

Introduction to the Theory of Discrete Systems

Luca de Alfaro
Department of Computer Engineering
UC Santa Cruz, USA

October 16, 2007

Chapter 1

Automata

Automata are a basic formalism for describing discrete systems. An automaton consists of a set of states, of an alphabet, and of a transition rule. At every state, the automaton scans the next input symbol, and moves to a new state. We distinguish between *deterministic* automata, in which the next state is uniquely determined by the current state and the input symbol, and *nondeterministic* automata, where the current state and input symbol determine a set of possible successor states.

One of the important roles of automata consists in defining languages. When an automaton is presented with an input string, it will either *accept* it or *reject* it. The language accepted by the automaton consists in the set of strings accepted by the automaton. The set of languages that can be defined via finite automata are called *regular* languages.

Finite automata find many applications in Computer Science. They are used to recognize and process text according to regular expressions, as lexers in compiler generation [1]. Automata also provide a basic model of discrete systems, and precisely, of systems subject to input behavior from the environment (the symbols read by the automaton). We overview here finite automata, starting from basic automata (deterministic and \exists -nondeterministic), and introducing later \forall -nondeterministic automata and alternating automata [?].

1.1 Deterministic Automata

Definition 1 (DFA). A *deterministic finite automaton* (DFA) $P = \langle \mathcal{S}_P, s_P^{init}, \mathcal{A}_P, \delta_P \rangle$ consists of the following components:

- \mathcal{S}_P is a finite set of *states*.

- $s_P^{init} \in \mathcal{S}_P$ is an *initial state*.
- \mathcal{A}_P is the finite *alphabet* of P .
- $\delta_P : \mathcal{S}_P \times \mathcal{A}_P \mapsto \mathcal{S}_P$ is the transition function of the automaton. ■

In order to define the language of P , it is convenient to define the *multi-step transition function* $\hat{\delta}_P : \mathcal{S}_P \times \mathcal{A}_P^* \mapsto \mathcal{S}_P$ inductively, by $\hat{\delta}_P(s, \varepsilon) = s$ for all $s \in \mathcal{S}_P$, and by $\hat{\delta}_P(s, a_0 a_1 \cdots a_{n+1}) = \delta_P(\hat{\delta}_P(s, a_0 \cdots a_n), a_{n+1})$ for $n \geq 0$. The size of a DFA P is given by $|P| = |\mathcal{S}_P| \cdot |\mathcal{A}_P|$.

Definition 2 (language of DFA). The language $\mathcal{L}(P, F)$ of a DFA P with set of accepting states $F \subseteq \mathcal{S}_P$ is given by

$$\mathcal{L}(P, F) = \{\sigma \in \mathcal{A}_P^* \mid \hat{\delta}_P(s_P^{init}, \sigma) \in F\} \quad \blacksquare$$

The following properties are easily proved.

Lemma 1. *The following assertions hold, for a DFA P and for all $F, F_1, F_2 \subseteq \mathcal{S}$:*

$$\mathcal{L}(P, F) = \mathcal{A}_P^* \setminus \mathcal{L}(P, \mathcal{S}_P \setminus F) \quad (1.1)$$

$$\mathcal{L}(P, F_1 \cup F_2) = \mathcal{L}(P, F_1) \cup \mathcal{L}(P, F_2) \quad (1.2)$$

$$\mathcal{L}(P, F_1 \cap F_2) = \mathcal{L}(P, F_1) \cap \mathcal{L}(P, F_2) \quad (1.3)$$

1.1.1 Operations on DFAs

Lemma 1 enables us to compute boolean operations among the languages defined by sets of final states *over a fixed DFA*. If the languages are specified by two distinct DFAs P and Q , we can perform the above binary boolean operations by first forming the product of the original automata. We require the two automata to share the same alphabet. If the automata have different alphabet, one must first modify the automata to describe the consequence of symbols not in the original alphabet (they could correspond to self-loops, or to transitions to a non-accepting sink state).

Definition 3 (product of DFAs). Given two DFAs P and Q with $\mathcal{A}_P = \mathcal{A}_Q$, their product $R = P \otimes Q$ is defined as follows, for all $s \in \mathcal{S}_P$, $t \in \mathcal{S}_Q$, and $a \in \mathcal{A}_Q$:

$$\mathcal{S}_R = \mathcal{S}_P \times \mathcal{S}_Q \quad (1.4)$$

$$\mathcal{A}_R = \mathcal{A}_P = \mathcal{A}_Q \quad (1.5)$$

$$s_R^{init} = \langle s_P^{init}, s_Q^{init} \rangle \quad (1.6)$$

$$\delta_R(\langle s, t \rangle, a) = \langle \delta_P(s, a), \delta_Q(t, a) \rangle \quad \blacksquare \quad (1.7)$$

	\neg	\cup	\cap	$= \emptyset?$	$= \mathcal{A}_P^*$?
DFA	$\mathcal{O}(1)$	$\mathcal{O}(P \cdot Q)$	$\mathcal{O}(P \cdot Q)$	$\mathcal{O}(P)$	$\mathcal{O}(P)$
\exists -NFA	$2^{ P }$	$\mathcal{O}(1)$	$\mathcal{O}(P \cdot Q)$	$\mathcal{O}(P)$	PSPACE($ P $)
\forall -NFA	$2^{ P }$	$\mathcal{O}(P \cdot Q)$	$\mathcal{O}(1)$	PSPACE($ P $)	$\mathcal{O}(P)$
AFA	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	PSPACE($ P $)	PSPACE($ P $)

Table 1.1: Costs of computing boolean operations over automata languages. DFA are deterministic finite automata, NFA are nondeterministic finite automata, and AFA are alternating automata.

Unions and intersections of DFAs can be done as explained in the following lemma.

Lemma 2. *For two DFAs P, Q with $\mathcal{A}_P = \mathcal{A}_Q$, we have:*

$$\begin{aligned} \mathcal{L}(P \otimes Q, F_P \times F_Q) &= \mathcal{L}(P, F_P) \cap \mathcal{L}(P, F_P) \\ \mathcal{L}(P \otimes Q, (F_P \times S_Q) \cup (F_Q \times S_P)) &= \mathcal{L}(P, F_P) \cup \mathcal{L}(P, F_P) \end{aligned}$$

The language $\mathcal{L}(P, F)$ of a DFA P with set F of final states is non-empty iff there is a state of F reachable from a state in S_P^{init} . Thus, checking emptiness can be easily done in time linear in P .

Table 1.1 summarizes the costs of these operations. The cost of complementation for DFAs is listed as $\mathcal{O}(1)$ because we can simply associate with a set of final states F a single bit, specifying whether the set of accepting states is F or $S_P \setminus F$.

1.2 Nondeterministic Automata

Definition 4 (NFA). A *nondeterministic finite automaton* (NFA) $P = \langle S_P, s_P^{init}, \mathcal{A}_P, \delta_P \rangle$ consists of the following components:

- S_P is a finite set of *states*.
- $S_P^{init} \subseteq S_P$ is a non-empty set of *initial states*.
- \mathcal{A}_P is the finite *alphabet* of P .
- $\delta_P : S_P \times \mathcal{A}_P \mapsto 2^{S_P} \setminus \emptyset$ is the transition function of the automaton. ■

The size of an NFA P is defined by $|P| = \sum_{s \in \mathcal{S}_P} \sum_{a \in \mathcal{A}_P} |\delta_P(s, a)|$. In order to define the language of an NFA P , we again define the *multi-step transition function* $\hat{\delta}_P : 2^{\mathcal{S}_P} \times \mathcal{A}_P^* \mapsto 2^{\mathcal{S}_P}$ inductively, by $\hat{\delta}_P(T, \varepsilon) = T$ for all $T \subseteq \mathcal{S}_P$, and by

$$\hat{\delta}_P(s, a_0 a_1 \cdots a_{n+1}) = \bigcup_{t \in \hat{\delta}_P(s, a_0 \cdots a_n)} \delta_P(t, a_{n+1})$$

for $n \geq 0$.

Definition 5 (language of NFA). The language $\mathcal{L}(P, F)$ of an NFA P with set of accepting states $F \subseteq \mathcal{S}_P$ is given by

$$\mathcal{L}(P, F_P) = \{\sigma \in \mathcal{A}_P^* \mid \hat{\delta}_P(S_P^{init}, \sigma) \cap F \neq \emptyset\} \quad \blacksquare$$

The following property holds.

Lemma 3. *The following assertions holds, for all NFAs P and all $F_1, F_2 \subseteq \mathcal{S}$:*

$$\mathcal{L}(P, F_1 \cup F_2) = \mathcal{L}(P, F_1) \cup \mathcal{L}(P, F_2)$$

Note, on the other hand, that a result similar to Lemma 1 does *not* hold for NFAs.

Problem 1. Give a counterexample to (1.3) for NFAs. \blacksquare

1.2.1 Operations on NFAs

1.2.1.1 Complementation

To construct an NFA that accepts the complement of the language accepted by a specified NFA, we need to construct a DFA that accepts the same language as the NFA. This procedure, called *subset construction*, may produce a deterministic automaton with state space exponentially larger than the one of the original nondeterministic automaton. Once the automaton is determinized via subset construction, we can complement its language using Lemma 1.

Definition 6 (subset construction). Given an NFA $P = \langle \mathcal{S}_P, s_P^{init}, \mathcal{A}_P, \delta_P \rangle$ the *determinization* of P is a DFA Q defined by:

$$\begin{aligned} \mathcal{S}_Q &= 2^{\mathcal{S}_P} \\ s_Q^{init} &= S_P^{init} \\ \mathcal{A}_Q &= \mathcal{A}_P \end{aligned}$$

and, for all $T \subseteq \mathcal{S}_P$ and $a \in \mathcal{A}_P$,

$$\delta_Q(T, a) = \bigcup_{t \in T} \delta_P(t, a). \quad \blacksquare$$

Theorem 1 (determinization). *Let $P = \langle \mathcal{S}_P, s_P^{init}, \mathcal{A}_P, \delta_P \rangle$ be an NFA, and let Q be its determinization. For a set $F_P \subseteq \mathcal{S}_P$ of final states of P , let $F_Q = \{T \subseteq \mathcal{S}_P \mid T \cap F_P \neq \emptyset\}$. Then, $\mathcal{L}(P, F_P) = \mathcal{L}(Q, F_Q)$.*

1.2.1.2 Intersection

To construct an NFA that accepts the intersection of languages accepted by two NFAs, we again form the *product* of NFAs.

Definition 7 (product of NFAs). Given two NFAs P and Q with $\mathcal{A}_P = \mathcal{A}_Q$, their product $R = P \otimes Q$ is defined as follows, for all $s \in \mathcal{S}_P$, $t \in \mathcal{S}_Q$, and $a \in \mathcal{A}_Q$:

$$\mathcal{S}_R = \mathcal{S}_P \times \mathcal{S}_Q \tag{1.8}$$

$$\mathcal{A}_R = \mathcal{A}_P = \mathcal{A}_Q \tag{1.9}$$

$$s_R^{init} = s_P^{init} \times s_Q^{init} \tag{1.10}$$

$$\delta_R((s, t), a) = \delta_P(s, a) \times \delta_Q(t, a) \quad \blacksquare \tag{1.11}$$

Theorem 2 (intersection, NFAs). *Given two NFAs P and Q , together with sets of final states F_P and F_Q , let $F = F_P \times F_Q$. Then, $\mathcal{L}(P \otimes Q, F) = \mathcal{L}(P, F_P) \cap \mathcal{L}(Q, F_Q)$.*

1.2.1.3 Union

Nondeterminism leads to a very simple algorithm for the construction of an NFA representing the union of two other NFAs.

Theorem 3 (union, NFAs). *Given two NFAs P and Q with $\mathcal{A}_P = \mathcal{A}_Q$, and with sets of initial states S_P^{init} and S_Q^{init} , and sets of final states F_P and F_Q , let R be the NFA obtained by letting*

$$\begin{aligned} \mathcal{S}_R &= (\mathcal{S}_P \times \{0\}) \cup (\mathcal{S}_Q \times \{1\}) \\ S_R^{init} &= (S_P^{init} \times \{0\}) \cup (S_Q^{init} \times \{1\}) \\ F_R &= (F_P \times \{0\}) \cup (F_Q \times \{1\}) \end{aligned}$$

and, for all $a \in \mathcal{A}_R$, all $s \in \mathcal{S}_P$, and all $t \in \mathcal{S}_Q$,

$$\begin{aligned}\delta_R(s, a) &= \delta_P(s, a) \times \{0\} \\ \delta_R(t, a) &= \delta_Q(t, a) \times \{1\}.\end{aligned}$$

Then, $\mathcal{L}(R, F_R) = \mathcal{L}(P, F_P) \cup \mathcal{L}(Q, F_Q)$. ■

1.2.1.4 Emptiness check

The language $\mathcal{L}(P, F)$ of an NFA P with set F of final states is non-empty iff there is a state of F reachable from a state in S_P^{init} . Thus, checking emptiness can be easily done in time linear in P .

1.2.2 \forall -nondeterministic automata

An NFA accepts a string if there is *at least one way* to resolve nondeterminism and accept the string. It is natural to consider the dual of this notion: an automaton that accepts a string if *all* ways to resolve nondeterminism while reading that string lead to a final state. Formally, a \forall -NFA is defined in the same way as an NFA; its language is defined as follows.

Definition 8 (language of \forall -NFA). The language $\mathcal{L}(P, F)$ of a \forall -NFA P with set of accepting states $F \subseteq \mathcal{S}_P$ is given by

$$\mathcal{L}(P, F_P) = \{\sigma \in \mathcal{A}_P^* \mid \hat{\delta}_P(S_P^{init}, \sigma) \subseteq F\} \quad \blacksquare$$

The union (resp. intersection) of \forall -NFAs can be constructed similarly to the intersection (resp. union) of NFAs; again, complementation requires a preliminary determinization of the automaton.

Problem 2. Adapt the subset construction (Definition 6) and the determinization theorem (Theorem 1) for \forall -NFA. ■

1.3 Alternating Automata

In NFAs, the transition relation is disjunctive: at a state s where $\delta(s, a) = \{t_1, t_2\}$, if we receive a , it suffices for *one* of the two a -successors t_1, t_2 to lead to a final state in order for the input string to be accepted: intuitively, we can write $\delta(s, a) = t_1 \vee t_2$. Similarly, the transition relation of \forall -NFAs is disjunctive: at a state s where $\delta(s, a) = \{t_1, t_2\}$, if we receive a , *both* a -successors t_1, t_2 must lead to a final state in order for the input string to

be accepted: intuitively, we can write $\delta(s, a) = t_1 \wedge t_2$. It is then natural to consider a generalization of these two cases, in which $\delta(s, a)$ is a general boolean formula over the successor states. The resulting automata are called *alternating automata* [?]. To define them, given a set B , we denote by $\text{PosBool}(B)$ the set of all boolean formulas formed by combining with \wedge and \vee (but not \neg) the elements of B . Furthermore, for $\phi \in \text{PosBool}(B)$ and $C \subseteq B$, we say that C *satisfies* ϕ if ϕ holds when every $b \in B$ is interpreted as *true*, and every $b \notin C$ interpreted as *false*.

Definition 9 (alternating automaton). An *alternating finite automaton* (AFA) $P = \langle \mathcal{S}_P, s_P^{init}, \mathcal{A}_P, \delta_P \rangle$ consists of the following components:

- \mathcal{S}_P is a finite set of *states*.
- $\mathcal{S}_P^{init} \subseteq \mathcal{S}_P$ is a non-empty set of *initial states*.
- \mathcal{A}_P is the finite *alphabet* of P .
- $\delta_P : \mathcal{S}_P \times \mathcal{A}_P \mapsto \text{PosBool}(\mathcal{S}_P)$ is the transition function.

The size of an alternating automaton P is given by $|P| = \sum_{s \in \mathcal{S}_P} \sum_{a \in \mathcal{A}_P} |\delta_P(s, a)|$, where we indicate with $|\phi|$ the size (number of symbols) of a propositional formula ϕ .

There are two alternative ways to define the language of an alternating automaton: one based on an inductive definition of the language, the other based on the definition of *executions*. We give here the former. Given a boolean formula ϕ with propositions in q_1, \dots, q_n , and sets A_1, \dots, A_n , denote by $\hat{\phi}[A_i/q_i]_{1 \leq i \leq n}$ the set expression obtained from ϕ by replacing \vee with \cup , \wedge with \cap , and q_i with A_i , for $1 \leq i \leq n$. For instance, if ϕ is $q_1 \vee (q_2 \wedge q_3)$, then $\hat{\phi}[A_1/q_1, A_2/q_2, A_3/q_3]$ is $A_1 \cup (A_2 \cap A_3)$. The language of an alternating automaton can then be defined as follows.

Definition 10 (language of alternating automaton). Given an AFA P together with a set of final states $F \subseteq \mathcal{S}_P$, we define inductively for $n \geq 0$ and $s \in \mathcal{S}_P$ the language $\mathcal{L}_P(s, n, F)$, corresponding to the strings of length n that can be accepted starting from s :

$$\mathcal{L}_P(s, 0, F) = \begin{cases} \varepsilon & \text{if } s \in F; \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{L}_P(s, n + 1, F) = \bigcup_{a \in \mathcal{A}_P} a \cdot \widehat{\delta(s, a)}[\mathcal{L}_P(t, n, F)/t]_{t \in \mathcal{S}_P}.$$

Then,

$$\mathcal{L}(P, F) = \bigcup_{s \in S_P^{init}} \bigcup_{n \geq 0} \mathcal{L}_P(s, n, F). \quad \blacksquare$$

Problem 3 (union, complementation, intersection of alternating automata languages). Give algorithms that, given alternating automata P , Q and sets of final states $F_P \subseteq \mathcal{S}_P$, $F_Q \subseteq \mathcal{S}_Q$, produce alternating automata with the languages:

1. $\mathcal{A}_P^* \setminus \mathcal{L}(P, F_P)$.
2. $\mathcal{L}(P, F_P) \cup \mathcal{L}(Q, F_Q)$.
3. $\mathcal{L}(P, F_P) \cap \mathcal{L}(Q, F_Q)$. \blacksquare

Problem 4. For $n \geq 0$, show how to construct an alternating automaton accepting the language $\mathcal{L}_n = \{\sigma \cdot \sigma \mid \sigma \in \mathcal{A}^n\}$; the size of the automaton should be polynomial in n . Prove that your construction is correct. \blacksquare

Problem 5 (size of DFA, NFA, \forall -NFA, and alternating automata). Give the following examples, or explain why the following cannot be done:

1. Give an example of a family $\{\mathcal{L}_n\}_{n \geq 0}$ of languages, such that there is an NFA with size polynomial in n to recognize \mathcal{L}_n , but such that the smallest DFA to recognize \mathcal{L}_n has size exponential in n .
2. Give an example of a family $\{\mathcal{L}_n\}_{n \geq 0}$ of languages, such that there is a \forall -NFA with size polynomial in n to recognize \mathcal{L}_n , but such that the smallest DFA to recognize \mathcal{L}_n has size exponential in n .
3. Give an example of a family $\{\mathcal{L}_n\}_{n \geq 0}$ of languages, such that there is a \forall -NFA with size polynomial in n to recognize \mathcal{L}_n , but such that the smallest NFA to recognize \mathcal{L}_n has size exponential in n .
4. Give an example of a family $\{\mathcal{L}_n\}_{n \geq 0}$ of languages, such that there is an alternating automaton with size polynomial in n to recognize \mathcal{L}_n , but such that the smallest NFA or \forall -NFAs to recognize \mathcal{L}_n has size exponential in n . \blacksquare

Problem 6 (regularity of alternating automata languages). Give a language-preserving translation from alternating automata to NFA. Such a translation proves that the language accepted by an alternating automaton is regular, that is, the class of languages that can be accepted by alternating automata and NFA (and DFA) coincide. Hint: do some kind of subset construction, on either \cup or \cap . \blacksquare

Problem 7 (Satisfiability and automata operations). Consider a set $Q = \{q_1, q_2, \dots, q_n\}$ of propositions. A truth assignment to Q can be represented by a string in $\{\mathbf{T}, \mathbf{F}\}^n$: for example, for $n = 3$, the string \mathbf{FTF} represents the assignment $q_1 = \mathbf{F}$, $q_2 = \mathbf{T}$, $q_3 = \mathbf{F}$. Let $\text{Bool}(Q)$ be the set of all boolean formulas over the set of predicates Q constructed using \wedge , \vee , and \neg . A formula $\phi \in \text{Bool}(Q)$ defines a language $\mathcal{L}(\phi) \subseteq \{\mathbf{T}, \mathbf{F}\}^n$ consisting of the truth assignments that satisfy ϕ .

1. Give an algorithm that, given $\phi \in \text{Bool}(Q)$, constructs a DFA accepting $\mathcal{L}(\phi)$. The algorithm should use only DFAs. By checking emptiness of the DFA for $\mathcal{L}(\phi)$, it is possible to check satisfiability of ϕ . If you use this algorithm as a satisfiability checker, what is its resulting complexity, expressed in terms of n and $|\phi|$?
2. Give an algorithm that, given $\phi \in \text{Bool}(Q)$, constructs an NFA accepting $\mathcal{L}(\phi)$. The algorithm should use only NFAs. By checking emptiness of the NFA for $\mathcal{L}(\phi)$, it is possible to check satisfiability of ϕ . If you use this algorithm as a satisfiability checker, what is its resulting complexity, expressed in terms of n and $|\phi|$? Is it better than the one for DFAs?
3. Give an algorithm that, given $\phi \in \text{Bool}(Q)$, constructs an alternating automaton accepting $\mathcal{L}(\phi)$. The algorithm can use DFAs, NFAs, or alternating automata. By checking emptiness of the alternating automaton for $\mathcal{L}(\phi)$, it is possible to check satisfiability of ϕ . What can you say about the complexity of checking emptiness of alternating automata? ■

1.3.1 Emptiness of alternating automata

The translation from AFA to NFA of Problems 6 yields an exponential-time decision procedure for emptiness of an AFA language: first, we translate the AFA into an NFA, then we check for emptiness of the NFA. Hence, the problem is in EXPTIME. The problem of checking AFA emptiness is also in PSPACE: to obtain a PSPACE algorithm, we can do the translation from AFA to NFA on the fly, while checking for the existence of accepting paths. From Problem 7 it follows that checking emptiness of AFAs is NP-hard. In fact, checking emptiness of AFAs is PSPACE-complete, since the problem of checking universality for NFAs is PSPACE-complete, and NFA universality can be trivially reduced to AFA emptiness.

Chapter 2

Transition Systems

Finite automata are models for language *acceptors*: they are used to define (regular) languages via the set of string they can accept. A *transition system* is a model for language *generators*, and thus, for system that *generate* behavior. There are three main differences between transition systems and automata. The first is that transition systems, as mentioned above, generate behavior, rather than accept it. The second difference is that transition systems do not have final states: they generate infinite behaviors. In this, they are similar to models of physical systems, whose differential equations describe behavior that goes on forever. The third difference is that transition systems are usually nondeterministic: the system has the choice of actions or transitions to perform.

2.1 Enumerative Representation

Definition 1 (transition system). A *transition system* $P = \langle S_P, s_P^{init}, \mathcal{A}_P, Steps_P \rangle$ consists of the following components:

- S_P is a set of *states*.
- $s_P^{init} \subseteq S_P$ is a set of *initial states*. We require that s_P^{init} contains at most one state. If $s_P^{init} = \emptyset$, then P is called *empty*.
- \mathcal{A}_P is a set of *actions*.
- $Steps_P \subseteq S_P \times \mathcal{A}_P \times S_P$ is a set of *steps*. ■

When no confusion arises, we will write $P = \langle S, s^{init}, \mathcal{A}, Steps \rangle$, for brevity. We let

$$\Gamma(s) = \{a \in \mathcal{A} \mid \exists t \in S. \langle s, a, t \rangle \in Steps\}.$$

Furthermore, if $\langle s, a, t \rangle \in Steps$, we write $s \xrightarrow{a} t$. We also indicate by $\rho_P = \{(s, t) \in S \times S \mid \exists a \in \mathcal{A}_P. s \xrightarrow{a} t\}$ the transition relation induced when any action can be taken.

Paths and Traces

A *path* of a transition system is a sequence of interleaved actions and states that can be followed starting from the initial state.

Definition 2 (paths). A path is an infinite sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots$ such that $s_k \xrightarrow{a_k} s_{k+1}$ for all $k \geq 0$. We indicate by $Paths_P(s_0)$ the set of paths of P with initial state s_0 . ■

Thus, unlike the case of finite automata, the behaviors of transition systems are *infinite*: we use transition systems as models of systems that exhibit a never-terminating behavior, similarly to physical systems.

We say that the transition system is *deterministic* if $\langle s, a, t \rangle, \langle s, a, t' \rangle \in Steps$ implies $t = t'$.

An *action trace* of a transition system is an infinite sequence of actions that can be followed; similarly, a *state trace* of a transition system is a sequence of states that can be followed.

Definition 3 (state and action traces and languages).

- An *action trace* of P from $s_0 \in \mathcal{S}$ is a sequence of actions $a_0, a_1, a_2, \dots \in \mathcal{A}^*$ such that there is a corresponding path $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots$. The *action language* from s_0 , denoted by $\mathcal{L}_{\mathcal{A}}(s_0)$, is the set of all action traces of P from s_0 .
- An *action trace* of P from $s_0 \in \mathcal{S}$ is a sequence of states $s_0, s_1, s_2, \dots \in \mathcal{A}^*$ such that there is a corresponding path $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \cdots$. The *state language* from s_0 , denoted by $\mathcal{L}_{\mathcal{S}}(s_0)$, is the set of all state traces of P from s_0 . ■

Given a transition system P , its set $Reach(P)$ of *reachable states* consists of all states that can be reached from some initial state by taking transitions of P .

Definition 4 (reachable states). Given a transition system $P = \langle S_P, s_P^{init}, \mathcal{A}_P, Steps_P \rangle$ we say that a state $s \in S_P$ is *reachable* if there is a state trace $s_0, s_1, \dots, s_k, \dots$ of P with $s_k = s$ for some $k \geq 0$. We denote the set of reachable states of P by $Reach(P)$. ■

2.2 Symbolic Representation

While the enumerative representation given above is convenient from a mathematical point of view, it is impractical for the description of complex transition systems. For instance, assume that we want to use a transition system to describe the behavior of a piece of hardware containing 40 latches. As each latch can be in 2 states, the state-space of the system has size 2^{40} , and it would be highly impractical to describe its behavior by providing explicitly the state-transition graph. We will introduce here a *symbolic* representation for transition systems, which provides a compact way to model systems that can have large state spaces. As we will see, the symbolic representation leads not only to a more compact description of a system, but also to verification algorithms that are in many cases more efficient, and to more natural notions of system composition.

2.2.1 State variables

The symbolic representation is based on the idea of representing the state of the transition systems via a set of state variables \mathcal{V} . We consider a global (possibly infinite) set \mathcal{W} of variables, and we assume that each variable $x \in \mathcal{W}$ can assume a value in a set $\mathcal{D}(x)$ of values; $\mathcal{D}(x)$ is called the *domain* of x . We will assume that $\mathcal{D}(x)$ is a finite set, restricting our attention to finite-state systems. An important special case is the one in which all variables are *boolean*, that is, their domain is the two-element set $\{\mathbf{F}, \mathbf{T}\}$.

2.2.2 States

Given a finite set $\mathcal{V} \subseteq \mathcal{W}$ of variables, a *state* s over \mathcal{V} is a function that associates with each variable $x \in \mathcal{V}$ a value $s(x) \in \mathcal{D}(x)$; we denote by $\mathcal{S}[\mathcal{V}]$ the set of all possible states over the variables \mathcal{V} . Note that, formally, the type of $s \in \mathcal{S}[\mathcal{V}]$ is $\prod_{x \in \mathcal{V}} (x \mapsto \mathcal{D}(x))$. Given a state $s \in \mathcal{S}[\mathcal{V}]$ and a subset $\mathcal{U} \subseteq \mathcal{V}$ of variables, we denote by $s[\mathcal{U}] \in \mathcal{S}[\mathcal{U}]$ the restriction of s to the variables in \mathcal{U} : precisely, $s[\mathcal{U}]$ is defined by $s[\mathcal{U}](x) = s(x)$ for all $x \in \mathcal{U}$. For any two set \mathcal{V}, \mathcal{U} of variables, and states $s \in \mathcal{S}[\mathcal{V}]$ and $t \in \mathcal{S}[\mathcal{U}]$, we write $s \simeq t$ if $s(x) = t(x)$ for all shared variables $x \in \mathcal{V} \cap \mathcal{U}$.

2.2.3 State predicates

We assume a logical language *Lang* in which assertions about the values of the variables in \mathcal{W} can be written. For example, if all variables are boolean, then *Lang* can be taken to be predicate logic with the addition of

the quantifiers \forall and \exists over the booleans. We say that a formula $\phi \in \text{Lang}$ is *over* a set \mathcal{V} of variables if it only involves variables of \mathcal{V} ; such a formula is also called a *predicate* over \mathcal{V} . We denote by $\text{Preds}[\mathcal{V}]$ the set of all formulas over the set of variable \mathcal{V} . Given a formula ϕ over \mathcal{V} and a state $s \in \mathcal{S}[\mathcal{V}]$, we write $s \models \phi$ to denote the fact that ϕ is true under the interpretation that assigns to every variable $x \in \mathcal{V}$ the value $s(x)$. In particular, a formula ϕ over \mathcal{V} defines the set of states $\llbracket \phi \rrbracket_{\mathcal{V}} = \{s \in \mathcal{S}[\mathcal{V}] \mid s \models \phi\}$.

Example 1. Consider the set of boolean variables $\mathcal{V} = \{x, y, z\}$. The set $\mathcal{S}[\mathcal{V}]$ consists of $2^3 = 8$ elements. If we take Lang to be propositional logic, then the formula $x \wedge \neg y$ is satisfied by the two states $(x = \text{T}, y = \text{F}, z = \text{F}), (x = \text{T}, y = \text{F}, z = \text{T}) \in \mathcal{S}[\mathcal{V}]$. If we take Lang to be quantified boolean formulas, then the formula $\exists w. (w \equiv x \wedge w \equiv \neg y \wedge w \equiv z)$ is satisfied by the two states $(x = \text{T}, y = \text{F}, z = \text{T}), (x = \text{F}, y = \text{T}, z = \text{F}) \in \mathcal{S}[\mathcal{V}]$. ■

2.2.4 Transition predicates

In order to be able to define *relations*, in addition to sets of states, we introduce the following notation. For each state variable x , we introduce a new variable $\circ x$ (read: “next x ”), with $\mathcal{D}(x) = \mathcal{D}(\circ x)$, that denotes the value of the state variable x in the successor state. Given a set $\mathcal{V} \subseteq \mathcal{W}$ of variables, we let $\circ\mathcal{V} = \{\circ x \mid x \in \mathcal{V}\}$ be the corresponding set of next variables. We denote the converse of \circ by \ominus (read: “previous”): precisely, we let $\ominus \circ x = x$ for all variables x . Given a predicate ϕ , we denote by $\circ\phi$ the result of replacing every variable x in ϕ with $\circ x$, and by $\ominus\phi$ the result of replacing every $\circ x$ in ϕ with x ; obviously, $\ominus \circ\phi = \phi$.

Intuitively, in a transition predicate the standard variables refer to the current state, and the next variables refer to the successor state. Given a predicate ρ over $\mathcal{V} \cup \circ\mathcal{U}$, and states $s \in \mathcal{S}[\mathcal{V}]$ and $t \in \mathcal{S}[\mathcal{U}]$, we write $(s, t) \models \rho$ to denote the fact that ρ is true when every $x \in \mathcal{V}$ has value $s(x)$, and every $\circ y \in \circ\mathcal{U}$ has value $t(y)$. A transition predicate $\rho \in \text{Preds}[\mathcal{V}, \neq \mathcal{U}]$ defines a relation

$$\llbracket \rho \rrbracket_{\mathcal{V}, \circ\mathcal{U}} = \{(s, t) \in \mathcal{S}[\mathcal{V}] \times \mathcal{S}[\mathcal{U}] \mid (s, t) \models \rho\}.$$

For brevity, we denote by $\text{Preds}[\mathcal{V}, \circ\mathcal{V}] = \text{TPreds}[\mathcal{V}]$ the set of transition predicates over the set \mathcal{V} of variables.

Example 2. Consider the set of boolean variables $\mathcal{V} = \{x, y\}$. The transition predicate $(\circ x \equiv y) \wedge \neg \circ y$ defines the transition that copies the value of y into x , and sets y to F. ■

2.2.5 Symbolic transition system

A *symbolic transition system* $P = \langle \mathcal{V}_P, \mathcal{A}_P, \theta_P, \text{Trans}_P \rangle$ consists of the following components:

- A set $\mathcal{V}_P \subseteq \mathcal{W}$ of state variables.
- A set \mathcal{A}_P of actions.
- An initial predicate $\theta_P \in \text{Preds}[\mathcal{V}_P]$.
- A mapping $\text{Trans}_P : \mathcal{A}_P \mapsto \text{TPreds}[\mathcal{V}_P]$, associating with each action $a \in \mathcal{A}_P$ a transition predicate $\tau_a \in \text{TPreds}[\mathcal{V}_P]$.

A symbolic transition system defines a regular transition system $P = \langle S_P, s_P^{\text{init}}, \mathcal{A}_P, \text{Steps}_P \rangle$ as follows:

- $S_P = \mathcal{S}[\mathcal{V}]$;
- $s_P^{\text{init}} = \{s \in S_P \mid s \models \theta_P\}$;
- $\text{Steps}_P = \{(s, a, t) \in S_P \times \mathcal{A}_P \times S_P \mid (s, t) \models \text{Trans}_P(a)\}$.

We let $\tau_P = \bigvee_{a \in \mathcal{A}_P} \text{Trans}_P(a)$, so that τ_P is the predicate defining the transition relation ρ_P .

Problem 1. Draw the state-transition graph of the enumerative transition system corresponding to the following symbolic system:

- $\mathcal{V}_P = \{x, y\}$, where x is Boolean (and ranges over $\{\mathbf{T}, \mathbf{F}\}$, and y ranges over $\{0, 1, 2, 3\}$;
- $\mathcal{A}_P = \{a, b, c\}$;
- The transition relations are:

$$\tau_a = (x \wedge y < 3 \rightarrow x' \wedge y' = y + 1) \wedge (\neg x \wedge y > 1 \rightarrow x' \wedge y' = y - 1)$$

$$\tau_b = (x \wedge y = 3 \rightarrow y' = 0)$$

$$\tau_c = (x' \neq x) \quad \blacksquare$$

Problem 2. Give a symbolic representation for the module described informally as follows. The state variables are x, y, z , where x is boolean, and y, z range over the set of integers $\{0, 1, 2, \dots, 9\}$. There are three actions, a, b , and c . Action a is a reset action, and leads to the state $(\mathbf{F}, 0, 0)$ from any other state. Action b can be taken only when x is true; it can either increase y by 1, or increase z by 1, or set x to false. Action c can be taken when x is true and $z < 7$; it increases z by 1. \blacksquare

Chapter 3

Safety and Reachability Verification

A *safety property* specifies a subset of *safe* states of a system, and requires that the system never leaves this subset of safe states. Safety properties are the most common properties in system specification: whenever we can identify in the system error states that indicate malfunction, we can use safety properties to specify that no such malfunction should occur. Most informal simulation and debugging approaches to verification focus primarily on the verification of safety properties.

3.1 Safety Properties

A *safety property* asks that a system does not leave a subset of *safe* states. A transition system satisfies a safety property if all its reachable states are safe.

Definition 1 (safety property). Given a transition system $P = \langle S_P, s_P^{init}, \mathcal{A}_P, Steps_P \rangle$, a *safety property* is a formula $\Box\phi$, where $\phi \subseteq S_P$ is a subset of states of P . We say that P satisfies the property $\Box\phi$, written $P \models \Box\phi$, if $s \in \phi$ for all $s \in Reach(P)$. ■

The “ \Box ” in $\Box\phi$, pronounced “box”, is a *temporal logic* connective that means “always” [19]. Thus, the formula $\Box\phi$ means “always in ϕ ”, in accordance with the above definition. When $P \models \Box\phi$, the property ϕ is called an *invariant* of P , to underline the fact that ϕ holds throughout the behavior of P . The problem of verifying whether $P \models \Box\phi$ is known as the *invari-*

ant verification problem, and it is the most important problem in system verification.

We note that usually, the set ϕ is specified via a state predicate (see Section 2.2.3), so that the formula $\Box\phi$ is a proper logical formula, rather than a formula mixing temporal operators (\Box) with sets (ϕ). However, from our point of view, state predicates and sets of states are interchangeable, and we prefer the set-notation for this chapter, in order to focus on the methods, rather than on the details of the logical languages.

3.2 Invariant Verification and Reachability Analysis

From these definitions, it follows that in order to check whether a transition system P satisfies a safety property $\Box\phi$, it suffices to check whether there are any reachable states that do not satisfy ϕ . Hence, the algorithmic methods for invariant verification that have been proposed are based on the exploration of the state space of the transition system P : the goal is to determine the presence or absence of a path from an initial state of P to $\neg\phi$. The fundamental operators used in the exploration of the state space of P are the *successor operator* post and the *predecessor operator* pre .

Definition 2 (predecessor and successor operators). Given a set of states \mathcal{S} , a state $s \in \mathcal{S}$, and a transition relation $\rho \subseteq \mathcal{S} \times \mathcal{S}$, we let:

$$\begin{aligned}\text{post}(s, \rho) &= \{t \in \mathcal{S} \mid (s, t) \in \rho\} \\ \text{pre}(t, \rho) &= \{s \in \mathcal{S} \mid (s, t) \in \rho\}\end{aligned}$$

the sets of *successors* and *predecessors* of s according to ρ . We lift this notion to sets of states, by taking for $B \subseteq \mathcal{S}$:

$$\text{post}(B, \rho) = \bigcup_{s \in B} \text{post}(s, \rho) \tag{3.1}$$

$$\text{pre}(B, \rho) = \bigcup_{s \in B} \text{pre}(s, \rho). \blacksquare \tag{3.2}$$

Note that (3.1) and (3.2) can also be written more compactly as:

$$\text{post}(B, \rho) = B \circ \rho \qquad \text{pre}(B, \rho) = \rho \circ B .$$

The algorithms for invariant verification can be broadly divided into *enumerative* and *symbolic*. Enumerative algorithms operate on *states* as the

basic entities, and represent sets of states in terms of their individual states. Symbolic algorithms operate on *sets of states* as their basic entities, and use symbolic representations to encode efficiently large sets of states.

3.2.1 Enumerative algorithms

The two simplest algorithms for invariant verification are the *forward search* and *backward search* algorithms. The forward search algorithm searches from the initial states, until it either explores all reachable states, or it reaches a state that satisfies $\neg\phi$. The backward search algorithm is symmetrical: it searches back from the states that satisfy $\neg\phi$, until it either finds a state satisfying the initial condition, or until or states that can lead to $\neg\phi$ have been explored. Both of these algorithms are enumerative.

Algorithm 1 (invariant verification by forward search).

Input: A transition system P , and a predicate ϕ over \mathcal{S}_P .
Output: YES if $P \models \Box\phi$, and NO otherwise.

Initialization: Set the *frontier* F by $F = s_P^{init}$, set the target to $T = \mathcal{S}_P \setminus \phi$ and let the set of already-explored states be $R = \emptyset$.

While $F \neq \emptyset$ **and** $F \cap T = \emptyset$ **do:**

- Pick $s \in F$, and let $F := F \setminus \{s\}$ and $R = R \cup \{s\}$;
- let $F := F \cup (\text{post}(s, \rho_P) \setminus R)$;

Return: YES if $F = \emptyset$, and NO if $F \cap T \neq \emptyset$. ■

Algorithm 2 (invariant verification by backward search).

Input: A transition system P , and a predicate ϕ over \mathcal{S}_P .
Output: YES if $P \models \Box\phi$, and NO otherwise.

Initialization: Set the *frontier* F by $F = \mathcal{S}_P \setminus \phi$, set the target to $T = s_P^{init}$ and let the set of already-explored states be $R = \emptyset$.

While $F \neq \emptyset$ **and** $F \cap T = \emptyset$ **do:**

- Pick $s \in F$, and let $F := F \setminus \{s\}$ and $R = R \cup \{s\}$;
- let $F := F \cup (\text{pre}(s, \rho_P) \setminus R)$;

Return: YES if $F = \emptyset$, and NO if $F \cap T \neq \emptyset$. ■

As we can see, these algorithms for invariant verification take the form of search for counterexample. The behavior of the forward (resp. backward) search algorithm is determined by how the state s is selected from the frontier F . If F is implemented as a stack (LIFO: Last In, First Out), the algorithm performs depth-first search; if F is implemented as a queue (FIFO: First In, First Out), then the algorithm performs breath-first search. If the breath-first approach is chosen, it is also possible to combine these algorithms, alternating forward-search steps from the initial region with backward-steps from $S_P \setminus \phi$. When $P \not\models \square\phi$, such an approach may result in fewer states being explored, compared with forward or backward reachability. These algorithms have many variations: for instance, the exploration of the state space may be performed in a randomized fashion, using a cache to keep track of which states have been visited. For an overview of other approaches, see the list of suggested readings. These forward and backward exploration algorithms are examples of *model-checking* algorithms, that verify a property of a system by exploring its state space.

3.2.2 The *state-explosion problem*

Consider a transition system with state space described by n variables: the size of the state space is 2^n . As this simple example indicates, there is in general an exponential dependency between the number of variables, and the size of the state space. In turn, the number of variables is usually (linearly) proportional to the “size” of the system, measured in terms of how large its design is. For example, in digital hardware we usually require one boolean variable for each 1-bit latch or register; the size of the state space becomes thus exponential in the number of latches and registers. This exponential growth in the size of the state space with a linear increase in the complexity of the design is known as the *state explosion problem*, and it is the foremost limitation to the application of model-checking methods to complex designs. Much research has been devoted to approaches that alleviate the effects of the state explosion problem, pushing forward the size limit for the systems that can be studied by model checking.

3.3 Symbolic Algorithms

Symbolic algorithms operate on *regions* rather than sets of states. Given a set of states S , a *region* σ is a *representation* for a subset $\llbracket\sigma\rrbracket \subseteq S$. For instance, if $S = \mathcal{S}[\mathcal{V}]$ for some set of variables \mathcal{V} , a possible choice is to identify regions with predicates over \mathcal{V} . We denote by Σ the set of all

regions. Note that distinct regions may correspond to the same set of states, that is, for $\sigma_1, \sigma_2 \in \Sigma$ we can have $\sigma_1 \neq \sigma_2$ and $\llbracket \sigma_1 \rrbracket = \llbracket \sigma_2 \rrbracket$. We assume that Σ contains at least regions σ_\emptyset and σ_S , such that $\llbracket \sigma_\emptyset \rrbracket = \emptyset$ and $\llbracket \sigma_S \rrbracket = S$; we assume also that Σ contains all regions that will be of interest in the specification of properties of interest, such as safety properties. Furthermore, we assume that we have at least the following operations defined over regions:

- *Equality and emptiness testing.* We must have tests $RegEq : \Sigma \times \Sigma \mapsto bool$ and $RegIsEmpty : \Sigma \mapsto bool$ such that for all $\sigma_1, \sigma_2 \in \Sigma$ we have (a) $RegEq(\sigma_1, \sigma_2) = \text{T}$ if $\llbracket \sigma_1 \rrbracket = \llbracket \sigma_2 \rrbracket$, and $RegEq(\sigma_1, \sigma_2) = \text{F}$ otherwise; and (b) $RegIsEmpty(\sigma_1) = \text{T}$ if $\llbracket \sigma_1 \rrbracket = \emptyset$, and $RegIsEmpty(\sigma_1) = \text{F}$ otherwise.
- *Union and Intersection.* We must have operators $RegAnd, RegOr : \Sigma \times \Sigma \mapsto \Sigma$, such that for all $\sigma_1, \sigma_2 \in \Sigma$ we have $\llbracket RegAnd(\sigma_1, \sigma_2) \rrbracket = \llbracket \sigma_1 \rrbracket \cap \llbracket \sigma_2 \rrbracket$, and $\llbracket RegOr(\sigma_1, \sigma_2) \rrbracket = \llbracket \sigma_1 \rrbracket \cup \llbracket \sigma_2 \rrbracket$,
- *Difference.* We must have an operator $RegDiff : \Sigma \times \Sigma \mapsto \Sigma$, such that for all $\sigma_1, \sigma_2 \in \Sigma$ we have $\llbracket RegDiff(\sigma_1, \sigma_2) \rrbracket = \llbracket \sigma_1 \rrbracket \setminus \llbracket \sigma_2 \rrbracket$.
- *Post and Pre.* For each transition relation ρ , we must have operators $Pre(\cdot, \rho), Post(\cdot, \rho) : \Sigma \mapsto \Sigma$ such that for all $\sigma_1, \sigma_2 \in \Sigma$ we have $\llbracket Post(\sigma, \rho) \rrbracket = \text{post}(\llbracket \sigma \rrbracket, \rho)$ and $\llbracket Pre(\sigma, \rho) \rrbracket = \text{pre}(\llbracket \sigma \rrbracket, \rho)$.

3.3.1 Symbolic invariant verification

Using regions, and the region operators mentioned above, we can write our first symbolic algorithms for invariant verification.

Algorithm 3 (invariant verification by symbolic forward search).

Input: A transition system P , and a region ϕ over S_P . **Output:** YES if $P \models \Box \llbracket \phi \rrbracket$, and NO otherwise.

Initialization: Set σ_0 so that $\llbracket \sigma_0 \rrbracket = s_P^{init}$, and $\sigma_{-1} = \sigma_\emptyset$.

For $k = 0, 1, 2, \dots$, **repeat:**

If $\neg RegIsEmpty(RegAnd(\sigma_k, \phi))$ return NO;
 If $RegEq(\sigma_k, \sigma_{k-1})$ return YES;
 $\sigma_{k+1} := RegOr(\sigma_k, Post(\sigma_k, \rho_P))$

The correctness of the algorithm stems from the fact that, for all $k \geq 0$, the region σ_k represents the set of states reachable from the initial region in at most k steps. We can also formulate a corresponding algorithm that

performs *backward search*: the algorithm proceeds backwards from the complement of the invariant $\neg\phi$, checking whether the initial condition can be reached:

Algorithm 4 (invariant verification by symbolic backward search).

Input: A transition system P , and a region ϕ over S_P . **Output:** YES if $P \models \Box[\phi]$, and NO otherwise.

Initialization: Set $\sigma_0 = \phi$, $\sigma_{-1} = \sigma_\emptyset$, and set σ_θ so that $\llbracket\sigma_\theta\rrbracket = s_P^{init}$. **For** $k = 0, 1, 2, \dots$, **repeat:**

If $\neg\text{RegIsEmpty}(\text{RegAnd}(\sigma_k, \sigma_\theta))$ return NO;
 If $\text{RegEq}(\sigma_k, \sigma_{k-1})$ return YES;
 $\sigma_{k+1} := \text{RegOr}(\sigma_k, \text{Pre}(\sigma_k, \rho_P))$

3.3.2 Improving the algorithms

Algorithm 3 can be further improved by observing that, when computing $\text{Post}(\sigma_k, \rho_P)$, we have in fact already computed $\text{Post}(\sigma_{k-1}, \rho_P) \subseteq \sigma_k$. Hence, the only portion of σ_k from which it is interesting to explore the successors is $\text{RegDiff}(\sigma_k, \sigma_{k-1})$, and we should replace the computation of $\text{Post}(\sigma_k, \rho_P)$ with that of $\text{Post}(\text{RegDiff}(\sigma_k, \sigma_{k-1}), \rho_P)$. Based on this idea, we obtain the following algorithm.

Algorithm 5 (invariant verification by symbolic forward search, improved).

Input: A transition system P , and a region ϕ over S_P . **Output:** YES if $P \models \Box[\phi]$, and NO otherwise.

Initialization: Set η so that $\llbracket\eta\rrbracket = s_P^{init}$, and $\sigma := \sigma_\emptyset$.

While $\text{RegIsEmpty}(\text{RegAnd}(\eta, \phi)) \wedge \neg(\text{RegIsEmpty}(\text{RegDiff}(\eta, \sigma)))$, **repeat:**

$\eta' := \text{RegDiff}(\eta, \sigma)$
 $\sigma := \text{RegOr}(\eta, \sigma)$
 $\eta := \text{Post}(\eta', \rho_P)$

Return: NO if $\neg\text{RegIsEmpty}(\text{RegAnd}(\eta, \phi))$, and YES otherwise. ■

Algorithm 4 can be improved in similar fashion.