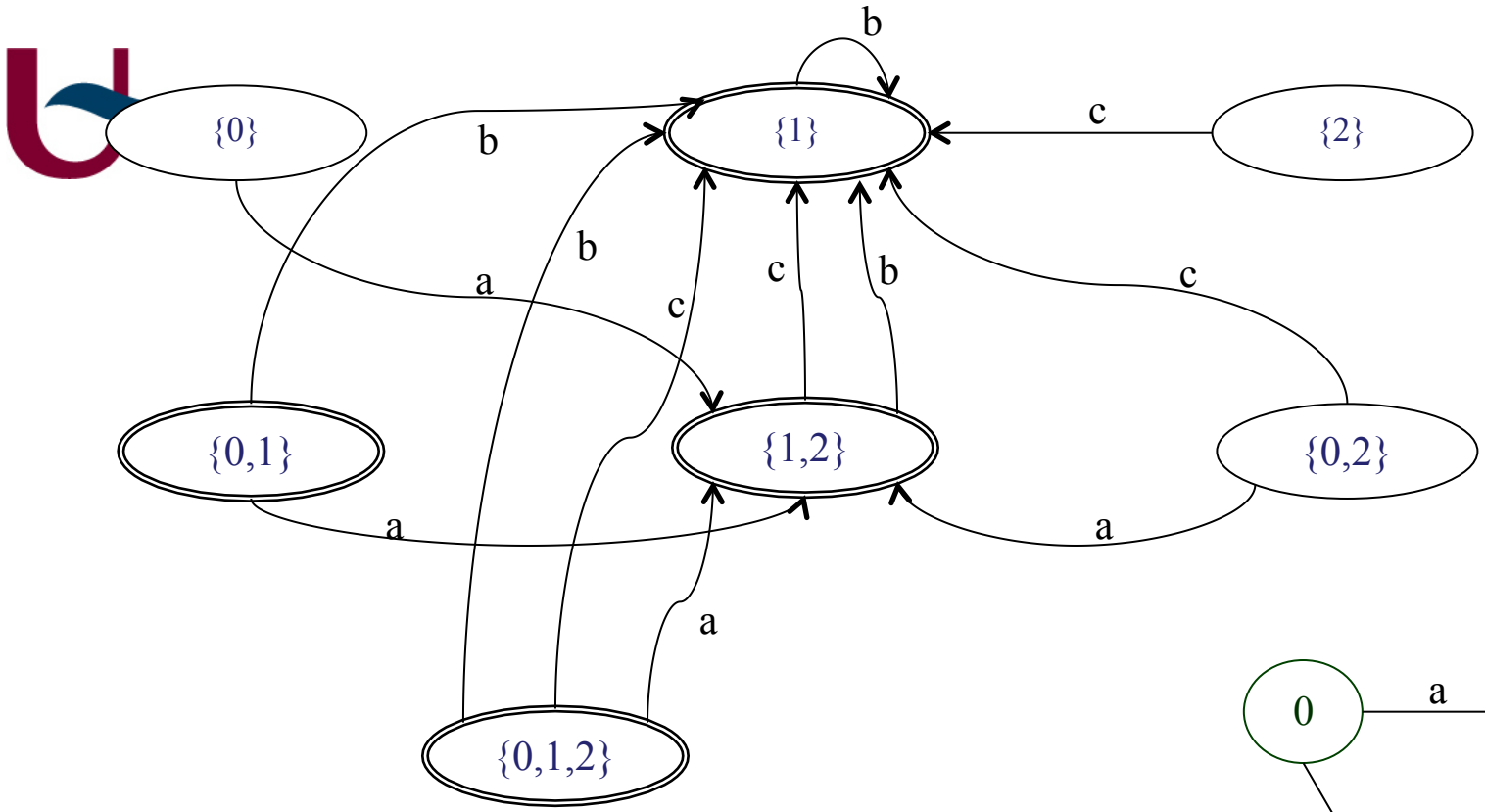

b
0
a
1
a
c
2

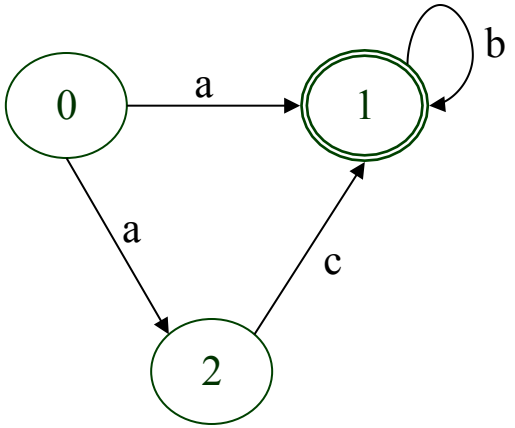

{0}
{1}
{2}
{0,1}
{1,2}
{0,2}
{0,1,2}
a
b
c

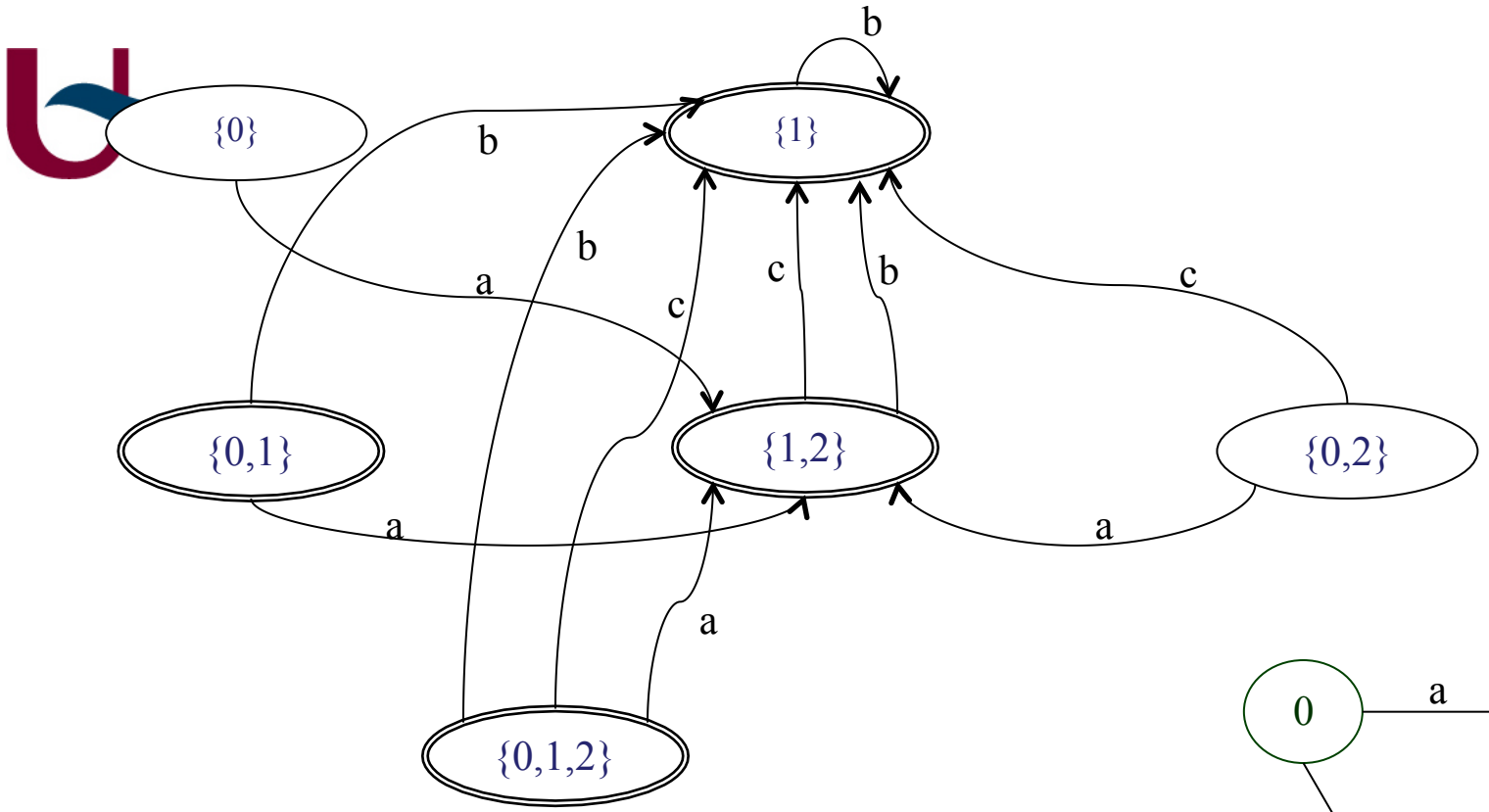

0
1
2
a
b
c

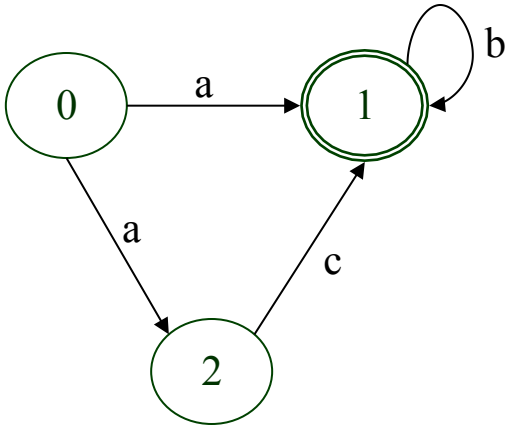

{0}
{1}
{2}
{0,1}
{1,2}
{0,2}
{0,1,2}
a
b
c

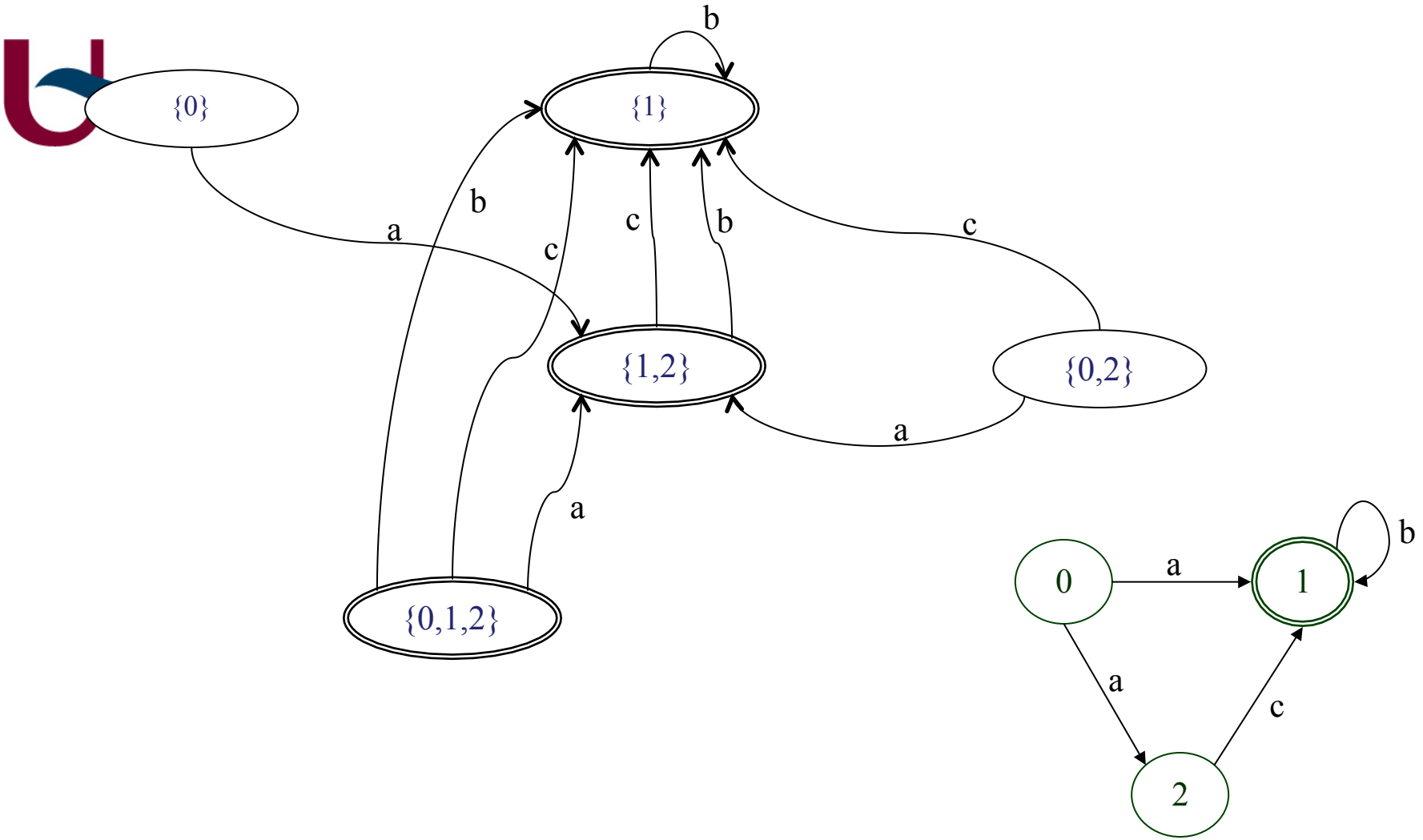

0
1
2
a
b
c

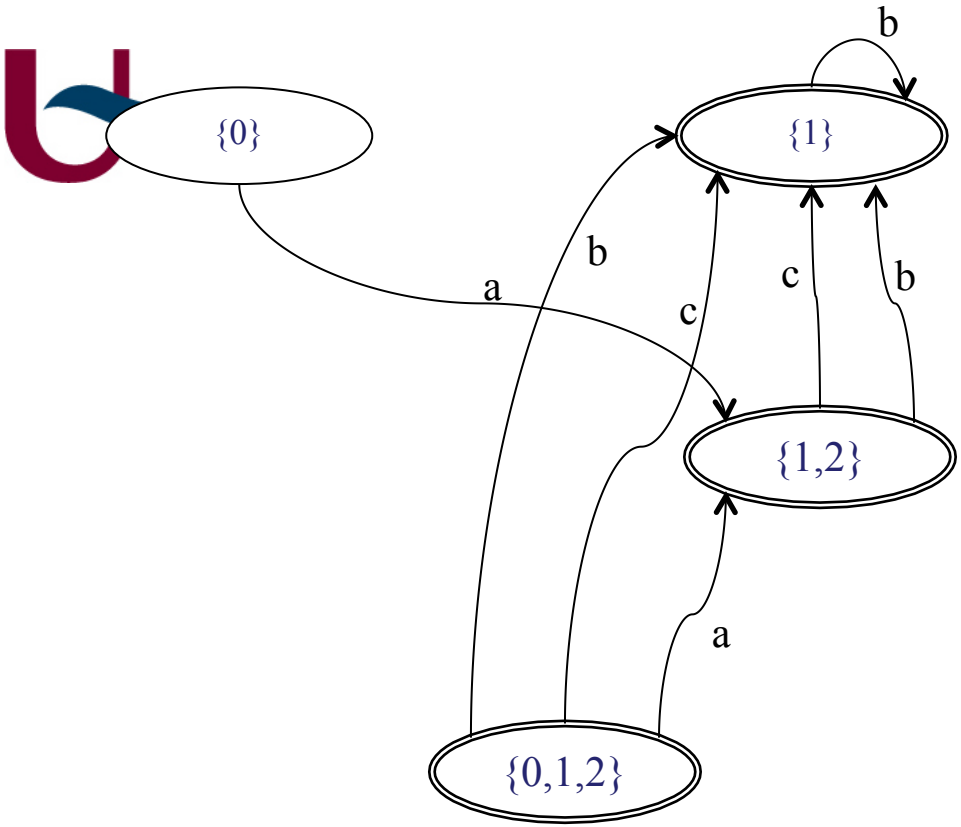

{0}
{1}
{2}
{0,1}
{1,2}
{0,2}
{0,1,2}
a
b
c

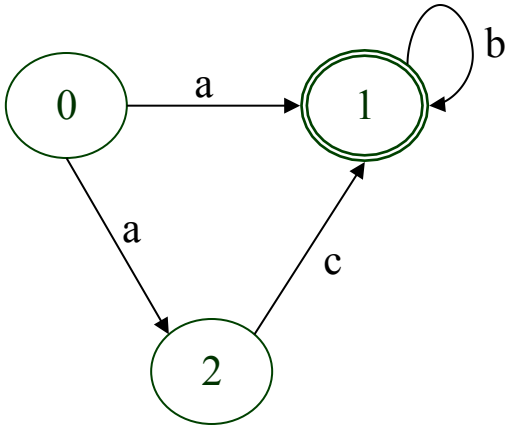

0
1
2
a
b
c

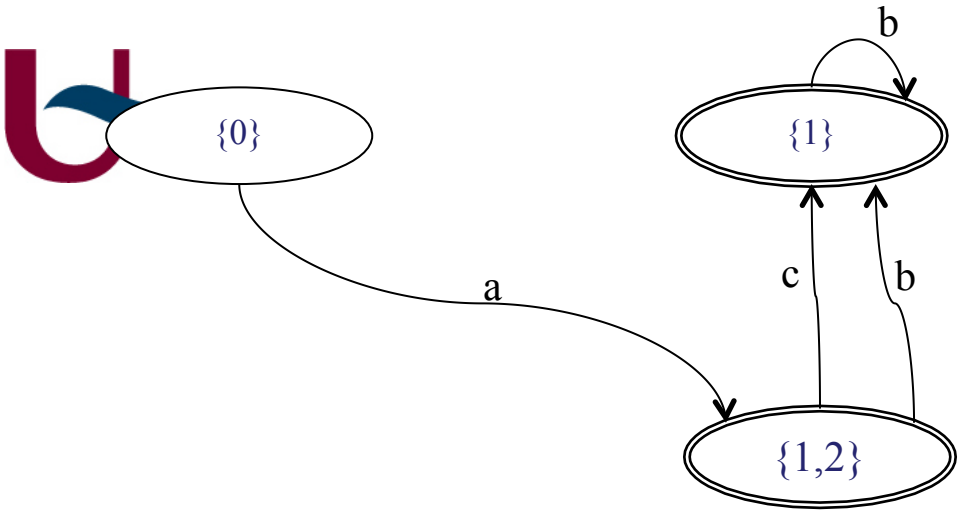

{0}
{1}
{0,1}
{1,2}
{0,2}
{0,1,2}
a
b
c

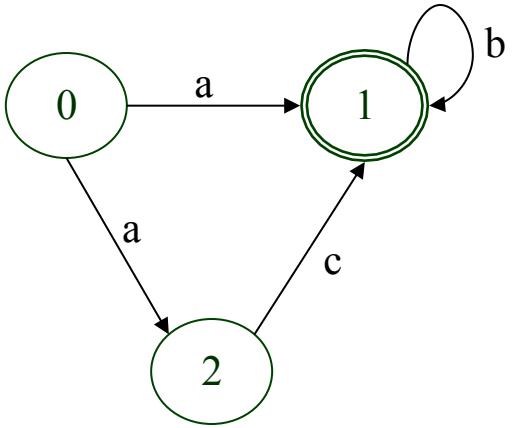

0
1
2
a
b
c


{0}
{1}
{1,2}
{0,2}
{0,1,2}
a
b
c
0
1
2

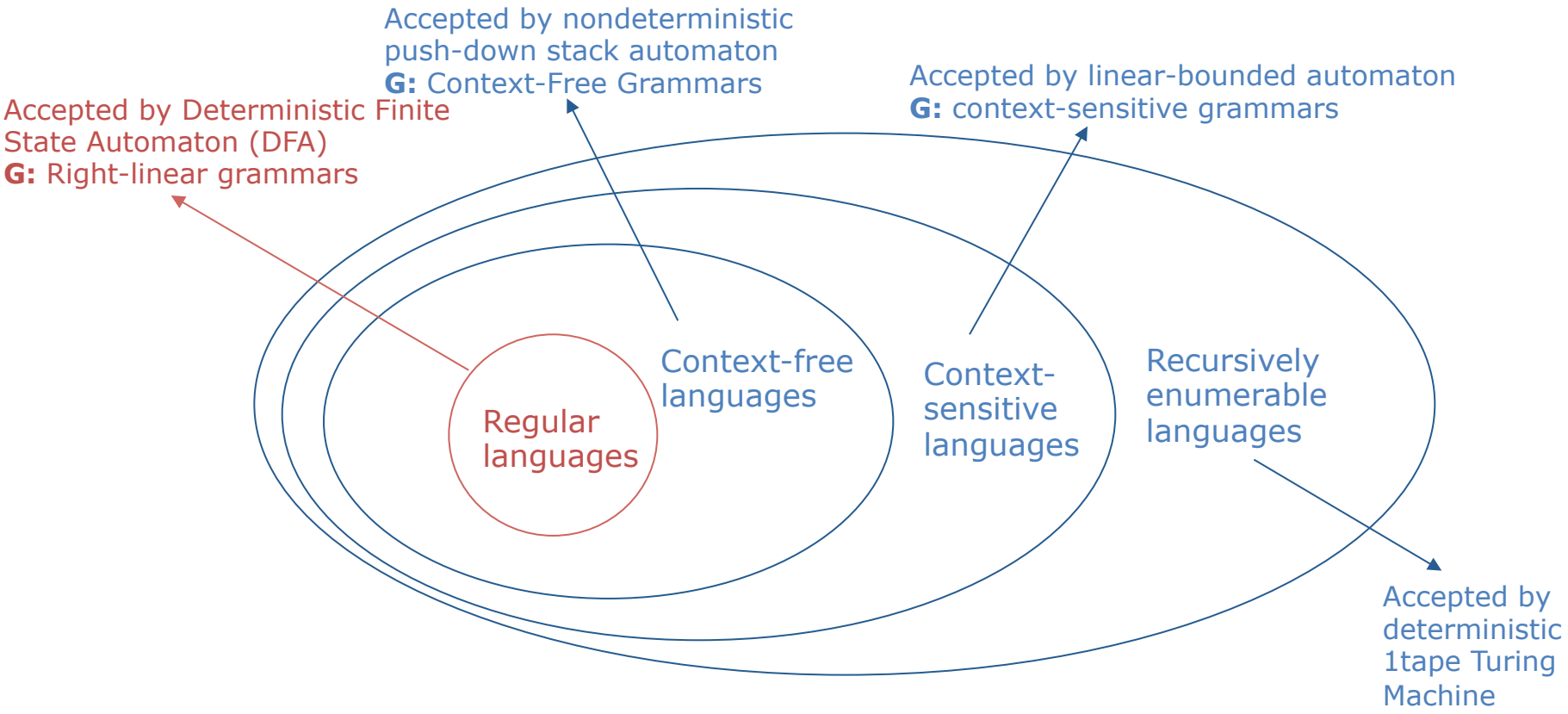

{0}
{1}
{1,2}
{0,1,2}
a
b
c
b
c
b
a


0
1
2
a
a
b
c


{0}
{1}
{1,2}
a
b
c
b

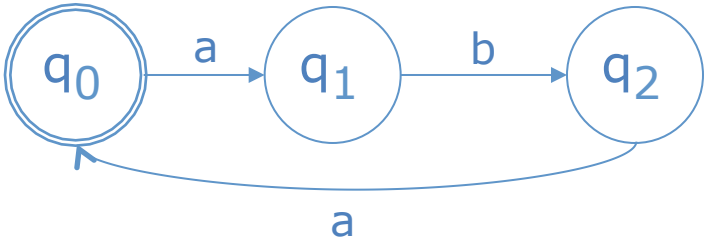

0
1
2
a
a
b
c

The Chomsky Hierarchy



Regular Languages

- Regular languages are recognized/generated by
  - Regular expressions /(aba)*/
  - Finite-state automata



  - Right-linear grammars
    - every state = non-terminal
    - every arc = terminal

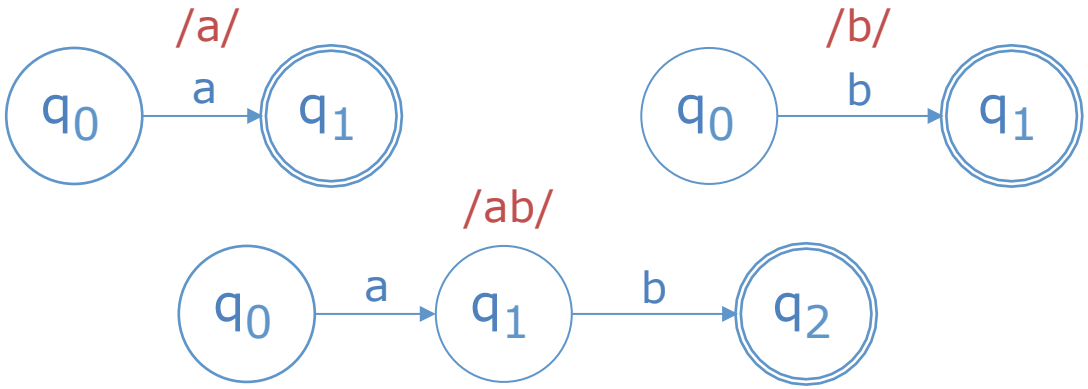
q0 → *a*q1
q1 → *b*q2
q2 → *a*q1

Regular Languages

- the empty language is a regular language
- A single symbol is a regular language
- If you have two regular languages L1 and L2
  - The concatenation of L1 and L2 is a regular language
  - The union of L1 and L2 is a regular language
  - The Kleene closure of L (L*) is a regular language

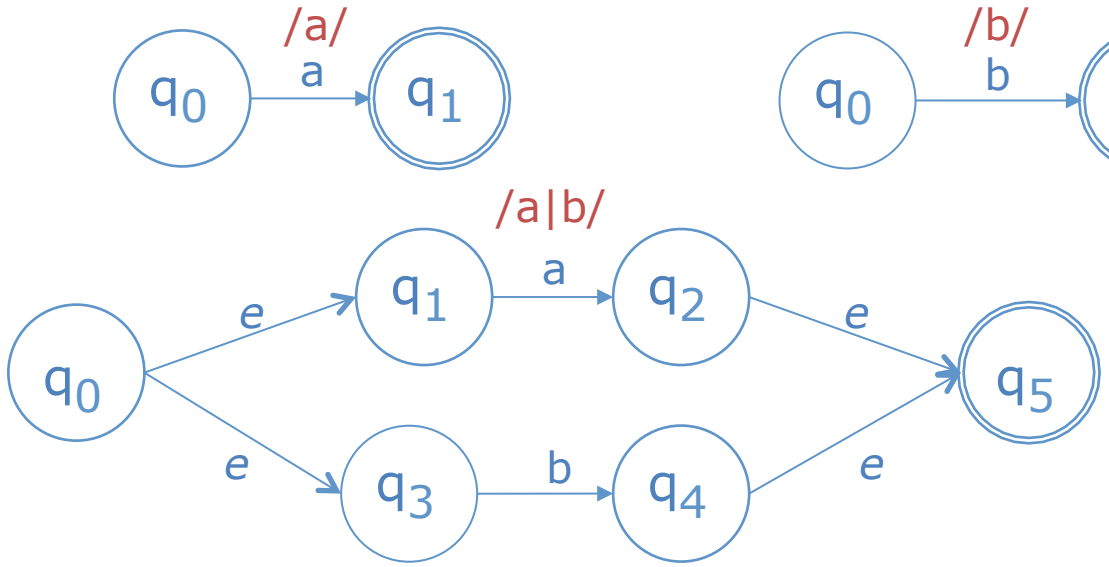
Regular Languages

- the empty language is a regular language
- A single symbol is a regular language
- If you have two regular languages L1 and L2
  - **The concatenation of L1 and L2 is a regular language**
  - The union of L1 and L2 is a regular language
  - The Kleene closure of L (L*) is a regular language



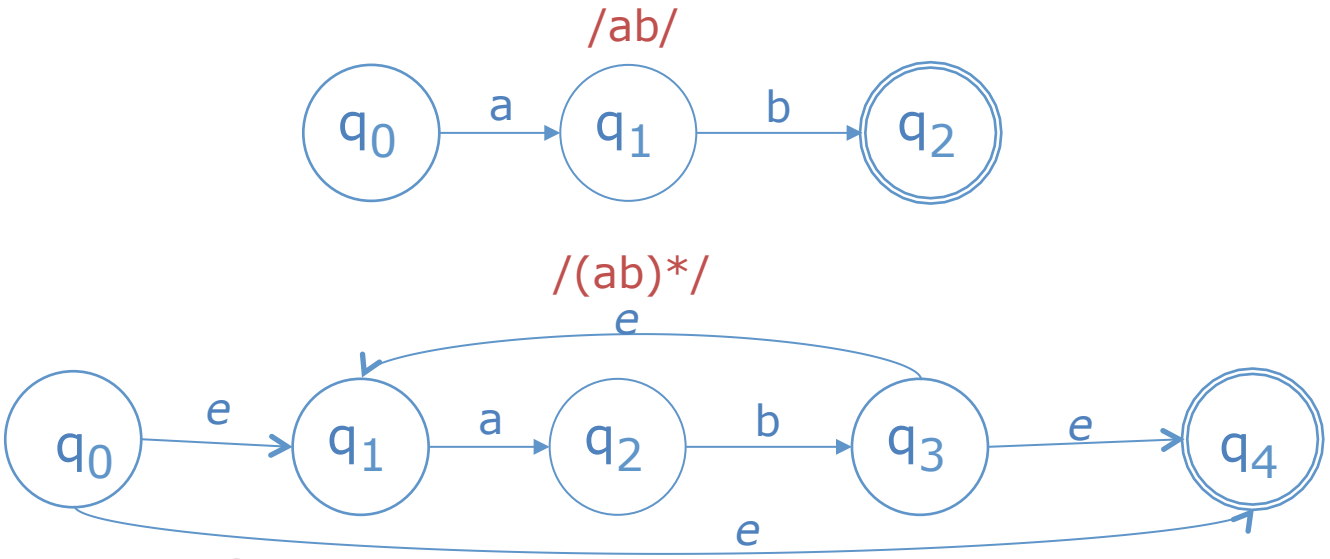
Regular Languages

- the empty language is a regular language
- A single symbol is a regular language
- If you have two regular languages L1 and L2
  - The concatenation of L1 and L2 is a regular language
  - **The union of L1 and L2 is a regular language**
  - The Kleene closure of L (L*) is a regular language



Regular Languages

- the empty language is a regular language
- A single symbol is a regular language
- If you have two regular languages L1 and L2
  - The concatenation of L1 and L2 is a regular language
  - The union of L1 and L2 is a regular language
  - **The Kleene closure of L (L*) is a regular language**



Extra Exercises (not mandatory)

- Do exercises on http://regex.sketchengine.co.uk/

Chapter 3

Morphology and Finite State Transducers

What is Morphology?

- The study of how words are composed of **morphemes**
  - **Morpheme**: *smallest meaning-bearing unit* in a language
  - **Stems**: core meaning units in a lexicon
  - **Affixes**: bits and pieces that combine with stems to modify their meanings and grammatical functions
    - Prefix — Immaterial — = in + material
    - Suffix — Trying — = try + ing
    - Infix — Absofuckinglutely — = absolutely + fucking
    - Circumfix — Unreadable — = un + read + able
    - Circumfix — Gedeeld — = ge + deel + d
  - **Allomorphs**: *inactive* vs *impossible*
    in + active *in + possible*

Two Broad Classes of Morphology

- Inflectional Morphology
  - Combination of stem and morpheme resulting in word of same class
  - Usually fills a syntactic feature (e.g. agreement)
  - E.g., plural –s, past tense –ed

- Derivational Morphology
  - Combination of stem and morpheme usually results in a word of a different class

    e.g., +ation in words such as computerization
  - Meaning of the new word may be hard to predict

    e.g. impossible, invaluable, inappropriate, infamous

Why is Morphology Important to the Lexicon?

Full Listing *vs* Minimal Redundancy

- true, truer, truest, truly, untrue, truth, truthful, truthfully, untruthfully, untruthfulness
- Untruthfulness = un- + true + -th + -ful + -ness
- These morphemes appear to be productive
- By representing knowledge about the **internal structure of words** and the **rules of word formation**, we can save room and search time.

Why is Morphology Important to the Lexicon?

Uygarlastiramayabileceklerimizdenmissinizcesine

urgar/civilized las/BECOME tir/CAUS ama/NEG yabil/POT ecek/FUT ler/3PL imiz/POSS-1SG den/ABL mis/NARR siniz/2PL cesine/AS-IF

Adverb meaning "(behaving) as if you were one of those whom we might not be able to civilize"

[Turkish, from Oflazer & Guzey 1994]

Morphological Parsing / Stemming

- Taking a surface input and breaking it down into its morphemes
- *foxes* breaks down into the morphemes *fox* (noun stem) and –es (plural suffix)
- *rewrites* breaks down into re- (prefix) and *write* (stem) and –s (suffix)

Parsing

- Taking a surface input and identifying its components and underlying structure
- Morphological parsing: parsing a word into its stem and affixes and identifying the parts and their relationships
  - Stem and **features**:
    - goose → goose +N +SG or goose +V
    - geese → goose +N +PL
    - gooses → goose +V +3SG
  - Bracketing: indecipherable → [in [[de [cipher]] able]]

Why parse words?

- For spell-checking
  - Is muncheble a legal word?
- To identify a word's part-of-speech (pos)
  - For **sentence parsing**, for **machine translation**, …
- To identify a word's stem
  - For **information retrieval**

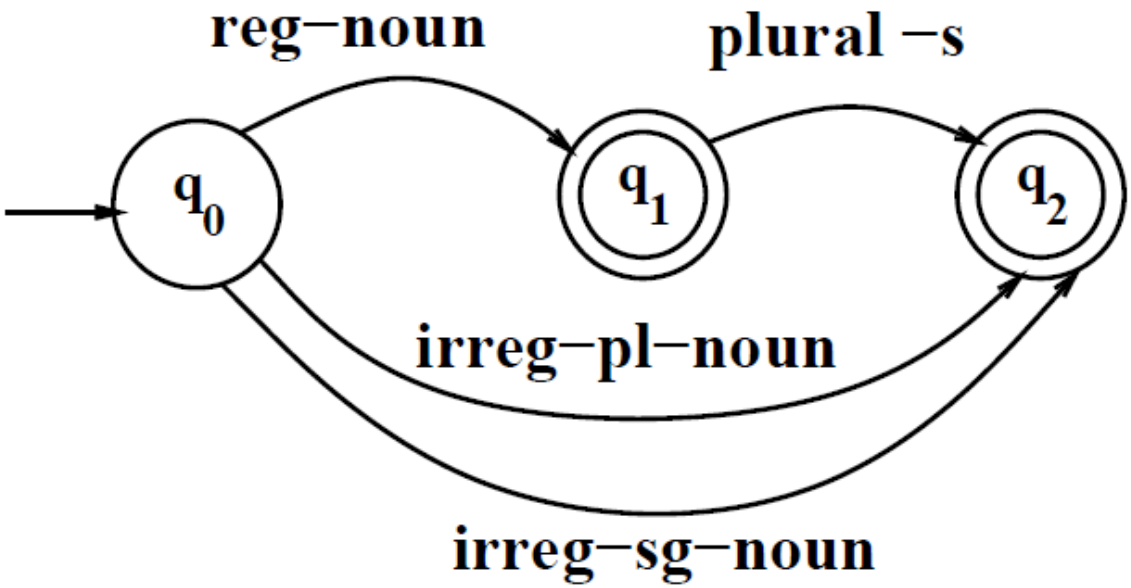
What do we need to build a morphological parser?

- **Lexicon**: stems and affixes (with corresponding class)
- **Morphotactics** of the language: model of how morphemes can be affixed to a stem. e.g., plural morpheme follows noun in English
- **Orthographic rules**: spelling modifications that occur when affixation occurs
  - in → il in context of l (in- + legal)

- **Finite State Transducers**

Morphotactic Models

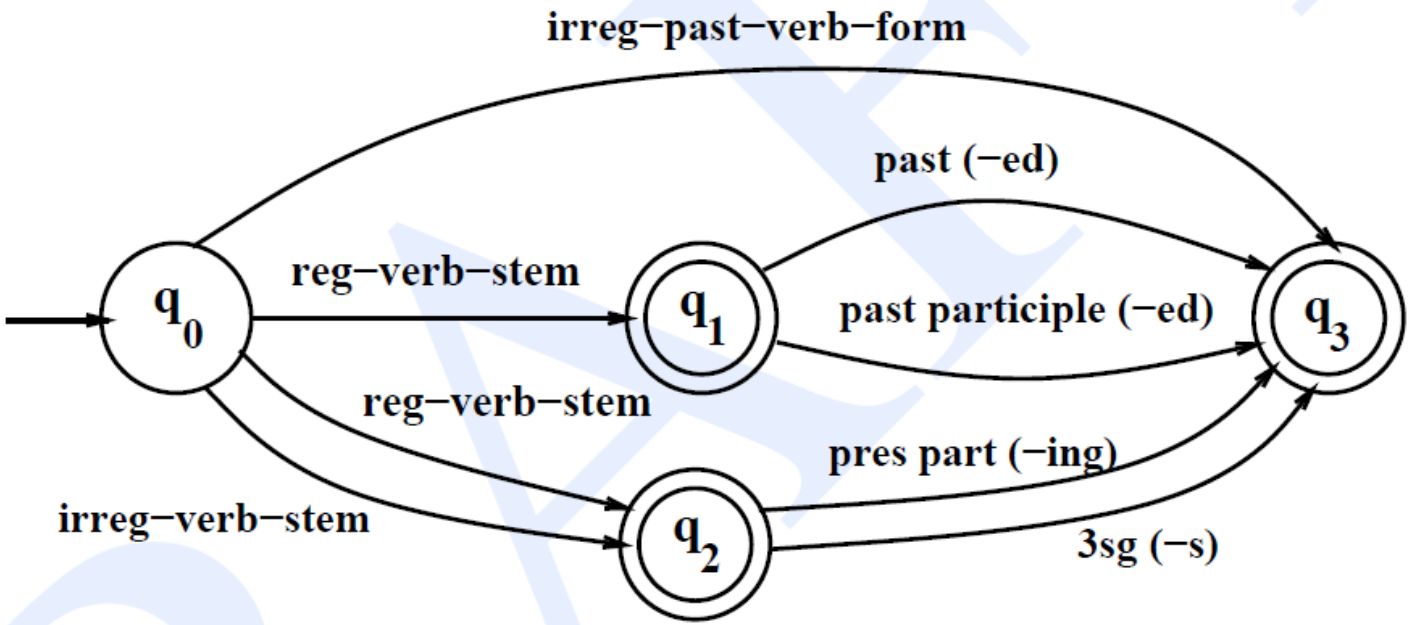
- English nominal inflection



- Inputs: cats, goose, geese

Morphotactic Models

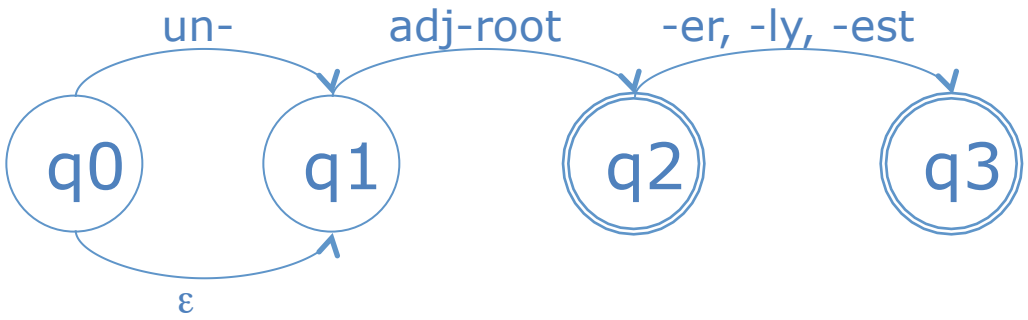
- English verbal inflection



- Inputs: look, looks, looked, looking, wrote, write, writes, writing

Morphotactic Models

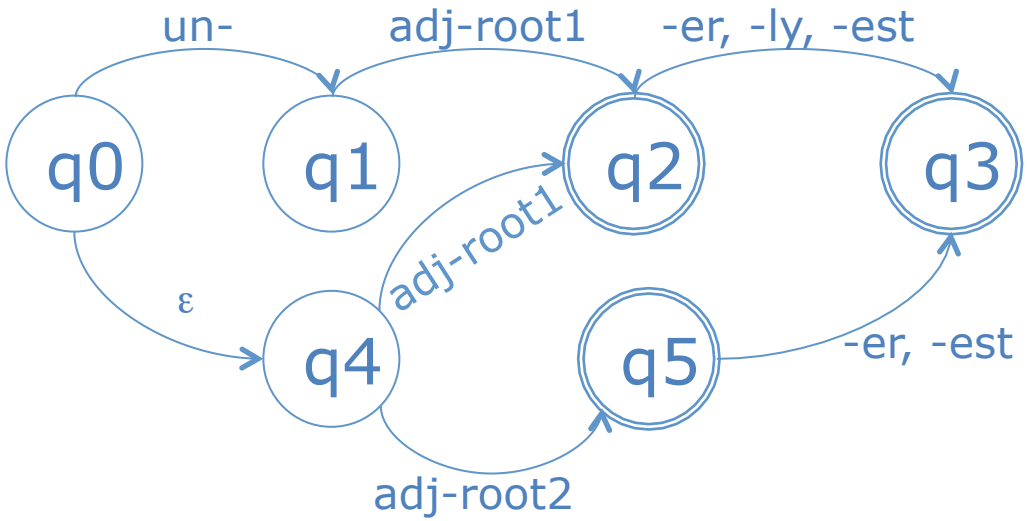
- Derivational morphology: adjective fragment



- adj-root: clear, happy, real, big, red
- BUT: *unbig, *redly, *realest

Morphotactic Models

- Derivational morphology: adjective fragment



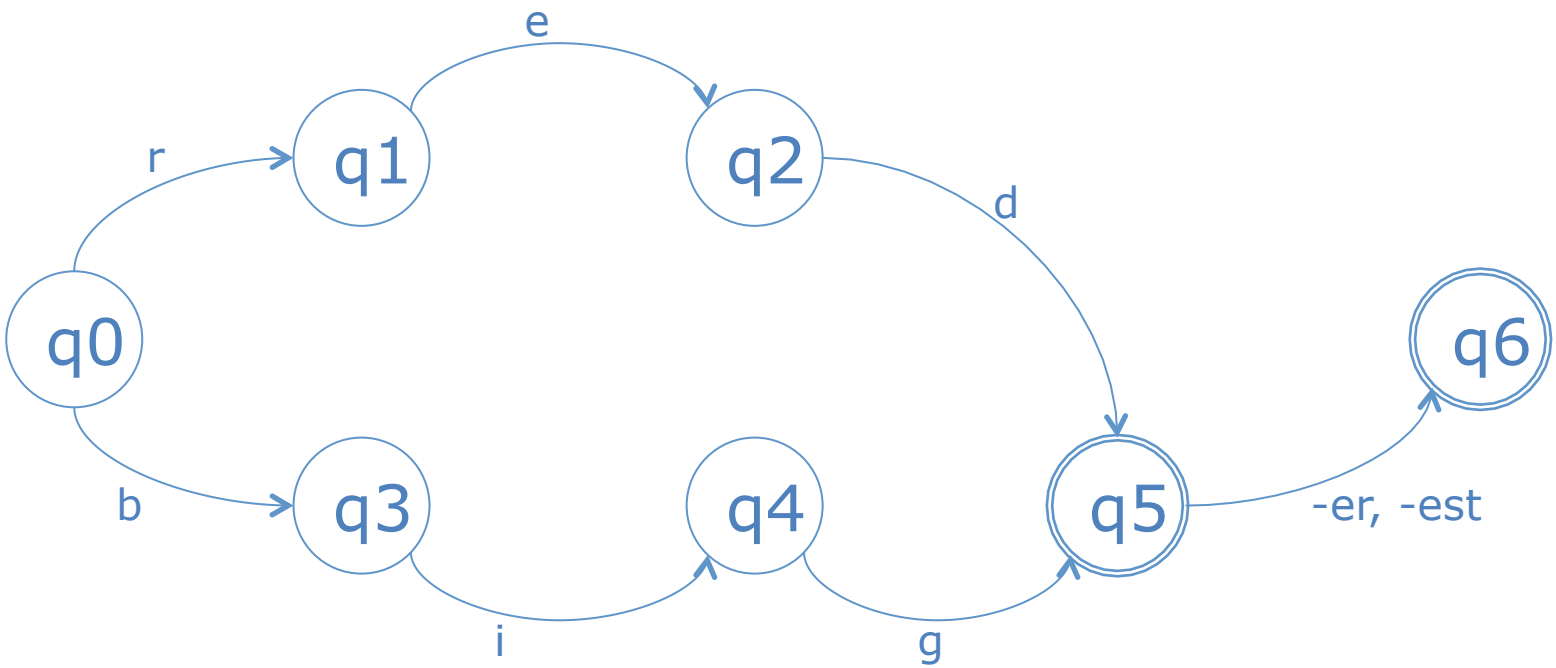
- Adj-root1: clear, happy, real
- Adj-root2: big, red

FSAs and the Lexicon

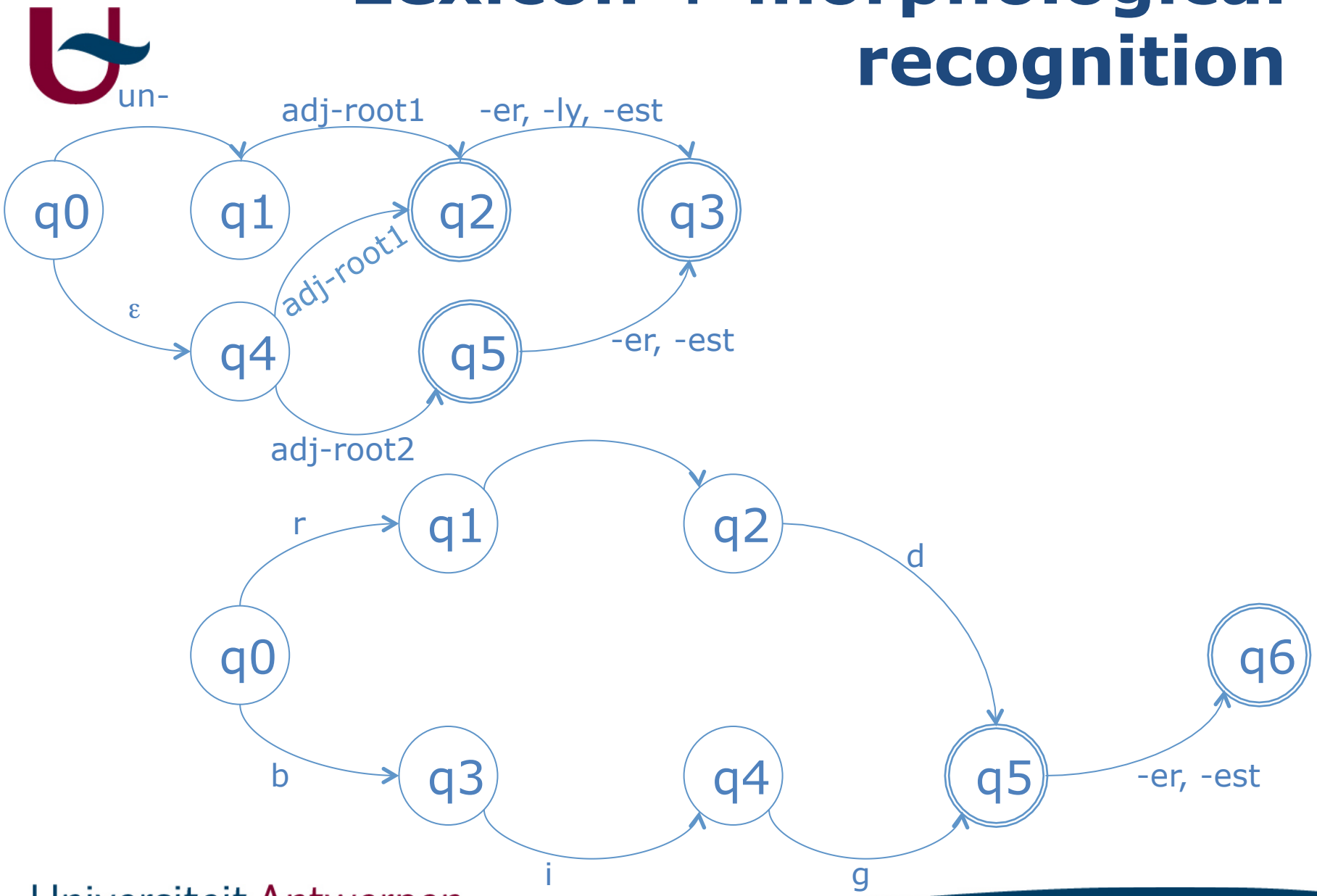
- First we capture the morphotactics
  - The rules governing the ordering of affixes in a language.
- Then we add in the actual words as classes or

- we can expand each non-terminal in our FSA into each stem in its class (e.g. adj_root_2 = {big, red}) and expand each such stem to the letters it includes (e.g. red → r e d, big → b i g)

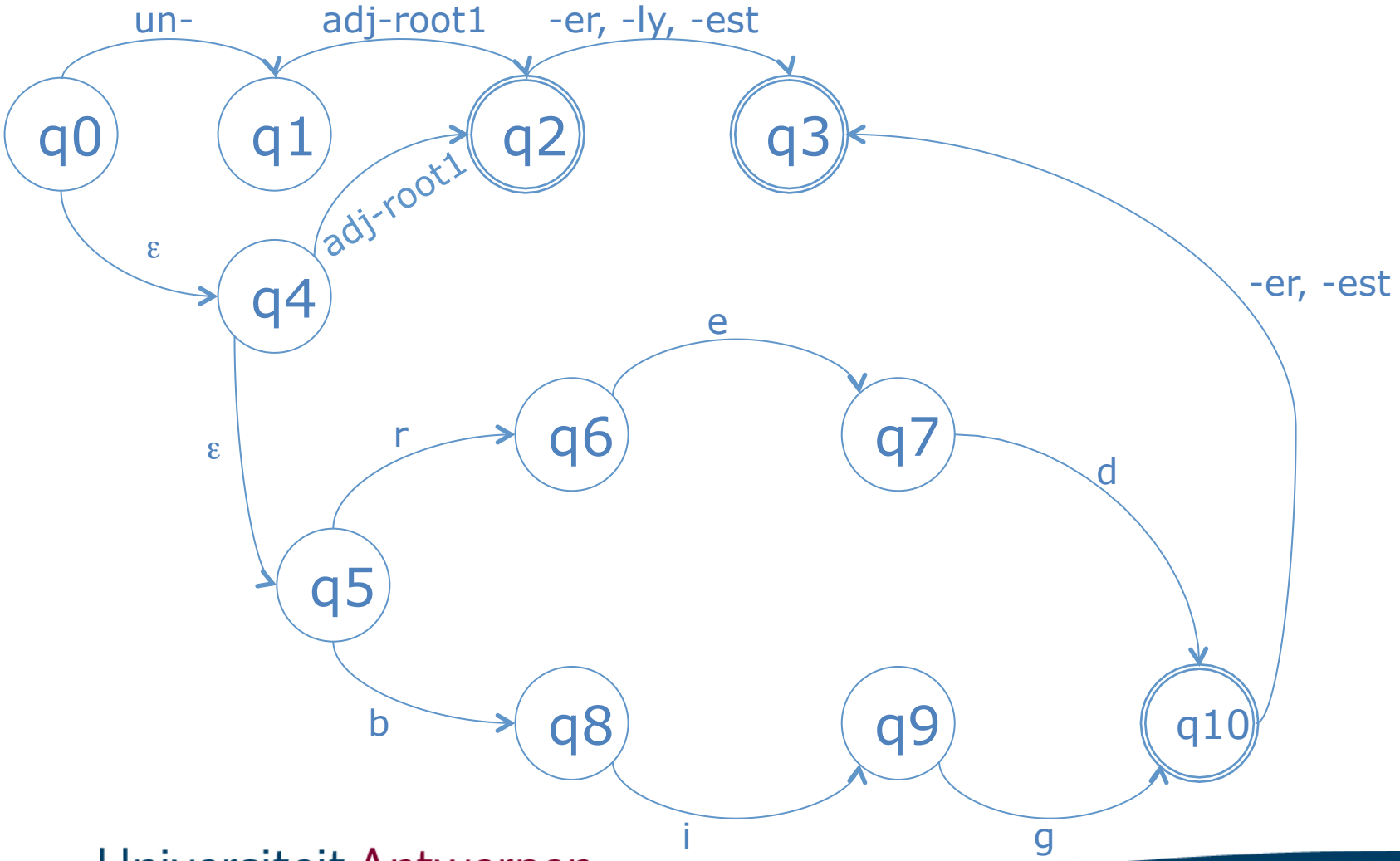
Lexicon + morphological recognition



Lexicon + morphological recognition



Lexicon + morphological recognition



Parsing/Generation vs. Recognition

- Recognition is usually not quite what we need.
  - FSA can only accept/generate strings
  - Usually if we find some string in the language we need to find the structure in it (parsing)
  - Example

    From "`cats`" to "`cat +N +PL`"

Finite State Transducers

- The simple story
  - Add another layer in our FSA
  - Add extra symbols to the transitions

  - On one tape we read "`cats`", on the other we write "`cat +N +PL`"

Parsing with Finite State Transducers

- cats → cat +N +PL
- Kimmo Koskenniemi's two-level morphology
  - Words represented as correspondences between **lexical** level (the morphemes) and **surface** level (the orthographic word)
  - Morphological parsing :building **mappings** between the lexical and surface levels

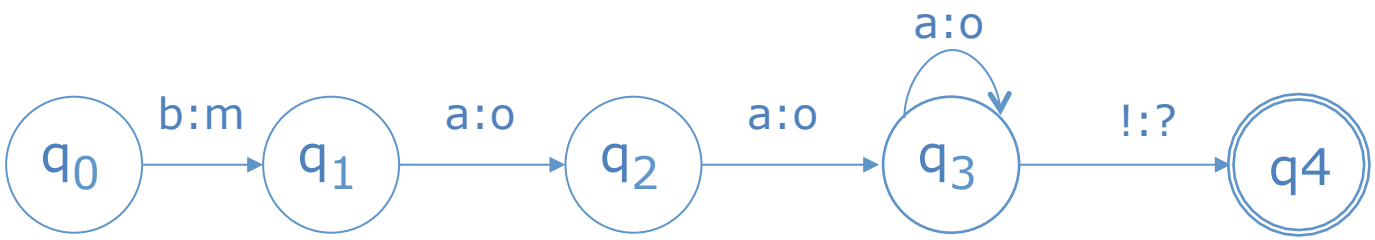
	c	a	t	s		
	c	a	t	+N	+PL	

Finite State Transducers

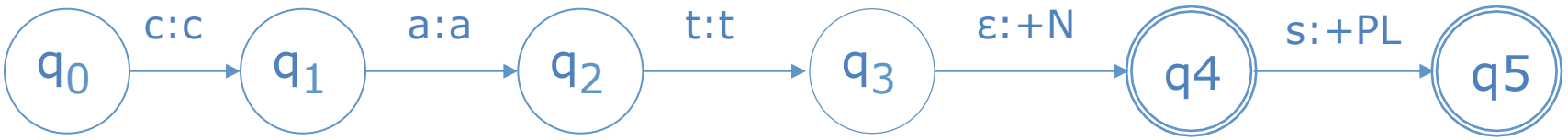
- FSTs map between one set of symbols and another using an FSA whose alphabet Σ is composed of pairs of symbols from input and output alphabets
- In general, FSTs can be used for
  - Translation (Hello: Ciao)
  - Parsing (cats: cat+N+PL)
  - Generation (Hello: How may I help you?)
  - To map between the lexical and surface levels of Kimmo's 2-level morphology

Finite State Transducers

- FST is a 5-tuple consisting of
  - Q: set of states {q0,q1,q2,q3,q4}
  - Σ: an alphabet of complex symbols, each an i/o pair s.t. i ∈ I (an input alphabet) and o ∈ O (an output alphabet) and Σ is in I x O
  - q0: a start state
  - F: a set of final states in Q {q4}
  - δ(q,i:o): a transition function mapping Q x Σ to Q
  - **Emphatic Sheep → Quizzical Cow**



Transitions



- c:c means read a c on one tape and write a c on the other
- ε:+N means read nothing on one tape and write +N on the other
- s:+PL means read s and write +PL

Summing Up

- FSTs provide a useful tool for implementing a standard model of morphological analysis, Kimmo's two-level morphology
  - Key is to provide an FST for each of multiple levels of representation and then to combine those FSTs using a variety of operators (cf AT&T FSM Toolkit)
  - Other (older) approaches are still widely used, e.g. the rule-based Porter Stemmer

Regular Expressions & Python

```
>>> import re
>>> wordlist = ['cat', 'cats', 'bat', 'bats', 'dog', 'dogs', 'sheep', 'child', 'children',
'shop', 'shops', 'rental', 'rentals']
>>> for word in wordlist:
...         print(word)
...
cat
cats
bat
bats
dog
dogs
sheep
child
children
shop
shops
rental
rentals
```

Regular Expressions & Python

```
>>> for word in wordlist:
...         if re.search('s',word):
...                 print(word+"[plural]")
...         else:
...                 print(word+"[singular]")
...
cat[singular]
cats[plural]
bat[singular]
bats[plural]
dog[singular]
dogs[plural]
sheep[plural]
child[singular]
children[singular]
shop[plural]
shops[plural]
rental[singular]
rentals[plural]
```

Regular Expressions & Python

```
>>> regex = r's'
>>> for word in wordlist:
...         if re.search(regex,word):
...                 print(word+"[plural]")
...         else:
...                 print(word+"[singular]")
...
cat[singular]
cats[plural]
bat[singular]
bats[plural]
dog[singular]
dogs[plural]
sheep[plural]
child[singular]
children[singular]
shop[plural]
shops[plural]
rental[singular]
rentals[plural]
```

Regular Expressions & Python

```
>>> regex = r's'
>>> for word in wordlist:
...         if re.search(regex,word):
...                 print(word+"[plural]")
...         else:
...                 print(word+"[singular]")
...
cat[singular]
cats[plural]
bat[singular]
bats[plural]
dog[singular]
dogs[plural]
sheep[plural]
child[singular]
children[singular]
shop[plural]
shops[plural]
rental[singular]
rentals[plural]
```

```
>>> def singplur(list,regex):
...         for word in list:
...                 if re.search(regex,word):
...                         print(word+"[plural]")
...                 else:
...                         print(word+"[singular]")
...
>>> singplur(wordlist,regex)
cat[singular]
cats[plural]
bat[singular]
bats[plural]
dog[singular]
dogs[plural]
sheep[plural]
child[singular]
children[singular]
shop[plural]
shops[plural]
rental[singular]
rentals[plural]
```

Regular Expressions & Python

Adjust the *regex* variable to get the correct output. You might also need to adjust the *singplur* function.

```
>>> def singplur(list,regex):
...         for word in list:
...                 if re.search(regex,word):
...                         print(word+"[plural]")
...                 else:
...                         print(word+"[singular]")
...
>>> singplur(wordlist,regex)
cat[singular]
cats[plural]
bat[singular]
bats[plural]
dog[singular]
dogs[plural]
sheep[plural]
child[singular]
children[singular]
shop[plural]
shops[plural]
rental[singular]
rentals[plural]
```

Regular Expressions & Python

Adjust the *regex* variable to get the correct output. You might also need to adjust the *singplur* function.

```
>>> regex = r'[s(ren)]$'
>>> def singplur(list,regex):
...         for word in list:
...                 if re.search('^sheep$',word):
...                         print(word+"[singular/plural]")
...                 elif re.search(regex,word):
...                         print(word+"[plural]")
...                 else:
...                         print(word+"[singular]")
...
>>> singplur(wordlist,regex)
cat[singular]
cats[plural]
bat[singular]
bats[plural]
dog[singular]
dogs[plural]
sheep[singular/plural]
child[singular]
children[plural]
shop[singular]
shops[plural]
rental[singular]
rentals[plural]
```

Regular Expressions & Python

re.search returns a true/false value for matching a regular expression **[FSA]**

re.findall returns a list with all matches for a regular expression

```
>>> word = 'supercalifragilisticexpialidocious'
>>> re.findall(r'[aeiou]', word)
['u', 'e', 'a', 'i', 'a', 'i', 'i', 'i', 'e', 'i', 'a', 'i', 'o', 'i', 'o', 'u']
```

re.sub takes 3 arguments: **[FST]**

argument 3: the string you want to process

argument 1: the regular expression that denotes the pattern you want to match

argument 2: the expression you want to replace it with

```
>>> re.sub(r'([aeiou]+)', r'<vowel=\1>', 'waterboarding')
'w<vowel=a>t<vowel=e>rb<vowel=oa>rd<vowel=i>ng'
```

Sentence Tokenization

Also check NLTK's Regular Expression Tokenizer (Ch3.7 in NLTK book)

re.split returns a list of items of a string, split according to a regular expression:

```
>>> raw = """'When I'M a Duchess,' she said to herself, (not in a very hopeful tone
... ... though), 'I won't have any pepper in my kitchen AT ALL. Soup does very
... ... well without--Maybe it's always pepper that makes people hot-tempered,'..."""
>>> re.split(r' ', raw)
["'When", "I'M", 'a', "Duchess,'", 'she', 'said', 'to', 'herself,', '(not', 'in', 'a', 'very', 'hopeful',
'tone\n...', 'though),', "'I", "won't", 'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL.',
'Soup', 'does', 'very\n...', 'well', 'without--Maybe', "it's", 'always', 'pepper', 'that', 'makes',
'people', "hot-tempered,'..."]
```

\t = tab \n = newline

```
>>> re.split(r'[ \t\n]+', raw)
["'When", "I'M", 'a', "Duchess,'", 'she', 'said', 'to', 'herself,', '(not', 'in', 'a', 'very', 'hopeful',
'tone', 'though),', "'I", "won't", 'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL.', 'Soup',
'does', 'very', 'well', 'without--Maybe', "it's", 'always', 'pepper', 'that', 'makes', 'people', "hot-
tempered,'..."]
```

\W = [^A-Za-z0-9_] \w =[A-Za-z0-9_]

```
>>> re.split(r'\W+', raw)
['', 'When', 'I', 'M', 'a', 'Duchess', 'she', 'said', 'to', 'herself', 'not', 'in', 'a', 'very', 'hopeful',
'tone', 'though', 'I', 'won', 't', 'have', 'any', 'pepper', 'in', 'my', 'kitchen', 'AT', 'ALL', 'Soup',
'does', 'very', 'well', 'without', 'Maybe', 'it', 's', 'always', 'pepper', 'that', 'makes', 'people', 'hot',
'tempered', '']
```

Regular Expressions & Python

Special symbols for regex in Python:

\b	Word boundary (zero width)
\d	Any decimal digit (equivalent to [0-9])
\D	Any non-digit character (equivalent to [^0-9])
\s	Any whitespace character (equivalent to [\t\n\r\f\v]
\S	Any non-whitespace character (equivalent to [^ \t\n\r\f\v])
\w	Any alphanumeric character (equivalent to [a-zA-Z0-9_])
\W	Any non-alphanumeric character (equivalent to [^a-zA-Z0-9_])
\t	The tab character
\n	The newline character

Word Segmentation

```
>>> suffixes = ['ly','able','ation']
>>> prefixes = ['un','in','re']
>>> words =
['unlikely','insurmountable','reorganization','incompletely','uncrushable','unworkable','resig
nation','inconsoleable','maintainable','unrecoverable']
>>> def segment(wordlist):
...         for word in wordlist:
...                 for suffix in suffixes:
...                         regex = r'('+suffix+')$'
...                         word = re.sub(regex,r'+\1',word)
...                 print(word)
...
>>> segment(words)
unlike+ly
insurmount+able
reorganiz+ation
incomplete+ly
uncrush+able
unwork+able
resign+ation
inconsole+able
maintain+able
unrecover+able
```

Extend this function so that also the prefixes are split off from the word

Word Segmentation

```
>>> suffixes = ['ly','able','ation']
>>> prefixes = ['un','in','re']
>>> words =
['unlikely','insurmountable','reorganization','incompletely','uncrushable','unworkable','resig
nation','inconsoleable','maintainable','unrecoverable']
>>> def segment(wordlist):
...         for word in wordlist:
...                 for suffix in suffixes:
...                         regex1 = r'('+suffix+')$'
...                         word = re.sub(regex1,r'+\1',word)
...                 for prefix in prefixes:
...                         regex2 = r'^('+prefix+')'
...                         word = re.sub(regex2,r'\1+',word)
...                 print(word)
>>> segment(words)
un+like+ly
in+surmount+able
re+organiz+ation
in+complete+ly
un+crush+able
un+work+able
re+sign+ation
in+console+able
maintain+able
un+recover+able
```

http://www.clips.uantwerpen.be/cl1415/morfsegment.py

Assignment

Download www.clips.ua.ac.be/cl1415/participles.py

Regular Expressions

1. Write a function that tokenizes the **text** and
2. returns a list of the past tense verbs in it
   - **Use a regular expression that DOES NOT just enumerate these verbs (although do take care of irregular verbs)!!!**
3. the past tense verbs should be morphologically decomposed to the best of your ability, e.g. 'strand+ed'
   'caught' or
   'catch+[past]'
   'decompos +ed' or
   'decompose +ed' or
   'de+compose+ed' or
   'de[prefix]+compose[root]+ed[past]'

DEADLINE: 24 November 2014

Send python code through e-mail to guy.depauw@uantwerpen.be

Don't hesitate to contact your helpline guy.depauw@uantwerpen.be