Software-Defined Networking and Advanced Network Control Programming

4 Mathematical Tools

Kai Gao
kaigao@scu.edu.cn

School of Cyber Science and Engineering
Sichuan University

Recap

Representative SDN Controllers

Controller	*Representative Feature*

Representative SDN Controllers

Controller	*Representative Feature*
NOX	First SDN controller, network-level programmatic control

Representative SDN Controllers

Controller	*Representative Feature*
NOX	First SDN controller, network-level programmatic control
Beacon	Multi-threaded SDN controller optimized for parallel event handling

Representative SDN Controllers

Controller	*Representative Feature*
NOX	First SDN controller, network-level programmatic control
Beacon	Multi-threaded SDN controller optimized for parallel event handling
Onix	Distributed network information base with customizable consistency levels

Representative SDN Controllers

Controller	*Representative Feature*
NOX	First SDN controller, network-level programmatic control
Beacon	Multi-threaded SDN controller optimized for parallel event handling
Onix	Distributed network information base with customizable consistency levels
Kandoo	Hierarchical distributed controller

Representative SDN Controllers

Controller	*Representative Feature*
NOX	First SDN controller, network-level programmatic control
Beacon	Multi-threaded SDN controller optimized for parallel event handling
Onix	Distributed network information base with customizable consistency levels
Kandoo	Hierarchical distributed controller
ONOS	Intent-based networking

Representative SDN Controllers

Controller	*Representative Feature*
NOX	First SDN controller, network-level programmatic control
Beacon	Multi-threaded SDN controller optimized for parallel event handling
Onix	Distributed network information base with customizable consistency levels
Kandoo	Hierarchical distributed controller
ONOS	Intent-based networking
OpenDaylight	Model-driven networking, widely used in industry and research

Intent-based Networking

The idea of intents is borrowed from mobile systems (such as Android)

Users specify what they want to do instead of how to do it, and multiple service instances can co-exist to realize the intent.

Intents realize loosely coupled and dynamic service binding.

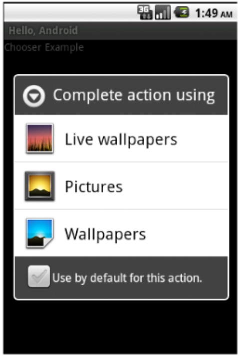


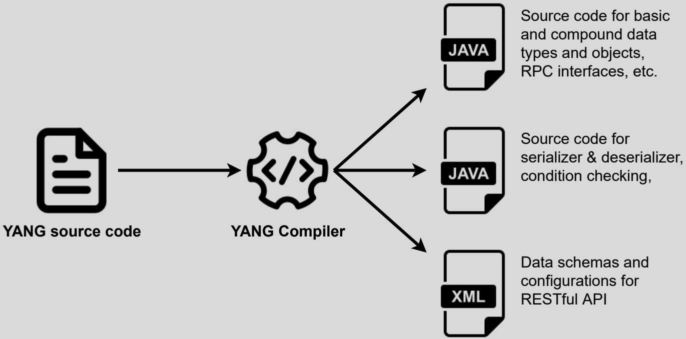
Figure 3: The user is prompted when an implicit Intent resolves to multiple Activities.

Erika Chin et al. “Analyzing Inter-Application Communication in Android”. In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services - MobiSys '11*. The 9th International Conference. Bethesda, Maryland, USA: ACM Press, 2011, p. 239. URL: http://portal.acm.org/citation.cfm?doid=1999995.2000018 (visited on 09/17/2021)

Model-driven Networking

MD-SAL provide an automation tool to handle the complexities of extending existing API or data models

With the model specification (e.g., the YANG modeling language), a compiler automatically generates source code and configuration files



YANG Language

YANG is *Yet Another Next Generation* modeling language for Network Configuration Protocol. It is first designed for state synchronization between devices and state storage but is also used for service layer abstraction.

```
Internet Engineering Task Force (IETF)                  M. Bjorklund, Ed.
Request for Comments: 6020                                 Tail-f Systems
Category: Standards Track                                    October 2010
ISSN: 2070-1721


            YANG - A Data Modeling Language for
       the Network Configuration Protocol (NETCONF)

Abstract

   YANG is a data modeling language used to model configuration and
   state data manipulated by the Network Configuration Protocol
   (NETCONF), NETCONF remote procedure calls, and NETCONF notifications.
```

```
module example1 {
  namespace "urn:examples:example1";
  prefix "example1";

  revision "2021-09-15" {
    description "Initial revision.";
  }

  typedef score {
    type uint8 {
      range "0..100";
    }
  }

  ...
}
```

Martin Björklund. *YANG - a Data Modeling Language for the Network Configuration Protocol (NETCONF).* RFC 6020. Oct. 2010. URL: https://rfc-editor.org/rfc/rfc6020.txt

本期学习目标

- 初步了解自动机理论：如何描述一个确定性有限状态自动机

本期学习目标

- 初步了解自动机理论：如何描述一个确定性有限状态自动机
- 通过示例了解状态机的使用

本期学习目标

- 初步了解自动机理论：如何描述一个确定性有限状态自动机
- 通过示例了解状态机的使用
- 掌握确定性有限状态自动机的乘积

Summary

In the coming lectures, we cover the following topics:

- Automata theory
- Linear programming
- Mixed integer linear programming
- Satisfactory theory

Summary

In the coming lectures, we cover the following topics:

- Automata theory
- Linear programming
- Mixed integer linear programming
- Satisfactory theory

In this lecture, you should

- see examples of automata theory

Summary

In the coming lectures, we cover the following topics:

- Automata theory
- Linear programming
- Mixed integer linear programming
- Satisfactory theory

In this lecture, you should

- see examples of automata theory
- get a basic sense of how to use automata to model the behaviors of a system/an algorithm

Summary

In the coming lectures, we cover the following topics:

- Automata theory
- Linear programming
- Mixed integer linear programming
- Satisfactory theory

In this lecture, you should

- see examples of automata theory
- get a basic sense of how to use automata to model the behaviors of a system/an algorithm
- understand deterministic finite automata

Summary

In the coming lectures, we cover the following topics:

- Automata theory
- Linear programming
- Mixed integer linear programming
- Satisfactory theory

In this lecture, you should

- see examples of automata theory
- get a basic sense of how to use automata to model the behaviors of a system/an algorithm
- understand deterministic finite automata
- understand how to compute the product automata

Automata Theory

Overview

Automata theory (自动机理论) is a foundation for many areas in computer science

- programming languages and compilers,

Overview

Automata theory (自动机理论) is a foundation for many areas in computer science

- programming languages and compilers,
- formal verification (形式化验证, e.g., protocol analysis),

Overview

Automata theory (自动机理论) is a foundation for many areas in computer science

- programming languages and compilers,
- formal verification (形式化验证, e.g., protocol analysis),
- other widely-used applications (e.g., regular expression/正则表达式)

Relation Between Automata and Languages

Different automatons (自动机) represents different languages

Relation Between Automata and Languages

Different automatons (自动机) represents different languages

Automaton	Language
finite state machine (NFA & DFA)	regular language
确定性/非确定性有限状态机	正则语言
pushdown automaton (PDA)	context-free language
下推自动机	上下文无关语言
linear-bounded automaton (LBA)	context-sensitive language
线性有界自动机	上下文有关语言
Turing machine	recursively enumerable language
图灵机	递归可枚举语言

Compilation of (Programming) Languages

An input (e.g., a program, an ASCII string) is valid in a language

$$\Updownarrow$$

The string can be accepted by the corresponding automaton

Example:

Compilation of (Programming) Languages

An input (e.g., a program, an ASCII string) is valid in a language

$$\Updownarrow$$

The string can be accepted by the corresponding automaton

Example:

If the syntax of a language is specified using context-free grammar (上下文无关文法), the parsing of any valid program can be realized using a Pushdown Automaton

Compilation of (Programming) Languages

An input (e.g., a program, an ASCII string) is valid in a language

$\Updownarrow$

The string can be accepted by the corresponding automaton

Example:

If the syntax of a language is specified using context-free grammar (上下文无关文法), the parsing of any valid program can be realized using a Pushdown Automaton

If one implements a pushdown automaton, it can potentially parse all valid programs for all languages that are defined in context-free grammar

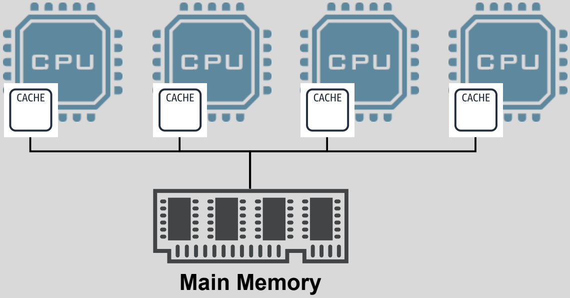
Application: Compiler-Compiler

Name	*Capability*	*Repo*
ANTLR	context-free grammar	`https://github.com/antlr/antlr4`
Yacc	context-free grammar	`https://www.tuhs.org/cgi-bin/utree.pl?file=V6/usr/source/yacc`
GNU Bison	context-free grammar	`https://git.savannah.gnu.org/cgit/bison.git`
JavaCC	context-free grammar	`https://github.com/javacc/javacc`

Model Behaviors of Protocols

Example: Write Invalidate Protocol

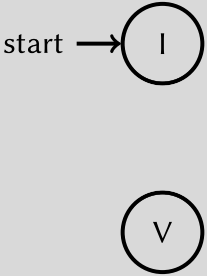
- Used for cache coherence in multi-processor computer
- Write-invalidate: invalidating all caches before the write
- Write-through: write directly to memory
- No-write-allocate: write-miss does not load cache



Write Invalidate Protocol

Each cache block is in one of two potential states:

- **Valid** (V): the cache block is up-to-date
- **Invalid** (I): the cache block is not valid



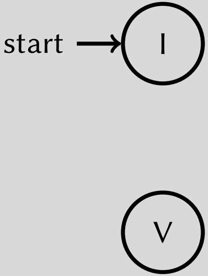
Write Invalidate Protocol

Each cache block is in one of two potential states:

- **Valid** (V): the cache block is up-to-date
- **Invalid** (I): the cache block is not valid

Events (transitions):

- Processor: read (PrRd), write (PrWr)
- Bus: read signal (BusRd), write signal (BusWr)



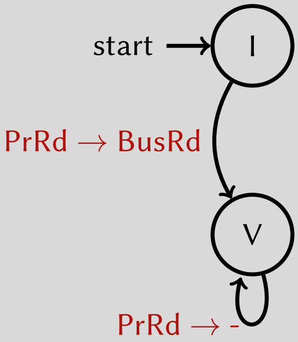
Write Invalidate Protocol

Each cache block is in one of two potential states:

- **Valid** (V): the cache block is up-to-date
- **Invalid** (I): the cache block is not valid

Events (transitions):

- Processor: read (PrRd), write (PrWr)
- Bus: read signal (BusRd), write signal (BusWr)



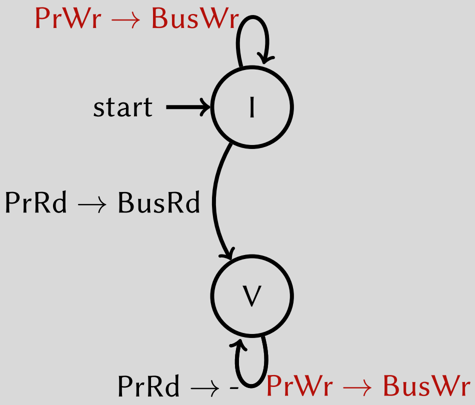
Write Invalidate Protocol

Each cache block is in one of two potential states:

- **Valid** (V): the cache block is up-to-date
- **Invalid** (I): the cache block is not valid

Events (transitions):

- Processor: read (PrRd), write (PrWr)
- Bus: read signal (BusRd), write signal (BusWr)



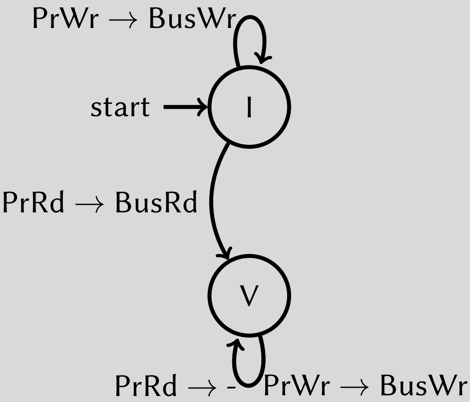
Write Invalidate Protocol

Each cache block is in one of two potential states:

- **Valid** (V): the cache block is up-to-date
- **Invalid** (I): the cache block is not valid

Events (transitions):

- Processor: read (PrRd), write (PrWr)
- Bus: read signal (BusRd), write signal (BusWr)



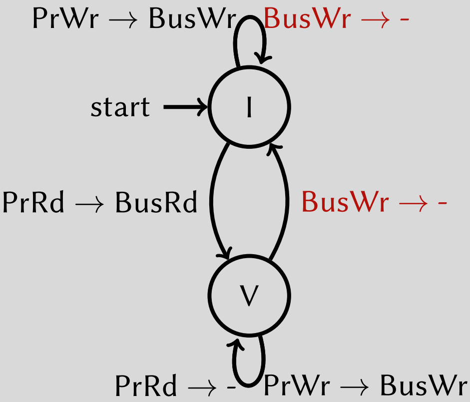
Write Invalidate Protocol

Each cache block is in one of two potential states:

- **Valid** (V): the cache block is up-to-date
- **Invalid** (I): the cache block is not valid

Events (transitions):

- Processor: read (PrRd), write (PrWr)
- Bus: read signal (BusRd), write signal (BusWr)



String Matching

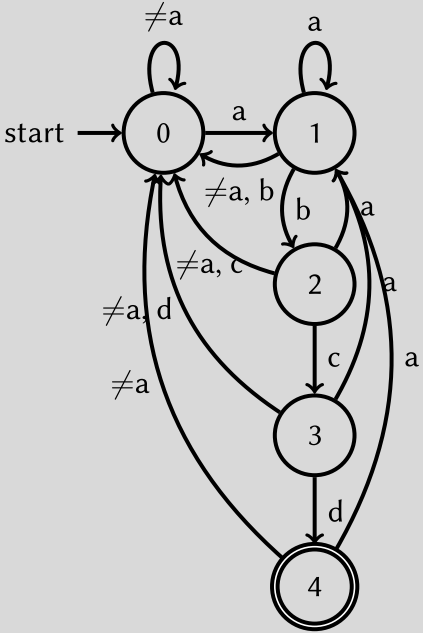
Example: Can a string `aaababcabcdefg` be matched by `abcd`?

String Matching

Example: Can a string `aaababcabcdefg` be matched by `abcd`?

Build an automata as the right:

- Five states:
  - 0: Match nothing
  - 1: Match `a`
  - 2: Match `ab`
  - 3: Match `abc`
  - 4: Match `abcd`: the accepting state
- If the accepting state is reached, a match of `abcd` is found



Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,
- a start state (初始状态), denoted as q_0

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,
- a start state (初始状态), denoted as q_0
- a set of final or accepting states (终止/接受状态) $F \subseteq Q$

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,
- a start state (初始状态), denoted as q_0
- a set of final or accepting states (终止/接受状态) $F \subseteq Q$

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,
- a start state (初始状态), denoted as q_0
- a set of final or accepting states (终止/接受状态) $F \subseteq Q$

Example: Pattern Matching Automaton for `abcd`

- $Q = \{0, 1, 2, 3, 4\}$

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,
- a start state (初始状态), denoted as q_0
- a set of final or accepting states (终止/接受状态) $F \subseteq Q$

Example: Pattern Matching Automaton for `abcd`

- $Q = \{0, 1, 2, 3, 4\}$
- $\Sigma = \{a, b, c, d, x\}$, where x denotes a symbol that is not `a`, `b`, `c` or `d`

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,
- a start state (初始状态), denoted as q_0
- a set of final or accepting states (终止/接受状态) $F \subseteq Q$

Example: Pattern Matching Automaton for `abcd`

- $Q = \{0, 1, 2, 3, 4\}$
- $\Sigma = \{a, b, c, d, x\}$, where x denotes a symbol that is not `a`, `b`, `c` or `d`
- transition function (see later)

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,
- a start state (初始状态), denoted as q_0
- a set of final or accepting states (终止/接受状态) $F \subseteq Q$

Example: Pattern Matching Automaton for `abcd`

- $Q = \{0, 1, 2, 3, 4\}$
- $\Sigma = \{a, b, c, d, x\}$, where x denotes a symbol that is not `a`, `b`, `c` or `d`
- transition function (see later)
- start state $q_0 = 0$

Basic Constructs of DFA

A deterministic finite automaton (DFA, 确定性有限状态自动机) consists of

- a finite set of states (状态), denoted as Q,
- *alphabet* (字母表) which is a finite set of input symbols, denoted as Σ,
- a transition (状态转移) function $\delta : Q \times \Sigma \mapsto Q$,
- a start state (初始状态), denoted as q_0
- a set of final or accepting states (终止/接受状态) $F \subseteq Q$

Example: Pattern Matching Automaton for `abcd`

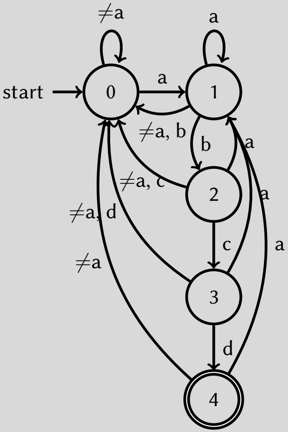
- $Q = \{0, 1, 2, 3, 4\}$
- $\Sigma = \{a, b, c, d, x\}$, where x denotes a symbol that is not `a`, `b`, `c` or `d`
- transition function (see later)
- start state $q_0 = 0$
- accepting state $F = \{4\}$

Representations of an Automaton

Graph Representation (图表示法)

- Each node represents a state. Q is the set of all nodes.

Example:

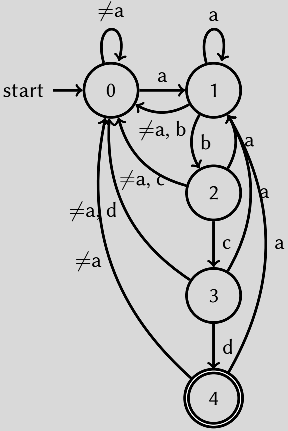


Representations of an Automaton

Graph Representation (图表示法)

- Each node represents a state. Q is the set of all nodes.
- Each edge represents a transition, the annotation indicates the input symbol: an edge from p to q with symbol x represents $\delta(p, x) = q$. Σ is the set of all edge annotations

Example:

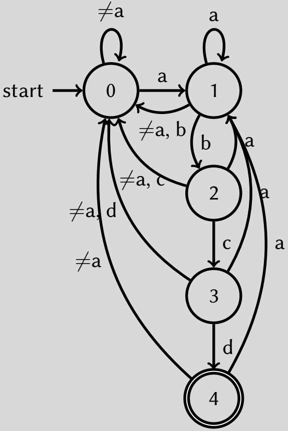


Representations of an Automaton

Graph Representation (图表示法)

- Each node represents a state. Q is the set of all nodes.
- Each edge represents a transition, the annotation indicates the input symbol: an edge from p to q with symbol x represents $\delta(p, x) = q$. Σ is the set of all edge annotations
- Start state q_0 is annotated with a "start" label

Example:

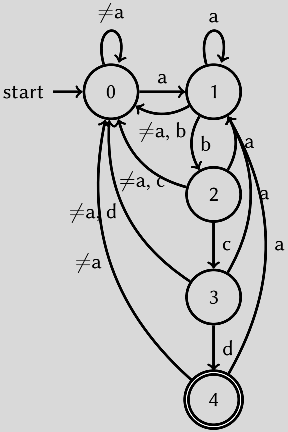


Representations of an Automaton

Graph Representation (图表示法)

- Each node represents a state. Q is the set of all nodes.
- Each edge represents a transition, the annotation indicates the input symbol: an edge from p to q with symbol x represents $\delta(p, x) = q$. Σ is the set of all edge annotations
- Start state q_0 is annotated with a "start" label
- Accepting states F are annotated by double circle.

Example:



Representations of an Automaton

The Table Representation (表表示法)

- Each index represents a state. Q is the set of all indices.

Example:

	a	b	c	d	x
$\rightarrow 0$	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
$*\ 4$	4	4	4	4	4

Representations of an Automaton

The Table Representation (表表示法)

- Each index represents a state. Q is the set of all indices.
- Each cell represents a transition, the column index indicates the input symbol: if the value of cell with row index p and column index x is q, it represents $\delta(p, x) = q$. Σ is the set of all column indices.

Example:

	a	b	c	d	x
$\rightarrow 0$	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
$*\,4$	4	4	4	4	4

Representations of an Automaton

The Table Representation (表表示法)

- Each index represents a state. Q is the set of all indices.
- Each cell represents a transition, the column index indicates the input symbol: if the value of cell with row index p and column index x is q, it represents $\delta(p, x) = q$. Σ is the set of all column indices.
- Start state q_0 is annotated with an arrow

Example:

	a	b	c	d	x
$\rightarrow 0$	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
$*\,4$	4	4	4	4	4

Representations of an Automaton

The Table Representation (表表示法)

- Each index represents a state. Q is the set of all indices.
- Each cell represents a transition, the column index indicates the input symbol: if the value of cell with row index p and column index x is q, it represents $\delta(p, x) = q$. Σ is the set of all column indices.
- Start state q_0 is annotated with an arrow
- Accepting states F are annotated with a star.

Example:

	a	b	c	d	x
$\rightarrow 0$	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
$* 4$	4	4	4	4	4

Product of Automata

How to build an automaton that can match `abcd` or `abd` by combining the automaton for `abcd` and the one for `abd`?

Product of Automata

How to build an automaton that can match `abcd` or `abd` by combining the automaton for `abcd` and the one for `abd`?

We need to compute the product (乘积) of automata.

Product of Automata

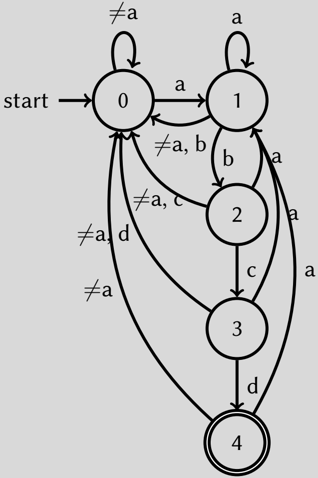
How to build an automaton that can match abcd or abd by combining the automaton for abcd and the one for abd?

We need to compute the product (乘积) of automata.

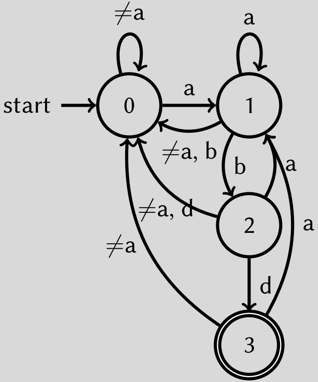
Note: the examples are deterministic finite automata (DFA, 确定性有限状态自动机). However, the product computation also applies to non-deterministic finite automatons (NFA, 非确定性有限状态自动机).

Product of Automata

Automaton for `abcd`



Automaton for `abd`



Product of Automata

For two automaton $(Q, \Sigma, \delta, q_0, F)$ and $(Q', \Sigma', \delta', q'_0, F')$, the product automaton consists of

- Set of states: $Q \times Q'$
- Alphabet: $\Sigma \times \Sigma'$ (not all combination is valid)
- Transition function: if $q = \delta(p, x)$ and $q' = \delta'(p', x')$, $(q, q') = \delta_P((p, p'), (x, x'))$
- Start state: (q_0, q'_0)
- Accepting states: depending on the combination logic

Product of Automata

For two automaton $(Q, \Sigma, \delta, q_0, F)$ and $(Q', \Sigma', \delta', q'_0, F')$, the product automaton consists of

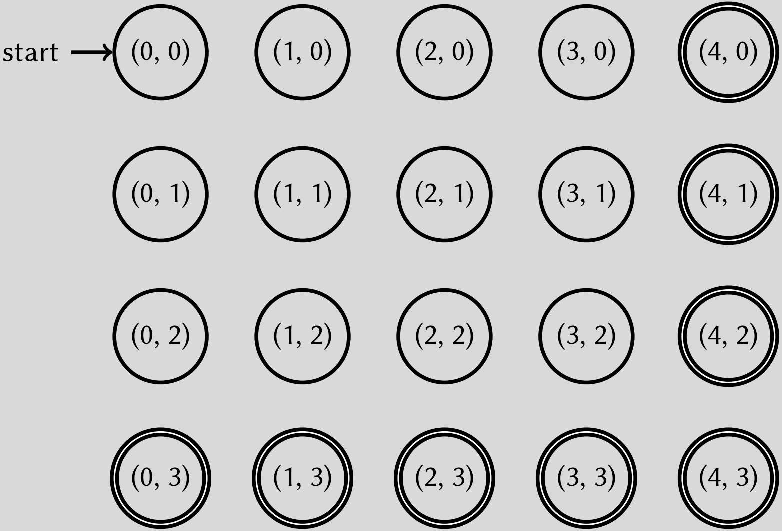
- Set of states: $Q \times Q'$
- Alphabet: $\Sigma \times \Sigma'$ (not all combination is valid)
- Transition function: if $q = \delta(p, x)$ and $q' = \delta'(p', x')$, $(q, q') = \delta_P((p, p'), (x, x'))$
- Start state: (q_0, q'_0)
- Accepting states: depending on the combination logic

Example: Multiple Pattern Matching

- Input symbol constraint: The input symbols to both automaton must be the same
- Accepting state: a state with at least one accepting state in the original automaton is an accepting state

Example: Multiple Pattern Matching

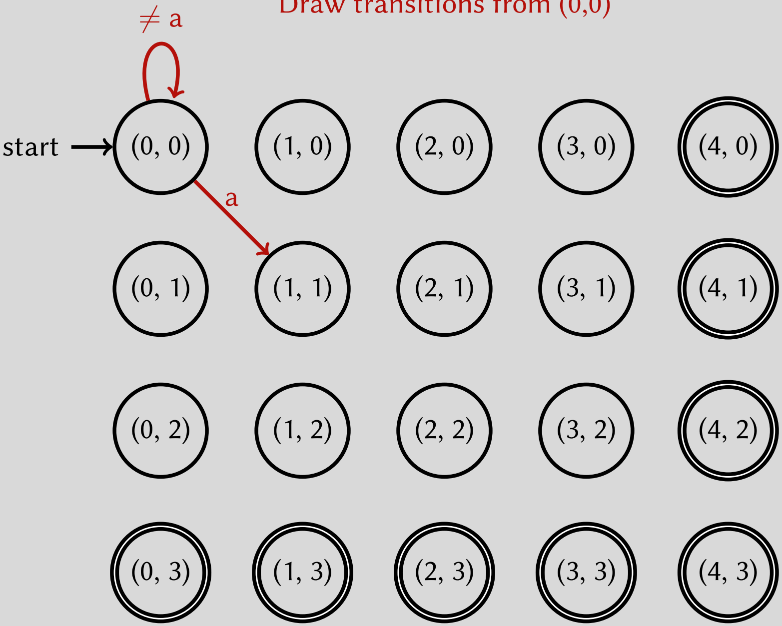
	a	b	c	d	x
	abcd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	0
	abd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	0	3	0
3	1	0	0	0	0



Example: Multiple Pattern Matching

	a	b	c	d	x
	abcd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	0
	abd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	0	3	0
3	1	0	0	0	0

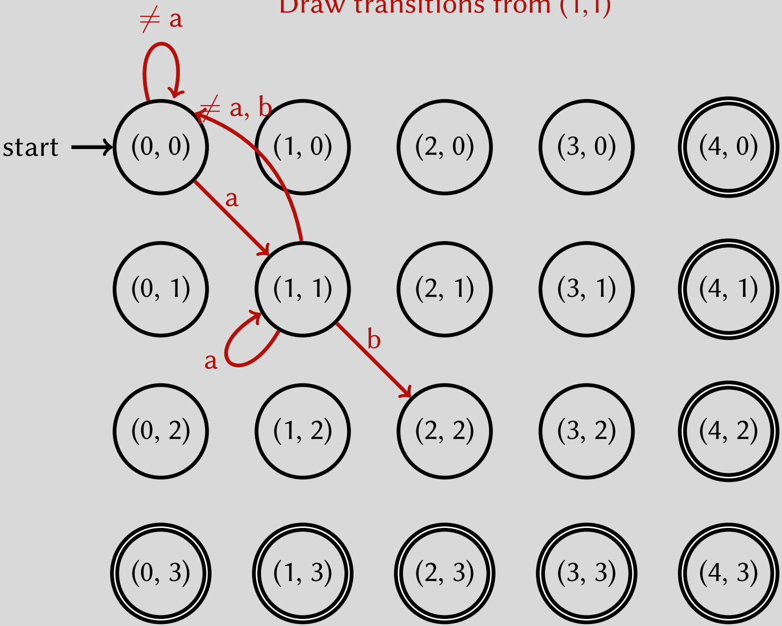
Draw transitions from (0,0)



Example: Multiple Pattern Matching

	a	b	c	d	x
	abcd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	0
	abd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	0	3	0
3	1	0	0	0	0

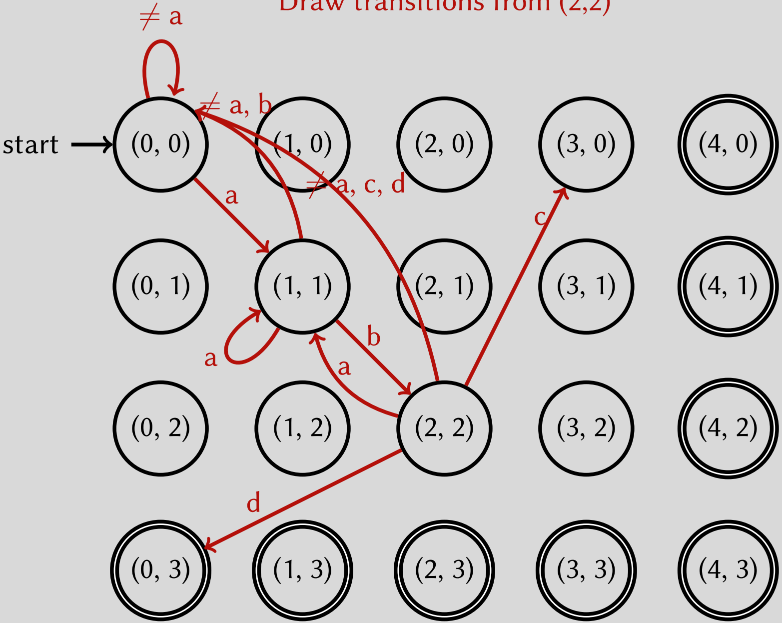
Draw transitions from (1,1)



Example: Multiple Pattern Matching

	a	b	c	d	x
	abcd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	0
	abd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	0	3	0
3	1	0	0	0	0

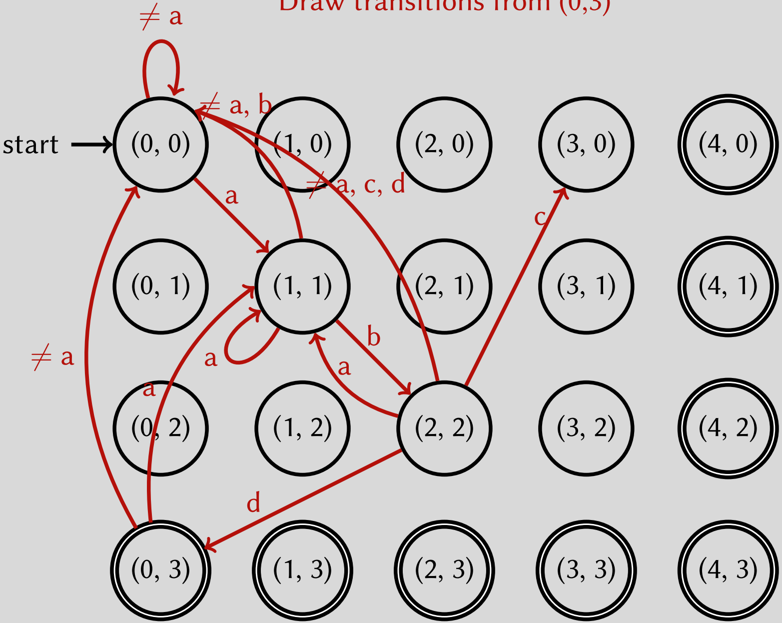
Draw transitions from (2,2)



Example: Multiple Pattern Matching

Draw transitions from (0,3)

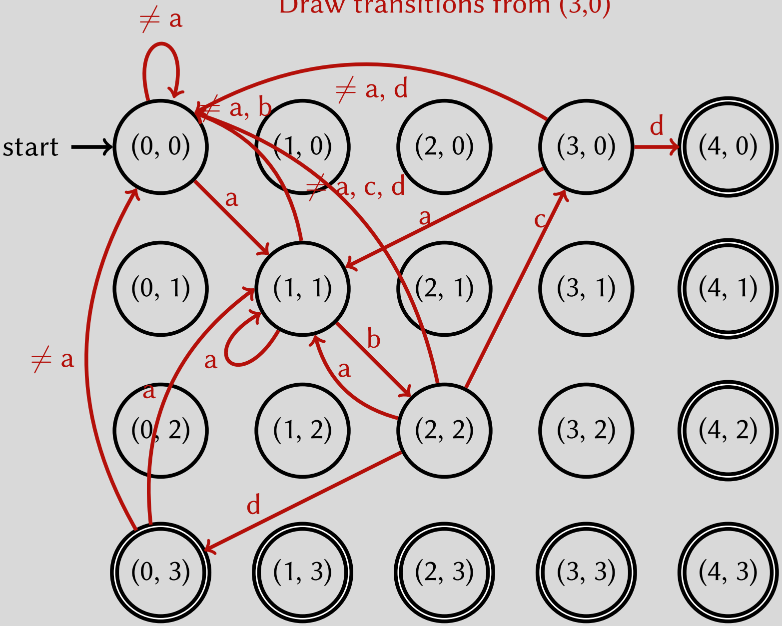
	a	b	c	d	x
	abcd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	0
	abd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	0	3	0
3	1	0	0	0	0



Example: Multiple Pattern Matching

	a	b	c	d	x
	abcd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	0
	abd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	0	3	0
3	1	0	0	0	0

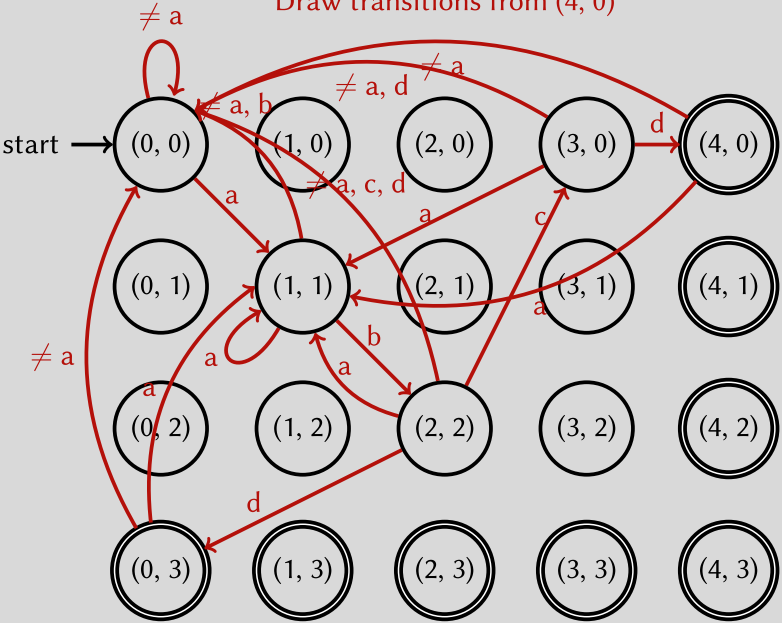
Draw transitions from (3,0)



Example: Multiple Pattern Matching

	a	b	c	d	x
	abcd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	0
	abd				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	0	3	0
3	1	0	0	0	0

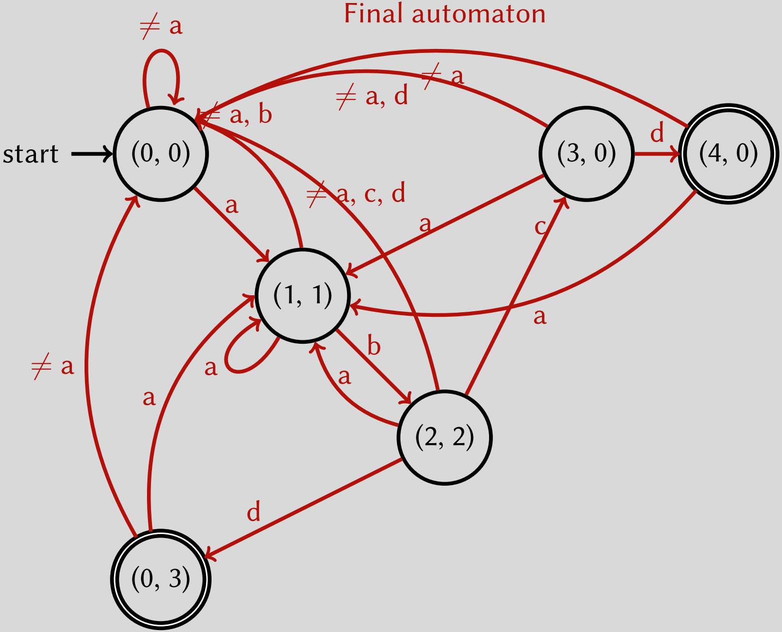
Draw transitions from (4, 0)



Example: Multiple Pattern Matching

	a	b	c	d	x
	`abcd`				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	3	0	0
3	1	0	0	4	0
4	1	0	0	0	0
	`abd`				
0	1	0	0	0	0
1	1	2	0	0	0
2	1	0	0	3	0
3	1	0	0	0	0

Final automaton



Product of Automata

Both depth-first search and breadth-first search can work

Breadth-first search:

1. The initial queue contains only the start state (e.g., $(0, 0)$)
2. In each iteration, if the queue is not empty, take one state from the queue. Otherwise terminate.
3. Mark the state as visited and enumerate all possible transitions from this state
4. If the new state is not visited, add it to the queue

Example

visited state:

#iter	*queue*	*transitions*
0	$\{(0,0)\}$	-

Example

visited state: (0, 0)

#iter	*queue*	*transitions*
0	$\{(0, 0)\}$	-
1	$\{(0, 0)\}$	$(0, 0) \rightarrow (0, 0), (1, 1)$

Example

visited state: (0, 0) , (1, 1)

#iter	*queue*	*transitions*
0	$\{(0,0)\}$	-
1	$\{(0,0)\}$	$(0,0) \rightarrow (0,0), (1,1)$
2	$\{(1,1)\}$	$(1,1) \rightarrow (0,0), (1,1), (2,2)$

Example

visited state: (0, 0) , (1, 1) , (2, 2)

#iter	*queue*	*transitions*
0	$\{(0,0)\}$	-
1	$\{(0,0)\}$	$(0,0) \rightarrow (0,0), (1,1)$
2	$\{(1,1)\}$	$(1,1) \rightarrow (0,0), (1,1), (2,2)$
3	$\{(2,2)\}$	$(2,2) \rightarrow (0,0), (1,1), (0,3), (3,0)$

Example

visited state: (0, 0) , (1, 1) , (2, 2) , (0, 3)

#iter	*queue*	*transitions*
0	$\{(0,0)\}$	-
1	$\{(0,0)\}$	$(0,0) \rightarrow (0,0), (1,1)$
2	$\{(1,1)\}$	$(1,1) \rightarrow (0,0), (1,1), (2,2)$
3	$\{(2,2)\}$	$(2,2) \rightarrow (0,0), (1,1), (0,3), (3,0)$
4	$\{(0,3), (3,0)\}$	$(0,3) \rightarrow (0,0), (1,1)$

Example

visited state: (0, 0) , (1, 1) , (2, 2) , (0, 3) , (3, 0)

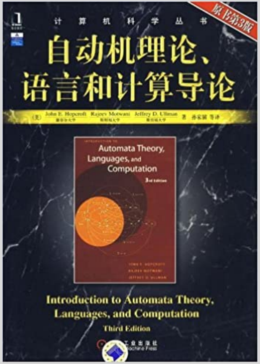
#iter	*queue*	*transitions*
0	$\{(0,0)\}$	-
1	$\{(0,0)\}$	$(0,0) \rightarrow (0,0), (1,1)$
2	$\{(1,1)\}$	$(1,1) \rightarrow (0,0), (1,1), (2,2)$
3	$\{(2,2)\}$	$(2,2) \rightarrow (0,0), (1,1), (0,3), (3,0)$
4	$\{(0,3), (3,0)\}$	$(0,3) \rightarrow (0,0), (1,1)$
5	$\{(3,0)\}$	$(3,0) \rightarrow (0,0), (1,1), (4,0)$

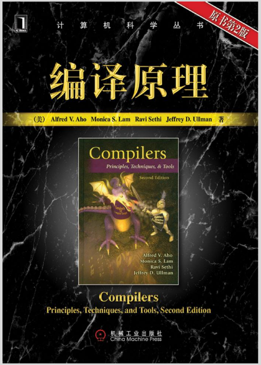
Example

visited state: (0, 0) , (1, 1) , (2, 2) , (0, 3) , (3, 0) , (4, 0)

#iter	*queue*	*transitions*
0	$\{(0,0)\}$	-
1	$\{(0,0)\}$	$(0,0) \rightarrow (0,0), (1,1)$
2	$\{(1,1)\}$	$(1,1) \rightarrow (0,0), (1,1), (2,2)$
3	$\{(2,2)\}$	$(2,2) \rightarrow (0,0), (1,1), (0,3), (3,0)$
4	$\{(0,3), (3,0)\}$	$(0,3) \rightarrow (0,0), (1,1)$
5	$\{(3,0)\}$	$(3,0) \rightarrow (0,0), (1,1), (4,0)$
6	$\{(4,0)\}$	$(4,0) \rightarrow (0,0), (1,1)$

Further Reading





John E Hopcroft, Rajeev Motwani, and JR Ullman. 自动机理论, 语言和计算导论. 原书第 3 版. 北京: 机械工业出版社, 2008

Alfred V Aho, Monica S Lam, and Jeffrey D Ravi Sethi. 编译原理. 原书第 2 版. 机械工业出版社, 2009

The End

Summary

In the coming lectures, we cover the following topics:

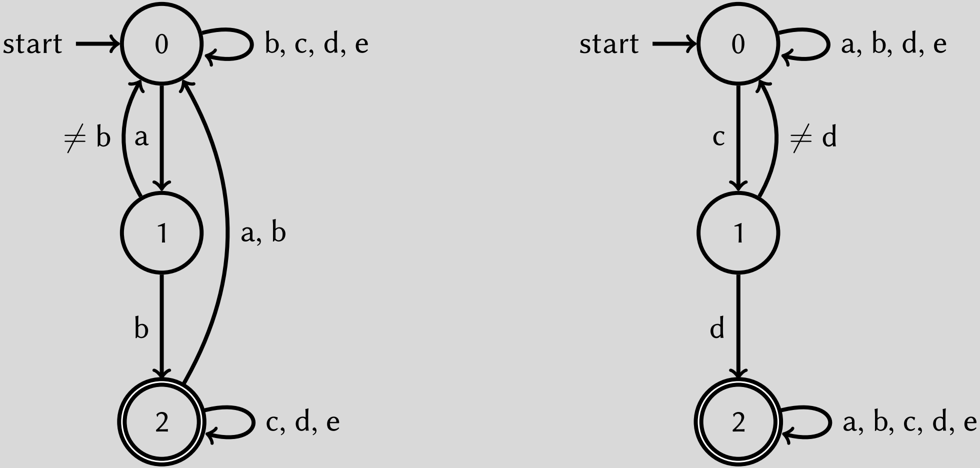
- Automata theory
- Linear programming
- Mixed integer linear programming
- Satisfactory theory

In this lecture, you should

- see examples of automata theory
- get a basic sense of how to use automata to model the behaviors of a system/an algorithm
- understand deterministic finite automata
- understand how to compute the product automata

Quiz

画出下列两个确定性有限状态自动机的乘积并给出计算过程



Thanks!

kaigao@scu.edu.cn

References I

[1] Alfred V Aho, Monica S Lam, and Jeffrey D Ravi Sethi. 编译原理. 原书第 2 版. 机械工业出版社, 2009.

[2] Martin Björklund. *YANG - a Data Modeling Language for the Network Configuration Protocol (NETCONF).* RFC 6020. Oct. 2010. URL: `https://rfc-editor.org/rfc/rfc6020.txt`.

[3] Erika Chin et al. "Analyzing Inter-Application Communication in Android". In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services - MobiSys '11.* The 9th International Conference. Bethesda, Maryland, USA: ACM Press, 2011, p. 239. URL: `http://portal.acm.org/citation.cfm?doid=1999995.2000018` (visited on 09/17/2021).

[4] John E Hopcroft, Rajeev Motwani, and JR Ullman. 自动机理论, 语言和计算导论. 原书第 3 版. 北京: 机械工业出版社, 2008.