Verification and Validation Techniques

Lecture Notes

Luca Simonetti

Università degli Studi di Udine · A.A. 2025–2026

Contents

I Automata, Infinite Words, and Games 8

1 Finite-State Automata and Regular Languages 9
1.1 Alphabets, Words, and Languages . . . 9
1.2 Deterministic Finite Automata . . . 11
1.2.1 Completeness and Sink States . . . 13
1.3 Non-Deterministic Finite Automata . . . 14
1.4 Determinizing an NFA . . . 16
1.4.1 Regular Languages . . . 17
Notation Introduced in This Chapter . . . 19
Exercises . . . 19

2 Regular Languages and Decision Problems 20
2.1 Regular Expressions and Kleene's Theorem . . . 20
2.1.1 Regular Expressions . . . 20
2.1.2 The ε-NFA Model . . . 22
2.1.3 Kleene's Theorem . . . 26
2.2 Closure Properties . . . 28
2.2.1 Complementation . . . 28
2.2.2 Intersection . . . 29
2.3 Decision Problems . . . 30
2.3.1 Reachability as a Fixpoint . . . 32
2.4 The Myhill–Nerode View . . . 32
Notation Recap . . . 35
Exercises . . . 35

3 Automata over Infinite Words 37
3.1 Infinite Words . . . 38
3.2 From Finite Languages to Infinite Languages . . . 39
3.3 Büchi Automata . . . 39
3.4 Closure and Expressions . . . 41
3.5 Emptiness and Lassos . . . 43
Notation Recap . . . 45
Exercises . . . 45

4 Complementation of omega-Regular Languages 46
4.1 Why the Finite Trick Fails 47
4.2 Congruences and Saturation 47
4.3 The Automaton Congruence 49
4.4 Ramsey Decomposition 51
4.5 Closure under Complement 53
4.6 Model Checking Consequence 54
Exercises 55

5 Synthesis and Infinite Games 57
5.1 From Verification to Synthesis 57
5.2 The Game View 59
5.3 Automata-Theoretic Synthesis Pipeline 61
5.4 Reachability Games and Attractors 63
5.5 Müller and Parity Games 65
5.6 Formal Verification in Practice 68
Notation Recap 69
Exercises 70

6 Planning as Model Checking 71
6.1 Action-Based and Timeline-Based Planning 72
6.2 Planning Domains 72
6.3 Plans as Execution Structures 74
6.4 Preimage Operators 76
6.5 Backward Fixpoint Algorithms 77
6.6 Symbolic Representation 79
6.7 Action Description Languages 80
6.8 Towards Model Checking 81
Exercises 82

II Model Checking, Temporal Logic, and Verification Tools 83

7 Reactive Systems, SMV, and Explicit Model Checking 84
7.1 Why Reactive Systems? 84
7.1.1 Closed vs. Open Systems 85
7.1.2 The cost of failure 85
7.2 The Model Checking Problem 86
7.2.1 What model checking is 86
7.2.2 The quality of model and specification 87
7.2.3 Universal and existential model checking 87
7.2.4 A brief historical note 88
7.2.5 Verification vs. validation 89
7.2.6 The state-explosion problem 89
7.3 Kripke Structures 89
7.4 Specifications: Safety and Liveness 91
7.4.1 Two kinds of “always good” 91
7.5 Counterexamples: From Bugs to Witnesses 93
7.5.1 Finite prefixes for safety violations 93
7.5.2 Lassos for liveness violations 93
7.6 The SMV Modelling Language 94
7.6.1 State variables and types 95

7.6.2 The `ASSIGN` block . . . 95
7.6.3 Expression syntax . . . 95
7.6.4 The `case` expression . . . 95
7.6.5 Formal semantics of SMV . . . 96
7.6.6 Adding a self-loop: implicit modelling with `TRANS` . . . 97
7.6.7 Integer-range example: modulo-4 counter . . . 98
7.6.8 Non-determinism in SMV . . . 98
7.6.9 Input variables . . . 99
7.6.10 The `define` macro . . . 100
7.7 Simulation with NuXMV . . . 100
7.8 Modules: Hierarchical Modelling . . . 101
7.8.1 The mechanical decimal counter . . . 102
7.8.2 Properties of the mechanical counter . . . 102
7.9 Asynchronous Composition . . . 103
7.9.1 Synchronous vs. asynchronous . . . 103
7.9.2 Producer–consumer example . . . 104
7.10 Fairness Constraints . . . 105
7.10.1 Temporal hierarchy: four key concepts . . . 105
7.10.2 Formal definitions . . . 106
7.10.3 Encoding fairness in NuSMV . . . 107
7.10.4 Fair paths and model checking under fairness . . . 107
7.11 LTL, CTL, and CTL*: an Overview . . . 108
7.12 Formal Methods in Context . . . 109
7.12.1 Three categories of formal methods . . . 109
7.12.2 Static analysis and abstract interpretation . . . 109
7.12.3 The verification–synthesis spectrum . . . 109
Exercises . . . 110

8 S1S, the Büchi Theorem, and Deterministic Büchi Automata 112
8.1 The Time-Flow Perspective . . . 113
8.2 The Logic S1S . . . 113
8.2.1 Simple example: every a is followed by a b . . . 113
8.2.2 $\mathrm{S1S}_A$: S1S over an alphabet . . . 114
8.2.3 Semantics: words as structures . . . 115
8.3 What S1S Can Express . . . 115
8.3.1 Defining $x < y$ using only $+1$ . . . 115
8.3.2 Parity: positions of A in even locations . . . 116
8.3.3 Counting between consecutive occurrences . . . 117
8.3.4 The key pattern: infinitely many occurrences . . . 118
8.4 Free Variables and the Encoding $\{0,1\}^n$ . . . 118
8.4.1 Why free variables complicate things . . . 119
8.4.2 Words as value assignments . . . 119
8.5 The Büchi Theorem . . . 120
8.6 Proof: ω-regular $\Rightarrow$ S1S-definable . . . 120
8.6.1 The main idea: encode a run as a family of sets . . . 120
8.6.2 The three conditions . . . 121
8.7 Closure under Projection . . . 121
8.8 Proof: S1S-definable $\Rightarrow$ ω-regular . . . 122
8.8.1 Step 1: S1S $\rightarrow$ $\mathrm{S1S}_0$ (syntactic simplification) . . . 122
8.8.2 Step 2: $\mathrm{S1S}_0$ $\rightarrow$ NBA (base cases and closure) . . . 123
8.9 Decidability of S1S . . . 124
8.9.1 Complexity: the non-elementary lower bound . . . 125

8.10 Extensions of S1S . . . 125
8.10.1 S1S over finite words . . . 125
8.10.2 WS1S: Weak Monadic Second-Order Theory . . . 126
8.10.3 Decidable extensions . . . 126
8.10.4 Undecidable extensions . . . 126
8.11 Deterministic Büchi Automata: Limitations . . . 127
8.11.1 Definition . . . 127
8.11.2 What DBAs can and cannot do . . . 127
8.11.3 The witness pair: infinitely many vs. finitely many a's . . . 128
8.11.4 Why L_{fin} has no DBA . . . 128
8.11.5 Tracing the NBA for L_{fin} on $abab\cdots$ . . . 129
8.12 Towards More Expressive Deterministic Automata . . . 129
Exercises . . . 130

9 Deterministic Omega-Automata and Star-Free Languages 132
9.1 The Quest for Determinism in Infinite Verification . . . 132
9.1.1 An Intuitive Example: Non-Deterministic Guessing vs. Real-Time Control . . . 133
9.1.2 Why the Subset Construction Fails on Büchi Automata 133
9.2 Deterministic Büchi Automata (DBA) and the Limit of Vectorial Closure . . . 135
9.2.1 Vectorial Closure . . . 135
9.2.2 The DBA Characterization Theorem . . . 136
9.2.3 The Limit of DBAs: The Finitely Many a's Language . . . 137
9.3 Deterministic Muller Automata (DMA) and McNaughton's Theorem . . . 139
9.3.1 Muller Acceptance Condition . . . 140
9.3.2 Closure Properties of DMAs . . . 141
9.3.3 The DMA Characterization Lemma . . . 142
9.3.4 McNaughton's Theorem and the Closure Bottleneck . . . 142
9.3.5 Corollary: Weak S1S (WS1S) = S1S . . . 144
9.4 Deterministic Rabin Automata, Parity, and Safra's Construction 145
9.4.1 Rabin Acceptance Condition . . . 145
9.4.2 Parity Acceptance and the Equivalence DMA ≡ DRA ≡ Parity . . . 146
9.4.3 Safra's Determinization (Roadmap) . . . 147
9.5 The First-Order Fragment of S1S and Star-Free Languages . . . 148
9.5.1 The Signature FO-S1S[<] . . . 148
9.5.2 Star-Free Languages . . . 149
9.5.3 Translation: Star-Free $\Longrightarrow$ FO-S1S[<] . . . 149
9.5.4 Translation: FO-S1S[<] $\Longrightarrow$ Star-Free (McNaughton-Papert) 150
9.6 Exercises . . . 152

10 Tree Languages and Tree Automata 153
10.1 Introduction to Trees and Branching Time . . . 153
10.1.1 Unfolding a Transition System . . . 154
10.1.2 Formalizing Trees: Domains and Valuations . . . 154
10.1.3 Paths, Subtrees, and Frontiers . . . 155
10.1.4 Simplifying Focus: Binary Coding of K-ary Trees . . . 156
10.2 Regular Tree Languages and Concatenation . . . 157
10.2.1 Tree Concatenation (Substitution) . . . 157
10.2.2 Tree Kleene Star (Iterated Substitution) . . . 159

10.2.3 Regular Tree Languages 160
10.2.4 Tuple Concatenation and Omega-Closure 161
10.2.5 Concatenation of Infinite Trees 162
10.3 Tree Automata on Finite Trees 163
10.3.1 Top-Down Tree Automata 163
10.3.2 The Expressive Limit of Deterministic Top-Down Automata 164
10.3.3 Bottom-Up Tree Automata 166
10.3.4 Bottom-Up Determinization (Subset Construction) . . . 167
10.3.5 Decision Problems and Closure Properties 169
10.4 Automata on Infinite Trees 170
10.4.1 Büchi Tree Automata 171
10.4.2 The Complementation Barrier for Büchi Tree Automata 173
10.4.3 Rabin Tree Automata 174
10.4.4 Rabin's Decidability Theorem and S2S 175
10.4.5 Weak S2S Characterization 177
10.4.6 From Tree Automata to Temporal Logics 178
Exercises 179

11 Linear Temporal Logic 180
11.1 The Logic of Linear Time 181
11.1.1 Modal Logic Foundations 181
11.1.2 Verification vs. Validation 182
11.1.3 Historical Motivation: Classic Software and Hardware Bugs 182
11.1.4 Broader AI Applications of LTL 182
11.1.5 Dimensions and Classification of Temporal Logics . . . 182
11.2 Formal Syntax and Semantics of LTL 183
11.2.1 Derived Temporal Operators 184
11.3 Equivalences and Temporal Expansions 184
11.3.1 Duality of Next and Eventually 184
11.3.2 Expansion (Unrolling) Rules 185
11.3.3 Concrete Trace Evaluations 185
11.3.4 Declarative vs. Operational Ordering Blow-Up 187
11.4 Expressive Power and Boundaries 188
11.4.1 Kamp's Theorem 188
11.4.2 The Modulo Counting Barrier 188
11.5 Linear Temporal Logic with Past 188
11.5.1 Syntax and Semantics 188
11.5.2 Gabbay's Equivalence vs. Succinctness 189
11.6 LTL Satisfiability: The Tableau Construction 191
11.6.1 Negation Normal Form (NNF) 191
11.6.2 Classification of Formulas: Alpha and Beta Formulas . 191
11.6.3 Closure and Atoms 192
11.6.4 Tableau Graph 193
11.7 Fulfilling Paths and SCS Reduction 193
11.7.1 Induced Paths and the Infinite Postponement Problem 193
11.7.2 Fulfilling Paths 194
11.7.3 Reduction to Reachable Strongly Connected Subgraphs 194
11.7.4 Tableau Pruning Rules 195
11.7.5 Detailed Case Study: $\phi = \Box P \wedge \Diamond\neg P$ 195
11.8 Automata-Theoretic Translation 196
Exercises 197

12 LTL Model Checking and Fairness **199**
12.1 P-Validity and P-Satisfiability . . . 199
12.1.1 The Behavior Graph Construction . . . 200
12.2 Direct Algorithms and SCC Search . . . 202
12.2.1 On-the-fly emptiness checking . . . 202
12.2.2 Strongly connected components . . . 203
12.2.3 Complexity . . . 203
12.3 Fairness . . . 204
12.3.1 Transition-Based Justice and Compassion . . . 204
12.3.2 The Adequate Subgraph . . . 204
12.3.3 Reading a fair counterexample . . . 205
12.4 Summary . . . 205
Exercises . . . 206

13 CTL and Symbolic Model Checking **207**
13.1 Branching Time and CTL . . . 207
13.1.1 Why branching time . . . 207
13.1.2 Syntax . . . 208
13.1.3 Semantics . . . 208
13.1.4 The microwave oven example . . . 209
13.1.5 The CTL restriction . . . 210
13.2 Explicit State CTL Model Checking . . . 211
13.3 CTL* and Expressiveness . . . 211
13.3.1 Positioning CTL* . . . 211
13.3.2 Why CTL and LTL are incomparable . . . 212
13.3.3 The satisfiability boundary . . . 212
13.4 Explicit and Symbolic Model Checking . . . 212
13.4.1 The state-space explosion . . . 212
13.4.2 Symbolic transition systems . . . 213
13.4.3 Symbolic reachability . . . 215
13.5 Bounded Model Checking and OBDDs . . . 215
13.5.1 Bounded model checking . . . 215
13.5.2 From binary decision trees to OBDDs . . . 216
13.5.3 The restrict function . . . 218
13.5.4 Why ordering matters . . . 219
13.6 CTL by Predicate Transformers and Fixed Points . . . 219
13.6.1 Predecessor transformers . . . 219
13.6.2 Fixed-point characterisations . . . 219
13.6.3 The iteration view . . . 220
13.6.4 A worked reading of the formulas . . . 221
13.6.5 Worked derivation: *AF* as a least fixed point . . . 221
13.7 Summary . . . 222
Exercises . . . 222

14 Learning Regular Languages **224**
14.1 The Myhill–Nerode Foundation . . . 225
14.1.1 The equivalence relation . . . 225
14.1.2 Characterizing regular languages . . . 226
14.1.3 Examples of equivalence classes . . . 226
14.2 Passive vs. Active Learning . . . 227
14.2.1 Passive learning and its limitations . . . 227
14.2.2 Active learning: The MAT framework . . . 227

14.3 Angluin's L^* Algorithm . . . 229
14.3.1 The observation table structure . . . 229
14.3.2 Table properties: Minimality and Completeness . . . 230
14.3.3 DFA Construction . . . 230
14.3.4 The L^* loop . . . 231
14.4 Walkthrough with a Concrete Example . . . 231
14.4.1 Iteration 1 . . . 232
14.4.2 Iteration 2 . . . 232
14.4.3 Iteration 3 . . . 233
14.4.4 Iteration 4 . . . 234
14.5 Correctness, Termination, and Complexity . . . 235
14.5.1 Complexity bounds . . . 235
14.5.2 Proof of correctness and termination . . . 235
Exercises . . . 237

15 Security Protocol Verification with Tamarin **238**
15.1 From State Machines to Protocol Histories . . . 239
15.1.1 Case study: Designing a secure key exchange . . . 239
15.2 Multiset Rewriting: Terms, Facts, and Rules . . . 241
15.2.1 Terms . . . 241
15.2.2 Facts . . . 242
15.2.3 Rewrite rules . . . 242
15.2.4 A concrete toy protocol . . . 243
15.2.5 The Dolev-Yao Attacker Rules . . . 244
15.3 Protocol Roles and Traces . . . 245
15.3.1 A role-level view of the toy handshake . . . 245
15.3.2 Traces . . . 246
15.3.3 A classical protocol pattern . . . 247
15.4 Equational Theories and Unification . . . 247
15.4.1 Semantic Equivalence in Attacker Knowledge . . . 247
15.4.2 The Unification Problem . . . 247
15.4.3 Subterm-Convergent Theories . . . 248
15.4.4 Non-Subterm-Convergent Theories and Undecidability 248
15.5 Property Verification in Tamarin . . . 249
15.5.1 Secrecy . . . 249
15.5.2 Authentication and correspondence . . . 249
15.5.3 Reachability and existence . . . 250
15.5.4 Forward Secrecy . . . 250
15.5.5 Anonymity and Observational Equivalence . . . 251
15.5.6 The modelling discipline . . . 252
15.6 Summary . . . 252
Exercises . . . 253

PART I

Automata, Infinite Words, and Games

CHAPTER 1

Finite-State Automata and Regular Languages

Verification starts with a simple modelling decision. We describe a system by listing the situations it can be in, and the events that move it from one situation to another. This gives us a directed graph with a finite set of nodes, but with enough structure to process input words. That structure is a **finite-state automaton**.

Automata are the first modelling language of the course: states are the possible configurations of a system, and transitions are the steps it can take.

This chapter builds the basic language. We begin with alphabets and words, define deterministic and non-deterministic finite automata, and prove the central fact that non-determinism does not add expressive power for finite words.

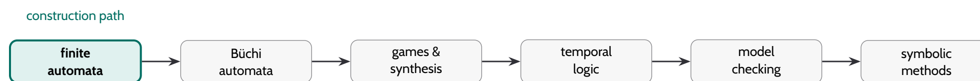


Figure 1.1: Where this chapter sits in the construction path of the book. Each step reuses the previous one as a representation or an algorithmic primitive.

Why start with finite words? Real-world systems like operating systems or network protocols often run continuously, effectively generating infinite *words. We build the foundations on finite sequences first, but will later extend this theory to infinite words to model systems that do not terminate.*

1.1 Alphabets, Words, and Languages

Before an automaton can read anything, we need to say what its inputs look like. For a turnstile, the relevant observations might be `coin` and `push`; for a tiny parity example, they might be just a and b. In both cases we first fix the symbols that may appear, and then we study finite sequences of those symbols.

> **Definition 1.1.1** (Alphabet and word)**.** An **alphabet** is a finite non-empty set of symbols, usually denoted by A.
>
> A **word** over A is a finite sequence of symbols from A. The set of all finite words over A is denoted by A^*.

Think of A as the vocabulary available to the automaton. If $A = \{a, b\}$, then aab, $baba$, and bb are words. The special word with no symbols is the **empty word**, written ε. It has length 0.

Example 1.1.2 (A small alphabet)**.** Let $A = \{\texttt{coin}, \texttt{push}\}$. A turnstile can observe words such as

$$\texttt{coin push push coin}.$$

The symbols are events. The word is the finite history of observed events.

Definition 1.1.3 (Language)**.** A **language** over A is any set of finite words:

$$L \subseteq A^*.$$

A language can be understood as a way of assigning a property to words: some words satisfy the property, while others do not. For example, over the alphabet $A = \{a, b\}$, we can form words such as a, b, ab, ba, and $abba$. The property "ends in a" selects exactly those words whose last symbol is a. Thus, the corresponding language is the set of all and only the words satisfying that property:

$$L = \{w \in A^* \mid w \text{ ends in } a\}.$$

In practice, the languages we care about are usually infinite. A language is still just a set of finite words, but it is typically an infinite subset of A^*.

In verification, this viewpoint is useful because a word can represent a behaviour, that is, a sequence of states, events, or actions. A language can therefore describe the set of behaviours allowed by a model, or the set of behaviours permitted by a specification. Saying that a behaviour belongs to the language means that it satisfies the property described by that language.

From the automata perspective, a language is something a machine can *accept*. An automaton reads an input word and decides whether that word belongs to the language. The languages accepted by finite automata are called **regular languages**. Later in the chapter we will see that deterministic and non-deterministic finite automata recognize the same class of languages, so "regular" is the right name for the whole finite-state setting.

Finite automata are only one presentation of regular behaviour. Later we will see regular expressions as an equivalent syntax for describing the same class of languages.

Definition 1.1.4 (Basic operations on words)**.** If $u, v \in A^*$, their **concatenation** is the word uv obtained by writing v immediately after u. The **length** of w is denoted $|w|$.

Concatenation is not commutative in general: usually $uv \neq vu$. It is also length-additive: $|uv| = |u| + |v|$.

The empty word behaves like a neutral element:

$$\varepsilon w = w\varepsilon = w.$$

This small fact is useful because many constructions need to handle the case where no input has been consumed yet.

The same operation extends from words to languages. If $L_1, L_2 \subseteq A^*$, then

$$L_1 L_2 = \{uv : u \in L_1,\ v \in L_2\}.$$

There is a small but important distinction here:

$$L\{\varepsilon\} = L, \qquad L\emptyset = \emptyset.$$

The singleton $\{\varepsilon\}$ contains one word, namely the empty word. The empty language $\emptyset$ contains no word at all.

Definition 1.1.5 (Kleene star)**.** If $W \subseteq A^*$ is a language, then

$$W^* = \{w_1 w_2 \cdots w_k : k \geq 0,\ w_i \in W \text{ for all } i\}$$

is the **Kleene closure** of W.

In plain language, W^* contains every word that can be built by concatenating finitely many blocks from W. The case $k = 0$ gives ε, so $\varepsilon \in W^*$ for every W.

It is often useful to name the finite powers explicitly:

$$W^0 = \{\varepsilon\}, \qquad W^{n+1} = WW^n, \qquad W^* = \bigcup_{n \geq 0} W^n, \qquad W^+ = \bigcup_{n \geq 1} W^n.$$

For example, if $W = \{a, b\}$ and we want to describe words containing at least one symbol, then W^+ is the natural choice. Using W^* would also include the empty word ε, which would have to be treated as a special case.

Example 1.1.6 (Words from the alphabet $\{a\}$)**.** Let $A = \{a\}$. Then $A^* = \{\varepsilon, a, aa, aaa, \ldots\}$ is the set of all finite sequences of a's. A^+ on the other hand is $\{a, aa, aaa, \ldots\}$, which excludes the empty word.

1.2 Deterministic Finite Automata

A deterministic automaton has no choices. Once we know the current state and the next input symbol, the next state is fixed. Before writing the general definition, let us build one concretely.

Example 1.2.1 (Words with an even number of a's)**.** Over $A = \{a, b\}$, consider the language of words containing an even number of occurrences of a. To decide membership we only need to track the parity of the number of a's seen so far. This suggests two states,

$$Q = \{q_{\text{even}}, q_{\text{odd}}\},$$

with a toggling between them and b leaving the state unchanged. The initial state is q_{even}, and it is also the only accepting state, because zero is even.

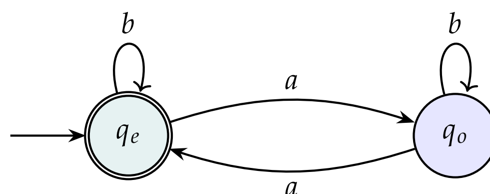


Figure 1.2: A DFA for words with an even number of a's.

In fig. 1.2, the state records only the information that matters for the property: parity. It forgets the exact number of a's, because the language does not need that much detail. The picture already contains every ingredient of a deterministic automaton: a finite set of states, an alphabet of inputs, a rule that gives the next state, a designated starting state, and a set of accepting states. We now formalize them.

Definition 1.2.2 (Deterministic finite automaton)**.** A **deterministic finite automaton** (DFA) is a tuple

$$\mathcal{A} = (Q, A, \delta, q_0, F)$$

where:
- Q is a finite set of states;
- A is a finite alphabet;
- $\delta : Q \times A \to Q$ is the transition function;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of accepting states.

The tuple separates the static shape of the automaton from its behaviour. The set Q tells us which configurations exist. The transition function δ tells us how the configuration evolves while reading a word. The accepting set F tells us which final states count as success.

Using a single initial state is mostly a convenience. Variants with several initial states can be converted into this form without changing the recognized language.

Example 1.2.3 (Every a is eventually followed by a b)**.** Over $A = \{a, b\}$, consider the language of all and only those words such that each occurrence of a is followed later by an occurrence of b:

$$L = \{w \in \{a, b\}^* : \text{for every position } i,\ w_i = a \Rightarrow \exists j > i \text{ with } w_j = b\}.$$

The word ε belongs to L, and so does every word made only of b's, because there is no occurrence of a to check. The word $aaab$ also belongs to L: the final b witnesses all three pending a's at once. By contrast, a, aba, and baa do not belong to L, because their last a has no later b.

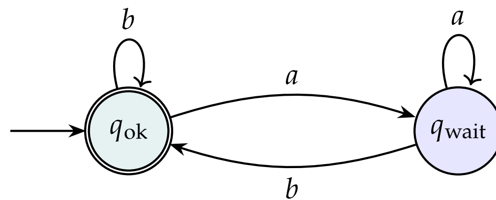


Figure 1.3: A DFA that remembers only whether there is an unresolved a.

The state q_{wait} means "some a has been seen and no later b has appeared yet". The automaton does not count the a's and the b's. It remembers only one bit of information: whether there is currently an unresolved a. This is exactly the finite-memory point.

The contrast with balanced parentheses is important. There, each opening parenthesis must be matched by the corresponding later closing parenthesis, possibly after nested openings. That requires an unbounded stack of pending openings, so the language is not regular; it belongs to the world of pushdown automata.

Definition 1.2.4 (Extended transition function)**.** The transition function extends from symbols to words by defining $\widehat{\delta} : Q \times A^* \to Q$ recursively:

$$\widehat{\delta}(q, \varepsilon) = q, \qquad \widehat{\delta}(q, wa) = \delta(\widehat{\delta}(q, w), a).$$

This simply says: to read the word wa, first read w, then process the last symbol a from the state reached after w. The sequence of states visited while reading a word is called a **computation** of the automaton.

Definition 1.2.5 (Language of a DFA)**.** The language accepted by a DFA $\mathcal{A} = (Q, A, \delta, q_0, F)$ is

$$L(\mathcal{A}) = \{w \in A^* : \widehat{\delta}(q_0, w) \in F\}.$$

So acceptance is a final-state test. The automaton consumes the whole word; if the unique resulting state is accepting, the word belongs to the language.

1.2.1 Completeness and Sink States

Sometimes automata are drawn with some transitions omitted. This is often done to keep the picture simple, but formally a DFA must have a *total* transition function. This means that for every state q and every input symbol $a \in A$, the transition $\delta(q, a)$ must be defined.

In a formal DFA, every pair (q, a) has exactly one next state because δ is a total function. In drawings, however, authors often omit some arrows. The issue is then not determinism but how to complete the picture.

An omitted transition should therefore be understood as a transition that goes to a rejecting "dead" or "sink" state. Once the automaton enters this sink state, it stays there forever, no matter which input symbol is read. Thus, any partially drawn DFA can be made into a formally correct DFA by adding a sink state and redirecting all missing transitions to it.

Definition 1.2.6 (Sink completion)**.** Given a partial deterministic automaton, its **completion** adds a fresh non-accepting sink state $\bot$. Every missing transition goes to $\bot$, and every symbol loops from $\bot$ back to $\bot$.

A missing transition is therefore not a self-loop. Replacing omitted arrows by self-loops can change the language. To read this definition, ask what the automaton should do when the drawing gives no arrow for the next input symbol. The operational test is simple: every missing case is sent to the rejecting state $\bot$, and once $\bot$ is reached, the word cannot be accepted.

Example 1.2.7 (Why omitted transitions are not self-loops)**.** Let $A = \{a, b\}$. Consider the partially drawn automaton with two states:

$$q_0 \xrightarrow{a} q_1, \qquad q_1 \in F.$$

No other transition is drawn. With sink completion, this automaton recognizes exactly the one-letter word a: any unexpected symbol, such as the first b in ba or the final b in ab, sends the computation to $\bot$ and the word is rejected.

If instead we completed the missing transitions by self-loops, the language would be different. The word ba would be accepted, because the missing b-transition from q_0 would keep the automaton in q_0, and the following a would move it to q_1. The word ab would also be accepted, because the missing b-transition from the accepting state q_1 would keep the automaton accepting. This is not the intended meaning of an omitted transition.

Figure 1.4 shows the completed version of the partial automaton from theorem 1.2.7. The sink state means "we have already violated the pattern". Once the automaton enters it, no future input can make the word acceptable again. This completion matters later, because complementation of deterministic automata works only when the automaton is complete.

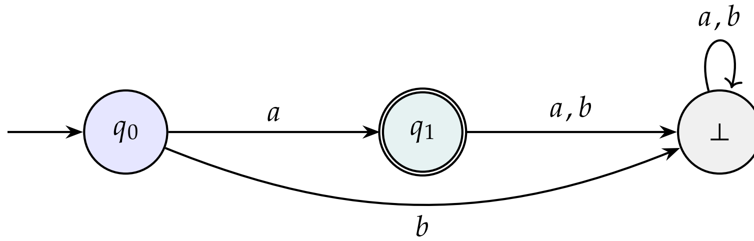


Figure 1.4: A concrete sink completion: every missing transition goes to the rejecting state $\perp$, and $\perp$ loops on every input symbol.

1.3 Non-Deterministic Finite Automata

Non-determinism changes the transition structure. The automaton may have several possible next states for the same state and input symbol. For example, while scanning a word, an automaton might keep reading as usual, or it might guess that the current symbol is the important one. The formal definition below records all such possible moves instead of choosing one immediately.

> **Definition 1.3.1** (Non-deterministic finite automaton)**.** A **non-deterministic finite automaton** (NFA) is a tuple
>
> $$\mathcal{A} = (Q, A, \Delta, q_0, F)$$
>
> where Q, A, q_0, and F are as for a DFA, and
>
> $$\Delta \subseteq Q \times A \times Q$$
>
> is a transition relation.

Equivalently, we can view the transition structure as a function

$$\Delta : Q \times A \to 2^Q.$$

A DFA is just the special case where every pair (q, a) has at most one outgoing transition. In an NFA, the same pair may lead to several possible next states.

The value $\Delta(q, a)$ is the set of all states that the automaton may move to after reading a in state q.

For computations it is convenient to extend this to sets of states. If $S \subseteq Q$, define

$$\widehat{\Delta}(S, \varepsilon) = S, \qquad \widehat{\Delta}(S, wa) = \bigcup_{p \in \widehat{\Delta}(S,w)} \Delta(p, a).$$

Starting from a single state q, we write $\widehat{\Delta}(q, w)$ for $\widehat{\Delta}(\{q\}, w)$. The set $\widehat{\Delta}(q_0, w)$ contains exactly the states reachable by some run on w.

> **Example 1.3.2** (Computing reachable states)**.** Consider the NFA over $A = \{a, b\}$ shown in fig. 1.5. Its initial state is q, and we use it only to compute reachability, so no accepting states are marked.

Let us compute $\widehat{\Delta}(q, bba)$. After the first b,

$$\widehat{\Delta}(q, b) = \{q_2, q_4\}.$$

From q_2 another b reaches q_1, while from q_4 another b reaches both q and q_5. Hence

$$\widehat{\Delta}(q, bb) = \{q, q_1, q_5\}.$$

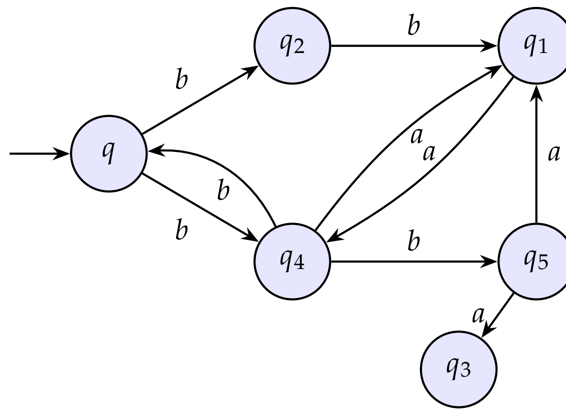


Figure 1.5: A small NFA used to compute $\widehat{\Delta}(q, bba)$ step by step.

The final a kills the branch currently in q, because q has no outgoing a-transition. The branch in q_1 reaches q_4, and the branch in q_5 reaches both q_1 and q_3. Therefore

$$\widehat{\Delta}(q, bba) = \{q_1, q_3, q_4\}.$$

Definition 1.3.3 (Run and acceptance for an NFA)**.** Let $w = a_0 a_1 \cdots a_{n-1}$. A **run** of an NFA on w is a sequence of states

$$r_0, r_1, \ldots, r_n$$

such that $r_0 = q_0$ and $(r_i, a_i, r_{i+1}) \in \Delta$ for every $0 \le i < n$.

The word w is accepted if there exists at least one run on w with $r_n \in F$.

This is the set-based view of non-determinism: we do not choose one branch and forget the others. At each prefix, we collect all states that are reachable by at least one run.

The key word is *exists*. An NFA may have many failing runs on the same input. The word is accepted as soon as one run reaches an accepting state after consuming the whole word. Equivalently,

$$w \in L(\mathcal{A}) \iff \widehat{\Delta}(q_0, w) \cap F \neq \emptyset.$$

Other acceptance choices are possible in principle, but the standard NFA semantics is existential: one successful computation is enough.

Example 1.3.4 (Guessing the last symbol)**.** Over $A = \{a, b\}$, the language of words whose last symbol is a can be recognized by a small NFA that stays in a scanning state, and whenever it reads a, guesses that this might be the last symbol. That guess moves to an accepting state with no outgoing transitions.

This example illustrates why non-determinism is convenient: the automaton does not need to know in advance which a is the last one. It can try all possibilities at once.

Notice the visual difference between fig. 1.2 and fig. 1.6. The DFA has exactly one outgoing transition for each symbol at each state. The NFA has two possible a-moves from q_s: keep scanning, or guess that this a is final. If you draw a second outgoing arrow labeled a from the same state to another state, you are already outside the DFA world, because one state-symbol pair would have two possible targets.

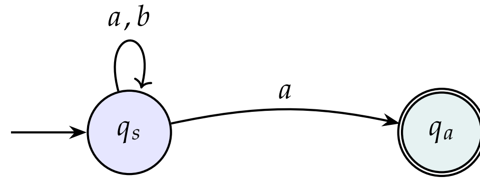


Figure 1.6: An NFA for words ending in a. The transition to q_a is the guess that the current a is the last symbol.

1.4 Determinizing an NFA

Every DFA is trivially an NFA: if the transition function δ is total, we can view it as a relation Δ in which each pair (q, a) has exactly one successor. So the class of languages recognizable by DFAs is contained in the class recognizable by NFAs.

The interesting question is the converse. Are there languages that an NFA can recognize but no DFA can? Does non-determinism genuinely add expressive power, or does it only add design convenience?

The answer is that it adds only convenience. If we restrict ourselves to DFAs, we do not lose any language.

Theorem 1.4.1 (Subset construction)**.** *For every NFA $\mathcal{A}$ there exists a DFA $\mathcal{D}$ such that*

$$L(\mathcal{D}) = L(\mathcal{A}).$$

■ **Formal details — Construction**

Let $\mathcal{A} = (Q, A, \Delta, q_0, F)$ be an NFA. We build a DFA $\mathcal{D} = (2^Q, A, \delta_D, \{q_0\}, F_D)$ whose states are subsets of Q. Each such subset S represents the set of NFA states that could simultaneously be active after reading some input prefix — the DFA keeps track of all possibilities at once rather than choosing one. (Some presentations write the DFA components as Q', Δ', q_0', F' and use P for a generic state; the objects are identical, only the names differ.)

Initial state. Before any input is consumed, the only active NFA state is q_0, so the DFA starts in the singleton $\{q_0\} \in 2^Q$.

Transition function. When the DFA is in state S and reads symbol a, it must account for every NFA transition that any state in S could take on a:

$$\delta_D(S, a) = \bigcup_{q \in S} \Delta(q, a).$$

The result is the set of all NFA states reachable from any member of S by consuming a.

Accepting states. The NFA accepts a word *existentially*: it succeeds if *at least one* active state is final. We mirror this in the DFA by declaring S accepting whenever it contains at least one NFA final state:

$$F_D = \{S \subseteq Q : S \cap F \neq \emptyset\}.$$

The construction is called the **power-set construction** because the DFA states are subsets of the NFA state set. Each deterministic state S is a subset of Q, so the DFA has at most $2^{|Q|}$ states. The intuition is direct: for every possible

subset of NFA states that could be simultaneously active, the DFA introduces one dedicated state. There is a price to pay in size, but not in expressive power — the two classes of automata define exactly the same set of languages.

This equivalence between DFA and NFA is specific to finite words. When we move to automata over infinite words in chapter 3, we will see that deterministic and non-deterministic variants are not *always equivalent: the acceptance condition matters, and for some conditions non-determinism genuinely adds expressive power.*

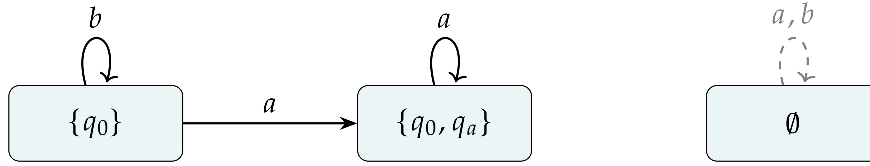


Figure 1.7: A determinized state is a set of currently possible NFA states.

Figure 1.7 visualizes the key bookkeeping move: one deterministic state stands for a whole set of possible NFA states. The picture is deliberately about sets, not individual branches.

Proof. We prove the following invariant by induction on $|w|$: for every word $w \in A^*$,

$$\widehat{\delta}_D(\{q_0\}, w) \;=\; \widehat{\Delta}(q_0, w).$$

A small but important remark about types. On the left-hand side, $\widehat{\delta}_D(\{q_0\}, w)$ is a single state of the DFA $\mathcal{D}$ — but every such state is, by construction, a subset of Q. On the right-hand side, $\widehat{\Delta}(q_0, w)$ is directly a subset of Q. The invariant claims these two subsets coincide.

Base case ($w = \varepsilon$). The DFA starts in its initial state $\{q_0\}$. The NFA, having consumed no input, can only be in q_0. So both sides equal $\{q_0\}$.

Inductive step. Suppose the invariant holds for some word w, so that

$$\widehat{\delta}_D(\{q_0\}, w) = \widehat{\Delta}(q_0, w) = S.$$

After reading one more symbol a, the DFA applies δ_D:

$$\widehat{\delta}_D(\{q_0\}, wa) \;=\; \delta_D(S, a) \;=\; \bigcup_{q \in S} \Delta(q, a).$$

This union is precisely $\widehat{\Delta}(q_0, wa)$: every state it contains is reached by extending some NFA run on w with one a-transition, and every such extension contributes its target to the union. The invariant therefore holds for wa.

The invariant immediately gives the language equivalence. The NFA accepts w if and only if $\widehat{\Delta}(q_0, w) \cap F \neq \emptyset$. By the invariant this is $\widehat{\delta}_D(\{q_0\}, w) \cap F \neq \emptyset$, which holds if and only if $\widehat{\delta}_D(\{q_0\}, w) \in F_D$ — exactly the DFA acceptance condition. Hence $L(\mathcal{D}) = L(\mathcal{A})$. □

1.4.1 Regular Languages

The equivalence between DFA and NFA lets us name the class of languages that finite automata recognize.

> **Definition 1.4.2** (Regular language)**.** A language $L \subseteq A^*$ is **regular** if there exists a DFA $\mathcal{A}$ with $L(\mathcal{A}) = L$.

This definition names the languages that can be checked with finite memory. To read it, ask whether there is some finite automaton whose accept/reject behaviour matches the property exactly. The operational test is: build a DFA, run it on the whole input word, and accept exactly when the final state lies in its accepting set.

theorem 1.4.1 makes this definition robust to the choice of model: a language is regular if and only if it is recognized by some NFA. We may therefore use DFA and NFA interchangeably when reasoning about regularity — typically NFAs when designing an automaton, DFAs when we need a single run on each input, for example to complement the language by swapping accepting and non-accepting states.

The class of regular languages is a *strict* subset of all languages over A. This boundary is exactly what the theory is about: regular languages are the properties that a finite agent can check by reading the input once, using only a bounded amount of memory. The next example shows a natural language that slips through this net.

Example 1.4.3 (A non-regular language: $a^n b^n$)**.** Over $A = \{a, b\}$, consider

$$L_= = \{a^n b^n : n \geq 0\},$$

the words consisting of a block of a's followed by a block with the same number of b's. This is precisely the matching problem that theorem 1.2.3 deliberately avoided: there, a single b could discharge any number of pending a's, whereas here each a demands a *distinct* later b.

Suppose a DFA with k states reads the unary block a^i for $i = 0, 1, \dots, k$. By the pigeonhole principle two of these prefixes, say a^i and a^j with $i < j$, must lead to the same state. From that shared state, the same continuation b^i must produce the same verdict for both $a^i b^i$ and $a^j b^i$. But $a^i b^i \in L_=$ while $a^j b^i \notin L_=$, a contradiction. Hence no DFA recognizes $L_=$, and $L_=$ is not regular.

We will turn this pigeonhole sketch into a clean characterization — the Myhill–Nerode theorem — in chapter 2.

Notation Introduced in This Chapter

Notation	Meaning
A	finite alphabet of input symbols
A^*	set of all finite words over A
ε	empty word
$L \subseteq A^*$	language over A
uv	concatenation of words u and v
$\lvert w \rvert$	length of the word w
W^*	Kleene closure of the language W
W^+	non-empty finite concatenations of words from W
$\mathcal{A}$	finite automaton
Q	finite set of states
q_0	initial state
F	set of accepting states
$\delta : Q \times A \to Q$	DFA transition function
$\widehat{\delta} : Q \times A^* \to Q$	extended DFA transition function on words
$L(\mathcal{A})$	language accepted by the automaton $\mathcal{A}$
$\bot$	rejecting sink state
$\Delta \subseteq Q \times A \times Q$	NFA transition relation
$\Delta : Q \times A \to 2^Q$	equivalent set-valued view of NFA transitions
$\widehat{\Delta}$	extended NFA transition map on words
2^Q	powerset of Q, used as the state space in determinization

Summary & Key Takeaways

- A language is a set of finite words over an alphabet.
- A DFA has one run per input word.
- An NFA may have many runs, and accepts if at least one succeeds.
- Every NFA can be determinized by tracking sets of possible states.
- The price of determinization can be exponential.
- The languages recognized by finite automata (DFA or NFA) are the *regular* languages; not every language is regular, as $\{a^n b^n : n \geq 0\}$ shows.

Exercises

Exercise 1 (Simulating a DFA). For the DFA in fig. 1.2, compute the reached state after reading each prefix of the word $abbaab$. Is the whole word accepted?

Exercise 2 (One subset-construction step). Use the NFA in fig. 1.6. Starting from the determinized state $\{q_s, q_a\}$, compute the next determinized state on input b. Explain why the accepting state disappears.

Exercise 3 (Design). Design a DFA over $\{a, b\}$ for the language of words that contain the substring ab. Use a sink-like state to record that the substring has already been seen.

CHAPTER 2

Regular Languages and Decision Problems

Regular languages are the finite-word behaviours recognized by finite automata. The point of the theory is not only recognition, but effective verification.

In chapter 1 we built finite automata to recognize finite-word behaviours: a turnstile event sequence, a parity check over $\{a, b\}$, and so on. Automata are an *operational* description—they tell us how to scan a word. But they are not the only way to describe the same languages, and they are not always the most convenient one.

This chapter adds three complementary viewpoints, each answering a different question that the operational model alone leaves open.

- *Can we describe a language algebraically, without drawing a graph?* Regular expressions (section 2.1) give such a syntax, and Kleene's theorem shows they define exactly the regular languages.
- *If we are given a property built from simpler ones by union, intersection, complement, can we still build an automaton for it?* Closure properties (section 2.2) answer yes, and the constructions are the backbone of every verification algorithm in the rest of the book.
- *Once automata encode properties, can we decide inclusion, equivalence, universality?* The decision problems of section 2.3 all reduce to a single reachability check—a finite-word prototype of model checking.

The chapter closes with the Myhill–Nerode theorem (section 2.4), which explains *why* some languages admit a finite-state description at all and yields the canonical minimal DFA. The guiding theme throughout is *reduction*: we reduce the hard-looking semantic questions to reachability-style emptiness checks that we actually know how to perform.

2.1 Regular Expressions and Kleene's Theorem

2.1.1 Regular Expressions

Drawing a DFA is an *operational* act: we specify, state by state and symbol by symbol, how the machine moves through a computation. The description answers the question "what does the machine do at each step?" rather than "what does the language look like?" For many purposes—communicating a property to a colleague, composing specifications from sub-properties, searching text—a more algebraic syntax is preferable, one that builds the language out of pieces using a few combinators and says nothing about machine states. This is what regular expressions provide.

The idea is to describe a language by saying how its words are built: start from individual letters, glue words together by concatenation, offer alternatives by union, and allow finite repetition by Kleene star. The following definition pins down exactly which expressions we allow.

Definition 2.1.1 (Restricted regular expressions)**.** The **restricted regular expressions** over an alphabet Σ are generated by:

$$\emptyset, \qquad \varepsilon, \qquad a \in \Sigma,$$

and, if r and s are regular expressions, by:

$$r + s, \qquad rs, \qquad r^*.$$

The semantics maps each expression to the language it describes. The base cases are: $\emptyset$ denotes the empty language (no word at all), ε denotes the language $\{\varepsilon\}$ containing only the empty word, and a symbol a denotes the singleton language $\{a\}$. For the combinators: $r + s$ denotes the union of the languages of r and s, rs denotes their concatenation (every word of r followed by every word of s), and r^* denotes finite repetition—zero or more concatenations of the language of r with itself, so r^* always includes the empty word.

Example 2.1.2 (A simple regular expression)**.** Over $\Sigma = \{a, b\}$, the expression

$$b^*(ab^*ab^*)^*$$

denotes the language of words with an even number of occurrences of a. The initial b^* consumes any number of b's before the first a, and each repetition of ab^*ab^* contributes two occurrences of a.

This is the same language recognized by the DFA in fig. 1.2. The expression describes the structure of the word; the automaton describes how to scan it.

The restricted syntax is not the only option. A more liberal variant admits Boolean operations directly inside expressions.

Definition 2.1.3 (General regular expressions)**.** The **general regular expressions** over Σ extend the restricted syntax of theorem 2.1.1 with two Boolean combinators: if r and s are general regular expressions, so are

$$r \cap s \qquad \text{and} \qquad \overline{r}.$$

The semantics extends naturally: $r \cap s$ denotes the intersection of the languages of r and s, and $\overline{r}$ denotes the complement of the language of r with respect to Σ^*.

Within the general syntax there is an important sublanguage obtained by dropping star rather than Boolean operations.

Definition 2.1.4 (Star-free expressions)**.** A **star-free expression** is a general regular expression that contains no occurrence of Kleene star. It may use union, concatenation, complement, and intersection freely, but repetition is forbidden.

■ **Formal details — Expressive power of the three classes**

The three syntactic classes—restricted, general, and star-free—are not equally expressive. General expressions add no power over restricted ones: once we can translate expressions to automata, closure under intersection and complement (section 2.2) lets us compile the Boolean operators away and return to a restricted expression. This is exactly theorem 2.1.13.

Removing star, however, does cost expressive power. Star-free expressions define a *strictly smaller* class: the language of words with an even number of a's (theorem 2.1.2) is regular but cannot be described without star. By Schützenberger's theorem (see chapter 9), the star-free languages are exactly the *aperiodic* regular languages—and, when logic enters the course, they reappear as the finite-word shadow of first-order definability.

2.1.2 The ε-NFA Model

Before proving the equivalence between expressions and automata, we need a more flexible automaton model that lets us wire pieces together without consuming input. The wiring device is the *ε-move*: a transition that changes state while the reading head stays put. Without it, building an automaton for the concatenation of two languages would require us to redesign the state spaces by hand—deciding which states of the first automaton should transfer control to the second, and carefully re-routing transitions so that the handoff happens at the right moment. With ε-moves, we simply add a free edge from the accepting state of the first automaton to the initial state of the second, and let a later clean-up pass dissolve the free edges.

Example 2.1.5 (Wiring two automata with an ε-move)**.** Suppose we have an automaton A_1 recognizing $\{a\}$ (one transition $q_0 \xrightarrow{a} q_1$ with q_1 accepting) and an automaton A_2 recognizing $\{b\}$ ($r_0 \xrightarrow{b} r_1$ with r_1 accepting). To build an automaton for the concatenation $\{ab\}$, we add a free edge $q_1 \xrightarrow{\varepsilon} r_0$: after A_1 finishes, the machine silently jumps to A_2 without consuming input. The result is an automaton with four states whose only accepted word is ab, built by wiring rather than redesigning.

The example above uses an edge labelled ε—a transition that fires without reading any symbol. Let us make this precise.

Definition 2.1.6 (ε-NFA)**.** An **ε-NFA** is an NFA whose transition function may also use the empty word ε as a label. Formally, the transition function has the shape

$$\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q,$$

so a transition in $\Delta(q, \varepsilon)$ changes state without consuming any input symbol.

Operationally, an ε-NFA behaves like an NFA with one extra freedom: at any moment it may follow an ε-labelled edge and jump to another state, while the input pointer does not move. This is purely a construction device—it gives us the wiring freedom needed to compose automata for union, concatenation, and Kleene star—but, as we show next, it does not add any expressive power.

Our goal is to eliminate ε-moves: we want an ordinary NFA that accepts the same language but uses only real-symbol transitions. The key step is to

pre-compute, for each state, the full set of states it can reach by following only ε-edges—so that we can "short-circuit" every chain of free moves into a single jump. This set is called the ε-closure.

Returning to theorem 2.1.5, from q_1 we can follow the ε-edge to r_0. Since every state trivially reaches itself in zero steps, the full set of states that q_1 can silently reach is $\{q_1, r_0\}$. Collecting all such silent destinations—both the state itself and everything reachable through one or more ε-edges—is exactly what the closure computes.

> **Definition 2.1.7** (ε-closure)**.** For a state q, the **ε-closure** $\mathsf{ECl}(q)$ is the set of states reachable from q by taking zero or more ε-moves. For a set of states S,
>
> $$\mathsf{ECl}(S) = \bigcup_{q \in S} \mathsf{ECl}(q).$$

Because Q is finite, $\mathsf{ECl}(q)$ is the least fixpoint of the monotone operator $X \mapsto \{q\} \cup \bigcup_{p \in X} \Delta(p, \varepsilon)$. This is the same fixpoint flavour that will return, scaled up, in the symbolic reachability algorithms of section 2.3.1.

■ **Formal details — Least fixpoint: what it means and how to compute it**

The ε-closure is defined by a self-referential condition—a state belongs to $\mathsf{ECl}(q)$ if it is reachable from other states that are already in $\mathsf{ECl}(q)$—so we need a principled way to solve it. The answer is to start small and expand until nothing new can be added.

How the iteration works. Begin with $\{q\}$, then repeatedly ask: can we reach any new state from the current set by taking one ε-step? If yes, add it and ask again; if no, stop. Written out:

$$X_0 = \{q\}, \qquad X_{i+1} = X_i \cup \bigcup_{p \in X_i} \Delta(p, \varepsilon).$$

Since we only ever add states and Q is finite, the sequence $X_0 \subseteq X_1 \subseteq \cdots$ must eventually stabilise: some step $k \leq |Q|$ produces $X_{k+1} = X_k$. That stable set is $\mathsf{ECl}(q)$. A set where re-applying the expansion leaves it unchanged is called a **fixpoint**.

Why least? There are many fixpoints. The entire state set Q is one: following ε-edges from every state can never escape Q. But Q overapproximates wildly—it claims every state is ε-reachable from q, which is almost never true.

The iteration above is disciplined: it starts with only q and admits a state only when an ε-path from q genuinely forces it in. Any other fixpoint must contain q and must be closed under ε-moves, so it must include everything our iteration accumulated. Our answer is therefore the **least** (i.e. smallest by $\subseteq$) fixpoint of the expansion operator—the tightest correct answer, denoted $\mu X. F(X)$ in the fixpoint calculus.

Connection to global reachability. The same pattern reappears in section 2.3.1: $\mathsf{Post}(X)$ plays the role of $\bigcup_{p \in X} \Delta(p, \varepsilon)$, the initial state q_0 plays the role of q, and the iteration $R_{i+1} = R_i \cup \mathsf{Post}(R_i)$ converges to the full reachable set R^*. Computing $\mathsf{ECl}(q)$ is therefore a miniature instance of that global reachability algorithm, narrowed to ε-labelled edges alone.

With the closure in hand we can define how an ε-NFA reads a whole word. The pattern is: take all free moves, read one real symbol, take all free moves again, and repeat.

Definition 2.1.8 (Extended transition function of an ε-NFA)**.** Let $A = (Q, \Sigma, \Delta, q_0, F)$ be an ε-NFA. The **extended transition function** $\widehat{\Delta} : 2^Q \times \Sigma^* \to 2^Q$ encodes the rhythm described above: close the current set under ε-moves, read one real symbol, close again, and repeat for each symbol in the word. Formally, by recursion on word length:

$$\widehat{\Delta}(S, \varepsilon) = \mathsf{ECl}(S),$$

$$\widehat{\Delta}(S, wa) = \mathsf{ECl}\left(\bigcup_{p \in \widehat{\Delta}(S,w)} \Delta(p, a) \right).$$

Reading the empty word simply resolves any pending free moves. For a word ending in a: find all states reachable after the prefix w, follow every real a-transition from those states, then close under ε-moves once more.

A word $w \in \Sigma^*$ is accepted by A if and only if

$$\widehat{\Delta}(\{q_0\}, w) \cap F \neq \emptyset.$$

Example 2.1.9 (Tracing $\widehat{\Delta}$ on the word ab)**.** We trace the definition on the ε-NFA from theorem 2.1.5, whose components are:

$$Q = \{q_0, q_1, r_0, r_1\}, \qquad \Sigma = \{a, b\}, \qquad q_0 \text{ initial}, \qquad F = \{r_1\}.$$

The transitions are $\Delta(q_0, a) = \{q_1\}$, $\Delta(q_1, \varepsilon) = \{r_0\}$, $\Delta(r_0, b) = \{r_1\}$, and every other entry is $\emptyset$.

We compute $\widehat{\Delta}(\{q_0\}, ab)$ by decomposing the word as $w = a$ and the last symbol b, then further decomposing $w = \varepsilon$ followed by a.

Step 1 — base case ($w = \varepsilon$).

$$\widehat{\Delta}(\{q_0\}, \varepsilon) = \mathsf{ECl}(\{q_0\}) = \{q_0\}.$$

State q_0 has no outgoing ε-edges, so its closure is just itself.

Step 2 — read a (inductive step with $w = \varepsilon$, $a = a$).

$$\widehat{\Delta}(\{q_0\}, a) = \mathsf{ECl}\Big(\bigcup_{p \in \{q_0\}} \Delta(p, a) \Big) = \mathsf{ECl}\big(\Delta(q_0, a)\big) = \mathsf{ECl}(\{q_1\}) = \{q_1, r_0\}.$$

After reading a we are in q_1; the ε-closure then adds r_0 because $q_1 \xrightarrow{\varepsilon} r_0$.

Step 3 — read b (inductive step with $w = a$, $a = b$).

$$\widehat{\Delta}(\{q_0\}, ab) = \mathsf{ECl}\Big(\bigcup_{p \in \{q_1, r_0\}} \Delta(p, b) \Big) = \mathsf{ECl}\big(\Delta(q_1, b) \cup \Delta(r_0, b)\big) = \mathsf{ECl}(\emptyset \cup \{r_1\}) = \{r_1\}.$$

From q_1 there is no b-transition ($\Delta(q_1, b) = \emptyset$); from r_0 we get r_1. Since r_1 has no outgoing ε-edges, the closure is $\{r_1\}$.

Acceptance check. $\widehat{\Delta}(\{q_0\}, ab) \cap F = \{r_1\} \cap \{r_1\} = \{r_1\} \neq \emptyset$, so ab is accepted.

> **Theorem 2.1.10** (ε-removal)**.** *For every ε-NFA A there exists an NFA A' without ε-moves such that $L(A) = L(A')$.*

Proof. Let $A = (Q, \Sigma, \Delta, q_0, F)$. We define an ordinary NFA $A' = (Q, \Sigma, \Delta', q_0, F')$ on the same states and alphabet. The construction encodes the slogan *take all free moves, read one symbol, take all free moves again* directly into the transition function:

$$\Delta'(q, a) = \mathsf{ECl}\left(\bigcup_{p \in \mathsf{ECl}(q)} \Delta(p, a)\right).$$

This is the best possible one-step simulation of A: starting from q, the run in A may first wander through ε-moves, arrive at some state p, consume a, and then again wander through ε-moves; $\Delta'(q, a)$ collects exactly the destinations reachable that way.

There is one boundary case. In A, the empty word is accepted if and only if some accepting state is reachable from q_0 by ε-moves alone, since the input pointer never moves. The new automaton would otherwise miss this case, because every transition of A' consumes a real symbol. We correct the final set:

$$F' = \begin{cases} F \cup \{q_0\}, & \mathsf{ECl}(q_0) \cap F \neq \emptyset, \\ F, & \text{otherwise.} \end{cases}$$

It remains to show $L(A') = L(A)$. Since A' and A share the same state set, it suffices to show that reading any non-empty word always lands them in the same reachable-state set; acceptance then coincides because F' was built to match F's verdict on every word (including ε, which is handled by the correction above). Formally, we prove the following invariant by induction on $|w|$: for every $w \in \Sigma^*$,

$$\widehat{\Delta}_{A'}(\{q_0\}, w) = \widehat{\Delta}_A(\{q_0\}, w),$$

where $\widehat{\Delta}_A$ is the extended function of theorem 2.1.8 and $\widehat{\Delta}_{A'}$ is the ordinary NFA extension of theorem 1.2.4 applied to A'. For $w = \varepsilon$: $\widehat{\Delta}_{A'}(\{q_0\}, \varepsilon) = \{q_0\}$ and $\widehat{\Delta}_A(\{q_0\}, \varepsilon) = \mathsf{ECl}(q_0)$; these sets need not be equal in general, but acceptance of the empty word is handled entirely by the F' correction above, so no further argument is needed here.

For the inductive step, suppose the invariant holds for w and let $S = \widehat{\Delta}_A(\{q_0\}, w)$. Before expanding the chain, note that S is already ε-closed: every output of $\widehat{\Delta}_A$ ends with an application of ECl, so $\mathsf{ECl}(S) = S$ and in particular $\mathsf{ECl}(p) \subseteq S$ for every $p \in S$. This is what licenses the third equality below. Then

$$\begin{aligned}
\widehat{\Delta}_{A'}(\{q_0\}, wa) &= \bigcup_{p \in S} \Delta'(p, a) && \text{(NFA extended function, using IH)} \\
&= \bigcup_{p \in S} \mathsf{ECl}\left(\bigcup_{r \in \mathsf{ECl}(p)} \Delta(r, a)\right) && \text{(definition of } \Delta'\text{)} \\
&= \mathsf{ECl}\left(\bigcup_{p \in S} \Delta(p, a)\right) && (\mathsf{ECl}(S) = S, \text{ so } \textstyle\bigcup_{p \in S} \mathsf{ECl}(p) = S) \\
&= \widehat{\Delta}_A(\{q_0\}, wa). && \text{(definition of } \widehat{\Delta}_A\text{)}
\end{aligned}$$

The third step is the key one: because S is already closed, ranging over $\mathsf{ECl}(p)$ for each $p \in S$ adds no states beyond S itself, so the inner per-p closures collapse into a single outer ECl over the direct a-successors. Intersecting with F' then yields exactly the same accepting verdict as intersecting with F in A. □

Example 2.1.11 (ε-removal on the concatenation automaton)**.** Consider the ε-NFA from theorem 2.1.5 for $\{ab\}$: states q_0, q_1, r_0, r_1, transitions $q_0 \xrightarrow{a} q_1$, $q_1 \xrightarrow{\varepsilon} r_0$, $r_0 \xrightarrow{b} r_1$, with r_1 accepting. The ε-closures are $\mathsf{ECl}(q_0) = \{q_0\}$, $\mathsf{ECl}(q_1) = \{q_1, r_0\}$, $\mathsf{ECl}(r_0) = \{r_0\}$, $\mathsf{ECl}(r_1) = \{r_1\}$. Applying the construction:

$$\Delta'(q_0, a) = \mathsf{ECl}\big(\Delta(q_0, a)\big) = \mathsf{ECl}(\{q_1\}) = \{q_1, r_0\},$$

$$\Delta'(q_1, b) = \mathsf{ECl}\big(\textstyle\bigcup_{p \in \{q_1, r_0\}} \Delta(p, b)\big) = \mathsf{ECl}(\{r_1\}) = \{r_1\}.$$

All other transitions are empty. Since $\mathsf{ECl}(q_0) \cap F = \emptyset$, the final set stays $F' = \{r_1\}$. The resulting NFA accepts exactly $\{ab\}$, with no ε-edges.

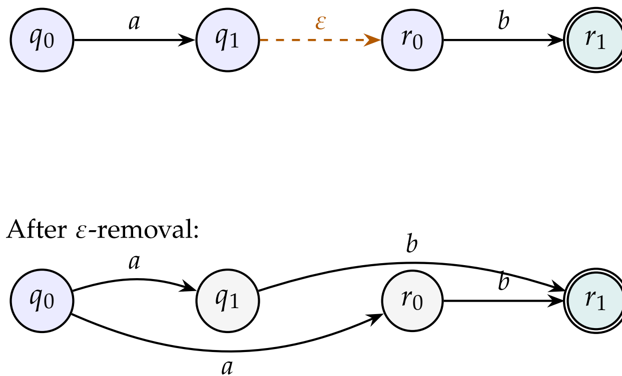


Figure 2.1: ε-removal applied to the concatenation automaton of theorem 2.1.5. Top: the original ε-NFA, with the free edge shown dashed in orange. Bottom: the resulting ordinary NFA—the ε-closure has been absorbed into the transition targets.

2.1.3 Kleene's Theorem

With ε-NFAs as a flexible intermediate model, we can equate the syntactic and operational views of regularity.

Theorem 2.1.12 (Kleene equivalence)**.** *A language $L \subseteq \Sigma^*$ is recognized by a finite automaton if and only if it is denoted by a regular expression.*

Proof. We prove both directions constructively.

Regular expression to automaton. We build an ε-NFA by structural induction on the expression. For $\emptyset$, use a single non-accepting state with no transitions. For ε, use an initial state that is already accepting. For a symbol a, use one transition labelled a from the initial state to a fresh accepting state.

Assume ε-NFAs have already been built for r and s, with disjoint state sets and a single accepting state each (this is without loss of generality, since we can always redirect outgoing edges through a fresh accepting sink via ε-moves).

- For $r + s$, create a fresh initial state with ε-moves to the initial states of the two automata, and ε-moves from each of their accepting states to a fresh accepting state. Figure 2.3 shows the wiring.
- For rs, connect the accepting state of the automaton for r to the initial state of the automaton for s by an ε-move; the accepting state of s becomes the accepting state of the combined automaton. Figure 2.2 shows the wiring.
- For r^*, add a fresh initial state that is already accepting, add an ε-move to the initial state of the automaton for r, and add an ε-move from the accepting state of r back to that initial state. Figure 2.4 shows the wiring.

Each construction mirrors exactly the semantic operation: choose a branch for union, run one machine after the other for concatenation, and repeat any finite number of times for star. By theorem 2.1.10 we can then eliminate ε-moves, determinize, and obtain a DFA. Hence every regular expression denotes an automaton-recognizable language.

Automaton to regular expression. Let A be a DFA with states $q_1, \ldots, q_n$, with q_1 initial. For $0 \le k \le n$, define $R^k_{i,j}$ to be the language of all words that take the automaton from q_i to q_j using only intermediate states among $q_1, \ldots, q_k$.

We prove by induction on k that each $R^k_{i,j}$ is denoted by a regular expression. For $k = 0$, no intermediate state is allowed, so a path from q_i to q_j is either the empty path (when $i = j$) or a single transition labelled by some $a \in \Sigma$. Hence $R^0_{i,j}$ is a finite union of symbols, together with ε when $i = j$; this is regular.

For the induction step, a path counted by $R^k_{i,j}$ either never visits q_k as an intermediate state—in which case it lies in $R^{k-1}_{i,j}$—or it visits q_k at least once. In the latter case, split the path at the first visit to q_k, then at each complete return to q_k, and finally at the last departure toward q_j. This yields the recurrence

$$R^k_{i,j} = R^{k-1}_{i,j} + R^{k-1}_{i,k}\,(R^{k-1}_{k,k})^*\,R^{k-1}_{k,j},$$

which is regular by the induction hypothesis.

Finally, the language of the automaton is

$$L(A) = \bigcup_{q_j \in F} R^n_{1,j},$$

again a regular expression. □

The accepting exit of the first automaton is spliced to the entry of the second by a single ε-move (in orange).

Figure 2.2: The ε-NFA construction for concatenation in the regex-to-automaton direction of theorem 2.1.12: the accepting state of r and the initial state of s are wired together by an ε-move, and the fresh accepting state f_s inherits acceptance.

A useful consequence of theorem 2.1.12, combined with the closure constructions we are about to develop in section 2.2, is that the two syntactic classes of theorems 2.1.1 and 2.1.3—restricted and general—recognize exactly the same languages.

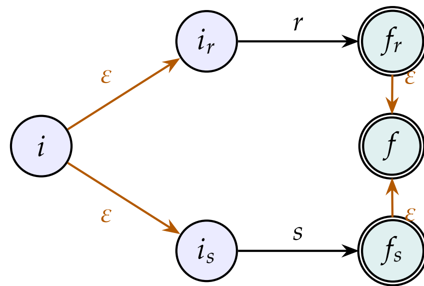


Figure 2.3: The ε-NFA construction for union $r + s$: a fresh initial state branches into the two sub-automata via ε-moves (orange), and both accepting states feed into a fresh common accepting state.

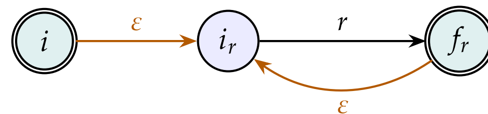


Figure 2.4: The ε-NFA construction for Kleene star r^*: a fresh initial state i (already accepting, to cover the empty word) has an ε-move into the sub-automaton for r, and the accepting state of r loops back to i_r via another ε-move.

Corollary 2.1.13 (Restricted and general expressions coincide)**.** *Restricted and general regular expressions denote the same class of languages.*

Proof. Every restricted expression is, by definition, a general one. Conversely, given a general expression, we translate each Boolean operator into an automaton construction: complementation becomes final-state swapping on a complete DFA, and intersection becomes the product construction of theorem 2.2.4. This produces a finite automaton, which theorem 2.1.12 converts back into a restricted expression. □

2.2 Closure Properties

Closure properties say that regular languages remain regular after standard operations. They are useful because verification constantly builds compound properties from simpler ones.

Theorem 2.2.1 (Closure of regular languages)**.** *Regular languages over finite words are closed under union, concatenation, Kleene star, complementation, and intersection.*

Union, concatenation, and star follow directly from regular expressions or ε-NFA constructions. Complementation and intersection are more important for verification, so we spell them out.

2.2.1 Complementation

If $A = (Q, \Sigma, \delta, q_0, F)$ is a complete DFA, its complement is

$$\overline{A} = (Q, \Sigma, \delta, q_0, Q \setminus F).$$

The same run is used; only the final verdict changes.

> *Observation* 2.2.2. The simple final-state swap works only for complete deterministic automata. Non-determinism and missing transitions both break the argument.

For an NFA, one accepting branch is enough to accept a word. Swapping final and non-final states would not turn "there exists an accepting branch" into "all branches reject". The standard route is therefore: determinize, complete, then swap final states.

> **Example 2.2.3** (Why swapping fails for NFAs). Consider the NFA over $\{a, b\}$ with states q_0 (initial) and q_1 (accepting), transitions $q_0 \xrightarrow{a} q_0$, $q_0 \xrightarrow{a} q_1$, and $q_0 \xrightarrow{b} q_0$. This NFA accepts every word containing at least one a. If we naively swap final and non-final states, q_0 becomes accepting and q_1 becomes non-accepting. Now the word a is still accepted—via the branch $q_0 \xrightarrow{a} q_0$, which stays in the (now accepting) state q_0. The swapped automaton therefore accepts both a and b^*, which is not the true complement $\{b\}^*$. The error is that non-determinism provides an alternative accepting branch that the swap cannot suppress.

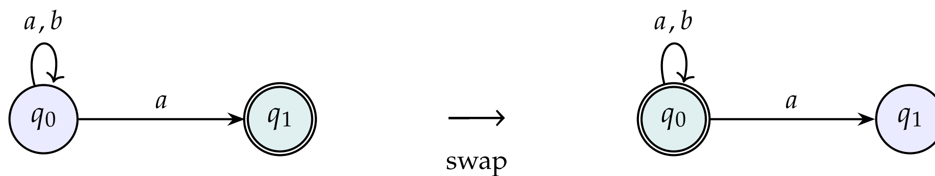


Figure 2.5: Naively swapping final states in an NFA does not produce the complement. Left: the original NFA accepts words with at least one a. Right: after swapping, the self-loop on q_0 still accepts every word, including those without any a.

2.2.2 Intersection

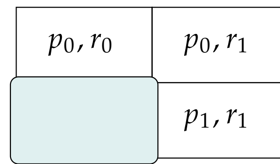
Intersection is handled by running two automata in lockstep.

> **Definition 2.2.4** (Product automaton). Let $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be DFAs. Their product for intersection is the DFA with:
>
> $$Q = Q_1 \times Q_2, \qquad q_0 = (q_1, q_2),$$
> $$\delta((p_1, p_2), a) = (\delta_1(p_1, a), \delta_2(p_2, a)), \qquad F = F_1 \times F_2.$$

The first component remembers where A_1 is; the second remembers where A_2 is. A word is accepted exactly when both components end in accepting states.

The word *end* is doing real work here. For finite words, both automata consume exactly the same input positions and there is one final moment at which acceptance is tested. This is why the simple accepting set $F_1 \times F_2$ is

correct. In chapter 3, the same idea must be modified: two Büchi automata may visit their accepting states at different times, so simultaneous final-state membership is no longer enough.



A product state stores one state from each automaton.

Figure 2.6: The state space of a product automaton is a Cartesian product.

Example 2.2.5 (Product of "ends in a" and "even a's")**.** Let A_1 be the two-state DFA over $\{a, b\}$ that accepts words ending in a: states p_0 (initial, not accepting) and p_1 (accepting), with $\delta_1(p_0, a) = p_1$, $\delta_1(p_0, b) = p_0$, $\delta_1(p_1, a) = p_1$, $\delta_1(p_1, b) = p_0$. Let A_2 be the two-state DFA that accepts words with an even number of a's: states r_0 (initial, accepting) and r_1 (not accepting), toggling on each a and staying put on each b.

The product automaton has four states $\{p_0, p_1\} \times \{r_0, r_1\}$. A word is accepted when both components accept simultaneously: p_1 (last letter was a) and r_0 (even count). So $F = \{(p_1, r_0)\}$.

The word a reaches (p_1, r_1)—last letter is a but the count is odd—so it is rejected. The word aa reaches (p_1, r_0) and is accepted. The word aab reaches (p_0, r_0): even count but the last letter is b, rejected. The product enforces both conditions at once.

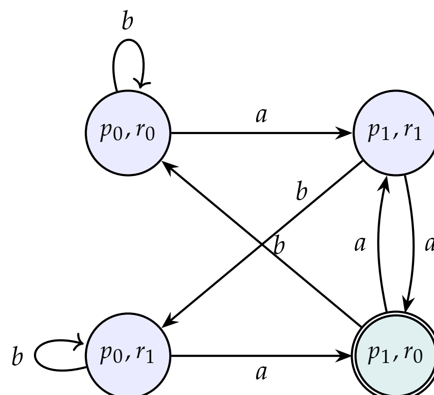


Figure 2.7: The product automaton for "ends in a" ∩ "even number of a's" from theorem 2.2.5. Only state (p_1, r_0) is accepting (teal, double border).

2.3 Decision Problems

We now have a rich toolkit for building automata: regular expressions give us a syntax, and closure under union, intersection, and complement lets us assemble compound properties from simpler ones. The natural next question is: once we hold such an automaton, what can we actually *ask* about it? Can

we decide whether two automata recognize the same language, or whether one language is contained in another? It turns out that all of these semantic questions reduce to a single graph problem—reachability—making them not only decidable but efficiently so.

Emptiness = satisfiability: if automata compile logical formulas, then checking whether $L(A) \neq \emptyset$ is exactly solving SAT.

Definition 2.3.1 (Core decision problems)**.** For finite automata over Σ^*, the standard decision problems are:

- **membership**: given A and w, is $w \in L(A)$?
- **emptiness**: is $L(A) = \emptyset$?
- **universality**: is $L(A) = \Sigma^*$?
- **inclusion**: is $L(A_1) \subseteq L(A_2)$?
- **equivalence**: is $L(A_1) = L(A_2)$?

Membership is simulation. Emptiness is reachability: does some final state occur on a path from the initial state?

Proposition 2.3.2 (Emptiness for finite automata)**.** *Let A be an NFA or DFA. Then $L(A) \neq \emptyset$ if and only if some accepting state is reachable from q_0 in the underlying directed graph.*

Proof. If an accepting state is reachable, the labels along that path form an accepted word. Conversely, an accepted word induces a run from q_0 to an accepting state. Thus non-emptiness and accepting-state reachability are the same question. □

There is another way to say the same thing, useful later when we move to symbolic model checking: if a finite automaton accepts something, then it accepts a short witness.

Theorem 2.3.3 (Small model property for finite automata)**.** *Let A be a DFA with n states. Then*

$$L(A) \neq \emptyset \iff \text{there exists } w \in L(A) \text{ with } |w| < n.$$

Proof. The right-to-left implication is immediate: an accepted word of any length proves non-emptiness.

For the other direction, assume $L(A) \neq \emptyset$ and choose an accepted word $z = a_1 a_2 \cdots a_m$ of minimum length. Let

$$p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_m} p_m$$

be the accepting run, with $p_0 = q_0$ and $p_m \in F$.

If $m \geq n$, then the sequence $p_0, p_1, \ldots, p_m$ contains at least $n + 1$ state occurrences but only n possible states. By the pigeonhole principle, $p_i = p_j$ for some $0 \leq i < j \leq m$. Split the word as

$$z = uvr,$$

where $u = a_1 \cdots a_i$, $v = a_{i+1} \cdots a_j$, and $r = a_{j+1} \cdots a_m$. The segment v labels a loop from p_i back to p_i. Removing that loop gives the shorter word ur, and the run still goes from q_0 to the same accepting state p_m. Thus ur is accepted, contradicting the minimality of z. Hence the minimal accepted word has length $m < n$. □

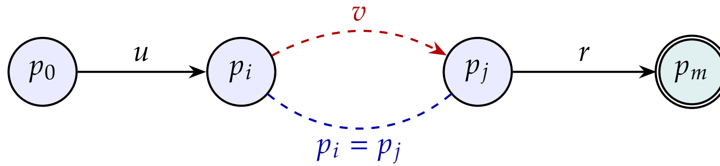


Figure 2.8: If an accepting run repeats a state, the loop can be removed to obtain a shorter accepted word.

The graph algorithm is better in practice, but the small-model theorem is conceptually useful: decidability follows because, in principle, we only need to inspect finitely many words shorter than n.

2.3.1 Reachability as a Fixpoint

Let $\mathsf{Post}(X)$ be the set of states reachable in one transition from some state in X. Starting from $R_0 = \{q_0\}$, define

$$R_{i+1} = R_i \cup \mathsf{Post}(R_i).$$

Because Q is finite, this increasing sequence eventually stabilizes at a set R^*. The language is non-empty exactly when

$$R^* \cap F \neq \emptyset.$$

This fixpoint view is the finite-state ancestor of the symbolic preimage/fixpoint algorithms used later for CTL and planning.

The other problems reduce to emptiness by using closure.

Formal details — Reductions

$$L(A) = \Sigma^* \iff L(\overline{A}) = \emptyset.$$

$$L(A_1) \subseteq L(A_2) \iff L(A_1) \cap \overline{L(A_2)} = \emptyset.$$

$$L(A_1) = L(A_2) \iff L(A_1) \subseteq L(A_2) \text{ and } L(A_2) \subseteq L(A_1).$$

For universality, we build the complement and ask whether any rejected word exists. For inclusion, we search for a counterexample:

$$w \in L(A_1) \setminus L(A_2).$$

If the intersection $L(A_1) \cap \overline{L(A_2)}$ is empty, then no such counterexample exists. Equivalence is just two inclusions, because two sets are equal precisely when neither contains an element missing from the other.

This is the finite-word prototype of model checking. The system automaton describes possible behaviours; the specification automaton describes allowed behaviours. Verification asks for language inclusion.

2.4 The Myhill–Nerode View

So far we have a powerful toolkit—regular expressions to describe languages, closure properties to combine them, and decision procedures to answer questions about them. But all of it rests on one assumption: the language is regular.

What if someone hands us a language defined by a predicate, not by an automaton? How do we know whether a finite-state description is even possible? And if it is, what is the smallest one?

The Myhill–Nerode theorem answers both questions at once, by reducing regularity to a single combinatorial test: count how many genuinely different "futures" the language distinguishes.

The central idea is best seen on an example before we formalize it.

Example 2.4.1 (Prefixes with the same future)**.** Consider the language L of words over $\{a, b\}$ that end in a. After reading the prefix ba, any continuation z that keeps the word in L must itself end in a. After reading the prefix aa, the situation is identical: exactly the same continuations work. So from L's point of view, ba and aa are indistinguishable—an automaton can merge them into a single state.

By contrast, the prefix b differs from ba: the empty continuation $z = \varepsilon$ puts $b \notin L$ but $ba \in L$. No automaton can afford to confuse them.

In fact, every prefix falls into one of two groups: those ending in a (which need continuation ending in a or the empty continuation) and those ending in b or the empty word (which always need more input ending in a). Two groups means two states—exactly the minimal DFA for "ends in a."

The example suggests a general recipe: group prefixes by their continuation behaviour. If the number of groups is finite, a DFA exists; if it is infinite, no finite automaton suffices. We now make this precise.

Definition 2.4.2 (Right congruence of a language)**.** For a language $L \subseteq \Sigma^*$, define $x \equiv_L y$ if and only if, for every continuation $z \in \Sigma^*$,

$$xz \in L \iff yz \in L.$$

Two prefixes are equivalent when no future suffix can separate them. After reading either prefix, the automaton needs exactly the same information to decide all possible continuations.

Definition 2.4.3 (Right-invariant equivalence)**.** An equivalence relation $\sim$ on Σ^* is **right-invariant** if

$$x \sim y \implies xz \sim yz \quad \text{for every } z \in \Sigma^*.$$

Operationally, right-invariance says that equivalence is preserved by appending: if two prefixes look the same to the language now, then reading one more symbol cannot make them look different. This is exactly the property that lets an automaton store a single state for both prefixes—after the append, both transitions lead to the same next state.

The relation $\equiv_L$ is right-invariant by definition: if no continuation separates x and y, then no longer continuation can separate xz and yz either.

Example 2.4.4 (The automaton relation)**.** Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Define

$$x \equiv_A y \iff \widehat{\delta}(q_0, x) = \widehat{\delta}(q_0, y).$$

This relation has at most $|Q|$ classes. It is right-invariant because once two prefixes put the automaton in the same state, appending the same suffix makes the two future computations identical.

Theorem 2.4.5 (Myhill–Nerode theorem)**.** *For any language $L \subseteq \Sigma^*$, the following three statements are equivalent:*

1. *L is a regular language.*
2. *L is the union of some equivalence classes of a right-invariant equivalence relation of finite index.*
3. *The specific right congruence relation $\equiv_L$ has finite index.*

Proof. We prove the equivalence in a cycle: (1) $\implies$ (2) $\implies$ (3) $\implies$ (1).

(1) $\implies$ (2). Assume L is regular, so it is recognized by some DFA $A = (Q, \Sigma, \delta, q_0, F)$. The automaton relation $\equiv_A$ (defined by $x \equiv_A y \iff \widehat{\delta}(q_0, x) = \widehat{\delta}(q_0, y)$) is a right-invariant equivalence relation and has at most $|Q|$ classes. Since L is exactly the set of words reaching states in F, L is exactly the union of the $\equiv_A$-classes corresponding to those final states.

(2) $\implies$ (3). Assume L is a union of classes of some right-invariant equivalence relation $\sim$ of finite index. If $x \sim y$, then for any continuation z, we have $xz \sim yz$. Since L is a union of $\sim$-classes, xz and yz must either both be in L or both outside L. This means $x \sim y \implies x \equiv_L y$. Thus, every $\sim$-class is fully contained in a single $\equiv_L$-class. Since $\sim$ has finite index, $\equiv_L$ can have at most the same finite number of classes.

(3) $\implies$ (1). Assume $\equiv_L$ has finite index. Build the quotient DFA

$$A_L = (\Sigma^*/\equiv_L, \Sigma, \delta, [\varepsilon], F_L)$$

where

$$\delta([w], a) = [wa] \quad \text{and} \quad F_L = \{[w] : w \in L\}.$$

The transition is well-defined because $\equiv_L$ is right-invariant: if $w \equiv_L w'$, then $wa \equiv_L w'a$. The accepting set is well-defined because taking the empty continuation $z = \varepsilon$ in the definition of $\equiv_L$ shows that equivalent words are either both in L or both outside L.

After reading a word w, the quotient automaton is in state $[w]$. Thus it accepts exactly the words whose equivalence class belongs to F_L, which is exactly L. □

Proposition 2.4.6 (Minimality)**.** *The quotient automaton built from $\equiv_L$ has the minimum possible number of states among all DFAs recognizing L.*

Proof. Let A be any DFA recognizing L. If two words reach the same state in A, then they are $\equiv_L$-equivalent by the first half of the previous proof. Therefore a single state of A can represent only one Myhill–Nerode class. Since every class is reached by some word, A must have at least as many states as there are $\equiv_L$-classes. The quotient automaton has exactly that many states, so it is minimal. □

The minimal DFA for L is therefore canonical up to renaming of states: merge exactly the prefixes with the same continuation behaviour, and no more.

Notation Recap

Symbol	Meaning
$r+s, rs, r^*$	Union, concatenation, Kleene star of regular expressions
ε-NFA	NFA extended with ε-labelled (free) transitions
$\mathsf{ECl}(q)$	ε-closure: states reachable from q by ε-moves
$\overline{A}$	Complement automaton (swap final/non-final in a complete DFA)
$A_1 \times A_2$	Product automaton for intersection
$F_1 \times F_2$	Accepting set of the product (both components accept)
$\mathsf{Post}(X)$	States reachable in one transition from X
R^*	Reachability fixpoint from the initial state
$R^k_{i,j}$	Words taking q_i to q_j through intermediate states $q_1, \dots, q_k$
$x \equiv_L y$	Myhill–Nerode equivalence: x and y have the same continuations in L
A_L	Quotient (minimal) DFA built from $\equiv_L$

Summary & Key Takeaways

- Regular expressions and finite automata define the same finite-word languages.
- Complementation needs complete deterministic automata.
- Intersection is implemented by a product construction.
- Inclusion and equivalence reduce to emptiness.
- Myhill–Nerode explains why regular languages are exactly those with finitely many distinguishable prefixes.

Exercises

Exercise 1 (Complementation needs completeness). Draw a partial DFA over $\{a, b\}$ that accepts only the word a. If you swap final and non-final states without adding a sink state, what language do you get? Compare it with the true complement of $\{a\}$.

Exercise 2 (Product construction for the turnstile). The turnstile from chapter 1 has two events: `coin` and `push`. Let A_1 accept sequences in which every `push` is immediately preceded by a `coin`, and let A_2 accept sequences containing at least one `push`. Construct the product automaton for $L(A_1) \cap L(A_2)$ and list its accepting states. Give the shortest accepted word.

Exercise 3 (Myhill–Nerode classes for even-length words). For the language of words over $\{a, b\}$ whose length is even, list the $\equiv_L$-classes and build the quotient DFA.

Exercise 4 (ε-removal). Build an ε-NFA for the expression $a+b^*$ using the union construction of fig. 2.3. Then apply the ε-removal algorithm of theorem 2.1.10: compute all ε-closures and write down the resulting ordinary NFA.

Exercise 5 (Deciding inclusion). Let A_1 accept words over $\{a, b\}$ that start with a, and let A_2 accept words containing at least one a. Is $L(A_1) \subseteq L(A_2)$?

Justify your answer by constructing the automaton for $L(A_1) \cap \overline{L(A_2)}$ and checking emptiness.

Exercise 6 (Small model witness). A DFA with 4 states over $\{a, b\}$ has the following transitions: $\delta(q_0, a) = q_1$, $\delta(q_0, b) = q_0$, $\delta(q_1, a) = q_2$, $\delta(q_1, b) = q_0$, $\delta(q_2, a) = q_3$, $\delta(q_2, b) = q_0$, $\delta(q_3, a) = q_3$, $\delta(q_3, b) = q_3$, with $F = \{q_3\}$. Find the shortest accepted word. Verify that its length is less than the number of states, as guaranteed by theorem 2.3.3.

Exercise 7 (Reachability fixpoint). For the DFA in the previous exercise, compute the fixpoint sequence $R_0, R_1, R_2, \ldots$ from section 2.3.1 and verify that the sequence stabilizes. At which iteration does an accepting state first appear in R_i?

CHAPTER 3

Automata over Infinite Words

Finite-word automata are well suited for batch computations: a compiler receives a finite program, a parser receives a finite string, a protocol message has a finite body. Verification of reactive systems is different. Operating systems, controllers, servers, and embedded devices are meant to keep interacting with their environment.

Reactive systems do not produce one finite output and stop. Their behaviour is an infinite interaction with the environment.

Their behaviours are therefore not finite words but infinite words. To model them we keep the finite-state graph, but change the acceptance condition.

Where we are. Chapters 1 and 2 developed the finite-word theory: automata recognise finite behaviours, regular expressions describe the same languages algebraically, and emptiness reduces verification questions to graph reachability.

What this chapter adds. We move from finite behaviours to infinite executions. The graph remains finite, but acceptance must now describe what happens in the limit.

Where it leads. Büchi automata become the automata-theoretic model behind liveness, model checking, complementation, and temporal logic in the following chapters.

Chapter map.

- Section 3.1 introduces infinite words as executions.
- Section 3.2 lifts language operations to ω-languages.
- Section 3.3 defines Büchi automata and their runs.
- Section 3.4 gives closure constructions and ω-regular expressions.
- Section 3.5 reduces non-emptiness to reachable lassos.

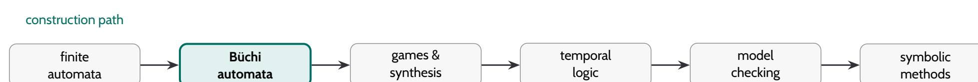


Figure 3.1: Where this chapter sits in the construction path of the book. Each step reuses the previous one as a representation or an algorithmic primitive.

Running example — request/acknowledgement traces

Throughout the infinite-word chapters, think of a reactive component that observes requests and acknowledgements forever. A typical liveness property is "acknowledgement occurs infinitely often", and the counterexamples are ultimately periodic traces where acknowledgements eventually disappear.

3.1 Infinite Words

Consider the request/acknowledgement component from the running example. One possible execution is

$$\texttt{req}\ \texttt{ack}\ \texttt{req}\ \texttt{req}\ \texttt{ack} \cdots .$$

The dots are not an omitted ending. They are part of the model: the component is expected to keep receiving inputs and producing observations forever. This is why we need words with positions $0, 1, 2, \ldots$ rather than words with a last symbol.

Definition 3.1.1 (ω-word)**.** Let Σ be a finite alphabet. An ω**-word** over Σ is an infinite sequence

$$\alpha = \alpha_0\alpha_1\alpha_2 \cdots \quad \text{with each } \alpha_i \in \Sigma.$$

The set of all infinite words over Σ is denoted by Σ^ω.

An ω-word is a function from the natural numbers to Σ: position 0 gets a symbol, position 1 gets a symbol, and so on without end. The key operational difference from finite words is that there is no "last symbol" to inspect, so every property of an ω-word must be phrased in terms of what happens eventually or repeatedly, never in terms of how the word ends.

We use u, v, w for finite words and Greek letters such as α, β for infinite words. This distinction matters because finite words can be consumed completely, while infinite words have no last position.

Definition 3.1.2 (Segments and suffixes)**.** For an ω-word α, the finite segment $\alpha[i, j]$ denotes the word

$$\alpha_i\alpha_{i+1} \cdots \alpha_{j-1}.$$

The infinite suffix $\alpha[i, \omega]$ denotes

$$\alpha_i\alpha_{i+1}\alpha_{i+2} \cdots .$$

The convention $\alpha[i, i] = \varepsilon$ keeps formulas uniform: a segment with equal endpoints consumes no symbols.

Definition 3.1.3 (Infinitely often)**.** For an infinite word $\alpha \in \Sigma^\omega$, define

$$\inf(\alpha) = \{a \in \Sigma : a \text{ occurs infinitely often in } \alpha\}.$$

Because Σ is finite and α is infinite, at least one symbol must occur infinitely often. The set $\inf(\alpha)$ records the long-run behaviour and forgets finite prefixes.

Example 3.1.4 (A long-run property)**.** Let

$$\alpha = ab\ a\ bb\ a\ bbb\ a\ bbbb \cdots .$$

Then a occurs infinitely often and b occurs infinitely often, so $\inf(\alpha) = \{a, b\}$.

3.2 From Finite Languages to Infinite Languages

We now have infinite words, but no way to describe sets of them. Rather than starting from scratch, we can lift familiar finite-word operations to the infinite setting. Some infinite languages can be built directly from finite-word languages.

Definition 3.2.1 (ω-power)**.** If $W \subseteq \Sigma^*$, then

$$W^\omega = \{w_0 w_1 w_2 \cdots : w_i \in W \text{ for every } i \geq 0\}$$

is the set of infinite concatenations of blocks from W.

Usually we assume $\varepsilon \notin W$ when using W^ω, because empty blocks do not help construct an infinite word and can obscure the intended decomposition.

Definition 3.2.2 (Vectorial closure)**.** For $W \subseteq \Sigma^*$, define

$$\vec{W} = \{\alpha \in \Sigma^\omega : \alpha[0, n] \in W \text{ for infinitely many } n\}.$$

The language $\vec{W}$ contains the infinite words whose finite prefixes hit W infinitely often.

Example 3.2.3 (Vectorial closure on the running example)**.** Over $\Sigma = \{\texttt{req}, \texttt{ack}\}$, let W be the set of finite words that end with `ack`. Then $\vec{W}$ is the set of infinite traces such that infinitely many finite prefixes end with `ack`—in other words, exactly the traces in which `ack` occurs infinitely often.

The vectorial closure makes the connection to acceptance explicit. It is a natural way to express recurrence: not just "a good prefix appears", but "good prefixes keep appearing". For a deterministic Büchi automaton this is exactly the operational shape of acceptance: if W is the set of finite prefixes that leave the automaton in an accepting state, then the accepted ω-language is $\vec{W}$. We will return to this observation in chapter 9, where it becomes the precise characterisation of deterministic Büchi languages.

3.3 Büchi Automata

A Büchi automaton has the same graph shape as an NFA—a finite set of states, labelled transitions, an initial state, and a set of accepting states—but it reads an infinite input rather than a finite one. Since the input never ends, there is no "final state after the last symbol". Instead, the automaton asks a long-run question: does the run keep returning to an accepting state, or does it eventually stay away forever? The tuple below captures exactly this setup: the graph is finite, but the acceptance condition talks about what happens in the limit.

Definition 3.3.1 (Büchi automaton)**.** A **Büchi automaton** is a tuple

$$A = (Q, \Sigma, \Delta, q_0, F)$$

where Q is finite, $\Delta \subseteq Q \times \Sigma \times Q$, $q_0 \in Q$, and $F \subseteq Q$ is the set of accepting states.

Read the tuple as a finite directed graph whose edges are labelled by symbols: Q is the set of nodes, Δ is the set of labelled edges, q_0 marks the starting node, and F marks the checkpoints that the run must keep visiting. Everything is identical to an NFA except the role of F: in an NFA, F says where a run must *end*; in a Büchi automaton, F says where a run must *return* infinitely often.

Definition 3.3.2 (Run and Büchi acceptance)**.** Let $\alpha = \alpha_0\alpha_1\alpha_2\cdots$. A run of A on α is an infinite sequence of states

$$\rho = \rho_0\rho_1\rho_2\cdots$$

such that $\rho_0 = q_0$ and $(\rho_i, \alpha_i, \rho_{i+1}) \in \Delta$ for every $i \geq 0$.

The run is **accepting** if

$$\inf(\rho) \cap F \neq \emptyset.$$

The word α is accepted if there exists at least one accepting run on α.

For finite words, we ask where the run ends. For infinite words, there is no end. Büchi acceptance instead asks whether the run visits some accepting state infinitely many times.

This is also why "membership" changes character. For a finite word we can simulate the automaton to the last symbol and inspect the final state. For an ω-word there is no last symbol to wait for. The decision procedures therefore reason about the finite graph of the automaton, looking for a finite pattern that can repeat forever.

Definition 3.3.3 (ω-regular language)**.** A language $L \subseteq \Sigma^\omega$ is ω**-regular** if it is accepted by some Büchi automaton.

This is the infinite-word counterpart of "regular language". It tells us that the class of properties expressible by Büchi automata has a name and will turn out to have the same robust characterisations (expressions, closure, decidable emptiness) that regular languages enjoy.

Example 3.3.4 (Infinitely many acknowledgements)**.** Over $\Sigma = \{\texttt{req}, \texttt{ack}\}$, the property "`ack` occurs infinitely often" is Büchi-recognizable. Use two states: a non-accepting scanning state and an accepting state entered whenever the current symbol is `ack`. The accepting state is visited infinitely often exactly when infinitely many `ack` symbols appear.

Figure 3.2 shows the whole construction. The accepting state is not a final destination; it is a checkpoint that the run must keep returning to.

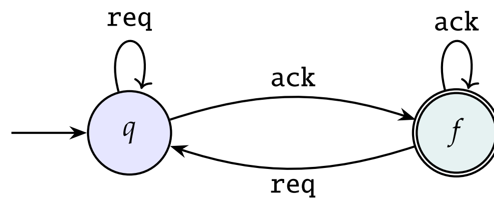


Figure 3.2: A Büchi automaton for infinitely many acknowledgements.

Example 3.3.5 (Parity between occurrences)**.** Here is a richer example, adapted from Wolfgang Thomas. Over the alphabet $\Sigma = \{a, b, c\}$, consider the language:

> "Between any two consecutive occurrences of a, there is an even number of occurrences of b and c combined."

For instance, $a\,b\,c\,a$ is accepted (two symbols between the a's), but $a\,b\,b\,b\,a$ is rejected (three symbols). Words with fewer than two occurrences of a vacuously satisfy the condition.

The Büchi automaton uses its states to track the parity of the non-a symbols since the last a:

- q_0 (initial, accepting): The empty prefix, or reading b, c before the first a.
- q_1 (accepting): The last symbol was a, or we have seen an even number of b/c symbols since the last a.
- q_2 (non-accepting): We have seen an odd number of b/c symbols since the last a.

If the automaton reads an a while in q_2, it transitions to a rejecting sink (omitted for clarity), because a forbidden odd-length block has been sealed by an a.

Figure 3.3 makes the memory explicit: after the first a, the automaton only needs to remember whether the current block length is even or odd.

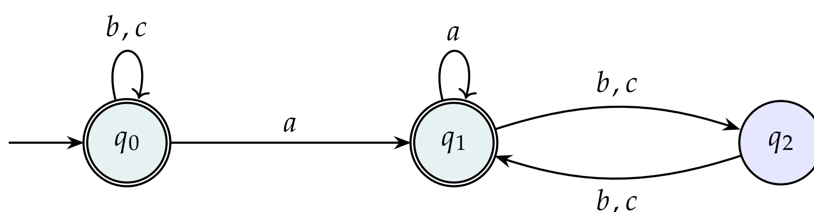


Figure 3.3: A Büchi automaton verifying the parity of $\{b, c\}$ blocks between occurrences of a.

3.4 Closure and Expressions

The examples above show how to design individual automata. For verification, however, we also need to combine specifications. If one automaton describes "requests are acknowledged infinitely often" and another describes a fairness assumption about the environment, then model checking will ask about their conjunction, disjunction, or about a finite initialisation phase followed by an

infinite behaviour. Closure properties tell us that these combinations stay inside the same automata model.

Theorem 3.4.1 (Basic closure properties). *ω-regular languages are closed under finite union and intersection. Moreover, if $U \subseteq \Sigma^*$ is regular and $L \subseteq \Sigma^\omega$ is ω-regular, then $U \cdot L$ is ω-regular; if $V \subseteq \Sigma^*$ is regular, then V^ω is ω-regular.*

The constructions mirror the finite case, but one detail changes: acceptance is about visiting final states infinitely often. In product constructions, we must ensure that both automata satisfy their Büchi conditions, not just that both components are accepting at the same single time.

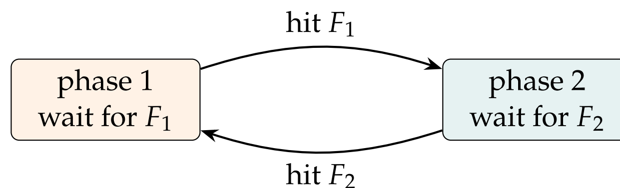
■ **Formal details — Construction ideas**

For $U \cdot L$, first run a finite-word automaton for U. Whenever it reaches an accepting state, nondeterministically switch to a Büchi automaton for L. The switch consumes no future symbols; it only marks where the finite prefix ends.

For V^ω, start from an automaton for the finite language V. Assume, after the standard harmless normalization, that it has no ε-moves and that no transition enters its initial state. Whenever a transition would enter an accepting state after reading a symbol a, duplicate that effect by sending the same a-transition back to the initial state of the V-automaton, passing through a Büchi accepting marker. Each visit to the marker means that one finite block from V has just been completed. Infinitely many marker visits mean infinitely many completed V-blocks.

For intersection of two Büchi automata, the naive product with $F_1 \times F_2$ is too strict: one run may visit F_1 at even times and the other may visit F_2 at odd times. The correct product stores a phase bit. In phase 1 it waits until the first component visits F_1, then switches to phase 2; in phase 2 it waits until the second component visits F_2, then switches back to phase 1. The accepting states are the states that complete one of these obligations. An accepting run must therefore complete both obligations infinitely often.

Figure 3.4 depicts only the control idea of the intersection product. The actual product also carries the two automaton states; the phase records which acceptance obligation is currently being checked.



The product accepts only if both phases are completed infinitely often.

Figure 3.4: Asynchronous Büchi intersection is handled by alternating obligations.

Definition 3.4.2 (ω-regular expression). An **ω-regular expression** denotes a finite union of languages of the form

$$U_i V_i^\omega,$$

where each $U_i, V_i \subseteq \Sigma^*$ is regular.

Each term $U_i V_i^\omega$ describes a two-phase behaviour: a finite initialisation from U_i, followed by an infinite repetition of blocks from V_i. The full expression is a finite union of such terms, so it says "the system does one of finitely many things: starts with some prefix, then loops forever in some pattern". This is the infinite-word analogue of a regular expression.

Theorem 3.4.3 (Büchi-expression equivalence). *A language $L \subseteq \Sigma^\omega$ is accepted by a Büchi automaton if and only if it can be denoted by an ω-regular expression.*

Proof. We again prove both directions constructively.

Expression to automaton. It is enough to handle one term UV^ω, because Büchi automata are closed under finite union. Let A_U be an automaton for the regular finite-word language U, and let A_V be an automaton for V. The Büchi automaton first simulates A_U. Whenever A_U reaches an accepting state, the automaton may non-deterministically switch to a copy of A_V. From then on, each time the A_V copy accepts one block from V, the automaton returns to the beginning of the A_V copy and passes through a Büchi accepting marker state.

Thus a run is accepting exactly when it reads one prefix from U and then completes infinitely many consecutive blocks from V.

Automaton to expression. Let $A = (Q, \Sigma, \Delta, q_0, F)$ be a Büchi automaton. For states $p, q \in Q$, let $W_{p,q}$ be the regular language of finite words that label some path from p to q in the underlying finite automaton. This language is regular because it is accepted by the same finite graph with initial state p and accepting state q.

If α is accepted, then some accepting run visits at least one state $f \in F$ infinitely many times. Split the run at successive visits to that same f. The finite prefix before the first selected visit is labelled by a word in $W_{q_0,f}$, and every later block is labelled by a non-empty word in $W_{f,f}$. Therefore

$$\alpha \in W_{q_0,f}(W_{f,f} \setminus \{\varepsilon\})^\omega.$$

Taking the union over all $f \in F$ gives an ω-regular expression containing every accepted word.

Conversely, any word in one of these products describes a path from q_0 to f, followed by infinitely many non-empty paths from f back to f. Following those paths gives a run that visits f infinitely often, hence is accepting. So the expression denotes exactly $L(A)$. □

3.5 Emptiness and Lassos

The most important algorithmic fact is that Büchi emptiness is still a graph problem.

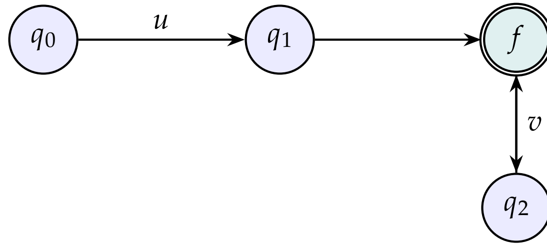
Proposition 3.5.1 (Büchi emptiness). *Let A be a Büchi automaton. Then $L(A) \neq \emptyset$ if and only if some accepting state is reachable from q_0 and belongs to a directed cycle.*

Proof. If an accepting state f is reachable and lies on a cycle, follow the path to f once and then repeat the cycle forever. This produces an infinite run that visits f infinitely often.

Conversely, any accepting run visits some accepting state f infinitely many times. Between two visits to f there is a non-empty finite path from f back to f, hence a cycle reachable from q_0. □

Figure 3.5 is the finite witness hidden inside the infinite run: a prefix reaches an accepting cycle, and the cycle can then be repeated without end.

A counterexample in model checking often has this shape: a finite prefix followed by a repeated cycle.



A reachable accepting cycle yields an infinite accepting run.

Figure 3.5: The lasso witness for non-emptiness: a finite prefix followed by a repeated cycle.

Definition 3.5.2 (Ultimately periodic word)**.** An ω-word is **ultimately periodic** if it has the form

$$uv^\omega$$

for finite words $u \in \Sigma^*$ and non-empty $v \in \Sigma^*$.

Every non-empty ω-regular language contains an ultimately periodic word. The graph-theoretic lasso in theorem 3.5.1 is the automata reason: a finite path gives u, and the repeated cycle gives v^ω.

Example 3.5.3 (Languages without periodic words)**.** Consider the infinite word

$$\beta = a\, b\, a\, b^2\, a\, b^3\, a\, b^4 \cdots .$$

Both a and b occur infinitely often, but the gaps between consecutive a's keep growing. The word is not ultimately periodic: in an ultimately periodic word, the distances between repeated occurrences are eventually governed by one fixed period.

Can we build a non-empty language that is not ω-regular?

We could simply take the singleton language $\{\beta\}$. If it were ω-regular, by the lasso property, it would have to contain an ultimately periodic word. But its only word is β, which is not ultimately periodic. So $\{\beta\}$ is not ω-regular.

That argument is correct, but it may feel too easy: a finite automaton for one fixed aperiodic word already sounds impossible. A more robust counterexample should still have many words.

To make the obstruction robust, consider Wolfgang Thomas's example:

$$L_\beta = \{u\beta : u \in \Sigma^*\}.$$

This language is infinite: we can prepend any finite prefix to β. Is L_β ω-regular? If it were, it would have to contain an ultimately periodic word.

But every word in L_β shares the suffix β, meaning it eventually behaves exactly like β and cannot be ultimately periodic.

Thus, L_β is not ω-regular. The absence of an ultimately periodic word is a structural barrier to ω-regularity, even for large infinite languages.

Notation Recap

Symbol	Meaning
Σ^ω	Set of all infinite words over Σ
$\alpha[i, j]$	Finite segment $\alpha_i \cdots \alpha_{j-1}$
$\alpha[i, \omega]$	Infinite suffix from position i
$\inf(\alpha)$	Symbols occurring infinitely often in α
W^ω	Infinite concatenation of blocks from $W \subseteq \Sigma^*$
$\vec{W}$	Vectorial closure: ω-words whose prefixes hit W infinitely often
uv^ω	Ultimately periodic word: finite prefix u, repeated cycle v
$(Q, \Sigma, \Delta, q_0, F)$	Büchi automaton tuple
F	Accepting states (must be visited infinitely often)

Summary & Key Takeaways

- Infinite behaviours are modelled as words in Σ^ω.
- Büchi automata accept runs that visit final states infinitely often.
- ω-regular languages are exactly finite unions of UV^ω.
- Emptiness reduces to finding a reachable accepting cycle.
- The lasso shape is the finite witness behind many infinite counterexamples.

Exercises

Exercise 1 (Computing an infinity set). Let

$$\alpha = a\,b\,b\,a\,b\,b\,b\,a\,b\,b\,b\,b \cdots,$$

where the number of b's between consecutive a's increases by one. Compute $\inf(\alpha)$. Is α ultimately periodic?

Exercise 2 (Büchi acceptance). Modify fig. 3.2 to accept the property "only finitely many `req` symbols occur". Hint: the automaton will need a non-deterministic guess.

Exercise 3 (Finding a lasso). Given a Büchi automaton graph, explain how you would use strongly connected components to decide emptiness. Which SCCs are relevant?

Exercise 4 (Writing an ω-regular expression). Over $\Sigma = \{\texttt{req}, \texttt{ack}\}$, write an ω-regular expression for the language "`ack` occurs infinitely often". Write a second expression for "`ack` occurs only finitely often".

CHAPTER 4

Complementation of ω-Regular Languages

Model checking is, at its core, an inclusion question. Given a system automaton A_{sys} and a specification automaton A_{spec}, we ask whether every behaviour allowed by the system is also allowed by the specification:

For finite words, complementing a DFA is a one-line trick. For Büchi automata, this is where the theory becomes genuinely subtle.

$$L(A_{sys}) \subseteq L(A_{spec}).$$

In chapter 2 we reduced the finite-word version of this question to emptiness, using the identity

$$L(A_{sys}) \cap \overline{L(A_{spec})} = \emptyset.$$

The same reduction works for ω-words, provided we can complement a Büchi automaton. Intersection and emptiness are already manageable: intersection uses the asynchronous product of fig. 3.4, and emptiness reduces to finding a reachable accepting cycle (theorem 3.5.1). The hard part is complementation.

For finite words, complementing a complete DFA is literally free: swap the accepting and non-accepting states, and the sink completion of chapter 1 does the rest. For Büchi automata this route is closed, and the detour is non-trivial. This chapter develops the classical algebraic proof that ω-regular languages are nonetheless closed under complement. The roadmap is a chain of refinements, each section answering a question forced by the previous one:

- why does the finite-word trick fail, and what replaces it? (section 4.1);
- finite-index congruences, which summarize finite words by finitely many labels (section 4.2);
- transition profiles, the concrete congruence induced by a Büchi automaton (section 4.3);
- a Ramsey-style decomposition, which represents every infinite word as a prefix followed by infinitely many congruent blocks (section 4.4);
- the saturation construction, which turns these ingredients into an effective automaton for the complement (section 4.5);
- and finally the decidability of inclusion, the model-checking payoff (section 4.6).

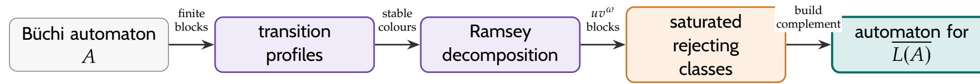


Figure 4.1: The complementation proof is not a state-flipping construction. It compresses finite blocks into profiles, uses Ramsey stability to describe infinite words by ultimately periodic blocks, and then builds the complement from the saturated rejecting classes.

4.1 Why the Finite Trick Fails

For a finite-word language recognized by a DFA, complementation is a syntactic operation: complete the automaton with a sink, then swap accepting and non-accepting states. Correctness is immediate, because on each finite word exactly one of "the run ends in F" and "the run ends in $Q \setminus F$" holds.

Two things break for Büchi automata. First, the automata are non-deterministic, so there is no single run whose endpoint we can flip. Second, and more fundamentally, deterministic Büchi automata are strictly weaker than non-deterministic ones, so we cannot even determinize first and then swap.

Example 4.1.1 (The complement of the running request/ack language)**.** Recall from theorem 3.3.4 the Büchi automaton for "`ack` occurs infinitely often" over $\Sigma = \{\texttt{req}, \texttt{ack}\}$. Its complement is

$$\overline{L} = \{\alpha : \text{only finitely many } \texttt{ack} \text{ occur in } \alpha\},$$

the property that from some position onwards no acknowledgement ever reappears.

It is easy to recognize $\overline{L}$ *non-deterministically*: guess the position after which no more `ack` arrives, then stay in an accepting state while reading only `req`. But a *deterministic* Büchi automaton cannot recognize $\overline{L}$. To accept, it would have to visit an accepting state infinitely often along every word of $\overline{L}$; yet while it is still seeing `ack` symbols (which any finite prefix of a word in $\overline{L}$ may contain) it has no deterministic way to know whether this `ack` is the last one. By the time it can be certain, finitely many accepting visits may already have been spent.

The lesson is not that complementation is impossible. The class of ω-regular languages *is* closed under complement, and we will prove it. The lesson is that the proof cannot be the finite-word proof: there is no state-swapping shortcut. We need machinery that reasons about infinite runs without ever "waiting for the end". Finite-index congruences are exactly that machinery.

4.2 Congruences and Saturation

The algebraic route partitions finite words into finitely many classes, then uses products of classes to describe infinite words.

The intuition is familiar from everyday arithmetic: we often care not about the exact value of a number but about a coarse summary — even or odd, positive or negative, its residue modulo 7. Two numbers sharing the same summary can be swapped freely in any calculation that depends only on the summary. A congruence on Σ^* does the same thing for words: it assigns a

finite label to every string, and the label of a concatenation depends only on the labels of the pieces.

Definition 4.2.1 (Congruence)**.** An equivalence relation $\equiv$ on Σ^* is a **congruence** if

$$u \equiv v \text{ and } u' \equiv v' \quad \Longrightarrow \quad uu' \equiv vv'.$$

It has **finite index** if it has finitely many equivalence classes.

Congruence means that only the class matters when concatenating. If two blocks are interchangeable now, they remain interchangeable in any larger finite context.

Example 4.2.2 (Parity of length, a finite-index congruence)**.** Over any alphabet, declare $u \equiv_{\text{par}} v$ when $|u|$ and $|v|$ have the same parity. There are two classes, even-length and odd-length words. Concatenating two words of known parity gives a word whose parity is determined by a tiny addition table (even + even = even, odd + even = odd, and so on), so $\equiv_{\text{par}}$ is a congruence of index 2.

Parity is far too coarse to be useful for verification, but it captures the shape of the idea: a congruence summarizes each finite word by a label drawn from a finite set, and concatenation acts on these labels through a finite table. The automaton congruence we build later in this chapter has exactly this shape, with a much richer label set.

Definition 4.2.3 (Saturation)**.** Let $\equiv$ be a finite-index congruence on Σ^*. An ω-language $L \subseteq \Sigma^\omega$ is **saturated** by $\equiv$ if, for every pair of equivalence classes U, V, the language

$$UV^\omega$$

is either entirely contained in L or disjoint from L.

This is a strong finiteness property. It says that once we know the class of a finite prefix and the class repeated forever, the membership of the resulting infinite word is fixed.

Proposition 4.2.4 (Saturation and complement)**.** *If a language L is saturated by a finite-index congruence $\equiv$, then its complement $\overline{L}$ is saturated by the same congruence.*

Proof. Fix two equivalence classes U, V. Since L is saturated, UV^ω is either contained in L or disjoint from L. In the first case UV^ω is disjoint from $\overline{L}$; in the second it is contained in $\overline{L}$. Thus the same all-or-nothing condition holds for the complement. □

Lemma 4.2.5 (Complement from saturation)**.** *If L is saturated by a finite-index congruence $\equiv$, then $\overline{L}$ is ω-regular.*

Proof. First note that every equivalence class of a finite-index congruence is a regular finite-word language. Indeed, build a DFA whose states are the

equivalence classes, whose initial state is $[\varepsilon]$, and whose transition is $[u] \xrightarrow{a} [ua]$. Congruence makes this transition well-defined.

There are only finitely many pairs of classes (U, V). Since L is saturated, each product UV^ω is either contained in L or disjoint from L. Therefore

$$\overline{L} = \bigcup \{UV^\omega : UV^\omega \cap L = \emptyset\}.$$

This is a finite union. Each U and V is regular, so each UV^ω is ω-regular, and finite unions of ω-regular languages are ω-regular. □

So complementation reduces to two tasks: finding a finite-index congruence that saturates $L(A)$, and showing that such a congruence always exists. The next two sections supply both. First we build, from the automaton A itself, a concrete finite-index congruence $\approx_A$ that knows about accepting visits. Then we invoke a Ramsey-style decomposition to argue that $\approx_A$ does saturate $L(A)$.

4.3 The Automaton Congruence

Let $A = (Q, \Sigma, \Delta, q_0, F)$ be a Büchi automaton. A finite word can be summarized by what it does to pairs of states. The summary must record not only reachability between states, but also whether accepting states were seen along the way: this is the new ingredient that finite automata never needed and that Büchi acceptance forces on us.

> **Definition 4.3.1** (Path profile notation)**.** For $p, q \in Q$ and $u \in \Sigma^*$, write
>
> $$p \xrightarrow{u} q$$
>
> when there is a path from p to q labelled u. Write
>
> $$p \xrightarrow{u}_F q$$
>
> when there is such a path that visits at least one accepting state in F.

How to read this: $p \xrightarrow{u} q$ answers "can the automaton reach q from p by reading u?", while $p \xrightarrow{u}_F q$ answers the sharper question "can it do so while passing through an accepting state?". The first relation is ordinary reachability; the second is the acceptance-aware refinement that Büchi complementation will need.

Here "visits" includes the endpoints of the finite path. Thus, for $u = \varepsilon$, we have $p \xrightarrow{\varepsilon}_F p$ exactly when $p \in F$. The convention is harmless but important: when transition profiles are concatenated, accepting information at the gluing state is not silently lost.

The second relation is the one that finite-word automata do not need. For Büchi acceptance, a repeated block is useful only if it can be traversed while seeing acceptance.

Definition 4.3.2 (Transition profile)**.** For a finite word $u \in \Sigma^*$ and states $p, q \in Q$, record two facts:
- whether there exists a path from p to q labelled u;
- whether there exists such a path that visits some state in F.

Two words are equivalent, written $u \approx_A v$, if they have the same answers for every pair (p, q).

The operational test is: given two finite words u and v, check every ordered pair of states (p, q) and ask three questions — is q reachable from p via u? via v? and in each case, does an accepting path exist? If u and v give identical answers on every pair, they are $\approx_A$-equivalent. Equivalent words are interchangeable as finite blocks inside any infinite run without affecting acceptance.

The relation $\approx_A$ has finite index because each word determines a finite Boolean table over $Q \times Q$. It is also a congruence: the transition profile of uv is obtained by composing the profiles of u and v.

Proposition 4.3.3 (Index bound)**.** *If A has n states, then $\approx_A$ has at most 3^{n^2} equivalence classes.*

Proof. Fix a finite word u and a pair of states (p, q). With respect to that pair, there are three relevant statuses:
1. no path labelled u goes from p to q;
2. some path labelled u goes from p to q, but no such path visits an accepting state;
3. some path labelled u goes from p to q and visits an accepting state.

There are n^2 ordered pairs (p, q). A word therefore determines a matrix with n^2 entries, each entry chosen from at most three statuses. There are at most 3^{n^2} such matrices. Two words with the same matrix are $\approx_A$-equivalent, so the number of equivalence classes is at most 3^{n^2}. □

Example 4.3.4 (What the profile remembers)**.** Suppose $Q = \{p, q, r\}$ and $F = \{r\}$. For a block u, the profile does not remember the exact path labelled by u. It remembers whether u can take p to q, p to r, and so on, and whether an accepting state was seen along the way.

This is exactly the information needed to reason about repeating finite blocks inside an infinite run. Before proving the saturation property, let us make the definition concrete on a small automaton that we will reuse throughout the chapter.

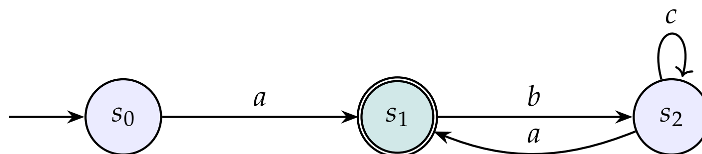


Figure 4.2: A small Büchi automaton A to illustrate transition profiles. The only accepting state s_1 is shown in teal. The automaton accepts the ω-words in which the block ab occurs infinitely often, separated by arbitrary c's.

Example 4.3.5 (Two profiles on a small automaton)**.** Consider the Büchi automaton A of fig. 4.2, with $F = \{s_1\}$. We compute the transition profiles of two short words.

For $u = c$, the only non-trivial entry is $s_2 \xrightarrow{c} s_2$, and it does *not* visit F (the path stays in $s_2 \notin F$). Every other pair (p, q) has no c-labelled path.

For $u = ab$, we have $s_0 \xrightarrow{ab}_F s_2$ along the path $s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2$, which visits $s_1 \in F$, and $s_2 \xrightarrow{ab}_F s_2$ along the path $s_2 \xrightarrow{a} s_1 \xrightarrow{b} s_2$, which also visits s_1. No other pair has an ab-labelled path: in particular s_1 has no outgoing a-transition, so $s_1 \xrightarrow{ab} q$ never holds.

The two words c and ab therefore lie in different $\approx_A$-classes: the profile of c has a single non-F entry at (s_2, s_2), while the profile of ab has two F-entries. This is the kind of finite summary that $\approx_A$ attaches to every finite word.

■ **Formal details — Replacement principle**

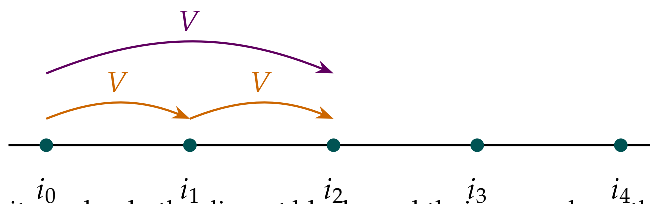
If $u \approx_A v$, then every finite path fact about u that matters for Büchi acceptance also holds for v with the same endpoints:

$$p \xrightarrow{u} q \iff p \xrightarrow{v} q, \qquad p \xrightarrow{u}_F q \iff p \xrightarrow{v}_F q.$$

Thus a run built from u-blocks can be imitated block by block with v-blocks, preserving the accepting visits that make the infinite run Büchi-accepting. This is the local reason why transition profiles saturate ω-regular languages.

4.4 Ramsey Decomposition

The saturation lemmas above reduce complementation to a question about class products UV^ω. But a generic ω-word α need not be literally of the form uv^ω; it may have no period at all. To use saturation on *every* ω-word, we need to know that α can nonetheless be partitioned into finite blocks that all share one congruence class. This is a combinatorial fact about infinite words and finite colourings; it is the content of Ramsey's theorem.



Homogeneity makes both adjacent blocks and their merge have the same class.

Figure 4.3: Ramsey homogeneity: all selected segments carry the same congruence class.

The picture in fig. 4.3 is the image to keep while reading the proof. We cut the infinite word at an increasing sequence of positions $i_0 < i_1 < i_2 < \cdots$, and the segments between consecutive cuts all carry the same congruence colour V. Because the congruence has finite index, Ramsey's theorem guarantees

that such an infinite monochromatic sequence of cuts always exists — it is a pigeonhole argument played out in two dimensions over the positions of α.

Lemma 4.4.1 (Homogeneous decomposition)**.** *Let $\equiv$ be a finite-index congruence on Σ^*. Every ω-word α can be decomposed as*

$$\alpha = u_0 u_1 u_2 \cdots$$

where $u_0 \in \Sigma^*$, each u_i for $i \geq 1$ is non-empty, and all blocks u_i with $i \geq 1$ belong to the same $\equiv$-class V. Moreover, the class can be chosen idempotent:

$$V \cdot V \subseteq V.$$

Proof. Colour each pair of positions (i, j) with the $\equiv$-class of the finite segment $\alpha[i, j]$. Since there are finitely many classes, the infinite Ramsey theorem gives an infinite set of positions

$$i_0 < i_1 < i_2 < \cdots$$

such that all segments $\alpha[i_m, i_n]$ have the same colour. Taking $u_0 = \alpha[0, i_0]$ and $u_k = \alpha[i_{k-1}, i_k]$ gives the desired decomposition.

To see idempotence, take any two words $x, y \in V$. Since V is an equivalence class and $\equiv$ is a congruence, the class of xy is the same as the class of a concatenation of two consecutive homogeneous segments:

$$\alpha[i_0, i_1]\alpha[i_1, i_2] = \alpha[i_0, i_2].$$

But $\alpha[i_0, i_2]$ has the same homogeneous colour V. Hence $xy \in V$, so $V \cdot V \subseteq V$. □

The point is conceptual: although the word may not be periodic, it can be partitioned into blocks that are indistinguishable by the finite congruence, and the repeated block class is closed under concatenating its own blocks.

Example 4.4.2 (Ramsey decomposition on the running automaton)**.** Return to the automaton A of fig. 4.2. Consider the ω-word

$$\alpha = a \underbrace{bc}_{v_1} \underbrace{abc}_{v_2} \underbrace{abc}_{v_3} \cdots$$

which is *not* ultimately periodic (we could scatter extra c's irregularly), yet we can cut it after the initial a and then into blocks $bc, abc, abc, \ldots$ The blocks abc all have the same $\approx_A$-profile: they take s_0 to s_2 while visiting $s_1 \in F$, and take s_2 to s_2 while visiting $s_1 \in F$. The first block bc may differ, but every block from the second onward shares the same class V. This is exactly the decomposition promised by theorem 4.4.1: a finite prefix $u_0 = a$, then infinitely many blocks from the same idempotent class V.

Two remarks will help when reading the proof below. First, the idempotence $V \cdot V \subseteq V$ is what lets us treat the infinite suffix as a genuine V^ω rather than a looser vectorial closure. Second, although the boundary states of the accepting run live in the finite set Q and therefore repeat, we never need to identify which state repeats: we only count, infinitely often, segments that visit F. This is the abstract, infinity-driven step that the proof exploits.

On the whiteboard, the professor visualized this Ramsey argument by defining when two positions k and k' "merge". They merge at some future position m if the segments $\alpha[k, m]$ and $\alpha[k', m]$ share the same congruence class. Because the index is finite, one can build an infinite sequence of positions where each segment merges into the same repeating class V, and merge adjacent marked segments to yield a block still in V.

4.5 Closure under Complement

We now assemble the pieces: the transition-profile congruence $\approx_A$ of section 4.3, the saturation lemmas of section 4.2, and the Ramsey decomposition of section 4.4. The crucial step is to show that $\approx_A$ actually saturates $L(A)$. We isolate that fact as a lemma so that the complementation theorem becomes a clean assembly.

Lemma 4.5.1 (The automaton congruence saturates $L(A)$)**.** *Let A be a Büchi automaton and let $\approx_A$ be its transition-profile congruence. For any two $\approx_A$-classes U, V, if $UV^\omega \cap L(A) \neq \emptyset$ then $UV^\omega \subseteq L(A)$.*

Proof. Pick an accepted word $\alpha \in UV^\omega \cap L(A)$ and decompose it as $\alpha = u_0u_1u_2\cdots$ with $u_0 \in U$ and $u_i \in V$ for $i \geq 1$. Because α is accepted, fix an accepting run and write the states it visits at the block boundaries as

$$q_0 = s_0 \xrightarrow{u_0} s_1 \xrightarrow{u_1} s_2 \xrightarrow{u_2} \cdots$$

Infinitely many of these block segments visit an accepting state; otherwise the whole run would visit F only finitely often and would not be Büchi-accepting.

Now take any other word $\beta = v_0v_1v_2\cdots \in UV^\omega$, with $v_0 \in U$ and $v_i \in V$ for $i \geq 1$. Because $u_0 \approx_A v_0$, the word v_0 can drive q_0 to the same boundary state s_1. Because each $u_i \approx_A v_i$, each v_i can drive s_i to s_{i+1}. Moreover, by the definition of $\approx_A$, whenever the original u_i-segment visits F, the replacement v_i-segment can be chosen to visit F as well. Stitching these replacement paths together produces a run of A on β that visits F infinitely often, hence is accepting.

We have shown that an arbitrary $\beta \in UV^\omega$ is accepted, so $UV^\omega \subseteq L(A)$. ◻

Example 4.5.2 (Saturation on the running automaton)**.** In the automaton of fig. 4.2, let U be the $\approx_A$-class of a and V the class of abc. The word $\alpha = a(abc)^\omega$ is accepted: the run visits $s_1 \in F$ during every abc block. By theorem 4.5.1, the entire product UV^ω is contained in $L(A)$. Now let V' be the class of c. No word in UV'^ω is accepted, because blocks from V' never visit s_1. Hence $UV'^\omega \subseteq \overline{L(A)}$: this class product belongs entirely to the complement.

Theorem 4.5.3 (Complementation of Büchi-recognizable languages)**.** *If $L \subseteq \Sigma^\omega$ is ω-regular, then $\overline{L}$ is ω-regular.*

Proof. Let A be a Büchi automaton with $L = L(A)$ and let $\approx_A$ be its transition-profile congruence.

Saturation. By theorem 4.5.1, each class product UV^ω either misses L or is contained in L. By theorem 4.2.4, the same all-or-nothing property holds for $\overline{L}$.

Coverage. By theorem 4.4.1 applied to $\approx_A$, every ω-word α can be written as $u_0u_1u_2\cdots$ with $u_0 \in U$ and $u_i \in V$ for $i \geq 1$, for some pair of $\approx_A$-classes (U, V) with $V \cdot V \subseteq V$. Hence every α belongs to at least one class product UV^ω.

Conclusion. Split the finitely many class products into those contained in L and those disjoint from L. The latter are exactly those contained in $\overline{L}$, and every word of $\overline{L}$ lies in one of them. Therefore

$$\overline{L} = \bigcup \{UV^\omega : UV^\omega \cap L = \emptyset\},$$

a finite union. Each class U and V is a regular finite-word language (recognized by a finite automaton built from the graph of A), so each UV^ω is ω-regular by theorem 3.4.1, and finite unions of ω-regular languages are ω-regular. □

The proof is non-constructive only in appearance. It actually describes an effective procedure for building a Büchi automaton that recognizes $\overline{L(A)}$, and that procedure is the algorithmic content behind the model-checking reductions at the end of the chapter.

■ **Formal details — The complement automaton by saturation**

Construction. Given $A = (Q, \Sigma, \Delta, q_0, F)$:

1. Enumerate the finitely many $\approx_A$-classes. Each class is a regular language: it is a Boolean combination of the languages $W_{p,q} = \{u : p \xrightarrow{u} q\}$ and $W^F_{p,q} = \{u : p \xrightarrow{u}_F q\}$, each recognized by the finite graph of A with initial state p and accepting state q (and, for $W^F_{p,q}$, an extra flag recording whether F has been seen).
2. For each ordered pair of classes (U, V) with $V \cdot V \subseteq V$, build a Büchi automaton $B_{U,V}$ for $U \cdot V^\omega$ using the closure constructions of theorem 3.4.1.
3. Retain exactly those $B_{U,V}$ for which $L(B_{U,V}) \cap L(A) = \emptyset$, decided by a reachable-accepting-cycle search on the asynchronous product (theorem 3.5.1).
4. Take the finite union of the retained $B_{U,V}$.

Invariant. After step 3, each retained $B_{U,V}$ recognizes a product $UV^\omega \subseteq \overline{L(A)}$, and each discarded $B_{U,V}$ recognizes a product $UV^\omega \subseteq L(A)$. This all-or-nothing split is exactly the content of theorem 4.5.1; the emptiness test in step 3 merely detects which side each pair falls on. The invariant holds initially (no pair has been classified) and is preserved every time step 3 classifies one more pair, because the classification is total: there is no borderline case.

Termination. There are at most 3^{n^2} classes by theorem 4.3.3, hence at most 3^{2n^2} pairs to test. Each emptiness test is a graph search on a product of polynomial size. The procedure therefore always halts.

Complexity. The number 3^{n^2} of profile classes dominates the blow-up: the complement automaton produced by this construction has $2^{\mathcal{O}(()n^2)}$ states. Sharper (Safra-style) constructions bring the bound down to $2^{\mathcal{O}(()n \log n)}$ states, which is asymptotically optimal. Either way, complementation is intrinsicly exponential in the number of states — the free finite-word lunch is genuinely over.

> **Proposition 4.5.4** (Ultimately periodic fingerprint). *Let $L_1, L_2 \subseteq \Sigma^\omega$ be ω-regular languages. If L_1 and L_2 contain the same ultimately periodic words, then $L_1 = L_2$.*

Proof. Suppose not. Then either $L_1 \setminus L_2$ or $L_2 \setminus L_1$ is non-empty. By closure under complement and intersection, that difference is ω-regular. Every non-empty ω-regular language contains an ultimately periodic word, so the difference contains an ultimately periodic word. This contradicts the assumption that L_1 and L_2 agree on all ultimately periodic words. □

4.6 Model Checking Consequence

The main verification payoff is the decidability of language inclusion.

Corollary 4.6.1 (Büchi inclusion). *Given Büchi automata A and B, it is decidable whether*

$$L(A) \subseteq L(B).$$

Proof. Construct an automaton for $\overline{L(B)}$, then an automaton for $L(A) \cap \overline{L(B)}$, and finally test Büchi emptiness by looking for a reachable accepting cycle. □

The same reduction schema decides the other classical problems: universality $L(A) = \Sigma^\omega$ reduces to $\overline{L(A)} = \emptyset$, and equivalence $L(A) = L(B)$ reduces to two inclusions. Each bottoms out in a reachable-accepting-cycle search, which runs in time linear in the product graph. The cost is hidden in complementation: as noted in the construction above, the whole pipeline is dominated by the $2^{\mathcal{O}()n \log n)}$ state blow-up of the best known complementation procedure. This is the price of moving from finite to infinite words.

This is the automata-theoretic backbone of model checking: search for an execution of the system that violates the specification. If none exists, inclusion holds.

Notation Recap

$\equiv, \approx_A$	congruence on Σ^*; automaton-induced congruence
$[u]_\equiv$	equivalence class of the finite word u
U, V	equivalence classes (sets of finite words)
UV^ω	class product: a word from U followed by ω-many from V
$p \xrightarrow{u} q$	there is a path from p to q labelled u
$p \xrightarrow{u}_F q$	such a path exists and visits an accepting state
3^{n^2}	upper bound on the number of $\approx_A$-classes ($n = \lvert Q \rvert$)
$\overline{L}$	complement $\Sigma^\omega \setminus L$

Summary & Key Takeaways

- Büchi complementation cannot be done by simply swapping final states; deterministic Büchi automata are strictly weaker than non-deterministic ones.
- A finite-index congruence summarizes each finite word by a label from a finite set, with concatenation acting through a finite table.
- Saturation makes each class product UV^ω all-or-nothing; the complement is then a finite union of class products.
- The transition-profile congruence $\approx_A$ is a concrete, computable, finite-index congruence with at most 3^{n^2} classes.
- A Ramsey decomposition writes every ω-word as a prefix followed by infinitely many congruent blocks, so saturation covers every word, not just the ultimately periodic ones.
- Assembling the pieces yields an effective — though $2^{\mathcal{O}()n^2)}$-state naive — automaton for the complement, and decidability of inclusion, universality, and equivalence.
- Ultimately periodic words are a fingerprint: two ω-regular languages agreeing on them are equal.

Exercises

Exercise 1 (Finite-index class automaton). Let $\equiv$ be equivalence modulo parity of word length. Build the DFA whose states are the two equivalence classes and whose transitions append one symbol. Which class contains ε?

Exercise 2 (Why profiles need accepting information). In the automaton of fig. 4.2, find two finite words u and v such that $s_2 \xrightarrow{u} s_2$ and $s_2 \xrightarrow{v} s_2$ (same endpoints), but one path visits F and the other does not. Write the transition profile of each word and show that if we dropped the accepting-path column, the two profiles would coincide. Construct an ω-word accepted via u-blocks and argue that replacing them with v-blocks breaks acceptance.

Exercise 3 (Inclusion reduction). Write the three automata operations needed to decide $L(A) \subseteq L(B)$ for Büchi automata. Which operation is the technically hardest one?

Exercise 4 (The complete profile congruence). Consider the Büchi automaton A of fig. 4.2. As discussed in lecture, the transition-profile congruence $\approx_A$ induces exactly nine equivalence classes over Σ^*.

1. Find all nine $\approx_A$-classes. For each class, provide a regular expression describing the words it contains (for example, one class contains $b, bab, babab \dots$ and can be written as $(ba)^*b$).
2. Verify that this is indeed a congruence by picking two classes, concatenating them, and showing that the result falls cleanly into another single class.
3. Express $L(A)$ exactly as a union of four class products $U_i(V_i)^\omega$, choosing U_i and V_i from your nine classes.

CHAPTER 5

Synthesis and Infinite Games

Chapters 1–4 worked in one direction: we were *given* a system, modelled its behaviours as finite or infinite words, and asked whether those behaviours satisfied a specification. The system was part of the input. Synthesis flips the arrow. We are *given* only the specification, and we must *construct* a finite-state system that satisfies it against every behaviour the environment may produce.

Verification asks whether a given system is correct. Synthesis asks for a correct system to be constructed automatically.

The flip has practical teeth. Later in this chapter we will meet the railway interlocking problem studied at FBK for the Italian railway network: starting from safety requirements written in temporal logic, we want a controller whose correctness is guaranteed by construction, not merely tested after the fact. Synthesis is the most ambitious form of this idea.

The right mathematical framework for the flip is *infinite games*. The environment and the system take turns forever; a play is an infinite sequence of moves; the specification decides who wins. This chapter presents the game-theoretic view behind Church's synthesis problem and the Büchi–Landweber solution pipeline: from a logical specification to an automaton, from an automaton to a game arena, and from a winning strategy to a synthesized transducer.

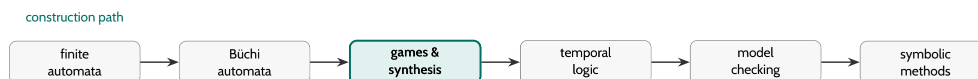


Figure 5.1: Where this chapter sits in the construction path of the book. Each step reuses the previous one as a representation or an algorithmic primitive.

> ■ **Running example — environment requests and system grants**
>
> The game chapters use the same request/acknowledgement intuition with a controller added. The environment raises requests; the system chooses grants; the winning condition says which infinite interactions are acceptable.

5.1 From Verification to Synthesis

Before defining synthesis in the abstract, let us start with a small problem that shows the flavour of the task.

Example 5.1.1 (A three-property specification)**.** An infinite binary input stream $\alpha = \alpha_0\alpha_1\cdots$ arrives one bit at a time. We must build a machine that produces an infinite binary output stream $\beta = \beta_0\beta_1\cdots$ satisfying three requirements simultaneously:

(i) *Echo ones:* for every t, if $\alpha_t = 1$ then $\beta_t = 1$.

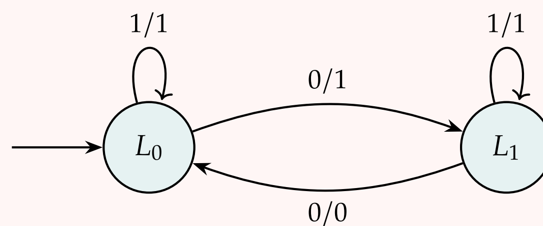
(ii) *No consecutive output zeros:* for every t, not both $\beta_t = 0$ and $\beta_{t+1} = 0$.

(iii) *Liveness:* if α contains infinitely many zeros, then β must also contain infinitely many zeros.

Condition (i) alone is trivially solved by the constant function $\beta_t = 1$ for all t: every input one is echoed, and no input constrains the output on zeros. This candidate also satisfies (ii), but it fails (iii): if the input has infinitely many zeros, the constant-one output has *no* zeros at all.

Can we do better? The identity function $\beta_t = \alpha_t$ satisfies (i) and (iii) but fails (ii) whenever two consecutive input zeros appear.

The trick is to alternate: upon seeing an input zero, output zero if the *previous* output on an input zero was one, and output one otherwise. This requires exactly one bit of memory—which zero-output was last. The result is a two-state Mealy transducer:



State L_0 means "the last output on an input zero was 0"; L_1 means "the last output on an input zero was 1". On input 1, both states output 1 and stay put. On input 0, the machine alternates between outputting 0 and 1, switching state each time. All three conditions are met for every input stream.

The example already exposes the two structural constraints that any synthesis solution must satisfy. First, the machine reacts *bit by bit*: when the n-th input arrives, the n-th output must be produced immediately, using only the input prefix seen so far. Second, the machine is *finite-state*: all relevant history is compressed into finitely many memory states (here, two). The rest of this section makes both constraints precise.

Definition 5.1.2 (Reactive synthesis)**. Reactive synthesis** asks whether there exists a finite-state controller that, for every infinite input sequence produced by the environment, generates an output sequence satisfying a given specification.

The important phrase is "for every input sequence". The controller is not allowed to choose a friendly environment. It must react correctly to all behaviours that the environment may produce.

Definition 5.1.3 (Mealy transducer)**.** A **Mealy transducer** is a finite-state machine whose output at each step depends on the current state and the current input symbol.

This captures causality. At time n, the system may use the input history up to time n, but not future inputs. A specification that can only be satisfied by looking into the future is not realizable by a reactive device.

Not every specification that *has* a solution as a relation between two infinite sequences can be realized by a causal, finite-state device. Three failure modes illustrate the constraints:

Example 5.1.4 (Satisfiability is not realizability)**.**

(a) *Future dependence.* The requirement "output exactly the next input bit" ($\beta_t = \alpha_{t+1}$) is satisfiable as a relation, but not realizable: the system would need to know tomorrow's input today. This violates the bit-to-bit constraint.

(b) *Unbounded memory.* The requirement $\beta_{2t} = \beta_{2t+1} = \alpha_t$ for all t forces the machine to remember α_t until time $2t$. As t grows, the gap between input and output grows without bound, so no finite memory suffices.

(c) *Infinite lookahead.* "Output all ones if the input contains infinitely many ones; otherwise output all zeros." No finite prefix of the input determines which case applies, so no causal device can commit to its very first output bit.

Definition 5.1.5 (Church's synthesis problem)**.** Given a logical specification $\Phi(X, Y)$, where X are input streams controlled by the environment and Y are output streams controlled by the system, decide whether there is a finite-state transducer M such that for every input stream x, the output stream $M(x)$ satisfies

$$\Phi(x, M(x)).$$

If such an M exists, construct it.

Read the quantifier carefully: it is $\forall x\, \Phi(x, M(x))$, not $\exists x\, \Phi(x, M(x))$. The controller cannot pick a friendly environment. This universal quantifier is what makes synthesis harder than satisfiability: a satisfying witness is a single pair (x, y), but a realizing transducer must defend against every x. In the rest of the chapter we will turn this universal quantifier into a game in which the environment is an adversary.

5.2 The Game View

The universal quantifier over environment inputs is naturally read as an adversary. Two players take turns forever: the environment picks an input bit, the system replies with an output bit, the environment picks again, and so on. After infinitely many rounds the joint input/output stream either satisfies Φ or violates it. If it satisfies it, the system wins; if it violates it, the environment wins. This is the game-theoretic reading of realizability.

Before defining the arena formally, consider the running example we will carry through the chapter.

Example 5.2.1 (Request-grant game, informally)**.** The environment may, at any round, raise a request r. The system may, at any round, emit a grant g. The specification is

every request is eventually granted.

Think of the environment as a client and the system as a server. The client is free to ask whenever it wants; the server must answer eventually, no matter how aggressive the client becomes. This is exactly the request/acknowledgement trace shape of chapter 3, but now viewed as a confrontation rather than as a single trace to be checked.

Definition 5.2.2 (Game arena)**.** A **game arena** is a finite directed graph

$$G = (V_E, V_S, E)$$

whose vertices are split between environment vertices V_E and system vertices V_S. At an environment vertex, the environment chooses the next move. At a system vertex, the system chooses the next move.

A play is an infinite path through the arena. The winning condition selects which infinite paths are winning for the system.

Definition 5.2.3 (Strategy)**.** A **strategy** for a player tells the player which move to take after each finite history ending in one of that player's vertices.

Strategies may use memory. A **positional** strategy depends only on the current vertex. A **finite-state** strategy depends on the current vertex plus a finite memory state.

A strategy for the system is **winning** from a vertex v if every infinite play that starts in v and is consistent with that strategy satisfies the winning condition, no matter how the environment resolves its own choices. This universal quantification is the game version of realizability: a controller is correct only if it survives all legal environment behaviours.

On the whiteboard (and in Wolfgang Thomas's standard tutorial, which the lectures follow), a positional strategy for the system is drawn by making exactly one outgoing edge from each system vertex ***bold****. Because the environment is adversarial,* all *outgoing edges from an environment vertex are treated as possible moves.*

Definition 5.2.4 (Winning region)**.** The **winning region** of a player is the set of vertices from which that player has a winning strategy.

For the finite games used here, the winning regions of the two players partition the arena. This determinacy fact is what lets synthesis return a sharp answer: either the system has an implementation, or the environment has a counter-strategy.

Example 5.2.5 (Request-grant game, formalized)**.** Return to theorem 5.2.1. Concretely, the arena vertices are pairs (*state*, *last input*): the environment vertex records whether r was just raised, and the system vertex records the input plus whether the system has yet to grant. The specification

every request is eventually granted

is not a finite-prefix property. A finite play cannot prove that a request *will never* be granted; only the infinite continuation can. The game must therefore be evaluated over infinite plays, which is exactly why we need the machinery of this chapter rather than the finite-word automata of chapter 1.

Definition 5.2.6 (Finite-state strategy)**.** A **finite-state strategy** is implemented by a finite memory machine. It updates its memory after each move and chooses the next system move from the current arena vertex and current memory value.

The strategy machine is not the arena itself. The arena describes all possible interactions between system and environment. A strategy selects one outgoing system edge at each system-controlled situation, while environment edges remain possible because the controller does not control them.

5.3 Automata-Theoretic Synthesis Pipeline

The last section gave us arenas, strategies, and winning regions in the abstract. But where does the arena come from? In synthesis the arena is not handed to us: we have to build it from the specification. The Büchi–Landweber solution is the pipeline that turns a logical specification into a solved game, and ultimately into a transducer.

■ **Formal details — Pipeline**

Step 1. Write the specification in a logic over infinite words, such as S1S or a temporal logic fragment.

Step 2. Translate the specification into an equivalent deterministic automaton with an acceptance condition suitable for games, such as a Müller or parity condition.

Step 3. Split each round into an environment move followed by a system move, obtaining a finite game arena.

Step 4. Solve the infinite game.

Step 5. If the system player has a winning strategy, extract a finite transducer from that strategy.

Step 2 reuses the automata constructions of chapters 3 and 4: by Büchi's theorem every S1S-definable specification has an equivalent deterministic Müller automaton. Step 3, the focus of this section, is the bridge from automata to games. Steps 4 and 5 occupy the rest of the chapter.

The pipeline differs from model checking in one crucial way. Model checking takes a system automaton as input. Synthesis constructs the system from a winning strategy.

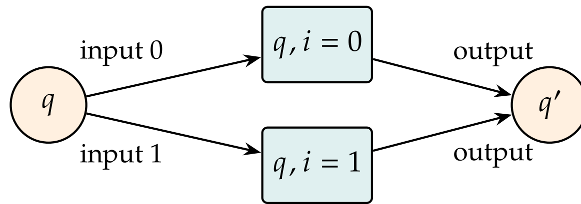
Definition 5.3.1 (Automaton-to-game construction)**.** Let $\mathcal{A} = (Q, \Sigma \times \Gamma, q_0, \delta, \mathcal{F})$ be a deterministic Müller automaton over input/output pairs $(i, o) \in \Sigma \times \Gamma$. The **game arena induced by** $\mathcal{A}$ has vertex set $Q \cup (Q \times \Sigma)$,

with Q owned by the environment and $Q \times \Sigma$ owned by the system, and edges

$$q \xrightarrow{i} (q, i) \xrightarrow{o} q' \quad \text{whenever} \quad \delta(q, (i, o)) = q'.$$

The Müller winning condition of $\mathcal{A}$ becomes the system's winning condition, applied to the projection of a play onto its Q-vertices.

The construction unwraps each transition of $\mathcal{A}$ into two moves: the environment first reveals the input bit i, taking us to (q, i), and then the system picks the output bit o, taking us to the successor state $q' = \delta(q, (i, o))$. fig. 5.2 shows one such unwrapped transition. Note that several output choices may lead to the same q': in that case only the bit label differs, and the system is free to pick any of them.



One automaton transition labelled by an input/output pair becomes an environment choice followed by a system choice.

Figure 5.2: Splitting automaton labels into game moves.

The induced arena is not yet the synthesized controller. The controller is extracted only after solving the game: at each system vertex (q, i), the winning strategy tells us which output move to choose, and that choice is precisely the bit produced by the transducer.

Definition 5.3.2 (Müller condition)**.** A **Müller condition** specifies a family $\mathcal{F} \subseteq 2^Q$ of accepting sets. A run is accepting if

$$\inf(\rho) \in \mathcal{F}.$$

Büchi acceptance asks whether $\inf(\rho)$ intersects a fixed set F. Müller acceptance can inspect the whole set of states seen infinitely often. This additional expressiveness is useful when moving from non-deterministic automata to deterministic automata for synthesis.

Definition 5.3.3 (Weak Müller condition)**.** A **weak Müller condition** evaluates a play using the set $\mathrm{Occ}(\rho)$ of all states that occur at least once, rather than the set $\inf(\rho)$ of states that occur infinitely often.

The difference is subtle but important. A weak Müller objective can ask whether a state ever appears during the play. A full Müller objective asks what remains visible in the limit after all finite prefixes are forgotten.

Example 5.3.4 (Occurrence vs. infinity). Consider the play

$$p\, q\, r\, r\, r\, r \cdots .$$

Then $\mathrm{Occ}(\rho) = \{p, q, r\}$, while $\inf(\rho) = \{r\}$. A weak condition can remember that p and q appeared once. A Müller condition ignores them because they do not appear infinitely often.

■ **Formal details — Appearance record: from weak Müller to weak parity**

We can reduce a weak Müller game to a weak parity game by building a strategy automaton that records which arena states have appeared so far. This is the *appearance record* construction, and it serves as a warmup for the more involved latest appearance record used for full Müller games.

The appearance record automaton. Build a Mealy automaton whose state set is 2^V (the power set of arena vertices). The initial state is $\emptyset$. On reading a vertex v from the play, the automaton updates $R := R \cup \{v\}$. Because only unions are taken, R grows monotonically and stabilizes after at most $|V|$ steps. The final value of R is exactly $\mathrm{Occ}(\rho)$.

Coloring. The weak Müller winning condition asks whether $\mathrm{Occ}(\rho) \in \mathcal{F}$. We encode this as a weak parity condition by assigning each appearance record R a color:

$$color(R) = \begin{cases} 2|R| & \text{if } R \in \mathcal{F}, \\ 2|R| - 1 & \text{otherwise.} \end{cases}$$

Since R only grows, the sequence of colors along a play is non-decreasing. The play eventually stabilizes on a final color $c = color(\mathrm{Occ}(\rho))$. The system wins the weak parity game if and only if the maximum color seen (equivalently, the final color) is even, which happens precisely when $\mathrm{Occ}(\rho) \in \mathcal{F}$.

Memory cost. The expanded arena has $|V| \cdot 2^{|V|}$ vertices (original vertex plus appearance record), so the finite-state strategy uses at most $2^{|V|}$ memory states—the bound stated in theorem 5.5.6 for the weak case. A positional winning strategy on the expanded arena projects to a finite-state strategy on the original arena via the game simulation of theorem 5.5.7.

5.4 Reachability Games and Attractors

Step 4 of the synthesis pipeline is "solve the infinite game". Before tackling Müller or parity objectives, let's solve the simplest non-trivial case: reachability. The construction we develop here—the *attractor*—will reappear inside the parity-game solver of section 5.5, so it earns its own section.

Definition 5.4.1 (Reachability game). In a reachability game, the system wins a play if the play eventually visits a target set $T \subseteq V$.

The system does not care what happens after the first visit to T; it only needs to force the play there. The request-grant specification of theorem 5.2.5 is *not* a reachability property (grants must keep happening forever), but its bounded variant "grant within N steps" is, and that bounded variant is exactly what the attractor construction below reasons about, one bound at a time.

The winning region is computed by a least-fixed-point recursion called the attractor.

Definition 5.4.2 (Attractor)**.** The system attractor to T is the least set $\mathsf{Attr}(T)$ such that:

- $T \subseteq \mathsf{Attr}(T)$;
- if $v \in V_S$ has some successor in $\mathsf{Attr}(T)$, then $v \in \mathsf{Attr}(T)$;
- if $v \in V_E$ has all successors in $\mathsf{Attr}(T)$, then $v \in \mathsf{Attr}(T)$.

The two predecessor clauses reflect control. At a system vertex, one good successor is enough because the system chooses. At an environment vertex, all successors must be good because the environment chooses and the system must survive every alternative. Because V is finite, the least fixed point is reached after at most $|V|$ predecessor sweeps, and each sweep adds at least one vertex or terminates.

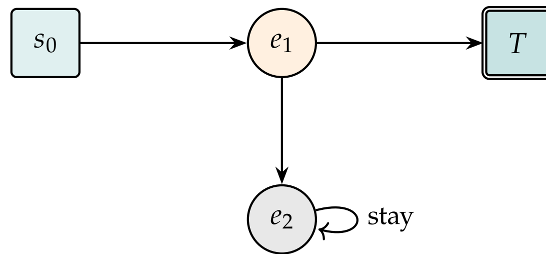


Figure 5.3: At environment vertices, every possible successor must lie in the attractor for the vertex itself to qualify. Here e_1 fails the test because the environment can escape to the gray sink e_2.

fig. 5.3 shows why environment vertices are the hard case: even if one successor leads to T, the environment will simply choose the other successor and escape. That is why we demand *all* successors to be in the attractor.

Theorem 5.4.3 (Attractor correctness)**.** *Let $G = (V_E, V_S, E)$ be a finite game arena with target $T \subseteq V$. A vertex v belongs to $\mathsf{Attr}(T)$ if and only if the system has a winning strategy for the reachability objective from v.*

Proof. The two directions are proved separately.

Winning strategy from the attractor. Define the *rank* $r(v)$ of a vertex $v \in \mathsf{Attr}(T)$ as the least k such that v enters $\mathsf{Attr}_k(T)$, where $\mathsf{Attr}_0(T) = T$ and

$$\mathsf{Attr}_{k+1}(T) \;=\; \mathsf{Attr}_k(T) \;\cup\; Pre_S(\mathsf{Attr}_k(T)) \;\cup\; Pre_E(\mathsf{Attr}_k(T)),$$

with Pre_S (resp. Pre_E) the existential (resp. universal) predecessor operator from theorem 5.4.2. We exhibit a strategy that strictly decreases the rank at every system move and never increases it at an environment move.

Induction on $r(v)$. If $r(v) = 0$ then $v \in T$ and the system has already won. If $r(v) = k+1 > 0$ and $v \in V_S$, the existential clause gives a successor $w \in \mathsf{Attr}_k(T)$; the strategy picks w, and by induction the system wins from w. If $v \in V_E$, the universal clause forces *every* successor w to satisfy $w \in \mathsf{Attr}_k(T)$; whichever move the environment picks, the induction hypothesis applies at w. In both cases the rank strictly decreases at each step, so after at most $r(v)$ steps the play reaches T.

No winning strategy outside the attractor. Let $U = V \setminus \mathsf{Attr}(T)$. By definition of the least fixed point, U is *closed under the dual predecessor*: every $v \in U \cap V_E$ has at least one successor in U, and every $v \in U \cap V_S$ has *all* successors in U. (Otherwise v would have entered the attractor on the next sweep.) The environment exploits this by always moving into U when it is its turn, and it is free to do so at every environment vertex. From a system vertex in U, every system move stays in U. So whatever the system does, the play remains in U forever and never visits T. Hence no system strategy wins from any $v \in U$. □

The proof shows more than the statement: it gives a *positional* winning strategy for the system inside the attractor (always move to a successor of strictly smaller rank), and a *positional* counter-strategy for the environment outside it (always move to a successor that stays in U). Reachability games are therefore *positionally determined*.

5.5 Müller and Parity Games

Reachability asks only "will we get there?". Most synthesis specifications are sharper: they care about what happens *forever*, not just about the first visit. "Every request is eventually granted" must hold infinitely often; "the system must not stay in error mode forever" forbids an infinite suffix. These are Müller-style objectives, and they need richer winning conditions than reachability.

> **Definition 5.5.1** (Parity condition)**.** In a parity game, each vertex has a priority in $\mathbb{N}$. The winner is determined by the highest priority seen infinitely often: one player wins when that priority is even, the other when it is odd.

Parity games are central to synthesis for two reasons. First, every Müller objective can be encoded as a parity objective over a finite memory expansion of the arena, which we now construct via latest appearance records. Second, parity games admit *positional* winning strategies, so the resulting controller is finite-state.

> **Definition 5.5.2** (Latest appearance record)**.** The **latest appearance record** (LAR) is a finite memory structure that keeps the arena states seen so far ordered by recency. When a state is visited again, it is moved to the front of the list. The **hit position** is the position in the old list where the revisited state was found, or 0 if it was not in the list yet. The **hit set** is the prefix of the old list ending at that hit position (empty when the hit position is 0).

The hit set is the key invention. After the play stabilises (every state that ever appears again has appeared at least once), the hit set is exactly the set of states seen since the previous visit of the current state—a finite-window proxy for the set of states visited infinitely often.

fig. 5.4 shows a single update step: visiting b when the LAR is $[a, b, c]$ moves b to the front, leaving the new list $[b, a, c]$. The hit position was 2 (counting from 1), so the hit set is $\{a, b\}$—the part of the recency list visited since the previous occurrence of b, plus b itself.

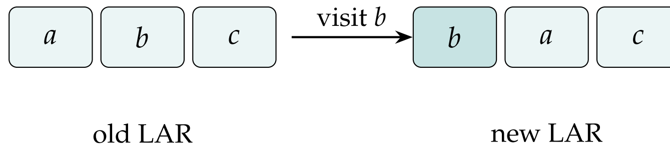


Figure 5.4: Latest appearance record: when b is visited, it moves to the front of the recency list. With a most-recent-first convention, the hit set is the prefix of the old list up to b, here $\{a, b\}$.

Example 5.5.3 (Walking through an LAR)**.** Take an arena on four states $\{a, b, c, d\}$. Let's track the LAR, the hit position, and the hit set for the sequence of visits a, c, c, d, b, c, d:

visit	LAR after	hit pos.	hit set
a	$[a]$	0	$\emptyset$
c	$[c, a]$	0	$\emptyset$
c	$[c, a]$	1	$\{c\}$
d	$[d, c, a]$	0	$\emptyset$
b	$[b, d, c, a]$	0	$\emptyset$
c	$[c, b, d, a]$	3	$\{b, d, c\}$
d	$[d, c, b, a]$	3	$\{c, b, d\}$

The first visit to each state has hit position 0 and empty hit set: the state was not yet in the list. When c is visited the second time, it is already at the front, so the hit position is 1 and the hit set is $\{c\}$. When c is visited the third time (after d and b), it is in position 3 of the old LAR $[b, d, c, a]$. It moves to the front, and the hit set is $\{b, d, c\}$. The short prefixes above do not yet reveal the limit behaviour; they only show the update rule. Once the play has stabilised, recurring visits to states inside the eventual loop produce hit sets whose largest recurring records identify exactly the set of states appearing infinitely often.

The LAR turns a Müller condition into a parity condition. Given a Müller family $\mathcal{F} \subseteq 2^Q$, colour each LAR by

$$color(\text{LAR}) = \begin{cases} 2 \cdot |\text{hit set}| & \text{if the hit set is in } \mathcal{F}, \\ 2 \cdot |\text{hit set}| - 1 & \text{otherwise.} \end{cases}$$

With n arena states, the LAR memory has at most $n! \cdot n$ records: an ordering of states, plus the hit position. The construction is an *exponential blow-up*, but it stays finite.

Example 5.5.4 (Request-grant as a Müller game)**.** Return to the request-grant arena of theorem 5.2.5. Suppose the arena has three relevant states: *idle* (no pending request), *pending* (request raised, not yet granted), and *granted* (grant just issued). The Müller family for "every request is eventually granted" contains exactly the sets that include *granted*:

$$\mathcal{F} = \{S \subseteq Q : granted \in S\}.$$

As the play loops through these states, the LAR tracks recency. If the system keeps granting, *granted* periodically returns to the front of the list, producing a hit set that contains *granted* and therefore belongs to $\mathcal{F}$—an even color. If the environment keeps requesting but the system never grants, *granted* sinks to the tail of the LAR and vanishes from the recurring hit sets, producing an odd color. The parity condition thus captures exactly the liveness requirement.

Theorem 5.5.5 (Büchi–Landweber; positional determinacy of parity games)**.** *Every parity game on a finite arena is determined, and the winner has a* positional *winning strategy from every vertex in their winning region.*

Formal details — Proof roadmap (Zielonka's recursive algorithm)

The proof, due to Zielonka (1998), is by induction on the number of distinct priorities. Let c be the greatest priority, on a vertex q.

If c is even, the system tries to attract the play to q infinitely often. Let $A = \mathsf{Attr}_S(\{q\})$ be the system attractor of q. Inside A the system uses the attractor strategy of theorem 5.4.3 to force visits to q, whose even priority wins. Outside A, the environment is trapped in the subgame $G \setminus A$, which has strictly fewer priorities; the induction hypothesis gives the answer there. Combining the two regions gives the system's winning region.

If c is odd, the same attractor argument is run with the roles swapped, this time for the environment. In both cases the winning strategy is positional: the attractor strategy is positional, the induction hypothesis is positional, and the two combine without extra memory.

The full argument carefully treats the case where one player cannot force the play back into the top-priority vertex; we then recurse on the residual subarena where the top priority is absent.

Corollary 5.5.6 (Müller games are finite-state determined)**.** *Every Müller game on a finite arena is determined, and the winner has a* finite-state *winning strategy from every vertex in their winning region. The strategy uses at most $n! \cdot n$ memory states.*

The decision problem "does the system win from vertex v?" lies in NP ∩ coNP, *but whether it can be solved in polynomial time remains a long-standing open question. The problem is equivalent to model checking the modal μ-calculus, so a polynomial algorithm for parity games would settle that question too.*

Proof. By the LAR construction above, the Müller game G is simulated by a parity game G' on the expanded state space $V \times \mathrm{LAR}$. theorem 5.5.5 gives a positional winning strategy in G'. By the simulation correspondence of theorem 5.5.7 below, that positional strategy is a finite-state strategy in G, with the LAR as memory. The number of memory values is bounded by $n! \cdot n$. □

Determinacy is what makes synthesis a decision problem. If the system does not have a winning strategy from the initial vertex q_0, then by determinacy the environment has a counter-strategy, and the specification is not realizable. The pipeline of section 5.3 therefore returns a sharp yes/no answer, plus a witnessing transducer when the answer is yes.

For the request-grant game of theorem 5.5.4, the system does have a winning strategy from the initial vertex: always grant within finitely many steps. After the LAR-to-parity reduction and Zielonka's algorithm, that strategy is positional in the expanded arena and projects back to a finite-state controller—a Mealy transducer that solves the original synthesis problem, just as the two-state machine solved the three-property example that opened this chapter.

At this point we are using one automata-theoretic result as a black box. The logic-to-automaton translation for S1S gives an automaton over infinite words, and McNaughton's theorem lets us take it in deterministic Müller form. The details belong to Part II (chapters 8 and 9); here we only need the interface: a finite deterministic monitor whose set of states visited infinitely often decides whether the play satisfies the specification.

Definition 5.5.7 (Game simulation)**.** A game $G = (V_E, V_S, E, W)$ is **simulated** by a game $G' = (V'_E, V'_S, E', W')$ via a finite-state transducer $S = (S, s_0, \delta)$ when:
1. $V'_E \subseteq S \times V_E$ and $V'_S \subseteq S \times V_S$;
2. $(r, p) \longrightarrow (s, q)$ is an edge of G' iff $p \longrightarrow q$ is an edge of G and $s = \delta(r, q)$;
3. the projection $(r_0, p_0)(r_1, p_1) \cdots \mapsto p_0 p_1 \cdots$ sends winning plays for G' bijectively onto winning plays for G.

Game simulation is the formal analogue of "solving one game by reducing it to another". A positional strategy in G' projects to a finite-state strategy in G: the controller maintains $r \in S$ internally and consults the positional choice prescribed for (r, p). This is exactly the mechanism by which the LAR parity game delivers a finite-state strategy for the original Müller game.

■ **Formal details — Closing the synthesis loop**

Combining the pieces of this chapter end-to-end:
1. the S1S specification $\Phi(X, Y)$ becomes a deterministic Müller automaton (Büchi plus McNaughton; see chapters 8 and 9);
2. the automaton becomes an arena (theorem 5.3.1) with the Müller objective as winning condition;
3. the arena game is solved by reduction to parity (theorem 5.5.6);
4. the resulting positional strategy in the expanded arena is a finite-state strategy in the original arena;
5. the system component of that strategy is a Mealy transducer realizing Φ, i.e. a solution to theorem 5.1.5.

If the initial vertex q_0 lies outside the system's winning region, the specification is not realizable, and the environment's winning strategy is a counterexample showing why.

5.6 Formal Verification in Practice

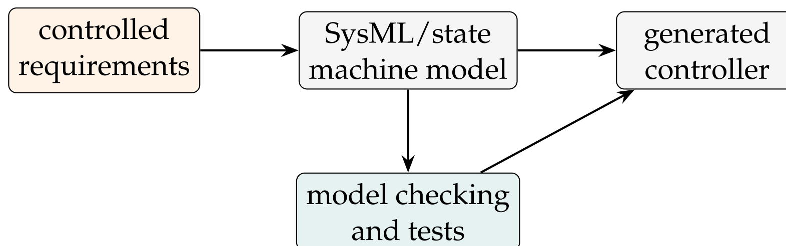
The same automata and game ideas appear in industrial verification. A typical safety-critical example is railway interlocking: the controller must prevent incompatible train movements and unsafe signal settings.

■ **Intermezzo — Railway Interlocking**

Legacy railway systems were often implemented with relay circuits. When such systems are migrated to software, the engineering challenge is not only to generate code, but to prove that the generated controller preserves the safety constraints of the original infrastructure.

Model-based design, software model checking, deductive verification, and reverse engineering of relay logic all fit the same pattern: extract a finite model, state the property precisely, and use an automated method to rule out bad behaviours.

In an interlocking system, a *route* reserves a portion of track for a train. Two routes may be incompatible because they share a track segment, cross at a junction, or require conflicting switch positions. The core safety requirement is mutual exclusion: incompatible routes must never be cleared at the same time. Other requirements connect signals, points, level crossings, and train-detection circuits.



Traceability links each generated artifact back to an explicit safety requirement.

Figure 5.5: A typical model-based verification flow for interlocking software.

The verification difficulty is scale. A station layout with many track circuits and switches induces a large Boolean transition system. Tools therefore use symbolic representations, SAT/SMT solving, abstraction, and model-based testing to search for unsafe states without enumerating every concrete configuration by hand.

Synthesis pushes this idea further. Instead of verifying code after it has been written, we try to generate a correct-by-construction finite-state controller from the specification.

Notation Recap

Symbol	Meaning
$\Phi(X, Y)$	S1S specification with input streams X, output streams Y
M	Mealy transducer (finite-state machine with output)
$G = (V_E, V_S, E)$	Game arena: environment vertices, system vertices, edges
V_E, V_S	Vertices owned by the environment / the system
T	Target set in a reachability game
$\mathsf{Attr}(T)$	System attractor to T (least fixed point)
$\mathcal{F} \subseteq 2^Q$	Müller acceptance family
$\inf(\rho)$	States occurring infinitely often in play ρ
$\mathrm{Occ}(\rho)$	States occurring at least once in play ρ
LAR	Latest appearance record (recency-ordered state list)
hit position	Position in the old LAR where a revisited state is found
hit set	Prefix of the old LAR up to the hit position
W_S, W_E	Winning regions for the system / the environment

Summary & Key Takeaways

- Synthesis constructs a finite-state system from a specification.
- Causality separates realizability from ordinary satisfiability.
- Infinite games model the interaction between environment and system.
- Attractor computation explains the basic predecessor reasoning.
- Müller and parity games handle conditions about infinite behaviour.

Exercises

Exercise 1 (Causality). Classify the following specifications as causal or non-causal for a Mealy transducer: (i) output the current input bit; (ii) output the previous input bit; (iii) output the next input bit.

Exercise 2 (Attractor iteration). In fig. 5.3, compute the first attractor iteration starting from T. Why is e_1 not immediately added?

Exercise 3 (Strategy memory). Give an example of a property where reacting only to the current input symbol is insufficient, but one bit of memory is enough.

Exercise 4 (LAR update). Starting from the empty LAR on state set $\{a, b, c\}$, trace the latest appearance record, hit position, and hit set for the visit sequence b, c, a, b, c, b. What is the hit set at the last step, and which states would it identify as recurring if the pattern continued?

Exercise 5 (Solving a small parity game). Consider a four-vertex arena with $V_S = \{s_1, s_2\}$, $V_E = \{e_1, e_2\}$, priorities $s_1 \mapsto 2$, $e_1 \mapsto 1$, $s_2 \mapsto 2$, $e_2 \mapsto 3$, and edges $s_1 \to e_1$, $s_1 \to e_2$, $e_1 \to s_2$, $e_2 \to s_1$, $s_2 \to e_1$. Apply the attractor-based induction from the proof roadmap of theorem 5.5.5 to determine the winning region for each player and give a positional winning strategy for the system.

CHAPTER 6

Planning as Model Checking

Part I has built a toolbox: finite automata to recognise behaviours, omega-automata for infinite traces, congruences to compare languages, and games with attractors to synthesise controllers. This closing chapter changes lens. We step outside verification for a moment and look at *automated planning*, the classical AI problem of choosing actions that drive a world from an initial situation to a goal. The surprise, and the reason this chapter belongs here, is that planning *is* the same problem in disguise. A planning domain is a finite transition system, a goal is a target set of states, and a plan under uncertainty is exactly a strategy in a game against nature. The preimage operator that defined the system attractor in chapter 5 will reappear verbatim as the one-step engine of backward planning.

Planning can be read as a controller-synthesis problem: find a strategy that drives the system to a goal despite uncertainty.

A planning problem describes a world, a set of actions, and a goal. The task is to construct a plan that reaches the goal. When actions are non-deterministic or the environment can interfere, a plan is no longer just a finite sequence of actions; it is a strategy. We will classify the strength of such strategies — weak, strong, and strong-cyclic — and show that each corresponds to a familiar verification flavour: existential reachability, universal reachability, and reachability under fairness. This recasting of planning as model checking is what gives the chapter its name and what ties Part I together before we turn, in Part II, to the temporal logics and symbolic tooling that make these computations scale. Model checking is not the only view: as the professor pointed out, modern planning also translates into satisfiability checking (SAT), controller synthesis (building the state-machine that implements the plan), and execution monitoring (detecting when the environment deviates from assumptions and dynamically re-planning).

Where we are. The previous chapters built automata, language operations, complementation, and games. The key algorithmic pattern was already visible in reachability games: compute the states from which a player can force progress.

What this chapter adds. Planning recasts that pattern as an AI problem. Actions change a finite world, uncertainty plays the role of an adversary, and a plan is a strategy for reaching a goal.

Where it leads. The same preimage and fixpoint ideas will return in Part II as symbolic model checking, CTL fixed points, and tool-supported verification.

Chapter map.

- Section 6.1 compares action-based and timeline-based modelling.
- Section 6.2 formalises fluents, states, and actions.

- Section 6.3 explains plans as strategies.
- Sections 6.4 and 6.5 develop the backward operators and algorithms.
- Sections 6.6 and 6.7 show how compact symbolic descriptions replace explicit graphs.
- Section 6.8 connects planning back to model checking and closes Part I.

6.1 Action-Based and Timeline-Based Planning

Before formalising anything, we have to choose how to describe the world that is being controlled. Two families of models have emerged from decades of practice, and they emphasise different things. One looks at the world through the actions an agent performs; the other looks at it through the values that state variables take over time. The choice is not cosmetic: it shapes which kinds of constraint feel natural to write and which algorithms fit.

> **Definition 6.1.1** (Action-based planning)**. Action-based planning** models change as a sequence of discrete actions. A plan is built from operations such as `load`, `move`, or `cross`, each with preconditions and effects.

This is the graph view: states are nodes, actions are labelled transitions, and a plan is a path or policy leading to a goal. It is the view that connects directly to the automata and games of the previous chapters.

> **Definition 6.1.2** (Timeline-based planning)**. Timeline-based planning** models the evolution of state variables over time intervals. Instead of focusing on individual actions, it records when each component has a given value and which temporal constraints synchronize different components.

Timeline planning is natural for scheduling and control domains. For a spacecraft, for instance, an instrument may be *off*, then *warming up*, then *collecting data*; another component, such as a battery or data buffer, evolves on its own timeline but must remain compatible with the instrument timeline. This style was used for real NASA missions and is closer in spirit to control theory than to graph search.

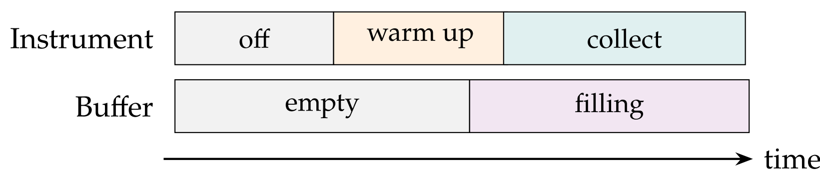


Figure 6.1: Timeline-based planning follows component values over intervals and synchronizes them by constraints.

The rest of this chapter focuses on the action-based view because it connects directly to transition systems and model checking. The timeline view is important in applications, but it requires richer temporal constraints.

6.2 Planning Domains

We now commit to the action-based view and build the formal model. Just as a finite automaton is a finite set of states with a labelled transition relation, an

action-based planning domain is a finite set of states equipped with actions that take a state to one or more successors. The two ingredients we add beyond pure automata are an *initial* set (where the world might start) and a *goal* set (where we want it to end up).

> **Definition 6.2.1** (Fluent and state). A **fluent** is a Boolean variable describing a property of the world. A **state** is a valuation of all fluents.

If the fluents are *atA*, *loaded*, and *green*, then a state records whether each of these facts is currently true. With n Boolean fluents, the explicit state space may contain up to 2^n states — our first hint that symbolic methods will eventually be needed.

> **Definition 6.2.2** (Inertial and non-inertial fluents). An **inertial fluent** keeps its value unless some action or event changes it. A **non-inertial fluent** is recomputed or controlled by the environment at each step.

The split between inertial and non-inertial fluents is the planning analogue of the system/environment split in chapter 5. The planner controls inertial fluents through its actions; non-inertial fluents are decided by nature. For the parcel domain below, the robot's location is inertial, while the colour of the traffic light is non-inertial from the robot's point of view.

> **Definition 6.2.3** (Action-based planning domain). An action-based planning domain consists of:
> - a finite set of states S;
> - a set of initial states $I \subseteq S$;
> - a set of goal states $G \subseteq S$;
> - a finite set of actions A;
> - a transition relation $T \subseteq S \times A \times S$.
>
> An action a is **enabled** at s when there exists at least one s' with $(s, a, s') \in T$. The set of possible successors of (s, a) is $\mathrm{Post}(s, a) = \{s' \in S : (s, a, s') \in T\}$.

The transition relation may be non-deterministic: the same action in the same state can have several possible successors. This is where planning begins to look like a game against the environment. The domain by itself fixes only the rules; the planner must still pick a move at each turn, and nature resolves the residual non-determinism.

■ Formal details — Modelling dimensions

Planning domains are often classified along three independent axes.

- **Deterministic vs. non-deterministic effects**: an action has one successor, or several possible successors.
- **Complete vs. partial observability**: the planner sees the exact current state, or only an observation compatible with several states.
- **Reachability vs. extended goals**: the goal is simply to reach G, or to satisfy temporal requirements along the way.

> *In the lecture, the professor illustrated this with a driving analogy: the destination is the goal state, but during the trip you must also respect a velocity threshold — a constraint that must hold* along the way, *not just at arrival.*

Throughout this chapter we work in the fully observable, reachability-goal case, where the differences between weak, strong, and strong-cyclic plans are already visible. richer combinations are tackled by the same symbolic machinery, just with more expressive logics — exactly the logics we will meet in Part II.

Example 6.2.4 (Parcel delivery with a traffic light)**.** We adopt the parcel-delivery domain used throughout the book. A robot must move a parcel from the train *station* to the *airport*. Between them sits a traffic light, which may be *green* or *red*. The fluent $pos \in \{station, light, airport\}$ records where the robot is; the fluent $color \in \{green, red\}$ records the light. The actions are `transport` (advance one step along the route) and `wait` (do nothing for one tick). The position is inertial; the colour is non-inertial. Crucially, `transport` is allowed at the light only when the colour is green, and a `transport` from the station may land at the light with either colour. fig. 6.2 shows the resulting six states.

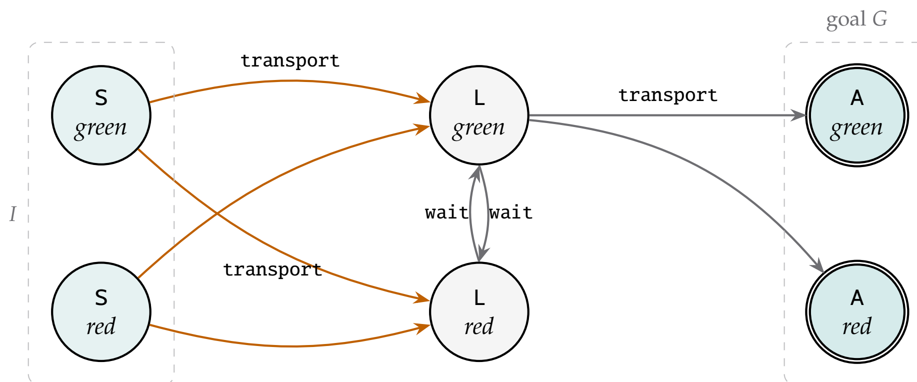


Figure 6.2: The parcel-delivery domain. Initial states $\mathsf{S}_{green}, \mathsf{S}_{red}$ are shown in teal, goal states $\mathsf{A}_{green}, \mathsf{A}_{red}$ are double-circled in teal. The orange `transport` edges from the station are non-deterministic: the environment chooses the colour on arrival. The grey `wait` loop between L_{red} and L_{green} is what will force us to consider strong-cyclic plans.

This little domain already contains every complication we need. The two initial states reflect uncertainty about the colour at the start. The transition from the station is non-deterministic, because the environment picks the colour on arrival. And the wait-loop between L_{red} and L_{green} means a naive "try to cross, then wait, then try again" policy can in principle loop forever. Each plan type in section 6.3 will be defined so that it copes with one of these complications.

6.3 Plans as Execution Structures

A domain tells us what *could* happen; a plan tells us what the planner *chooses* to do. The two together generate an execution structure, a subgraph of the domain in which only the chosen actions survive but every environmental outcome of each chosen action is kept.

Definition 6.3.1 (Plan as a state-action table)**.** A **plan** (or **state-action table**) for a domain is a relation $\pi \subseteq S \times A$ with $(s, a) \in \pi$ only when a is enabled at s. The plan is **deterministic** if for every s there is at most one a with $(s, a) \in \pi$; it is **universal** if for every $s \in S$ there is at least one such a.

In a fully deterministic domain, a plan collapses to the familiar object: a finite sequence $a_1, a_2, \ldots, a_k$ of actions, since at each step there is only one enabled action and only one possible successor. In a non-deterministic domain this is no longer enough. The fixed sequence may fail after an unlucky outcome, so the plan must be allowed to branch and react — exactly like a strategy in a game.

Definition 6.3.2 (Execution structure)**.** Let π be a plan and $I \subseteq S$ a set of initial states. The **execution structure** induced by π from I is the directed graph $K_\pi = (Q, T_\pi)$ where $Q \subseteq S$ and $T_\pi \subseteq Q \times Q$ are the minimal sets satisfying:

- $I \subseteq Q$;
- if $s \in Q$ and $(s, a) \in \pi$ and $(s, a, s') \in T$, then $s' \in Q$ and $(s, s') \in T_\pi$.

A node $s \in Q$ is **terminal** if it has no outgoing edge in T_π. An **execution path** from $s_0 \in I$ is a finite or infinite sequence $s_0, s_1, s_2, \ldots$ with $(s_i, s_{i+1}) \in T_\pi$ for every i, ending in a terminal state if it is finite. The structure is **acyclic** if every execution path is finite.

The definition makes a subtle but crucial move. The plan π lives in the world of state-action pairs; the execution structure lives in the world of state-state pairs. Closing π under all environmental outcomes produces a graph that contains every future the planner cannot rule out. Terminal states are where this graph stops: either because the planner reached its target, or because it reached a dead end with no enabled action.

Definition 6.3.3 (Weak, strong, and strong-cyclic solutions)**.** Let $K_\pi = (Q, T_\pi)$ be the execution structure induced by π from I, and let G be the goal set.

- π is a **weak solution** if, for every $s_0 \in I$, there *exists* an execution path from s_0 that ends in a terminal state belonging to G.
- π is a **strong solution** if K_π is acyclic and *every* terminal state of K_π belongs to G.
- π is a **strong-cyclic solution** if from every state of Q some terminal state of K_π is reachable, and every terminal state of K_π belongs to G.

These three notions increase in strength. Weak planning is *existential*: one lucky path suffices. Strong planning is *universal and acyclic*: every path terminates, and every termination is a goal. Strong-cyclic planning sits between the two: cycles are tolerated, but only if from every state along them a goal remains reachable, and a fairness assumption rules out the run that loops forever while postponing the exit.

Example 6.3.4 (Weak, strong, and strong-cyclic on the parcel domain)**.** In fig. 6.2 no strong plan can exist. Whatever the planner does at L_{red}, the environmental outcome "stay red" is always available, so any execution structure retains the cycle $\mathsf{L}_{red} \longrightarrow \mathsf{L}_{green} \longrightarrow \mathsf{L}_{red}$. A weak plan exists trivially: a path that crosses only when green *can* reach the airport in finitely many steps. A strong-cyclic plan also exists, because from every state in the structure an airport state remains reachable, and under the fairness assumption that the green outcome cannot be postponed forever, the robot eventually crosses. This is exactly the situation safety-critical AI planning cares about: we cannot *force* success, but we can guarantee it under fair scheduling.

Definition 6.3.5 (Situated, universal, and conformant plans)**.** A **situated plan** assumes a known initial state. A **universal plan** gives an action choice for every reachable state in a set of possible states. A **conformant plan** must work without observing which of several possible initial states is the real one.

Remark 6.3.6 (Determinization)*.* A non-deterministic plan π is a weak (respectively strong, strong-cyclic) solution if and only if *every* deterministic sub-plan $\pi' \subseteq \pi$ that still covers all states of π is also a weak (respectively strong, strong-cyclic) solution. This universal quantifier over determinizations is what lets us search the smaller space of deterministic plans without loss of strength.

Strong cyclic planning is useful precisely when recovery loops are acceptable but strong guarantees are unachievable. If the robot repeatedly tries to cross while the light may nondeterministically be red or green, a strong plan is impossible because the red outcome can keep it stuck forever. A strong cyclic plan is still reasonable under the fairness assumption that the green outcome eventually occurs — the same kind of compassion constraint that we will revisit when we study fair model checking in Part II.

A useful mnemonic is the logic-flavour summary:

Plan type	Path quantifier	Acyclicity
weak	$\exists$ path reaching G	not required
strong	$\forall$ paths, terminal in G	required
strong-cyclic	$\forall$ states, $\exists$ path to G (under fairness)	cycles allowed

6.4 Preimage Operators

How do we compute, rather than just describe, the states from which the planner can win? The answer is the same trick that drove the attractor construction in chapter 5: work backwards from the goal, one step at a time, and ask which states can be forced into the current winning set by a single action choice. The two flavours of plan — weak and strong — give rise to two one-step predecessors, the *preimage operators*.

Definition 6.4.1 (Weak preimage)**.** The weak preimage of $X \subseteq S$ is

$$\mathsf{Pre}_w(X) = \{s \in S : \exists a \in A, \exists s' \in X \text{ such that } (s, a, s') \in T\}.$$

This is the one-step version of a weak plan: there exists an action and an outcome that reaches X. It captures the planner's ability to be *lucky* — the environment is allowed to pick the friendliest outcome of the chosen action.

Definition 6.4.2 (Strong preimage)**.** The strong preimage of $X \subseteq S$ is

$$\mathsf{Pre}_s(X) = \{s \in S : \exists a \in A,\ \mathrm{Post}(s, a) \neq \emptyset \text{ and } \mathrm{Post}(s, a) \subseteq X\}.$$

This is the one-step version of a strong plan. The planner chooses the action, but nature chooses the outcome. Therefore every possible successor must already lie inside X; one bad outcome is enough to disqualify s from being a strong predecessor.

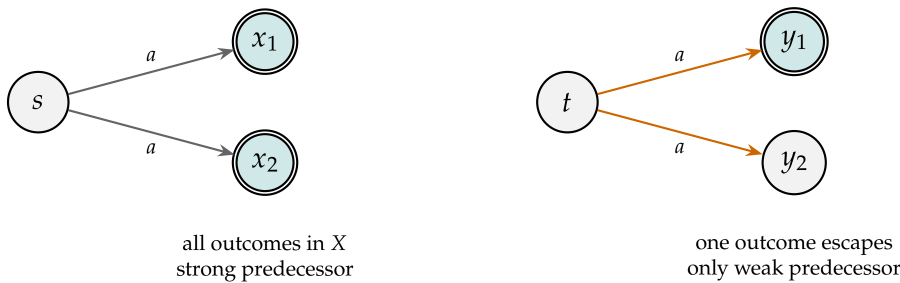


Figure 6.3: Strong vs. weak preimage. In graphite, $s \in \mathsf{Pre}_s(X)$: all outcomes of action a land in X. In orange, $t \notin \mathsf{Pre}_s(X)$ but $t \in \mathsf{Pre}_w(X)$: one outcome escapes, so a cannot *force* entry into X.

Figure 6.3 makes the operational difference explicit. The state s is a strong predecessor of X because action a pins both successors inside X. The state t is not: even though a has one successor in X, it has another outside, and an adversarial environment can pick that one. This is precisely the same "controllable predecessor" operator that built the system attractor in the games of chapter 5; only the names have changed.

6.5 Backward Fixpoint Algorithms

With the preimage operators in hand, the strong and weak planning problems reduce to least-fixpoint computations on the lattice $(2^S, \subseteq)$. The goal set is the seed; we repeatedly add states from which the planner can force progress into the current winning set; when nothing new appears, we have reached the fixpoint.

■ **Formal details — Strong backward planning**

Step 1. Initialize $W_0 = G$.

Step 2. Compute

$$W_{i+1} = W_i \cup \mathsf{Pre}_s(W_i).$$

Step 3. Stop when $W_{i+1} = W_i$.

Step 4. The resulting fixpoint W^* is the set of states from which the goal can be *forced*.

The weak variant is identical except that Pre_s is replaced by Pre_w.

This is the same logical shape as the attractor computation in chapter 5. At each step we add states from which the controller has an action whose possible outcomes are already known to be winning. The iteration terminates because S is finite and the sequence is monotone. By the Knaster–Tarski theorem the result is exactly the least fixpoint of the operator $X \mapsto G \cup \mathsf{Pre}_s(X)$.

Proposition 6.5.1 (Strong winning region)**.** *Let W^* be the fixpoint above. If an initial state belongs to W^*, then a strong plan exists from that state. If it does not belong to W^*, no memoryless strong plan based on the represented state space can force the goal using the available transitions.*

Proof. For the first claim, assign a rank to each state according to the first iteration in which it enters the sequence $W_0, W_1, \ldots$. States of rank 0 are already goals. If a state has rank $i+1$, then by definition it belongs to $\mathsf{Pre}_s(W_i)$, so there exists an action whose every successor lies in W_i. Choose such an action. No matter which successor occurs, the rank strictly decreases. Since ranks are natural numbers, every execution reaches rank 0, hence a goal.

For the second claim, let W^* be the fixpoint and consider a state $s \notin W^*$. For every available action, either it has no valid successor or it has some successor outside W^*; otherwise s would belong to $\mathsf{Pre}_s(W^*)$ and hence to W^* by the fixpoint property. An adversarial environment can always choose such an outside successor, keeping the execution outside W^* forever. Therefore no state-based strong plan can force the goal from outside the fixpoint. □

The extracted policy chooses, at each newly added state, one action that witnessed membership in $\mathsf{Pre}_s(W_i)$. Ranks double as a certificate: they bound the number of steps to the goal.

■ **Formal details — Strong-cyclic backward planning**

Strong-cyclic planning sits between weak and strong planning. A weak plan only needs one successful outcome; a strong plan must tolerate all outcomes. A strong-cyclic plan tolerates all outcomes *provided* the chosen policy never leaves a closed live region and fairness rules out postponing enabled progress forever.

The standard computation therefore alternates two requirements. Given a candidate region R, keep a non-goal state $s \in R$ only if there is an action a such that

$$\mathsf{Post}(s,a) \neq \emptyset, \qquad \mathsf{Post}(s,a) \subseteq R, \qquad \mathsf{Post}(s,a) \cap X \neq \emptyset$$

for the current progress set X. The first two clauses say that the policy is enabled and closed inside the candidate region; the third says that, under a fair resolution of nondeterminism, the policy has a way to make progress toward states already known to be closer to the goal. Starting from $X = G$ and iterating this progress test inside a greatest closed candidate yields the usual nested-fixpoint view of strong-cyclic planning.

Operationally, one can read the algorithm as a backward pass followed by pruning, but the pruning is not optional bookkeeping: it removes states for which every apparent weak path to the goal depends on first leaving the live region or on cycling forever without any fair progress

edge. The resulting policy has two invariants: every possible successor stays inside the region, and from every reachable non-goal state the goal is again reachable through policy edges.

This is the precise content behind the slogan “from every state, a goal remains reachable, and every terminal state is a goal”. The policy is not allowed to gamble on a single lucky branch: it must keep all nondeterministic outcomes inside the controlled region. Fairness then supplies the missing ingredient that distinguishes strong-cyclic planning from strong planning: if progress remains continuously available, an execution cannot ignore it forever.

6.6 Symbolic Representation

Explicit state graphs can be enormous. The parcel domain has six states, but a domain with forty Boolean fluents already hides up to 2^{40} states, and most realistic planning problems have many more. Symbolic model checking sidesteps the explosion by representing sets and transitions with Boolean formulas and manipulating whole sets at once. The preimage operators we defined set-theoretically translate cleanly into quantified Boolean formulas, where they can be evaluated by BDD-based or SAT-based engines.

Definition 6.6.1 (Symbolic transition relation)**.** Let x be the vector of current-state variables, a the action variables, and x' the next-state variables. A symbolic transition relation is a Boolean formula

$$T(x, a, x')$$

that is true exactly for valid transitions.

Why represent the chosen action with a vector of Boolean variables a rather than a single scalar? As the professor noted, assigning an independent Boolean variable to each action naturally models **concurrent actions**: multiple action variables can be true at once. If the domain forbids concurrent execution, the modeller must explicitly add a mutual-exclusion constraint (e.g., $\sum a_i \leq 1$) to the transition relation. When concurrency is ruled out structurally, a more compact scalar encoding (using $\lceil \log_2 |A| \rceil$ bits) suffices.

A set of states is represented by a formula $X(x)$. The strong preimage can then be written logically as

$$\mathsf{Pre}_s(X)(x) = \exists a \left(\exists x' \, T(x, a, x') \ \wedge \ \forall x' \, (T(x, a, x') \rightarrow X(x')) \right).$$

Read this formula carefully: the outermost $\exists a$ is the planner’s choice of action, the inner $\exists x' \, T$ checks that the action is actually enabled, and the universal $\forall x'$ is nature’s adversarial choice of outcome, which we force to land inside X.

The weak preimage has the existential version of the same shape:

$$\mathsf{Pre}_w(X)(x) = \exists a \, \exists x' \, \big(T(x, a, x') \wedge X(x')\big).$$

The strong preimage replaces the existential choice of outcome with a universal requirement over all outcomes of the chosen action. In implementation, these formulas are rarely evaluated by enumerating assignments: they are compiled into ordered binary decision diagrams (OBDDs), on which projection and Boolean operations are polynomial in the diagram size. Part II develops this symbolic machinery in detail.

Intermezzo — Why Symbolic Methods Matter

If a domain has 40 Boolean fluents, the explicit state space may contain 2^{40} states. Symbolic methods do not make the problem magically small, but they let us manipulate large regular sets of states at once, often using compact representations such as ordered binary decision diagrams. Many planning domains are mostly regular: a robot's dynamics factor over its fluents, and a single BDD node can summarise millions of concrete states that all behave identically.

6.7 Action Description Languages

Writing the transition relation $T(x, a, x')$ by hand is tedious and error-prone. Planning tools therefore let the modeller declare, at a higher level, what each action *requires* and what it *causes*; a compiler then expands those declarations into the symbolic transition relation used by the fixpoint algorithms. This is the planning analogue of writing a program in a high-level language and letting a compiler lower it to machine code.

> **Definition 6.7.1** (Action rule)**.** An action rule describes when an action is executable and how it changes fluents. It typically has a precondition part and an effect part.

A typical rule has the form

$$A \text{ causes } P \text{ if } Q,$$

meaning that when action A is executed in a state satisfying Q, the next state must satisfy effect P. Executability conditions say when an action is allowed; frame or inertia constraints say that inertial fluents keep their current value unless some rule changes them.

Example 6.7.2 (Compiling one action rule)**.** The rule

$$\texttt{load} \text{ causes } \mathit{loaded} \text{ if } \mathit{robotAtParcel}$$

contributes the implication

$$\texttt{load}(a) \wedge \mathit{robotAtParcel}(x) \implies \mathit{loaded}(x')$$

to the transition relation. Inertia contributes, for example,

$$\neg\mathsf{changesLoaded}(x, a) \implies \big(\mathit{loaded}(x') \leftrightarrow \mathit{loaded}(x)\big).$$

Example 6.7.3 (Rules for the parcel domain)**.** The parcel domain of fig. 6.2 can be written declaratively as follows. State invariants pin down what a state is:

$$\mathsf{always}\ \Big(\mathit{pos} = \mathit{station} \oplus \mathit{pos} = \mathit{light} \oplus \mathit{pos} = \mathit{airport}\Big).$$

Initial and goal states are declared with the predicates initially and goal:

$$\mathsf{initially}\ \big(\mathit{pos} = \mathit{station}\big), \qquad \mathsf{goal}\ \big(\mathit{pos} = \mathit{airport}\big).$$

The actions are compiled into transition constraints:

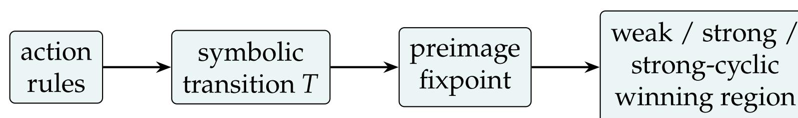
$$\texttt{transport}\ \text{causes}\ pos = light\ \text{if}\ pos = station,$$

$$\texttt{transport}\ \text{causes}\ pos = airport\ \text{if}\ pos = light \wedge color = green.$$

The colour *color* is non-inertial, so its evolution is left unconstrained by these rules; nature chooses it. A small set of declarative rules has thus generated the explicit six-state graph of fig. 6.2.

6.8 Towards Model Checking

The chapter has traced a single arc. Planning problems are finite transition systems with goals. Plans are state-action tables, and their strength is measured against the universal or existential path-quantification in the induced execution structure. Strong and weak solutions are characterised by preimage fixpoints; strong-cyclic solutions add a forward reachability pass on top. The whole pipeline compiles declaratively from an action description language into the symbolic transition relations of model checking:



The closing observation is that the weak and strong plan-existence questions are themselves expressible in branching-time temporal logic. The existence of a path to a goal is the CTL formula $EF\,G$; the universality of reaching a goal is $AF\,G$; the strong-cyclic guarantee, under fairness, is a fair variant of $AF\,G$. In other words, the very same CTL that will drive the symbolic model checkers of Part II already classifies the plans of this chapter. Planning *is* model checking on a finite transition system, with a reachability objective.

Summary & Key Takeaways

We have now closed Part I. Starting from finite words and regular languages, we added infinite words, complementation, infinite games, and finally planning. The throughline has been the same: a finite transition system, a target set, and a preimage-based backward search.

- Planning domains are finite transition systems with actions.
- Plans are state-action tables; under non-determinism they become strategies.
- Weak planning is existential; strong planning is universal; strong-cyclic planning adds fairness on top of reachability.
- Preimage operators are one-step controllable predecessors, reused from the attractor construction of chapter 5.
- Symbolic model checking represents large planning problems with Boolean formulas and computes their fixpoints without enumerating states.

Part I has built the theoretical machinery: finite and infinite automata, games, and the attractor/preimage reasoning that solves them. Part II turns to scale and practice. We will introduce temporal logics (LTL and CTL) as specification languages, study explicit and symbolic model-checking algorithms, and meet the toolchain (SMV, OBDDs, SAT-based bounded model checking, learning, and Tamarin) that turns the theory of these chapters into running verifiers. Many of the questions Part I leaves open — how do we specify temporal

goals beyond reachability, how do we handle fairness explicitly, how do we verify systems far larger than six states — are exactly what Part II is designed to answer.

Exercises

Exercise 1 (Weak vs. strong on the parcel domain). In fig. 6.2, list the contents of $\mathsf{Pre}_w(G)$ and $\mathsf{Pre}_s(G)$ for $G = \{\mathsf{A}_{green}, \mathsf{A}_{red}\}$ after a single iteration. Explain in one sentence why L_{red} belongs to the first set but not the second.

Exercise 2 (One fixpoint step). Let $G = \{g\}$ and suppose state s has an action whose only successors are g and another state already in G. Does s enter the strong winning set after one iteration? Justify by evaluating the membership condition for $\mathsf{Pre}_s(G)$.

Exercise 3 (Strong, weak, or strong-cyclic?). Consider the parcel domain of fig. 6.2 and the plan "at the light, if green then `transport`, else `wait`; at the station, `transport`". Classify it as weak, strong, or strong-cyclic, and explain which feature of fig. 6.2 rules out the stronger classifications.

Exercise 4 (Symbolic preimage). In the symbolic strong preimage formula

$$\exists a \, \big(\exists x' \, T(x, a, x') \wedge \forall x' \, (T(x, a, x') \rightarrow X(x'))\big),$$

identify which quantifier is controlled by the planner, which expresses action executability, and which expresses adversarial outcome selection by the environment.

PART II

Model Checking, Temporal Logic, and Verification Tools

CHAPTER 7

Reactive Systems, SMV, and Explicit Model Checking

Reactive systems do not stop, and their verification does not stop at finite input/output either: it reasons about every possible infinite interaction with the environment.

Part I built the mathematical machinery: finite automata to recognise finite behaviours, ω-automata to recognise infinite ones (chapter 3), games to synthesise controllers (chapter 5), and planning to compose actions under uncertainty (chapter 6). Throughout, the objects of study were idealised mathematical structures—graphs, automata, arenas—studied for their own sake. Part II now turns that machinery loose on *practice*: real programs with state variables, real specifications written in temporal logics, and real tools that push the verification automatically.

This chapter is the hinge. We introduce *reactive systems* as the class of programs we care about, formalise the *model checking* problem, and learn a concrete modelling language—SMV—with which we feed models to the NuXMV model checker. A useful way to read the chapter is as a single pipeline. A reactive system has infinite executions; a Kripke structure packages all those executions in one finite graph; specifications select which of those executions count as correct; counterexamples explain why a violated specification fails; and SMV is the engineering notation from which the graph, the specifications, and the witnesses are all generated.

7.1 Why Reactive Systems?

Most courses on program correctness focus on *functional programs*: a function receives some input, computes a result, and terminates. Proving it correct amounts to showing that the output satisfies a postcondition given the input. We are *not* primarily interested in that setting here.

> **Definition 7.1.1** (Reactive system). A *reactive system* is a program or device that maintains a *continuous, potentially infinite interaction* with its environment.

Intuition. The defining feature is not *what* the system computes but *when* it computes. A reactive system is always switched on: at every instant it must be ready to accept an input, produce an output, and move on. Correctness is therefore a statement about the entire infinite dialogue with the environment, not about a single terminating computation. This contrasts sharply with the

functional view of programs—factorials, sorting routines, Fourier transforms—where correctness is a relation between one input and one output and the program is allowed to halt as soon as the answer is produced.

Examples are everywhere: an operating system responding to user events, the flight-control software of an airplane, a communication protocol exchanging messages between two parties. The interaction never truly ends—even if the device is eventually switched off, correctness must hold for every prefix of that infinite dialogue.

The infinite-execution viewpoint. Think of a reactive system's execution as an infinite sequence of *time points*:

$$s_0 \to s_1 \to s_2 \to \cdots$$

Each s_i is a snapshot of the system's internal state. Reading the indices $0, 1, 2, \ldots$ as time positions, the execution is exactly an *ω-word* over the alphabet of state labels, in the sense of theorem 3.1.1 in Part I. This is more than an analogy: the request/acknowledgement traces we used to motivate Büchi automata in fig. 3.2 were already behaviours of an idealised reactive system. The sequence is potentially infinite, so proving a property of the system means proving something about this infinite flow—very much like reasoning about the natural numbers $\mathbb{N}$. This analogy is not accidental: we will see (chapter 8) that the theory of infinite executions is precisely the theory of the monadic second-order logic of one successor (S1S), which reasons directly over $\mathbb{N}$.

This is why we will study *temporal logics*: they are the right language for asserting facts about infinite sequences of states. The plural is intentional—there are many temporal logics, each with its own expressive power and algorithmic properties.

7.1.1 Closed vs. Open Systems

When we model and verify reactive systems, we must distinguish between two fundamental configurations of their environment:

- **Closed Systems**: A closed system is completely self-contained. Its transitions and state changes are determined entirely by its own internal logic and non-deterministic choices. There are no external inputs from the environment. In verification, closed systems are modeled directly as Kripke structures (where all choices are internal to the model).
- **Open Systems**: An open system maintains a continuous interaction with an environment that it does not control. Its transitions depend not only on its internal state, but also on external inputs supplied by the environment. To verify an open system, we must ensure that it satisfies its specification under *all* possible sequences of environment inputs. In synthesis, open systems are modeled as games between the system and the environment.

While real-world reactive systems are almost always open, we often analyze them as closed systems during model checking by incorporating a model of the environment directly into the state space.

7.1.2 The cost of failure

Reactive systems are frequently deployed in safety-critical domains where defects can be fatal or exceptionally costly. We return to the historical record

in section 7.2, where the failures of the Therac-25, Ariane 5, and the Pentium FDIV bug motivate the model-checking workflow directly.

7.2 The Model Checking Problem

The previous section explained why executions are infinite. Model checking adds the missing verification question: given a finite description of all possible executions, and a formal property of those executions, can a tool decide whether the property always holds?

7.2.1 What model checking is

> **Definition 7.2.1** (Model checking). *Model checking* is an automatic, exhaustive verification technique that takes two inputs—a *system model* M and a *specification* φ—and decides whether every behaviour of M satisfies φ. When the answer is negative, it produces a concrete *counter-example* trace.

Intuition. The model M is a finite description of *all* behaviours of the system; the specification φ is a temporal formula describing the behaviours we accept as correct. Model checking is the decision problem $M \models \varphi$, made effective by the finiteness of M. Crucially, the answer is not a probabilistic guarantee but a mathematical theorem that holds for every possible execution, including those driven by an adversarial environment.

Three features distinguish model checking from other verification approaches.

1. **Fully automatic (push-button).** No manual proof steps are needed. The engineer writes the model and the property; the tool gives a yes-or-no answer. This is in sharp contrast with deductive methods such as Hoare logic, which require hand-crafted invariants and proof obligations.
2. **Exhaustive.** Unlike testing, which covers a finite sample of behaviours, model checking verifies *all* possible behaviours. If the tool says "satisfied", the property holds in every execution, for every possible environment input, with 100% coverage.
3. **Counter-example generation.** When the property is violated, the model checker produces a concrete execution trace starting from an initial state and ending at the faulty behaviour. This trace can be *replayed* on the model to locate and fix the defect, enabling an iterative *refinement loop*: model $\rightarrow$ check $\rightarrow$ counter-example $\rightarrow$ fix model $\rightarrow$ re-check.

> *Remark* 7.2.2 (Universal vs. Existential Model Checking). The standard definition of model checking (as described above) is *universal*: it proves that a property holds in *all* possible executions of the system. However, model checking can also be used existentially. In *existential model checking*, we ask if there exists *at least one* execution satisfying a property—for example, finding if a security vulnerability or an information leak is reachable. This is particularly efficient using *Bounded Model Checking* (introduced in the early 2000s), which searches for paths up to a fixed length k.

fig. 7.1 sketches the overall workflow.

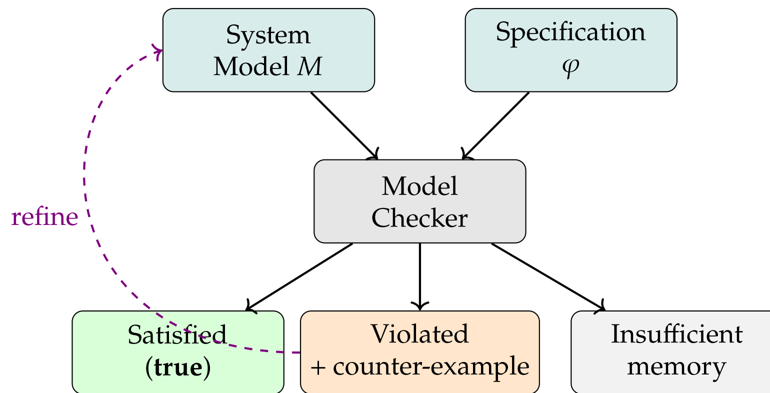


Figure 7.1: The model-checking workflow. The tool takes a system model and a specification as input. It outputs either a correctness certificate, a counter-example trace (enabling model refinement), or a memory-exhaustion signal. The dashed arrow shows the refinement loop.

7.2.2 The quality of model and specification

Model checking gives guarantees that are exactly as strong as the quality of its two inputs. A vivid failure mode:

> **Example 7.2.3** (Vacuously true specification)**.** Suppose the engineer writes a complex temporal formula φ that, due to a typo or logical error, contains the sub-formula $p \vee \neg p$. The whole formula is then a tautology. The model checker will always report "satisfied"—even if the system is completely broken—because φ is trivially true in every state. The result is *garbage out* for *garbage in*.

In practice, verifying the quality of specifications is as important as running the model checker itself. Techniques include:

- **Vacuity detection**: check whether φ is satisfied because some sub-formula is unreachable.
- **Mutation testing on specifications**: slightly mutate φ and check that the answer changes.
- **Sanity checks**: verify simple expected properties first (e.g., the initial state is reachable) before moving to complex ones.

7.2.3 Universal and existential model checking

The model checking we study is primarily *universal*: we ask whether *all* behaviours of the system satisfy φ. There is a dual variant, *existential model checking*, that asks whether *there exists* at least one behaviour satisfying φ.

Existential model checking is useful for security analysis: the property φ can express the *existence of an information leak* or a reachable dangerous state. The celebrated *Bounded Model Checking* (BMC) algorithm—introduced by Cimatti and collaborators in the early 2000s—is essentially an efficient existential model checker based on SAT solving. We will return to SAT-based and symbolic variants later in the model-checking part.

7.2.4 A brief historical note

The definition above is abstract, but the motivation is concrete. Model checking became important because many severe failures were not caused by a single wrong arithmetic result; they were caused by rare interactions between states, timing, concurrency, and environment choices—exactly the behaviours that are hard to cover with ordinary testing.

The field of formal verification developed in response to a string of software failures whose costs—in human lives and money—were enormous.

- **Therac-25** (1985–1987): a medical linear accelerator whose control software, written in assembly *without any safety controls*, contained a race condition (a bug triggered by the precise timing of access to a shared resource between two concurrent processes). The race caused massive radiation overdoses; several patients died. Formal verification of the concurrency would have detected the bug.
- **AT&T network outage** (early 1990s): a misinterpretation of the `break` statement in a C program propagated through a distributed telephone network, knocking out millions of customers and costing tens of millions of dollars.
- **Ariane 5 crash** (June 1996): the flight-control software attempted a data conversion from a 64-bit floating-point number to a 16-bit signed integer; the value was out of range, the exception was unhandled, and the rocket self-destructed seconds after launch.
- **Intel Pentium FDIV bug** (1994): an incorrect look-up table in the floating-point division unit produced wrong results for certain inputs. Intel recalled chips at a cost of approximately half a billion US dollars—a bug discovered *after release*, at the rightmost point of the cost-of-repair curve.

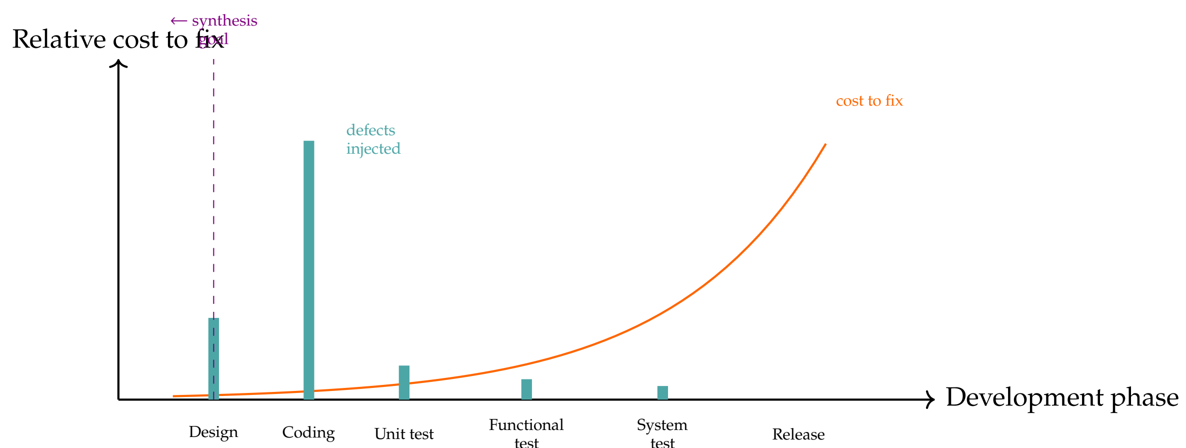


Figure 7.2: The cost to fix a defect grows exponentially with the development phase in which it is discovered. The majority of defects are injected during coding, but the cost of fixing them after release is orders of magnitude higher. Synthesis aims to eliminate bugs at the design phase entirely.

The theoretical groundwork for model checking was laid by Pnueli (1977), who introduced temporal logic for reactive programs, and by Clarke and Emerson (1980), who proved that the model-checking problem for CTL is decidable and gave the first algorithm. All three received the ACM Turing Award in 2007.

7.2.5 Verification vs. validation

Two terms that appear together but mean different things:

- **Verification**: *Are we building the product right?* Given a specification φ and a model M, does $M \models \varphi$? The inputs are formal.
- **Validation**: *Are we building the right product?* Does the specification capture *all* needs of the stakeholders? This is a question about completeness relative to an informal requirement, much harder to automate.

7.2.6 The state-explosion problem

Consider a system with n Boolean state variables. Each assignment of those variables is a distinct state, so the state space has size 2^n.

> **Example 7.2.4** (Weather-station counter)**.** A program records the daily rainfall in millimetres for each of the seven days of the week. Suppose the maximum daily rainfall is 1000 mm (one metre), so each of the seven locations stores a value in $\{0, 1, \ldots, 999\}$. The total number of states is $1000^7 = 10^{21}$—comparable to the estimated number of atoms in the observable universe.

The explosion becomes worse with concurrent systems: if we have N processes each with 2 states, the combined state space (their Cartesian product) has 2^N states. With 100 processes, that is 2^{100}, which is infeasible to store or enumerate.

The state-explosion problem is the central obstacle in model checking. Two responses:

- **Classical (explicit-state) model checking** (1980s–1990s): constructs and traverses the state graph explicitly. Works for moderate-size systems; fails for large ones.
- **Symbolic model checking** (1990s–today): represents sets of states as Boolean formulas or Binary Decision Diagrams (BDDs) and operates on them without constructing individual states. We will study this later when CTL and symbolic state-space traversal are developed.

7.3 Kripke Structures

We have now named the verification problem. The next step is to choose the mathematical object that the model checker actually analyses.

The most abstract model we use is the *Kripke structure*, the formal counterpart of the "state machine" idea. All the concrete modelling languages we will see—SMV, NuSMV, and others—ultimately produce Kripke structures as their semantic objects.

> **Definition 7.3.1** (Kripke structure / transition system)**.** A *Kripke structure* (also called a *transition system* or a *state machine*) is a tuple
>
> $$M = (AP, Q, Q_0, R, L)$$
>
> where

- AP is a finite set of *atomic propositions*—observable facts that may hold or not in each state;
- Q is a finite set of *states*;
- $Q_0 \subseteq Q$ is the (non-empty) set of *initial states*;
- $R \subseteq Q \times Q$ is the *transition relation*, required to be *total* (every state has at least one successor); and
- $L : Q \to 2^{AP}$ is the *labelling function*, mapping each state to the set of atomic propositions that hold there.

Intuition. Think of M as a directed graph whose nodes are labelled with sets of observable facts. A *path* through M starting from $q_0 \in Q_0$ is an infinite sequence $q_0, q_1, q_2, \ldots$ with $(q_i, q_{i+1}) \in R$ for all i. This formalises the "reactive execution" idea: the system never terminates and continuously transitions between states.

M encodes *all behaviours simultaneously*: different non-deterministic choices at branching points yield different paths. Model checking asks whether a temporal property holds on *every* path that starts from an initial state.

A finite graph can generate infinitely many behaviours of infinite length. For example, a two-state graph with a self-loop generates infinitely many paths (stay in state 0 forever, visit state 1 once then loop, twice then loop, etc.), each of infinite length. There are thus two orthogonal "degrees of infinity": the number of paths (infinite) and the length of each path (also infinite). This is exactly what the theory of ω-words from Part I was designed to handle.

Example 7.3.2 (Arbiter Kripke structure)**.** A resource arbiter has two state variables: `request` $\in \{false, true\}$ and `state` $\in$ {Ready, Busy}. There are four states (one per assignment). The initial condition fixes `state` = Ready, with `request` unconstrained, giving *two* initial states.

To see how this corresponds precisely to theorem 7.3.1, we can specify the components of the Kripke structure $M = (AP, Q, Q_0, R, L)$:

- $AP = \{r, ready, busy\}$, where r represents `request` = *true*, *ready* represents `state` = Ready, and *busy* represents `state` = Busy;
- Q = {RF, RT, BF, BT} corresponding to the four valuations:

RF = (Ready, *false*) RT = (Ready, *true*)
BF = (Busy, *false*) BT = (Busy, *true*)

- Q_0 = {RF, RT} because the system must start with `state` = Ready, while the initial value of `request` is unconstrained;
- The transition relation R consists of the directed edges shown in fig. 7.3;
- The labelling function L maps each state to its active propositions:

L(RF) = {*ready*} L(RT) = {*ready*, r}
L(BF) = {*busy*} L(BT) = {*busy*, r}

fig. 7.3 shows the structure for the *buggy* version of the arbiter (see theorem 7.6.1).

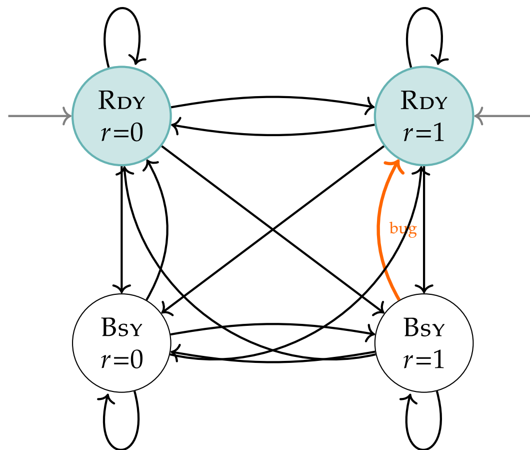


Figure 7.3: Kripke structure of the unconstrained arbiter (Example 7.3.2). Initial states (teal) have `state` = READY. The orange transition (BUSY, r=1) $\longrightarrow$ (READY, r=1) witnesses the bug: with a request pending, the unconstrained `TRUE` branch of the `case` lets the next state be READY rather than BUSY, violating **AG**(`request` $\longrightarrow$ **AX**(`state` = BUSY)).

7.4 Specifications: Safety and Liveness

A Kripke structure packages *all* behaviours of a system; a specification selects which of those behaviours we accept as correct. Choosing the specification is therefore as important as choosing the model: an overly weak property is satisfied vacuously, an overly strong property is never satisfied, and a property in the wrong shape may not even be expressible in the temporal logic we plan to use. In this section we introduce the classical Lamport taxonomy that splits temporal specifications into two families. The split is not just pedagogical: as the LTL material in chapter 11 will make algorithmic, safety and liveness properties are checked by qualitatively different searches.

7.4.1 Two kinds of "always good"

Before giving the formal definitions, let us build intuition on the arbiter of theorem 7.3.2. Two natural requirements are:

- *"Two processes are never granted the resource at the same time."* A violation is observed *at a single instant*: as soon as we see the bad configuration, no future behaviour of the system can deny that it happened.
- *"Every request is eventually followed by a grant."* A violation can never be observed at any finite instant: no matter how long the request has been pending, there is always a future in which the grant arrives. The only way to refute the property is to exhibit an *infinite* run in which the grant never comes.

The first property is a *safety* property; the second is a *liveness* property. The distinction is captured formally by asking what kind of finite prefix witnesses a violation.

Definition 7.4.1 (Safety property)**.** A property $P \subseteq (2^{AP})^\omega$ is a *safety property* if every behaviour that violates P has a *finite bad prefix*: for every $\sigma \notin P$ there exists $i \geq 0$ such that no extension of the prefix $\sigma[0, i]$ belongs to P.

Intuition. A safety violation is witnessed at a concrete instant. Once the bad thing has happened, no future behaviour of the system can "undo" it. Typical safety properties are mutual exclusion (two processes are never simultaneously in their critical sections), partial correctness of an arbiter (a grant is never asserted without a pending request), type correctness (a pointer is never dereferenced after being freed), and absence of deadlock. Mechanically, a safety property is checked by a *reachability* search: does any reachable state satisfy the bad condition?

Definition 7.4.2 (Liveness property)**.** A property $P \subseteq (2^{AP})^\omega$ is a *liveness property* if every finite prefix can be extended to a behaviour in P: for every finite word $u \in (2^{AP})^*$ there exists an infinite word $\sigma \in P$ with prefix u.

Intuition. A liveness property can never be refuted by a finite prefix: no matter how long we observe the system, there is always a future in which the property could still hold. Violations therefore require an *infinite* witness. Typical liveness properties are starvation freedom (every pending request is eventually granted), eventual delivery (every sent packet eventually arrives), and the absence of indefinite waiting. Mechanically, a liveness property is checked by a *repeated reachability* or *fair-cycle* search.

Example 7.4.3 (Safety vs. liveness in the arbiter)**.** Reconsider the arbiter of theorem 7.3.2 with atomic propositions $AP = \{r, ready, busy\}$.

- "Whenever the arbiter is Busy, it was Ready in the previous step" is a *safety* property: the violation is a single transition in which the invariant fails, and no extension of that prefix repairs it.
- "Every request is eventually followed by a busy state," formally $\mathbf{AG}(r \rightarrow \mathbf{AF}\, busy)$, is a *liveness* property: a finite trace with a pending request can always be extended with a later busy state, so no finite prefix alone witnesses a violation.

The CTL property we wrote in the SMV model (`AG (request -> AX (state = busy))`) is a hybrid: a *bounded* liveness property, itself a safety property in disguise, because the deadline "next state" turns every violation into a finite witness of length two.

Remark 7.4.4 (The decomposition theorem). Every temporal property decomposes, in a precise sense, into a safety and a liveness component (the Alpern–Schneider decomposition). This is more than a theoretical curiosity: in practice we check safety with forward reachability and liveness with cycle detection, so knowing which part of a specification we are dealing with determines the algorithm we run.

The req/ack traces of chapter 3 give us a familiar vocabulary for the two families: "every `req` is followed by `ack` within two steps" is a safety property,

while "`ack` occurs infinitely often"—the very property recognised by the Büchi automaton of fig. 3.2—is a liveness property. The temporal logics LTL and CTL of chapter 11 and the later CTL material will give us the syntax to write both kinds compactly; for now the informal vocabulary is enough.

7.5 Counterexamples: From Bugs to Witnesses

When the model checker reports `false`, the third desideratum of model checking kicks in: it must produce a *counterexample*, a concrete behaviour of M that starts in an initial state and violates φ. Counterexamples are what makes model checking an iterative engineering activity rather than a one-shot oracle: a readable witness points the engineer at the exact sequence of choices that produces the bug, supports an interactive *model* $\to$ *check* $\to$ *fix* loop, and survives as a regression test once the bug is fixed.

The *shape* of a counterexample depends on the kind of property that was violated.

7.5.1 Finite prefixes for safety violations

If φ is a safety property, then by theorem 7.4.1 a violation has a finite witness. Concretely, the model checker returns a finite path

$$q_0 \to q_1 \to \cdots \to q_k$$

with $q_0 \in Q_0$ and such that the bad event is observed at position k. No infinite extension of the prefix can repair the violation, so the prefix alone is the certificate.

Example 7.5.1 (A safety counterexample in the arbiter)**.** In the unconstrained arbiter of fig. 7.3 the bounded property $\mathbf{AG}(\texttt{request} \to \mathbf{AX}(\texttt{state} = \textsc{Busy}))$ fails. Reading the figure, a finite counterexample is the three-state trace

$$\mathsf{RF} \to \mathsf{RT} \to \mathsf{BT} \to \mathsf{RT},$$

i.e., start with no request; the environment raises a request (now at RT); the arbiter correctly moves to Busy (now at BT); but `request` is still true at BT, and the unconstrained `TRUE` branch of the `case` lets the arbiter return to Ready—violating the property at the last step. This is exactly the orange "bug" edge of fig. 7.3.

7.5.2 Lassos for liveness violations

If φ is a liveness property, no finite prefix can witness its violation: we need an *infinite* behaviour in which the good event simply never occurs. By the pigeonhole principle on the finite state space, every infinite path in a finite Kripke structure eventually enters a cycle. Hence a liveness counterexample is always an *ultimately periodic* path of the form

$$q_0 \to \cdots \to q_i \xrightarrow{\text{cycle}} q_i \to \cdots$$

where the cycle is repeated forever. This is precisely the *lasso* shape of theorem 3.5.2 and fig. 3.5: a finite prefix u followed by an infinitely repeated non-empty suffix v.

Example 7.5.2 (A liveness counterexample is a lasso). In the producer–consumer model of section 7.9.2, consider the liveness property "whenever the buffer is full, eventually a cell is freed." Without fairness the property fails, and the counterexample is the infinite path on which the producer runs at every step:

$$\underbrace{(\mathsf{empty}, \mathsf{empty})}_{\text{prefix}} \to \underbrace{(\mathsf{full}_1, \mathsf{empty}) \to (\mathsf{full}_1, \mathsf{full}_2)}_{\text{prefix}} \to \underbrace{(\mathsf{full}_1, \mathsf{full}_2) \to (\mathsf{full}_1, \mathsf{full}_2) \to \cdots}_{\text{lasso cycle}}.$$

The cycle is a single self-loop on the state with a full buffer; the consumer never runs. Reading the witness, the engineer identifies the unfair scheduling choice and adds a fairness constraint (section 7.10) to rule it out.

Remark 7.5.3 (Why the model checker emits a lasso). The algorithmic reason is the same as the Büchi emptiness check of theorem 3.5.1 in Part I: a finite graph contains an infinite path avoiding some set of states iff there is a reachable cycle that avoids that set. Searching for such a cycle is exactly what an LTL model checker does, and the lasso it returns is the finite encoding of an infinite counterexample. The LTL chapter will make this automata-theoretic search precise.

The practical takeaway is that every counterexample is ultimately periodic, regardless of whether the property was safety or liveness. For safety the cycle has length zero (a single state suffices); for liveness the cycle is the heart of the witness. In section 7.6 we will write the models that produce such counterexamples, and in section 7.7 we will replay them with NuXMV.

7.6 The SMV Modelling Language

Kripke structures are ideal for definitions, but too low-level for writing real examples by hand. Drawing the four-state graph of the arbiter already tried our patience; a 100-state model would be untenable. SMV is the bridge between the engineering and the mathematics: the modeller writes variables, initial conditions, and next-state constraints in a small, typed language; the tool turns them into the Kripke structure of section 7.3 on which the algorithms of section 7.5 operate. Specifications are written in the same notation, so that the model, the property, and the counterexample all live in one file.

SMV (Symbolic Model Verifier) is both a specification language and the name of one of the first model checkers (CMU, early 1990s). We use SMV as a *modelling language*; for actual verification we run NuSMV/NuXMV (developed at FBK Trento, freely available).

SMV supports:

- **Synchronous and asynchronous** process composition.
- **Finite-state and infinite-state** models.
- **Modular and hierarchical** descriptions via `module` (analogous to classes in OOP).
- **Non-determinism** for modelling the environment or underspecified components.
- **Multiple specification languages**: LTL, CTL, and others.

Every SMV model has an entry-point module named `main`.

7.6.1 State variables and types

State variables are declared under the keyword `VAR`:

```
VAR
  x : boolean;
  y : 0..9;
  z : {idle, running, done};
```

A *state* is an assignment of a value to every state variable. With n Boolean variables we get 2^n states; with integer-range or enumeration variables the count grows accordingly.

7.6.2 The `ASSIGN` block

The `ASSIGN` block specifies how each variable evolves. It contains three kinds of statement:

`init(v) := expr` The *initial value* of variable v must satisfy `expr`.

`next(v) := expr` The *next value* of v in any successor state must satisfy `expr`.

`INVAR expr` An *invariant*: `expr` must hold in *every* state (those that violate it are pruned from the state space).

Rules.

- Every variable may appear in at most one `init` and at most one `next` statement.
- There must be no *dependency cycles*: it is illegal to write `next(x) := next(y)` and `next(y) := next(x)` simultaneously.
- A `next` expression may refer to the *current* value of any variable, but nesting `next` inside another `next` is not allowed.

7.6.3 Expression syntax

Simple expressions (used in `init` and `INVAR`) may contain:

- Constants: integers, `TRUE`, `FALSE`, enumeration literals.
- Arithmetic: `+`, `-`, `*`, `/`, `mod`, `abs`, `max`, `min`.
- Boolean connectives: `!` (not), `&` (and), `|` (or), `->` (implies), `<->` (iff), `xor`.
- Comparison: `=`, `!=`, `<`, `<=`, `>`, `>=`.
- Other variables (but not `next(v)`).

Next expressions (used in `next`) extend simple expressions with `next(v)` for any variable v, allowing the successor value of one variable to depend on the successor of another.

7.6.4 The `case` expression

A `case` expression provides conditional assignment:

```
case
  condition_1 : value_1;
  condition_2 : value_2;
  ...
  TRUE        : value_default;
esac
```

It evaluates to the value corresponding to the *first* condition that is true. The final `TRUE` branch (the "otherwise" clause) ensures that at least one condition always holds; omitting it is a syntax error.

Example 7.6.1 (Arbiter in SMV). `MODULE main`

```
VAR
  request : boolean;
  state   : {ready, busy};
ASSIGN
  init(state)  := ready;
  next(state)  := case
    state = ready & request : busy;
    TRUE                    : {ready, busy};
  esac;
SPEC
  AG (request -> AX (state = busy))
```

The first branch says: "if currently ready and a request is present, move to busy." The `TRUE` branch says: "otherwise, go non-deterministically to either `ready` or `busy`." Note that `request` has no `init` or `next`: it is *unconstrained*, so the model checker considers every possible value at every step.

The specification `AG (request -> AX (state = busy))` reads (in CTL): "in all reachable states, if a request is present, then in *all next states* the arbiter is in state `busy`."

7.6.5 Formal semantics of SMV

We now spell out how an SMV module translates to a Kripke structure. This is the point where syntax becomes behaviour: every declaration and constraint either creates states, removes impossible states, or restricts the transition relation.

State space. Each state is an assignment of values to all state variables. States that violate any `INVAR` condition are removed.

Initial states. A valuation s is an initial state if and only if it satisfies the conjunction of all `init` constraints:

$$s \text{ is initial} \iff s \models \bigwedge_v \varphi_v^{init}$$

where φ_v^{init} is the formula on the right-hand side of `init(v) :=`. Variables with no `init` constraint contribute the trivially true condition $\top$, so they are unconstrained in the initial state.

Example 7.6.2 (Arbiter initial states). Only `state` has an `init`: `init(state) := ready`. Since `request` is unconstrained, the initial states are the two valuations:

$$\{(\texttt{request} = 0, \texttt{state} = \textsc{ready}), \quad (\texttt{request} = 1, \texttt{state} = \textsc{ready})\}.$$

Transitions. There is a transition from state s to state s' if and only if the pair (s, s') satisfies the formula obtained by replacing each `next(v) := expr` with the constraint $v' \leftrightarrow expr$, where v' denotes the value of v in s':

$$(s, s') \in R \iff (s \cup s') \models \bigwedge_{v} (v' \leftrightarrow next_expr_v).$$

Here, unprimed variables are interpreted under s and primed under s'. Variables with no `next` constraint are unconstrained in the successor: they can take any value from their type.

Example 7.6.3 (Flip-flop semantics).
```
MODULE main
VAR   x : boolean;
ASSIGN
  init(x) := FALSE;
  next(x) := !x;
```
The state space is $\{s_0, s_1\}$ where $s_0 : x = false$ and $s_1 : x = true$. The only initial state is s_0. The transition constraint is $x' \leftrightarrow \neg x$. Checking:

- (s_0, s_1): $x = 0$ so $\neg x = 1$; must have $x' = 1$, which holds in s_1. ✓
- (s_0, s_0): $\neg x = 1$ but $x' = 0$ in s_0. × (not a valid transition)
- (s_1, s_0): $\neg x = 0$; must have $x' = 0$, which holds in s_0. ✓
- (s_1, s_1): $\neg x = 0$ but $x' = 1$. ×

The unique transition structure is $s_0 \to s_1 \to s_0 \to \cdots$ with *no self-loops*. fig. 7.4 shows this.

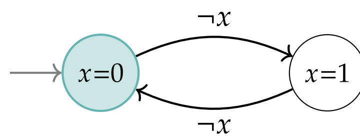


Figure 7.4: Kripke structure for the flip-flop of Example 7.6.3. The constraint `next(x) := !x` allows only alternating transitions; self-loops are excluded.

7.6.6 Adding a self-loop: implicit modelling with TRANS

Suppose we want to add a self-loop on state s_1 (i.e., allow x to remain `true`). Using `ASSIGN`, one approach is:

```
next(x) := case
  !x   : TRUE;
  TRUE: {TRUE, FALSE};
esac;
```

But this also adds a self-loop on s_0, which we may not want. The cleaner alternative is the *implicit modelling* style using the `TRANS` keyword:

```
MODULE main
VAR   x : boolean;
INIT  !x
TRANS !x -> next(x)
```

Reading the `TRANS` formula. The formula $\neg x \rightarrow$ `next`(x) says: "if x is currently false, then in the next step x must be true." When x is *true*, the antecedent of the implication is false, so the formula is trivially satisfied *regardless* of the next value of x. The model checker is therefore free to choose either `next`(x) = *true* or `next`(x) = *false*, giving a self-loop on s_1 but not on s_0.

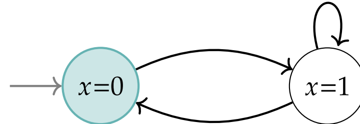


Figure 7.5: Adding a self-loop on s_1 via `TRANS !x -> next(x)`. The implication is vacuously true when x is true, so `next`(x) is unconstrained in s_1.

The two paradigms compared.

Assign paradigm	Trans/Init/Invar paradigm
Uses `ASSIGN` with `init(v)` and `next(v)`.	Uses `INIT`, `TRANS`, `INVAR` keywords.
Imperative style: "variable v evolves as `expr`".	Declarative style: "the joint behaviour of all variables satisfies this Boolean formula".
Easy to read; preferred for simple sequential models.	More expressive for complex relational constraints; preferred when multiple variables interact.
Both compile to the same Kripke structure.	(same)

7.6.7 Integer-range example: modulo-4 counter

```
MODULE main
VAR   x : 0..3;
ASSIGN
  init(x) := 0;
  next(x) := (x + 1) mod 4;
```

The state space is $\{0, 1, 2, 3\}$ with the unique deterministic cycle $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$. There are no self-loops and no branching.
Exercise: Rewrite this counter using only two Boolean variables b_1, b_0 (so that $x = 2b_1 + b_0$). Verify with the formal semantics that the resulting Kripke structure is isomorphic to Figure 7.6.

7.6.8 Non-determinism in SMV

The examples so far include deterministic counters and deliberately underspecified transitions. In model checking, this underspecification is not a defect

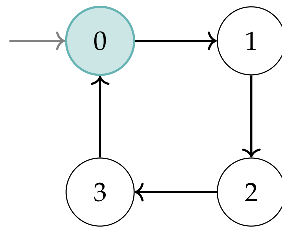


Figure 7.6: Kripke structure of the modulo-4 counter. The model is completely deterministic; there is a single cycle of length 4.

in the notation: it is how we represent choices made by the environment, an adversary, or a scheduler.

Non-determinism arises naturally in three ways.

1. **Unconstrained variable.** If no `init` or `next` is given for v, it may take any value in its type at any time step. Example: `request` in the arbiter.
2. **Set assignment.** `next(x) := {v1, v2}` lets the successor of x be either v_1 or v_2, non-deterministically.
3. **Underspecified `TRANS` formula.** A formula that does not pin down all variables in the successor state leaves some degrees of freedom, as we saw with the self-loop example.

Uses of non-determinism.

- **Modelling the environment.** The environment is typically outside the designer's control (e.g., which request arrives next, whether a network packet is lost). Declaring the corresponding variable unconstrained forces the model checker to verify the system under *all possible environments*, including adversarial ones.
- **Abstraction.** Non-determinism can replace a concrete sub-system that is not relevant to the property being checked, reducing the state space.
- **Concurrency.** An interleaving of N concurrent processes can be modelled by a non-deterministic choice of "which process goes next" at each step.

7.6.9 Input variables

A cleaner way to model environmental non-determinism is with *input variables* (keyword `IVAR`):

```
MODULE main
IVAR
  request : boolean;
VAR
  state : {ready, busy};
ASSIGN
  init(state)  := ready;
  next(state)  := case
    state = ready & request : busy;
    state = busy  & request : busy;
    TRUE                    : ready;
  esac;
LTLSPEC
  G (request -> X (state = busy))
```

Input variables *cannot* be constrained by `init` or `next`—attempting to do so is a syntax error. They behave like *labels on transitions* in the Kripke structure: the state space contains only the state variables, reducing from four to two states in this example.

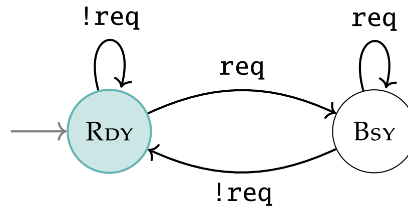


Figure 7.7: Kripke structure of the *fixed* arbiter with `request` as an input variable. There are only two states; input values label transitions. This is the correct model: from any state, whenever `request` is true, the next state is always Busy.

> *Remark* 7.6.4 (Modelling Finite Automata). Because input variables effectively act as labels on the transitions, they provide a natural way to model finite automata (like those from Part I) in SMV. To model an automaton with alphabet Σ, we can declare a set of Boolean input variables such that Σ is isomorphic to the power set of those variables (e.g., two input variables a, b can encode an alphabet of size four: $\emptyset, \{a\}, \{b\}, \{a, b\}$). The automaton's state becomes a single SMV state variable, and its transitions are encoded using `case` expressions over the input variables.

> *Remark* 7.6.5. Input variables cannot appear in CTL specifications under NuSMV, but can appear in LTL specifications. This is a tool-level restriction, not a fundamental one: it arises because CTL quantifies over paths using the A and E path quantifiers, which in NuSMV's implementation are not compatible with the "label on edge" semantics of input variables.

7.6.10 The `define` macro

```
DEFINE z := x xor y;
```

`DEFINE` introduces a *macro*: z is syntactic sugar for $x \oplus y$ and is *not* a state variable. The model checker performs *flattening* by substituting the definition at every use site before building the state space. Macros are useful for:

- Naming complex sub-expressions reused in multiple specifications.
- Defining carry-out signals or other derived signals in hardware models (see Section 7.8).

7.7 Simulation with NuXMV

Once the model has a precise semantics, the first practical check is usually not a full proof but a short execution. Simulation lets us inspect one path through the Kripke structure and catch modelling mistakes before asking the model checker to explore all paths. NuXMV provides an interactive simulation mode:

```
$ NuXMV -int model.smv
NuXMV > go
NuXMV > pick_state -v -i      -- choose initial state interactively
NuXMV > simulate -k 7 -v -i   -- simulate 7 steps, choose at each step
NuXMV > show_traces -v         -- display the generated trace
```

Example 7.7.1 (Simulating the flip-flop)**.** For the flip-flop module (theorem 7.6.3), `pick_state` reports only one available state ($x = false$). Simulating 7 steps with `simulate -k 7` produces the trace:

Step	0	1	2	3	4	5	6
x	0	1	0	1	0	1	0

This matches the deterministic alternation we derived from the semantics.

Example 7.7.2 (Simulating the modulo-4 counter)**.** Starting from $x = 0$ and simulating 23 steps yields: $0, 1, 2, 3, 0, 1, 2, 3, \ldots$ for 23 steps, arriving at $x = 3$. The NuXMV session:

```
NuXMV > go
NuXMV > pick_state -v         -- only state: x=0
NuXMV > simulate -k 23 -v -i
  State 1:  x = 1
  State 2:  x = 2
  State 3:  x = 3
  State 4:  x = 0
  ...
```

After simulation, model checking is invoked with:

```
NuXMV > check_spec         -- for CTL/LTL specs in the SPEC/LTLSPEC block
NuXMV > check_ctlspec -p "AG (request -> AX (state = busy))"
```

7.8 Modules: Hierarchical Modelling

Flat examples are useful for learning the semantics, but realistic systems are built out of repeated components. SMV modules let us keep that structure visible while still compiling the whole model to one flattened transition system.

Modules in SMV play the role of *classes* in object-oriented programming: a module is declared once and can be instantiated multiple times. Every module instantiation has its own copies of internal variables. Parameters are passed *by reference*: modifying a formal parameter inside a module modifies the actual parameter of the caller.

```
MODULE my_module(z)   -- formal parameter z
VAR
  x : boolean;
DEFINE
  y := z;             -- y is an alias for the caller's variable
ASSIGN
  init(x) := z;
  next(z) := TRUE;    -- this modifies the CALLER's variable!
```

The model checker performs *flattening*: each module instantiation is inlined, renaming variables with the instance path (e.g., `inst.x`). This does not add expressive power but improves readability.

7.8.1 The mechanical decimal counter

We model a 5-digit decimal counter (a mechanical device found at the end of industrial production lines). Pushing a button increments the count; a reset button returns all digits to zero.

Each digit is modelled by a `cell` module. The key idea is the *carry*: digit d_i must increment whenever digit d_{i-1} wraps from 9 back to 0 and there is a carry from its predecessor.

```
MODULE cell(carry_in, reset)
VAR
  value : 0..9;
DEFINE
  carry_out := (value = 9) & carry_in;
ASSIGN
  init(value) := 0;
  next(value) := case
    reset     : 0;
    carry_in  : (value + 1) mod 10;
    TRUE      : value;
  esac;
```

Reading the carry logic. `carry_out` is true when this digit is at 9 *and* a carry is arriving from the previous digit. That signals the digit after this one to increment. The least-significant digit `d0` always has `carry_in = TRUE` (it must always increment when the counter is stepped).

```
MODULE main
VAR
  d0 : cell(TRUE,          reset);
  d1 : cell(d0.carry_out,  reset);
  d2 : cell(d1.carry_out,  reset);
  d3 : cell(d2.carry_out,  reset);
  d4 : cell(d3.carry_out,  reset);
IVAR
  reset : boolean;
```

State-space size. 10^5 = 100,000 states (5 digits, each ranging 0–9). Without reset the graph is deterministic: a single cycle of length 100,000. With reset, the environment may push the reset button at any step, creating non-deterministic transitions back to state 00000.

7.8.2 Properties of the mechanical counter

We define three macros for readability:

```
DEFINE
  first_state := d0.value=0 & d1.value=0 & d2.value=0
```

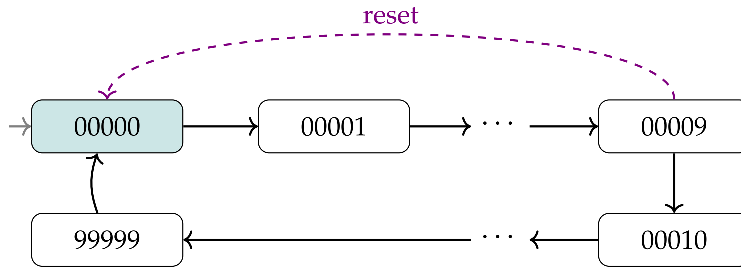


Figure 7.8: Simplified topology of the mechanical counter. Without reset, a single deterministic cycle of length 100,000. With reset (dashed), the environment can return to 00000 at any point.

```
              & d3.value=0 & d4.value=0;
last_state   := d0.value=9 & d1.value=9 & d2.value=9
              & d3.value=9 & d4.value=9;
state_81100 := d0.value=0 & d1.value=0 & d2.value=1
              & d3.value=1 & d4.value=8;
```

The four CTL properties below illustrate how the same model gives different answers depending on whether the reset input is included:

Property	Reading	No reset	With reset
EF(`last_state`)	*There exists a run reaching* 99999.	**T**	**T**
AF(`last_state`)	*All runs eventually reach* 99999.	**T**	**F**
AG(`state_81100` → **EX**(`first_state`))	*There is a 1-step path from* 81100 *to* 00000.	**F**	**T** (via reset)
AG AF(`first_state`)	*All runs visit* 00000 *infinitely often.*	**T**	**T**

For **AF**(`last_state`) with reset: the environment can always push reset before reaching 99999, so there exist runs that never arrive at 99999. Hence the property is false.

For **AG AF**(`first_state`) with reset: even if the user never pushes reset in some runs, the counter still cycles back through 00000 every 100,000 steps. In runs where reset is pushed, 00000 is visited even more often. So the property is true in all runs.

7.9 Asynchronous Composition

The decimal counter was hierarchical but still synchronous: every component advanced together in one global step. Concurrent systems add a different source of behaviour, namely the scheduler's choice of which process runs next.

7.9.1 Synchronous vs. asynchronous

In a *synchronous* composition, a single step of the combined system is a step taken *simultaneously* by all sub-processes. In an *asynchronous* composition, a step is taken by *at most one* process; the others remain idle (their variables do not change).

Asynchronous composition models concurrent processes that operate independently, such as those communicating over a network. Protocols are naturally asynchronous.

In SMV, prepending `process` to a module instantiation makes it asynchronous:

```
VAR
  prod : process producer(buf);
  cons : process consumer(buf);
```

NuSMV inserts a hidden variable `process_selector` that is non-deterministically assigned to one of the process names (or to `main`, meaning no process runs). The selected process has its `running` flag set to `TRUE`; the others have `running = FALSE` and their variables remain unchanged.

> *Remark* 7.9.1. The `process` keyword is deprecated in recent versions of NuSMV/NuXMV but still supported. A warning is emitted during the `go` step.

7.9.2 Producer–consumer example

Example 7.9.2 (Producer–consumer with shared buffer). A shared buffer has two cells, each either empty or containing one of three objects. A *producer* fills the leftmost empty cell; a *consumer* empties the leftmost non-empty cell.

```
MODULE main
VAR
  buf  : array 0..1 of {empty, o1, o2, o3};
  prod : process producer(buf);
  cons : process consumer(buf);
ASSIGN
  init(buf[0]) := empty;
  init(buf[1]) := empty;

MODULE producer(buffer)
ASSIGN
  next(buffer[0]) := case
    buffer[0] = empty : {o1, o2, o3};
    TRUE              : buffer[0];
  esac;
  next(buffer[1]) := case
    buffer[0] != empty & buffer[1] = empty : {o1, o2, o3};
    TRUE                                   : buffer[1];
  esac;

MODULE consumer(buffer)
ASSIGN
  next(buffer[0]) := empty;
  next(buffer[1]) := case
    buffer[0] != empty : buffer[1];
```

```
    TRUE                  : empty;
  esac;
```

The consumer always empties `buf[0]`. If `buf[0]` was non-empty, `buf[1]` is retained for the next consumption. If `buf[0]` was already empty, then `buf[1]` is also cleared (it was the leftmost non-empty cell and is now consumed).

Property without fairness.

$$\mathbf{G}\big(\texttt{buf[0]} \neq \texttt{empty} \wedge \texttt{buf[1]} \neq \texttt{empty} \longrightarrow \mathbf{F}(\texttt{buf[0]} = \texttt{empty} \vee \texttt{buf[1]} = \texttt{empty})\big)$$

"Whenever the buffer is full, eventually at least one cell is freed." This property is **false** without fairness: the scheduler can run the producer in every step and never execute the consumer, keeping the buffer full forever.

7.10 Fairness Constraints

The counter-example above is unrealistic: in any real system, a scheduler ensures that every process eventually gets CPU time. *Fairness constraints* formalise this assumption. They do not change the transition graph itself; instead, they say which infinite paths of the graph should count as admissible executions.

For this chapter, that is the whole point of fairness: it is an SMV-level *path filter*. We use justice and compassion to describe which infinite scheduler behaviours NuSMV should ignore before judging a property. We are not yet deriving the full LTL fairness model-checking algorithm; the later automata/tableau machinery in chapter 11 will explain how these "infinitely often" side conditions are checked.

7.10.1 Temporal hierarchy: four key concepts

Before defining justice and compassion, we must firmly understand four distinct temporal notions. Their confusion is one of the most common sources of errors in exams.

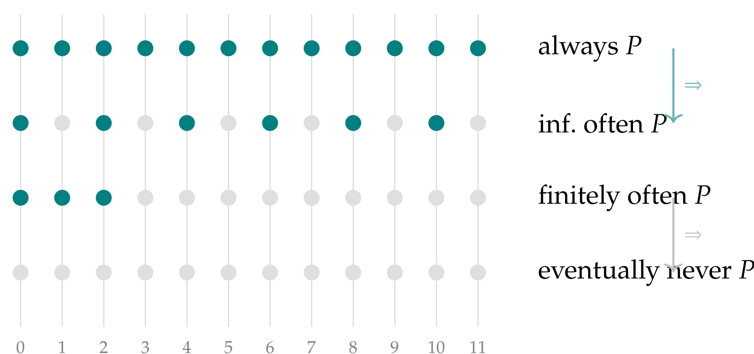


Figure 7.9: Four temporal behaviours of a property P along an infinite path. "Always P" implies "infinitely often P"; "finitely often P" means that, from some point on, P never holds again. The converse implications do not hold.

7.10.2 Formal definitions

Fix a model M with state machine $M' = (AP, Q, Q_0, R, L)$.

Definition 7.10.1 (Initialized path)**.** An *initialized path* π of M is an infinite sequence $\pi = q_0, q_1, q_2, \ldots$ in M' such that $q_0 \in Q_0$ and $(q_i, q_{i+1}) \in R$ for all $i \geq 0$. We write $\pi(i)$ for the i-th state.

Definition 7.10.2 (Enabled at position i)**.** A state $s \in Q$ is *enabled at position i of π* if there exists a transition from $\pi(i)$ to s, i.e., $(\pi(i), s) \in R$.

Definition 7.10.3 (Taken at position i)**.** A state s is *taken at position i of π* if $\pi(i+1) = s$, i.e., s is the *actual* successor chosen at step i.

Being *enabled* means the option exists; being *taken* means the option was actually exercised. In a deterministic system these coincide, but in a non-deterministic one many states may be enabled while only one is taken.

For a set $S \subseteq Q$, write $\text{Enabled}_S(i)$ when some state in S is enabled at position i, and $\text{Taken}_S(i)$ when the actual successor $\pi(i+1)$ belongs to S.

Definition 7.10.4 (Justice / Weak fairness)**.** A set $J \subseteq Q$ satisfies *justice* (weak fairness) on path π if:

$$\big(\exists k.\ \forall i \geq k.\ \text{Enabled}_J(i)\big) \implies \big(\forall k.\ \exists i \geq k.\ \text{Taken}_J(i)\big).$$

Equivalently (positive form):

> If states in J are *continuously available* from some time point on, then states in J must be *taken infinitely often*.

Definition 7.10.5 (Compassion / Strong fairness)**.** A set $C \subseteq Q$ satisfies *compassion* (strong fairness) on path π if:

$$\big(\forall k.\ \exists i \geq k.\ \text{Enabled}_C(i)\big) \implies \big(\forall k.\ \exists i \geq k.\ \text{Taken}_C(i)\big).$$

Concept	Condition	Required conclusion
Justice (weak)	*always* enabled (from time k)	taken *infinitely often*
Compassion (strong)	enabled *infinitely often*	taken *infinitely often*

Compassion is stronger than justice for the same scheduling choice: if something is enabled continuously from some point on, then it is enabled infinitely often, so strong fairness forces it to be taken infinitely often. The converse does not hold. A choice can be enabled infinitely often but with gaps; justice is silent in that situation, while compassion still requires infinitely many selections.

Example 7.10.6 (Fairness in the producer–consumer)**.** In the producer–consumer model, the set $S_{cons} = \{s \in Q \mid s \models \texttt{consumer.running}\}$ contains all states in which the consumer is the selected process.

The unfair run consists of the producer being selected in every step: `producer.running = TRUE, consumer.running = FALSE, ...` ad infinitum. In this run, the states in S_{cons} are enabled at every step (the scheduler could always pick the consumer) but never taken. This violates *justice*.

It also violates compassion: states in S_{cons} are enabled infinitely often (always, in fact) but taken zero times (finitely often).

Adding the fairness constraints:

```
JUSTICE  producer.running
JUSTICE  consumer.running
```

discards all paths that violate these constraints. The property "whenever the buffer is full, eventually a cell is freed" is now **true**: since the consumer must eventually run, it will free a cell, and this repeats.

Example 7.10.7 (A property that remains false even with fairness)**.** Consider the stronger LTL property:

$$\mathbf{G}(\texttt{full} \to \mathbf{F}\,\texttt{empty_buffer})$$

where `empty_buffer` := `buf[0] = empty` ∧ `buf[1] = empty`. This is **false** even under fairness: the producer can fill the buffer twice, then the consumer executes once (freeing one cell, satisfying justice), then the producer fills again, etc., keeping the buffer non-empty forever. Fairness requires that both processes run infinitely often—not that they alternate strictly.

7.10.3 Encoding fairness in NuSMV

```
-- Justice (weak fairness): P holds infinitely often
JUSTICE P

-- Compassion (strong fairness): if P infinitely often, then Q infinitely often
COMPASSION (P, Q)
```

Remark 7.10.8 (NuSMV terminology mismatch). In NuSMV, `JUSTICE P` means "P holds infinitely often on every fair path". This is a path constraint, not the enabled/taken definition above. It implements weak fairness for a scheduler only when P names the process that should be selected and that process is continuously enabled. For exam purposes, keep the semantic definition (Definitions 7.10.4 and 7.10.5) separate from the NuSMV keyword syntax.

7.10.4 Fair paths and model checking under fairness

A *fair path* with respect to sets $\mathcal{J}$ (justice constraints) and $\mathcal{C}$ (compassion constraints) is an initialized path that satisfies all constraints in $\mathcal{J}$ and $\mathcal{C}$.

Under fairness, model checking restricts attention to fair paths only: paths that violate any fairness constraint are excluded from consideration. As a consequence, a property that was false (because there existed an unfair counter-example) may become true (if every fair path satisfies it). This is the semantic preview needed for SMV: fairness filters the paths first, and the specification is evaluated only on the remaining paths.

7.11 LTL, CTL, and CTL*: an Overview

Throughout the chapter we have referred informally to LTL and CTL. We preview their relationship here, to be developed formally in chapter 11 and in the later CTL material. This section is only a map: Chapter 8 first explains the automata-theoretic foundation behind temporal specifications, and later chapters turn these names into full syntax, semantics, and algorithms.

Linear Temporal Logic (LTL). Formulas are evaluated on a single *infinite path* (a linear sequence of states). Operators include **X** (next), **U** (until), and their derived forms **F** (eventually) and **G** (always). Model checking LTL is **PSPACE-complete**.

Computation Tree Logic (CTL). Formulas are evaluated on the *computation tree*—the tree of all paths branching from each state. Each temporal operator is paired with a *path quantifier*: A (all paths) or E (exists a path). For example, **AF**φ means "on all paths, eventually φ"; **EF**φ means "there exists a path on which eventually φ". Model checking CTL is **polynomial** in the size of the state space and the formula.

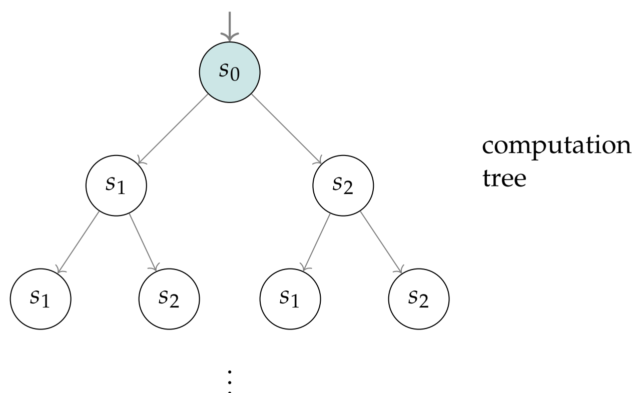


Figure 7.10: The computation tree rooted at s_0. CTL quantifies over branches of this tree (A/E). LTL refers to a single branch (a single execution).

CTL* (pronounced "CTL star"). A unifying formalism that subsumes both LTL and CTL. In CTL*, path quantifiers (A, E) and temporal operators (**X**, **U**, etc.) can be freely mixed. Model checking CTL* is **EXPSPACE-hard** and impractical, which is why LTL and CTL are used in practice instead.

Expressive power. LTL and CTL are *incomparable*: some properties expressible in LTL are not expressible in CTL, and vice versa. For example, "every state is eventually followed by p infinitely often" is an LTL property that has no CTL

equivalent. This is why both logics are studied and why no single sub-logic of CTL* suffices for all practical verification tasks.

7.12 Formal Methods in Context

We close the chapter by locating model checking among the other techniques that an engineer might use. This context matters because model checking is not a replacement for all validation activity: it is a precise answer to the specific question "does this finite model satisfy this formal property?"

7.12.1 Three categories of formal methods

1. **Deductive methods** (e.g., Hoare logic, dating back to Turing and Floyd): provide a formal proof that a property holds by annotating code with pre/post-conditions and loop invariants. Very expressive but require substantial manual effort; not easily automated for large systems.
2. **Simulation and testing**: unit testing, integration testing, functional testing. Fast and widely used but *incomplete*: a bug can hide in any unexplored behaviour.
3. **Model checking**: systematic, automatic, exhaustive. Combines the rigour of deductive methods with the automation of testing. Used daily at NASA, ESA, Intel, Boeing, and major software companies.

7.12.2 Static analysis and abstract interpretation

Two additional techniques deserve mention:

- **Static analysis**: analyses source code without executing it, often looking for security vulnerabilities (buffer overflows, null dereferences, race conditions). Unlike model checking, it usually does not reason about infinite executions.
- **Abstract interpretation**: a theoretical framework for static analysis based on approximating the program semantics. Closely related to the notion of over-approximation used in symbolic model checking.

7.12.3 The verification–synthesis spectrum

At one end: *testing* (a posteriori, incomplete). At the other: *synthesis* (a priori, complete by construction). Model checking occupies the middle: it is a posteriori (applied to an existing model) but exhaustive.

The overarching goal of the field is to move verification as early as possible into the design lifecycle. Synthesis—producing a system that is *correct by construction* from a declarative specification—is the culmination of this programme and will be studied in the final part of the course.

> **Summary & Key Takeaways**
>
> - A *reactive system* interacts continuously with its environment; its executions are infinite sequences of states.
> - *Model checking* takes a finite model M and a temporal specification φ and decides $M \models \varphi$ automatically, exhaustively, and with counterexample generation.
> - A *Kripke structure* encodes all behaviours of a system as a finite labelled graph; each infinite path is one execution.

- *Safety* properties are violated by a finite bad prefix; *liveness* properties require an infinite witness (a lasso).
- *SMV* is the modelling language compiled to Kripke structures; NUXMV is the tool that checks them.
- *Fairness* constraints (justice and compassion) filter out unrealistic scheduler behaviours before checking a property.
- The *state-explosion problem* is the central obstacle: symbolic methods address it by representing state sets as formulas.

Exercises

Exercise 1 (SMV semantics). For the SMV module:

```
VAR x : boolean; y : boolean;
ASSIGN
  init(x) := FALSE;    init(y) := FALSE;
  next(x) := y;        next(y) := !x;
```

List all four states, identify the initial state, and draw the full Kripke structure. How many distinct behaviours (infinite paths) does this model have?

Exercise 2 (Counter with Booleans). Rewrite the modulo-4 counter (section 7.6.7) using exactly two Boolean state variables b_1, b_0. Write the SMV `ASSIGN` block. Verify using the transition semantics that the state sequence is $00, 01, 10, 11, 00, \ldots$.

Exercise 3 (Arbiter fix). Fix the arbiter of theorem 7.6.1 by adding a clause: “if in state `busy` with a request, stay `busy`.” Write the corrected `case` expression. Verify that the CTL property $\mathbf{AG}(\texttt{request} \rightarrow \mathbf{AX}(\texttt{state} = \texttt{busy}))$ now holds by tracing through the Kripke structure.

Exercise 4 (Counterexample trace). In the unfixed arbiter, NuSMV produces the counterexample:

$$(\texttt{req} = 0, s = \textsc{Rdy}) \longrightarrow (\texttt{req} = 1, s = \textsc{Rdy}) \longrightarrow (\texttt{req} = 0, s = \textsc{Rdy}).$$

Explain step by step why this trace violates the CTL property. Which of the three desiderata of model checking is illustrated here?

Exercise 5 (Implicit modelling). The following `TRANS` formula is given: `TRANS (x & !next(x)) | (!x & next(x)) | (x & next(x))`. Draw the Kripke structure over $\{x = 0, x = 1\}$. Which transitions are allowed? Is there a self-loop on $x = 0$?

Exercise 6 (Ring oscillator). Model a ring inverter with N Boolean inverter cells using SMV `module`. For $N = 3$: (a) draw the Kripke structure; (b) identify two non-trivial LTL properties that hold; (c) state whether the property “every inverter changes its value infinitely often” is expressible in CTL.

Exercise 7 (Fairness). In theorem 7.9.2, add both `JUSTICE producer.running` and `JUSTICE consumer.running`. Determine the truth value of each of the following properties under fairness:
(a) “Whenever the buffer is full, eventually a cell is freed.”
(b) “Whenever the buffer is full, eventually the buffer is completely empty.”
(c) “The buffer is full infinitely often.”
For any false property, give a fair path that violates it.

Exercise 8 (Justice vs. compassion). Construct a small example (a 3- or 4-state Kripke structure with two non-deterministic choices at some state) where:

(a) A path violates justice for a given set J but does not violate compassion for any set.
(b) A path satisfies justice but violates compassion.

Hint for (b): the set C must be enabled infinitely often but only finitely often taken.

CHAPTER 8

S1S, the Büchi Theorem, and Deterministic Büchi Automata

The previous chapter introduced model checking as the central technique for verifying reactive systems. But what, mathematically, are we allowed to *say* about an infinite execution? This chapter answers that question by studying the *Monadic Second-Order Theory of One Successor* (S1S), a fragment of arithmetic powerful enough to express every ω-regular property, yet still decidable.

We will prove the landmark **Büchi Theorem** (1960): S1S and non-deterministic Büchi automata are two different faces of the same mathematical object—ω-regular languages. As a consequence, the truth of any S1S sentence is decidable by an algorithm. We will also characterise the *limits* of deterministic Büchi automata, which turn out to be strictly weaker than their non-deterministic counterparts—a fact that motivates the whole machinery of complementation and, ultimately, the need for richer acceptance conditions (Muller automata, studied in Chapter 9).

The chapter has one narrative spine. First we learn how S1S talks about positions of an infinite execution. Then we turn formulas with free set variables into ω-words over bit-vectors. That encoding lets every logical operation become an automata operation: Boolean connectives become closure constructions, and existential set quantification becomes projection. The Büchi theorem is the point where these translations meet. The final section on deterministic Büchi automata is a deliberate bridge: it keeps the theorem in view while explaining why the next chapter must enrich deterministic acceptance conditions rather than merely repeat the non-deterministic story.

Where we are. Chapter 7 introduced infinite executions as the behaviours of reactive systems. Earlier, chapter 3 gave us Büchi automata as operational recognisers for such behaviours.

What this chapter adds. S1S gives a logical language for talking directly about positions and sets of positions along an infinite execution. The Büchi theorem proves that this logical view and the automata view define the same ω-regular behaviours.

Where it leads. The proof explains why temporal logics can be compiled to automata. The deterministic Büchi limitation at the end also sets up the deterministic acceptance conditions of the next chapter.

Chapter map.

- Sections 8.1 to 8.3 introduce the time-flow view, the syntax of S1S, and representative specifications.
- Section 8.4 turns formulas with free variables into bit-vector words.
- Sections 8.5 to 8.8 prove both directions of the Büchi correspondence.
- Sections 8.9 and 8.10 extract decidability and mark the boundary of decidable extensions.
- Sections 8.11 and 8.12 explain why deterministic Büchi automata are too weak and why richer conditions are needed.

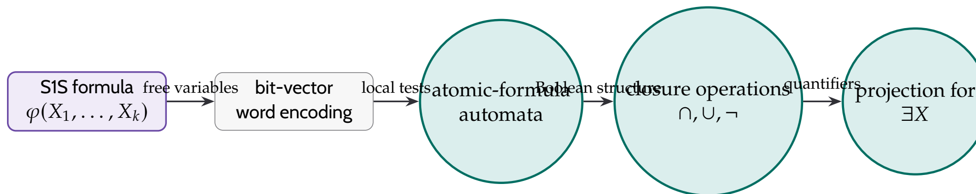


Figure 8.1: The automata-theoretic reading of S1S. Formulas are interpreted as languages of bit-vector words; logical connectives become automata closure operations, and existential second-order quantification becomes projection.

8.1 The Time-Flow Perspective

A reactive system produces an infinite sequence of states $s_0, s_1, s_2, \ldots$. The indices $0, 1, 2, \ldots$ are just the *natural numbers* $\mathbb{N}$: the set of positions of an ω-word. Every property we want to assert about the system is therefore a property of $\mathbb{N}$ equipped with its successor function.

This is not an accident. Historically, Büchi introduced non-deterministic automata over infinite words precisely in order to prove that a certain fragment of arithmetic—a logic about $\mathbb{N}$—is *decidable*. The resulting theory has since become the theoretical foundation of virtually every temporal logic used in model checking.

Preview. The two logics we will study in the next chapters—LTL and CTL—are *not* S1S. They are strictly less expressive, on purpose. LTL corresponds precisely to the *first-order* fragment of S1S (without second-order quantification). Sacrificing second-order quantification reduces decidability from *non-elementary* complexity to *PSPACE-complete* for LTL and *polynomial* for CTL—a trade-off that makes model checking tractable in practice. Understanding S1S is therefore not merely academic: it is the bedrock on which the whole temporal-logic edifice is built.

8.2 The Logic S1S

8.2.1 Simple example: every a is followed by a b

Example 8.2.1. Let $A = \{a, b, c\}$. Define the language:

$$L_1 = \{\alpha \in A^\omega \mid \text{every occurrence of } a \text{ is followed by some } b\}.$$

An $\mathrm{S1S}_A$ sentence for L_1:

$$\forall x.\ (x \in Q_a) \to (\exists y.\ x < y \wedge y \in Q_b).$$

Reading in natural language: "for every position x, if a holds at x, then there exists a future position $y > x$ where b holds."

Note how naturally the logic captures the English description: every universal quantifier over positions $\forall x$ corresponds to "for every time point", the predicate Q_a says "the symbol at that position is a", and $\exists y.\ x < y$ means "there is a future time point y".

The time-flow perspective gives us the intended structure: $0, 1, 2, \ldots$ with a successor operation. S1S is the formal language for talking about that structure. First-order variables name individual time positions; second-order variables name sets of positions, such as "all times where a request is pending".

8.2.2 $\mathrm{S1S}_A$: S1S over an alphabet

Let A be a finite alphabet (a finite set of symbols). We first define $S1S_A$, a version of the logic tailored to reason about ω-words over A.

Terms. Terms are built from:

- the constant 0 (zero);
- *first-order variables* $x, y, z, \ldots$ (ranging over natural numbers, i.e., positions in a word); and
- the *successor function* $+1$, applied to any term.

So 0, x, $x + 1$, $y + 1 + 1$, etc. are all terms. There is no subtraction and no multiplication—these are the key restrictions that keep the logic decidable.

Atomic formulas. An atomic formula has one of four forms:

1. $t_1 = t_2$ (equality between terms)
2. $t_1 < t_2$ (ordering: term t_1 is less than term t_2)
3. $t \in X$ (set membership: position t belongs to the *second-order variable* X, which ranges over *sets* of natural numbers)
4. $t \in Q_a$ for each symbol $a \in A$ (the position t is labelled by a in the word being described)

The predicates Q_a are the interface between the logic and the words: they "read" the word at a given position.

Formulas. Full formulas are built from atomic formulas by:

- Boolean connectives: $\neg, \wedge, \vee, \to, \leftrightarrow$;
- *first-order* quantifiers: $\exists x, \forall x$ (over positions);
- *second-order* quantifiers: $\exists X, \forall X$ (over sets of positions).

First-order variables are written as lowercase x, y, z; second-order variables as uppercase X, Y, Z.

Definition 8.2.2 (S1S$_A$ sentence)**.** A *sentence* of S1S$_A$ is a formula with no free variables. A sentence has no parameters: it is either *true* or *false* over a given structure, with no room for ambiguity about the value of any variable.

8.2.3 Semantics: words as structures

The key insight is to interpret an ω-word $\alpha \in A^\omega$ as a mathematical *structure*.

Definition 8.2.3 (Word structure)**.** Let $\alpha \in A^\omega$. The *word structure* $\overline{\alpha}$ is the tuple

$$\overline{\alpha} = \big(\mathbb{N},\ 0,\ +1,\ <,\ \{Q_a\}_{a\in A}\big)$$

where:

- the domain is $\mathbb{N} = \{0, 1, 2, \ldots\}$;
- 0, $+1$, and $<$ have their standard arithmetic meanings;
- $Q_a = \{i \in \mathbb{N} \mid \alpha(i) = a\}$ is the set of positions at which symbol a appears in α.

Example 8.2.4 (Q_a for a periodic word)**.** Let $A = \{a, b\}$ and $\alpha = (ab)^\omega = ababab\cdots$. Then:

$$Q_a = \{0, 2, 4, 6, \ldots\} = \text{even positions}, \qquad Q_b = \{1, 3, 5, 7, \ldots\} = \text{odd positions}.$$

Definition 8.2.5 (Language of a formula)**.** The *language* of an S1S$_A$ sentence Φ, written $L(\Phi)$, is the set of all ω-words α such that $\overline{\alpha} \models \Phi$.

This is exactly the notion of language we know from Büchi automata—a set of ω-words—but now defined logically.

8.3 What S1S Can Express

Before proving the Büchi theorem, let us develop an intuition for the expressive power of S1S$_A$ through four examples of increasing difficulty. These patterns appear constantly in exam problems. They also foreshadow the proof: every example below isolates a logical move that later becomes an automata construction.

8.3.1 Defining $x < y$ using only $+1$

The structure $\overline{\alpha}$ includes the ordering $<$, but it is instructive to see that $<$ can actually be *defined* using only the successor function $+1$ and second-order variables—which shows the power of the monadic quantification.

Example 8.3.1 (Defining the ordering from the successor)**.** We want to write a formula $\varphi(x, y)$ with two free variables that holds if and only if $x < y$, using no occurrence of $<$.

Key idea. The set $\{x+1, x+2, x+3, \ldots\}$ is the *smallest* set that (a) contains $x+1$, and (b) is closed under $+1$. If y is in this set, then $y > x$.

Wrong first attempt (using $\exists X$):

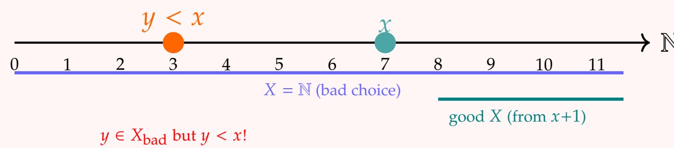
$$\exists X.\ \big[(x+1) \in X \ \wedge\ \forall z.\ z \in X \to (z+1) \in X \ \wedge\ y \in X\big].$$

This looks right, but is *wrong*. The problem: if $y < x$, we can still choose $X = \mathbb{N}$, which satisfies all the conditions (it contains $x + 1$, is closed under $+1$, and contains y). The existential is too weak—it lets us pick any set, including ones that are "too big".

Correcting the Existential Attempt: We *can* make the existential definition work by adding a "breakpoint". We need to force the set X to start strictly after x. We do this by simply forbidding $x \in X$:

$$\exists X.\ \big[x \notin X \ \wedge\ (x+1) \in X \ \wedge\ \forall z.\ z \in X \to (z+1) \in X \ \wedge\ y \in X\big].$$

This prevents the choice of $X = \mathbb{N}$ and forces X to be exactly $\{x+1, x+2, \ldots\}$ (or a subset of it, but closure under $+1$ makes it exactly that set).



With $\exists X$, one can always choose $X = \mathbb{N}$ (blue), which contains both $x + 1$ and y even when $y < x$.

Correct formula (using $\forall X$):

$$\varphi(x, y) \ \equiv\ \forall X.\ \big[(x+1) \in X \ \wedge\ \forall z.\ z \in X \to (z+1) \in X\big] \to y \in X.$$

Reading: "for *every* set X that contains $x + 1$ and is closed under $+1$, y must belong to X." Since the smallest such X is $\{x + 1, x + 2, \ldots\}$, and this set does *not* contain any $y \le x$, the formula holds if and only if $y > x$.

Why does the wrong case fail now? If $y < x$, take $X = \{x + 1, x + 2, \ldots\}$ (the *minimal* candidate). This set satisfies the hypothesis (contains $x + 1$ and is closed under $+1$), but does *not* contain y. So the implication is false, and $\varphi(x, y)$ is false. Correctness achieved.

Remark 8.3.2 (Second-order universals are essential here). The ordering $<$ is *not* definable from $+1$ in the *first-order* fragment of S1S (without second-order quantifiers). The inductive construction above fundamentally relies on the ability to quantify over the *set* of all successors, which is a second-order concept. This separation will reappear when we compare LTL to S1S.

However, the converse is true: we *can* define the successor $+1$ from the ordering $<$ in purely first-order logic. The successor of x is the strictly greater element y such that there is no other element z strictly between them:

$$y = x + 1 \iff x < y \ \wedge\ \neg\exists z.\ (x < z \wedge z < y).$$

8.3.2 Parity: positions of A in even locations

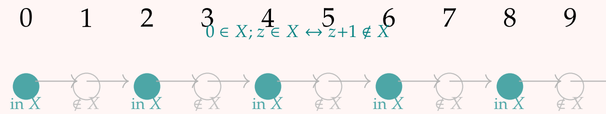
Example 8.3.3 (Even-position predicate)**.** Let $A = \{a, b\}$. The language:

$$L_2 = \{\alpha \in A^\omega \mid \exists \text{ an even position } i \text{ with } \alpha(i) = a\}.$$

First, we define a formula Even(y) asserting that y is even. The idea: the set of even numbers is the *unique* set X satisfying:

- Base case: $0 \in X$.
- Alternating step: $\forall z.\ z \in X \leftrightarrow (z+1) \notin X$.

Note that the $\leftrightarrow$ (if-and-only-if) forces a strict alternation: for every z that is in X, its successor is *not* in X, and vice versa. This produces exactly the set $\{0, 2, 4, 6, \ldots\}$.



Alternating definition of parity: starting from $0 \in X$, every even position is inside and every odd one outside.

Because the set is uniquely determined by these constraints, existential and universal quantification collapse: "there exists an X with these properties" and "for all X with these properties" give the same answer. The formula for L_2:

$$\exists y.\ (y \in Q_a) \wedge$$
$$\exists X.\ \big[0 \in X\ \wedge\ \forall z.\ (z \in X \leftrightarrow (z+1) \notin X)\big]\ \wedge\ y \in X.$$

Remark 8.3.4 (Parity is not first-order). The parity property is a canonical example of a property that is *not* expressible in the first-order fragment of S1S. Its formula makes essential use of a second-order variable X to simulate induction. This is directly connected to the fact that LTL cannot express "a occurs at an even position"—temporal operators correspond to first-order quantifiers, which are not powerful enough for parity.

8.3.3 Counting between consecutive occurrences

Example 8.3.5 (Even number of symbols between consecutive A's)**.** Let $A = \{a, b, c\}$. Define:

$$L_3 = \{\alpha \in A^\omega \mid \text{between every pair of consecutive } a\text{'s, there is an even number of } b\text{'s or } c\text{'s}\}.$$

(This is the same language for which we built a Büchi automaton in Part I.) The S1S sentence proceeds in two parts:

Part 1: "x and y are consecutive a's." Positions x and y are consecutive occurrences of a if: $x \in Q_a$, $y \in Q_a$, $x < y$, and there is no z with $x < z < y$ and $z \in Q_a$:

$$\text{consec}(x, y)\ \equiv\ x \in Q_a\ \wedge\ y \in Q_a\ \wedge\ x < y\ \wedge\ \neg\exists z.\ (x < z \wedge z < y \wedge z \in Q_a).$$

Part 2: "x and y have the same parity." Using an alternating set X starting at x (analogous to the parity example, but rooted at x):

$$\mathrm{sameParity}(x, y) \;\equiv\; \exists X.\, \big[x \in X \;\wedge\; \forall z.\, (z \in X \leftrightarrow (z+1) \notin X)\big] \;\wedge\; y \in X.$$

If $y = x + 2k$ (even gap), they have the same parity; if $y = x + 2k + 1$ (odd gap), different parity. So "even number of symbols in (x, y)" means $y - x$ is even, which is exactly $\mathrm{sameParity}(x, y)$.

The full S1S sentence:

$$\forall x, y.\, \mathrm{consec}(x, y) \to \mathrm{sameParity}(x, y).$$

8.3.4 The key pattern: infinitely many occurrences

Example 8.3.6 (Infinitely many a's)**.** The language $L_\infty = \{\alpha \in A^\omega \mid \alpha \text{ contains infinitely many } a\text{'s}\}$ is expressed by:

$$\Phi_\infty \;\equiv\; \forall x.\, \exists y.\, x < y \;\wedge\; y \in Q_a.$$

Reading: "for every time point x, there is a future time point y where a holds." This forces a to appear arbitrarily far in the future—equivalently, infinitely often.

Why not $\forall x.\, (x \in Q_a) \to \exists y.\, x < y \wedge y \in Q_a$? That formula has a trivially true antecedent: if there are *no* a's at all, then $x \in Q_a$ is always false and the implication holds vacuously. So the word b^ω (no a's anywhere) would satisfy that formula—which is wrong. The correct Φ_∞ has no conditional: it asserts that *regardless* of what symbol is at x, a future a must exist.

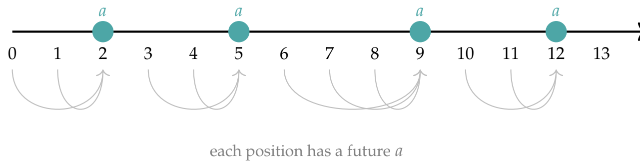


Figure 8.2: The $\forall x\, \exists y$ pattern for "infinitely many a's". Every position (including non-a ones) has a future occurrence of a to witness the existential. This pattern is the S1S counterpart of the LTL operator $\mathbf{GF}\, a$ ("globally eventually a").

Remark 8.3.7 ($\forall x\, \exists y = \mathbf{GF}$ in LTL). The pattern $\forall x.\, \exists y.\, x < y \wedge P(y)$ is one of the most recurring in S1S, model checking, and temporal logic. In LTL it corresponds to $\mathbf{G}(\mathbf{F}\, P)$ ("P holds infinitely often"). In fairness constraints, the same infinitely-often pattern appears in NuSMV `JUSTICE` clauses (Chapter 7).

8.4 Free Variables and the Encoding $\{0, 1\}^n$

So far we have treated $\mathrm{S1S}_A$ for sentences (no free variables). For the Büchi theorem proof, we need to handle *open formulas* with n free *second-order* variables $X_1, \ldots, X_n$. This is not just a technical detail: it is the mechanism that lets logic

and automata communicate. An automaton reads a word, so a valuation for the free variables must itself be presented as a word.

8.4.1 Why free variables complicate things

A sentence Φ is true or false over $\overline{\alpha}$—there are no parameters. But a formula $\Psi(X_1, X_2)$ with two free SO variables needs an *interpretation* of X_1 and X_2 (as sets of naturals) before it can be evaluated. A single ω-word does not carry this information.

8.4.2 Words as value assignments

The elegant solution: *encode* the value of all free variables inside the word itself. Instead of words over A, use words over $\{0,1\}^n$, where each position carries an n-bit tuple that specifies, for each variable X_k, whether that position belongs to X_k.

Definition 8.4.1 (Free-variable encoding)**.** Given a formula $\Phi(X_1, \ldots, X_n)$ with n free SO variables and an ω-word $\alpha \in (\{0,1\}^n)^\omega$, define:

$$P_k = \{i \in \mathbb{N} \mid \alpha(i)_k = 1\}$$

where $\alpha(i)_k$ is the k-th bit of the n-tuple at position i. The set P_k is the *interpretation* assigned to X_k by the word α.

Example 8.4.2 (Visualising the encoding)**.** Suppose $n = 2$ (variables X_1, X_2) and the word is:

$$\alpha = \underbrace{\begin{pmatrix}1\\1\end{pmatrix}}_{\alpha(0)} \underbrace{\begin{pmatrix}0\\1\end{pmatrix}}_{\alpha(1)} \underbrace{\begin{pmatrix}1\\0\end{pmatrix}}_{\alpha(2)} \underbrace{\begin{pmatrix}0\\0\end{pmatrix}}_{\alpha(3)} \underbrace{\begin{pmatrix}1\\0\end{pmatrix}}_{\alpha(4)} \cdots$$

Reading by rows: row 1 (top) encodes X_1; row 2 (bottom) encodes X_2.

$$X_1 = P_1 = \{0, 2, 4, \ldots\}, \qquad X_2 = P_2 = \{0, 1\}.$$

So this word represents the interpretation "X_1 is the set of even numbers; X_2 is the set $\{0,1\}$".

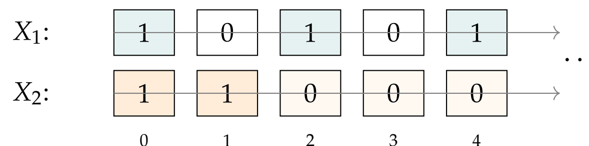


Figure 8.3: Encoding two free SO variables X_1 (top row, teal) and X_2 (bottom row, orange) as an ω-word over $\{0,1\}^2$. Each column is one position; a 1 means "this position belongs to the variable". The word uniquely determines the interpretation $X_1 = \{0, 2, 4, \ldots\}$ and $X_2 = \{0, 1\}$.

This encoding is key: a formula with n free SO variables becomes a *language* over $\{0,1\}^n$. Every closure property (union, intersection, complementation,

projection) for Büchi automata over $\{0,1\}^n$ then translates directly to a logical operation on formulas.

S1S$_A$ ≡ S1S. S1S$_A$ (with alphabet predicates Q_a) can be converted to S1S (pure arithmetic, no alphabet) by replacing each symbol $a \in A$ with a binary encoding of length $\lceil \log_2 |A| \rceil$. The two logics are expressively equivalent; we use whichever is more convenient.

8.5 The Büchi Theorem

At this point the two sides have the same shape. An S1S sentence defines a set of infinite words; a Büchi automaton recognises a set of infinite words. The theorem says that these two ways of describing sets are extensionally identical.

We can now state the central result.

> **Theorem 8.5.1** (Büchi, 1960)**.** *An ω-language $L \subseteq A^\omega$ is* ω-regular *(recognised by some non-deterministic Büchi automaton) if and only if it is* S1S-definable *(equal to $L(\Phi)$ for some S1S$_A$ sentence Φ).*

Both directions are constructive: the proofs give explicit algorithms for translating between automata and formulas. Before the proof, note what this theorem means:

- **Declarative ↔ Operational.** S1S is a *declarative* language (describe what you want); Büchi automata are *operational* (describe how to process a word step by step). The theorem says they define exactly the same class of languages.
- **Succintness.** S1S can be exponentially more succinct than Büchi automata: there exist S1S sentences of size n whose smallest equivalent Büchi automaton has exponentially many states.
- **Decidability.** The theorem implies that *all* Büchi automaton operations (emptiness, inclusion, etc.) have logical counterparts, and vice versa.

8.6 Proof: ω-regular ⟹ S1S-definable

This is the easier direction. An accepting run is already an infinite object indexed by time; S1S only has to guess that run using sets of positions and then check that the guessed run starts correctly, follows the transition relation, and satisfies the Büchi condition.

Given: NBA $\mathcal{A} = (Q, A, \delta, q_0, F)$ with states $Q = \{0, 1, \ldots, m\}$ (we name the initial state 0).

Goal: Write an S1S sentence $\Phi_\mathcal{A}$ such that $L(\Phi_\mathcal{A}) = L(\mathcal{A})$.

8.6.1 The main idea: encode a run as a family of sets

A run of $\mathcal{A}$ over a word α is an infinite sequence of states $\sigma = q_{i_0} q_{i_1} q_{i_2} \cdots$. We encode this run using $m+1$ second-order variables $Y_0, \ldots, Y_m$, one per state:

$$Y_i = \{ j \in \mathbb{N} \mid \sigma(j) = i \} = \text{“the set of positions where the run is in state } i\text{”}.$$

If the run visits state 3 at positions $0, 4, 17, \ldots$, then $Y_3 = \{0, 4, 17, \ldots\}$. The run is fully determined by the partition $Y_0, \ldots, Y_m$ of $\mathbb{N}$.

Non-determinism is encoded by the *choice* of these sets: different choices of $Y_0, \ldots, Y_m$ correspond to different runs of the (non-deterministic) automaton.

8.6.2 The three conditions

We write $\Phi_{\mathcal{A}} \equiv \exists Y_0 \cdots \exists Y_m.\ \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where:

φ_1: Initiality. The run starts in state $q_0 = 0$ and the sets are pairwise disjoint (the automaton is in exactly one state at each position):

$$\varphi_1 \equiv (0 \in Y_0) \wedge \bigwedge_{i \neq j} \neg\exists x.\ (x \in Y_i \wedge x \in Y_j).$$

φ_2: Consequentiality. The run follows the transition relation: at each position x, if the automaton is in state i and reads symbol a, it moves to state j:

$$\varphi_2 \equiv \forall x.\ \bigvee_{(i,a,j)\in\delta} \big(x \in Y_i \wedge x \in Q_a \wedge (x+1) \in Y_j\big).$$

Every position must be covered by exactly one transition.

φ_3: Acceptance. At least one final state is visited infinitely often (using the $\forall x\, \exists y$ pattern from Example 8.3.6):

$$\varphi_3 \equiv \bigvee_{i\in F} \forall x.\ \exists y.\ x < y \wedge y \in Y_i.$$

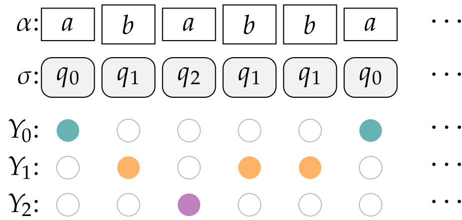


Figure 8.4: Encoding a run $\sigma = q_0q_1q_2q_1q_1q_0\cdots$ of a 3-state automaton as sets Y_0 (teal), Y_1 (orange), Y_2 (violet). At each position exactly one set contains a filled dot. This encoding converts the run into an interpretation of the S1S second-order variables.

> **Theorem 8.6.1** (Correctness). *$\overline{\alpha} \models \Phi_{\mathcal{A}}$ if and only if $\alpha \in L(\mathcal{A})$.*

Proof sketch. ($\Rightarrow$) If $\overline{\alpha} \models \Phi_{\mathcal{A}}$, then the existential witnesses $Y_0, \ldots, Y_m$ define a run of $\mathcal{A}$ over α (φ_1 gives initiality, φ_2 gives valid transitions, φ_3 gives acceptance). Hence $\alpha \in L(\mathcal{A})$. ($\Leftarrow$) If $\alpha \in L(\mathcal{A})$, there exists an accepting run σ. Setting $Y_i = \{j \mid \sigma(j) = i\}$ satisfies all three conditions. □

8.7 Closure under Projection

Before proving the other direction of the Büchi theorem, we need one more closure property of Büchi automata: *closure under projection*. This corresponds, logically, to *existential second-order quantification*. This is the central bridge from syntax to automata: forgetting one bit of the input word is the automata-theoretic version of saying "there exists a set of positions with this property".

Lemma 8.7.1 (Closure under projection). *Let $\mathcal{A}$ be an NBA over $\{0,1\}^n$ recognising $L(\Phi)$, where Φ has n free SO variables. Let $\Psi \equiv \exists X_i.\, \Phi$ (with $n-1$ free variables). Then there exists an NBA $\mathcal{A}'$ over $\{0,1\}^{n-1}$ recognising $L(\Psi)$.*

Construction. $\mathcal{A}'$ is obtained from $\mathcal{A}$ by *removing the i-th bit from every transition label*. If $\mathcal{A}$ has a transition $q \xrightarrow{(b_1,\ldots,b_n)} q'$, then $\mathcal{A}'$ has a transition $q \xrightarrow{(b_1,\ldots,b_{i-1},b_{i+1},\ldots,b_n)} q'$. □

Intuition. Existential quantification over X_i means: "there *exists* a value for the i-th row of the word such that Φ holds." By removing the i-th bit from the alphabet, we allow the automaton to process words of length $n-1$ and non-deterministically "guess" what the missing i-th bit could have been.

Example 8.7.2 (Projection reduces the alphabet). An NBA $\mathcal{A}$ over $\{0,1\}^2 = \{00, 01, 10, 11\}$ with transitions:

$$q_0 \xrightarrow{00} q_1, \quad q_1 \xrightarrow{11} q_1, \quad q_1 \xrightarrow{10} q_2, \quad q_2 \xrightarrow{11} q_2$$

After projecting onto the first component (removing the second bit):

$$q_0 \xrightarrow{0} q_1, \quad q_1 \xrightarrow{1} q_1, \quad q_1 \xrightarrow{1} q_2, \quad q_2 \xrightarrow{1} q_2.$$

The resulting $\mathcal{A}'$ over $\{0,1\}$ is *non-deterministic* (two transitions from q_1 on symbol 1), because the second component is now "free" to take any value.

8.8 Proof: S1S-definable $\Rightarrow$ ω-regular

This is the more involved direction: given an S1S formula, construct an NBA. The strategy is a two-step normalisation. First we reduce the logic to a tiny kernel with only set variables and two atomic predicates. Then we show that this kernel has base automata and is closed under the operations used to build formulas.

8.8.1 Step 1: S1S $\longrightarrow$ S1S$_0$ (syntactic simplification)

S1S$_0$ is an extremely restricted fragment:

- **Only second-order variables** (no first-order variables).
- **Only two atomic predicates**: $X \subseteq Y$ (subset relation) and $\mathrm{Succ}(X,Y)$ ("$X = \{x\}, Y = \{x+1\}$ are consecutive singletons").
- Boolean connectives and second-order quantifiers as usual.

Despite its extreme simplicity, S1S$_0$ is expressively equivalent to S1S. The translation proceeds through four rewriting steps:

Step 1a: Isolate uses of $+1$. Rewrite every atomic formula so that $+1$ appears only in an equality: "$x+1=y$". For example, $x+1 \in X$ becomes $\exists y.\,(y = x+1) \wedge (y \in X)$. Nested successors introduce extra auxiliary variables: $x+1+1 \in X$ becomes $\exists y.\,\exists z.\; y = x+1 \wedge z = y+1 \wedge z \in X$.

Step 1b: Remove the constant 0 and the relation <. Using the definitions from Section 8.3.1: 0 is the unique position x with no predecessor; $x < y$ is defined by the $\forall X$ formula from Example 8.3.1.

Step 1c: Eliminate first-order variables. Replace every first-order variable x with a second-order singleton variable X (a set containing exactly one element):

$$\begin{aligned} \exists x.\ \varphi(x) &\quad\mapsto\quad \exists X.\ \text{Singleton}(X) \wedge \varphi(X), \\ \forall x.\ \varphi(x) &\quad\mapsto\quad \forall X.\ \text{Singleton}(X) \rightarrow \varphi(X). \end{aligned}$$

Here Singleton(X) asserts X has exactly one element. While we could define it using Succ(X, Y), we can construct it more cleanly using *only* the subset relation $\subseteq$ by viewing sets as a lattice:

$$\text{Singleton}(X) \equiv \exists Y \subseteq X.\ \big(Y \neq X \ \wedge\ \neg\exists Z.\ (Y \subseteq Z \wedge Z \subseteq X \wedge Z \neq Y \wedge Z \neq X)\big).$$

Because Y is a strict subset of X with no sets strictly between them, X can only have exactly one more element than Y. Furthermore, by forcing Y to be the empty set (which we can define as $Y \subseteq W$ for all W), this completely constrains X to have exactly one element.

Walkthrough: Reducing a formula to S1S$_0$ Let's see this process in action at the whiteboard. Suppose we start with the simple property: "the successor of position x is labelled a", which is written as $x + 1 \in Q_a$.

1. **Isolate** $+1$: We extract the successor into an equality. $\exists y.\ (y = x+1 \wedge y \in Q_a)$.
2. **Eliminate FO variables**: We replace the positions x and y with singleton sets X and Y. $\exists X.\ \text{Singleton}(X) \wedge \exists Y.\ \text{Singleton}(Y) \wedge \ldots$
3. **Rewrite atomics**: The equality $y = x + 1$ becomes Succ(X, Y). The membership $y \in Q_a$ means the set Y is a subset of the set of positions labelled a, which is just $Y \subseteq Q_a$.

The final S1S$_0$ formula is:

$$\exists X.\ \exists Y.\ \text{Singleton}(X) \wedge \text{Singleton}(Y) \wedge \text{Succ}(X, Y) \wedge Y \subseteq Q_a.$$

This uses only second-order variables and the base predicates.

Step 1d: Rewrite remaining atomic formulas. After the above steps, every atomic formula has one of these forms, which are directly expressible in S1S$_0$:

- $X = Y \ \equiv\ X \subseteq Y \wedge Y \subseteq X$
- $X \neq Y \ \equiv\ \neg(X = Y)$
- $x + 1 = y \ \equiv\ \text{Succ}(\{x\}, \{y\})$ (using the singleton encoding)
- $x \in X \ \equiv\ \{x\} \subseteq X$

8.8.2 Step 2: S1S$_0$ → NBA (base cases and closure)

With S1S$_0$ in hand, we build NBAs recursively on the formula structure:

- **Boolean operators**: closure under union ($\vee$), intersection ($\wedge$), complementation ($\neg$).
- **Existential quantifier** $\exists X_i$: closure under projection (Lemma 8.7.1).

It remains to give NBAs for the two atomic predicates of S1S$_0$.

Automaton for $X \subseteq Y$. This formula has two free SO variables, so the alphabet is $\{0,1\}^2$. An ω-word over $\{0,1\}^2$ encodes X (first bit) and Y (second bit). $X \subseteq Y$ holds iff every position where the first bit is 1 also has the second bit equal to 1—i.e., the symbol $(1,0)$ never occurs.

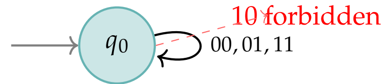


Figure 8.5: NBA for $X \subseteq Y$ over $\{0,1\}^2$. A single accepting state loops on all symbols except $(1,0)$ (which would mean "position in X but not in Y", violating $X \subseteq Y$). Words containing $(1,0)$ have no accepting run.

Automaton for $\mathrm{Succ}(X,Y)$. This asserts: $X = \{x\}, Y = \{x+1\}$ for some x. Three states suffice:

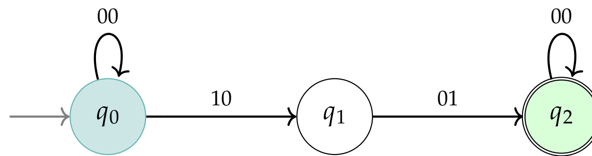


Figure 8.6: NBA for $\mathrm{Succ}(X,Y)$ over $\{0,1\}^2$. State q_0 waits for the single 1 in X (symbol 10: position in X, not yet in Y). State q_1 sees the single 1 in Y (symbol 01: position in Y, immediately after X's element). State q_2 (accepting) loops on 00 forever. Any other symbol causes a stuck run (no transition), rejecting the word.

With these two base automata and the closure properties, the recursive construction of an NBA for any $\mathrm{S1S}_0$ formula—and hence any S1S formula—is complete. This finishes the proof of Theorem 8.5.1.

8.9 Decidability of S1S

Theorem 8.9.1 (Decidability of S1S)**.** *Satisfiability and validity of S1S sentences are decidable. Equivalently, given an $\mathrm{S1S}_A$ sentence Φ, there are algorithms to decide whether $L(\Phi)$ is empty and whether $L(\Phi) = A^\omega$.*

Algorithm. Given an $\mathrm{S1S}_A$ sentence Φ:

1. Translate Φ to an NBA $\mathcal{A}$ using the construction of Section 8.8, so that $L(\mathcal{A}) = L(\Phi)$.
2. Decide *emptiness* of $\mathcal{A}$: check whether $L(\mathcal{A})$ is empty. Emptiness of an NBA is solvable in NLogSpace (polynomial time) by looking for a reachable accepting lasso. This decides satisfiability of Φ.
3. To decide validity, apply the same construction to $\neg\Phi$ and check emptiness. The sentence Φ is valid over all ω-words iff $L(\neg\Phi) = \emptyset$.

□

Pure S1S as a special case. If there are no alphabet predicates Q_a, the structure is just $(\mathbb{N}, 0, +1, <)$. There is then only one vacuous input word over $\{0,1\}^0 = \{\varepsilon\}$,

so satisfiability, truth in the pure structure, and non-emptiness of the translated automaton collapse to the same check.

8.9.1 Complexity: the non-elementary lower bound

The algorithm described above is correct but has enormous complexity.

> **Definition 8.9.2** (Tower function). The *tower function* $\exp(h, n)$ is defined by:
>
> $$\exp(0, n) = n, \qquad \exp(h, n) = 2^{\exp(h-1,n)}.$$
>
> So $\exp(1, n) = 2^n$, $\exp(2, n) = 2^{2^n}$, $\exp(3, n) = 2^{2^{2^n}}$, etc.

> **Theorem 8.9.3** (Stockmeyer). *The S1S decision problem requires time* $\Omega(\exp(h, n))$ *where n is the size of the formula and h is the number of* alternations *between* $\exists$ *and* $\forall$ *quantifiers. In general, h is unbounded, so no fixed-height tower suffices: the complexity is* non-elementary.

Why? Each universal second-order quantifier $\forall X$ requires a complementation of the current NBA. Büchi complementation introduces an exponential blow-up (from the Safra construction or similar). With k alternations, we apply complementation k times, giving a tower of height k. Since k can be arbitrarily large (there is no bound on how many alternations an S1S formula can have), the tower is arbitrarily high.

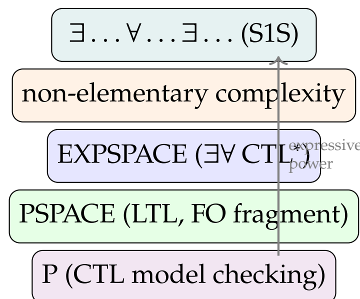


Figure 8.7: Complexity vs. expressive power in the logic hierarchy. Full S1S is non-elementary. Dropping second-order quantifiers gives the FO fragment ≡ LTL ≡ PSPACE. Dropping to CTL (a branching-time restriction) gives polynomial complexity. Every step up the ladder gains expressivity at an exponential cost.

8.10 Extensions of S1S

Over the decades, many extensions of S1S have been studied, some remaining decidable and some falling into undecidability. This boundary is pedagogically important: S1S is powerful because it can quantify over sets of time positions, but its decidability depends on very specific arithmetic restrictions.

8.10.1 S1S over finite words

Theorem 8.10.1 (Büchi, finite-word version)**.** *A language $L \subseteq A^*$ is* regular *(recognised by some NFA) if and only if it is* definable in S1S over finite domains*.*

S1S over finite domains is defined like S1S, but:

- The domain is $\{0, 1, \ldots, k\}$ for some k (the length of the word minus one), not $\mathbb{N}$.
- Second-order variables are interpreted as *finite* sets.
- The successor of the last element k is k itself (a self-loop).

This gives yet another characterisation of regular languages, alongside DFA, NFA, and regular expressions.

8.10.2 WS1S: Weak Monadic Second-Order Theory

Definition 8.10.2 (WS1S)**.** *WS1S* (Weak S1S) is obtained from S1S by requiring that every second-order variable is interpreted as a *finite* set.

At first glance this seems much weaker: many of the sets we defined (all even numbers, all successors of a position, etc.) are infinite and thus unavailable. The surprising result is:

Theorem 8.10.3. *WS1S and S1S are* expressively equivalent*: every ω-language definable in WS1S is also definable in S1S, and vice versa.*

The non-obvious direction (S1S $\subseteq$ WS1S) is proved via a detour through *deterministic Muller automata*. The intuition: although each set in WS1S is strictly finite, we can simulate infinity by considering a sequence of finite sets of *always increasing size*. The "limit" of these growing finite sets mathematically captures the infinite behavior without ever needing a single infinite set as an object.

The fundamental operation. The reference operation for Büchi automata is the *ω-power*: a finite word u followed by infinitely many copies of some suffix v (uv^ω). For Muller automata (which correspond to WS1S), the reference operation is the *vectorial closure*: infinitely many finite prefixes $v_1 v_2 v_3 \cdots$, each belonging to some set V. This structural difference reflects the finite-set restriction.

8.10.3 Decidable extensions

Both of the following extensions add a new predicate to S1S:

S1S + $\mathbf{P_k}$ (powers of k): Add a unary predicate P_k where $P_k(i)$ holds iff $i = k^j$ for some $j \geq 0$. For example, $P_2 = \{1, 2, 4, 8, 16, \ldots\}$. The resulting theory is decidable.

S1S + Factorial**:** Add a predicate that holds at positions $1, 2, 6, 24, 120, \ldots$ ($n!$ for $n \geq 0$). Still decidable.

8.10.4 Undecidable extensions

S1S + ×2 (doubling function): Add the function $i \mapsto 2i$ (multiply by two). Decidability is lost immediately. Multiplying by 2 gives enough power to simulate arbitrary multiplication, which leads to Peano arithmetic.

Peanoarithmetic ($\mathbb{N}$ with + and ×): Highly undecidable (Gödel incompleteness theorems).

The boundary is surprisingly delicate: adding "multiply by 2" destroys decidability, yet "powers of k" and "factorial" do not.

We now turn from logical expressiveness to a closely related operational question. The Büchi theorem says that non-deterministic Büchi automata capture exactly the S1S-definable languages, but model checking and synthesis often prefer deterministic monitors, because deterministic automata are easier to complement and to run inside games. The next section is therefore not a second full theory of deterministic ω-automata. It is the motivating obstruction that Chapter 9 will resolve with Muller, Rabin, and parity acceptance.

8.11 Deterministic Büchi Automata: Limitations

The goal here is intentionally narrow. We keep the Büchi acceptance condition and ask whether determinism alone is harmless. Over finite words the answer would be yes: every NFA can be determinised. Over infinite words the answer is no, and the failure already appears in the small language of words with finitely many a's. This witness prepares the more systematic deterministic theory of Chapter 9.

8.11.1 Definition

> **Definition 8.11.1** (Deterministic Büchi Automaton (DBA))**.** A *DBA* is a tuple $\mathcal{A} = (Q, A, \delta, q_0, F)$ where all components are as in an NBA except that $\delta : Q \times A \to Q$ is a *function* (not a relation). Every state-symbol pair has exactly one successor.

Operational meaning. At runtime, a DBA has no choices to make. Upon reading a symbol, the transition function deterministically forces the automaton into exactly one next state. It completely loses the ability to "guess" future inputs. For a finite word, this isn't a problem, because we can always traverse the whole word and check the end. But for an infinite word, the inability to branch into multiple potential futures and "wait and see" which one becomes successful fundamentally cripples its expressive power.

A DBA has a *unique run* on every ω-word: given α, the run $\sigma = q_0, q_1, q_2, \ldots$ is fully determined by $q_{i+1} = \delta(q_i, \alpha(i))$. Acceptance: the unique run is successful iff $\text{Inf}(\sigma) \cap F \neq \emptyset$. This uniqueness removes the automaton's ability to "guess" a future event. That is exactly what will fail in the witness language below.

8.11.2 What DBAs can and cannot do

> **Proposition 8.11.2.** *DBAs are closed under* union *and* intersection*.*

(Exercise: construct the product automaton and adjust the acceptance condition.)

Theorem 8.11.3 (DBAs are not closed under complementation)**.** *There exists a language L recognised by a DBA such that $A^\omega \setminus L$ is not recognised by any DBA. In particular, DBA languages are not closed under complement and form a* strict subset *of the ω-regular languages.*

8.11.3 The witness pair: infinitely many vs. finitely many a's

Definition 8.11.4. Let $A = \{a, b\}$ and:

$$L_{\text{fin}} = \{\alpha \in A^\omega \mid \alpha \text{ contains finitely many } a\text{'s}\}.$$

Proposition 8.11.5. *L_{fin} is ω-regular.*

Proof. A key observation: α contains finitely many a's if and only if α has a *last a*—some final position after which only b's appear. The following NBA captures this:

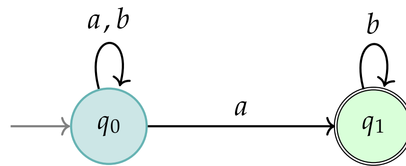


Figure 8.8: NBA for L_{fin} (finitely many a's). The automaton nondeterministically *guesses* the last occurrence of a by taking the transition $q_0 \to q_1$ and then never seeing a again.

Why is this correct? The automaton "guesses" which a is the last one. If it guesses correctly (transitions to q_1 on the last a and only sees b's thereafter), the run visits q_1 infinitely often. If it guesses wrong (transitions to q_1 too early, before the last a), the run gets stuck (no outgoing transition from q_1 on a). Since it is an NBA, the word is accepted iff at least one run is successful. □

The complement $\overline{L_{\text{fin}}}$ = "infinitely many a's" is accepted by the following DBA:

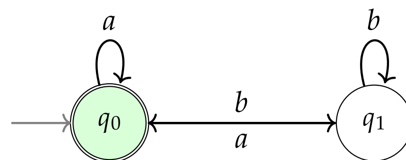


Figure 8.9: DBA for $\overline{L_{\text{fin}}}$ (infinitely many a's). The accepting state q_0 is revisited every time an a is read. The word is accepted iff q_0 is visited infinitely often, i.e., iff there are infinitely many a's.

8.11.4 Why L_{fin} has no DBA

Suppose for contradiction that some DBA $\mathcal{B}$ accepts L_{fin}. We will construct an infinite word with infinitely many a's that $\mathcal{B}$ is forced to accept.

Start with a finite prefix $u_0 = \varepsilon$. For any finite prefix u, the word ub^ω has only finitely many a's, hence belongs to L_{fin}. Since $\mathcal{B}$ is assumed to recognise L_{fin}, the unique run on ub^ω must visit accepting states infinitely often. Therefore, after the prefix u, there exists some number n such that the run reaches an accepting state after reading b^n.

Use this observation inductively. Having built u_r, choose n_r so that the run reaches an accepting state after $u_r b^{n_r}$, and set

$$u_{r+1} = u_r b^{n_r} a.$$

The limiting word

$$\beta = b^{n_0} a\, b^{n_1} a\, b^{n_2} a \cdots$$

contains infinitely many a's, so $\beta \notin L_{\text{fin}}$. But the unique run of $\mathcal{B}$ on β visits an accepting state after each block b^{n_r}, hence infinitely often. Thus $\mathcal{B}$ accepts β, a contradiction.

This proves that no DBA recognises L_{fin}.[1]

Theorem 8.11.3 now follows by taking $L = \overline{L_{\text{fin}}}$, the language "infinitely many a's": Figure 8.9 gives a DBA for L, while the argument above shows that its complement L_{fin} has no DBA.

8.11.5 Tracing the NBA for L_{fin} on $abab\cdots$

Example 8.11.6 (Non-acceptance of ab^ω)**.** The NBA of Figure 8.8 should *reject* $abab\cdots = (ab)^\omega$ (infinitely many a's). Let us verify:

Position	0	1	2	3	4	5	6	$\cdots$
Symbol	a	b	a	b	a	b	a	$\cdots$
Run (A):	q_0	q_0	q_0	q_0	q_0	$\cdots$		(not accepting)
Run (B):	q_0	q_1	*stuck*					(not accepting)

Run (A) stays in q_0 forever (never entering q_1): not accepting since $q_1 \in F$ is never visited. Run (B) enters q_1 after the first a then gets stuck (no transition from q_1 on a). Every possible run either stays in q_0 forever or gets stuck—none is accepting. So $(ab)^\omega \notin L(\text{NBA})$, as expected.

8.12 Towards More Expressive Deterministic Automata

The failure of DBAs to accept L_{fin} shows that determinism with the Büchi acceptance condition is too weak. For model checking, we *need* determinism (to complement specifications efficiently), but we also need full ω-regular expressiveness. This is the precise point at which the chapter hands off to deterministic ω-automata proper: the proof above is a warning sign, and the repair is to keep determinism while replacing the acceptance condition.

The solution is to change the acceptance condition. Two classical approaches:

Muller automata: Accept iff the set of states visited infinitely often is exactly one of a specified collection $\mathcal{F} \subseteq 2^Q$. This condition is more refined than Büchi's "at least one state in F infinitely often".

[1] A standard reference for deterministic automata on infinite words is W. Thomas, *Automata on Infinite Objects*, in *Handbook of Theoretical Computer Science*, vol. B, Elsevier, 1990.

Rabin automata: Accept iff there exists a pair (E_i, F_i) in a specified list such that F_i is visited infinitely often *and* E_i is visited only finitely often.

Both conditions, when combined with determinism, yield automata equivalent to NBAs in expressive power—which is exactly what we need. We will study these automata and their connection to WS1S in Chapter 9.

■ **Summary & Key Takeaways**

- *S1S* is the monadic second-order logic of $(\mathbb{N}, 0, +1, <)$; it reasons about positions of an infinite execution using first-order variables (positions) and second-order variables (sets of positions).
- The **Büchi theorem** (1960): an ω-language is S1S-definable iff it is ω-regular (recognised by an NBA). Both directions are constructive.
- The key encoding: free second-order variables become rows of a bit-vector word over $\{0, 1\}^n$; logical operations become automata closure operations; existential quantification becomes projection.
- S1S decidability follows from the Büchi theorem and NBA emptiness, but the complexity is *non-elementary* (a tower of exponentials whose height equals the number of quantifier alternations).
- *Deterministic* Büchi automata are strictly weaker: they cannot recognise "finitely many a's". Richer acceptance conditions (Muller, Rabin) are needed for deterministic ω-regular recognition.

Exercises

Exercise 1 (S1S formula writing). Write S1S_A sentences for the following ω-languages over $A = \{a, b\}$:

(a) α contains *no* a's.
(b) α contains *finitely many* a's. *Hint:* "there exists a last a".
(c) α contains *at least two* a's.
(d) Every b is immediately preceded by an a.

Exercise 2 (Ordering definability). Show that the formula $\varphi(x, y)$ from theorem 8.3.1 correctly defines $x < y$ by:

(a) Verifying that $\varphi(3, 7)$ is true over $\mathbb{N}$.
(b) Constructing an explicit counterexample set X that refutes the *existential* version of the formula when $y = 1$ and $x = 5$.

Exercise 3 (Büchi theorem, easy direction). Given the NBA $\mathcal{A}$ with states $Q = \{q_0, q_1\}$, initial state q_0, $F = \{q_1\}$, and transitions $q_0 \xrightarrow{a} q_1$, $q_1 \xrightarrow{b} q_0$, write the S1S_A sentence $\Phi_{\mathcal{A}}$ using the construction of section 8.6. Identify φ_1, φ_2, φ_3 explicitly.

Exercise 4 (S1S_0 translation). Translate the formula $\exists x.\ x \in Q_a \wedge (x + 1) \in Q_b$ into S1S_0 following the four steps of section 8.8. Which auxiliary SO variables do you introduce, and why?

Exercise 5 (Base automata). Draw the NBA for $X \subseteq Y$ (fig. 8.5) and verify that it accepts the word $(01)^\omega$ but rejects $(10)^\omega$. Verify the run on $00\,(10)\,11^\omega$ and determine whether it is accepted.

Exercise 6 (DBA closure). Let L_1 and L_2 be recognised by DBAs $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively.

(a) Construct a DBA for $L_1 \cup L_2$.
(b) Construct a DBA for $L_1 \cap L_2$.

(c) Explain why the same product construction does *not* work for complementation.

Exercise 7 (NBA for finitely many a's). Trace the run of the NBA in fig. 8.8 on the word ab^3ab^ω.

(a) List all possible runs.
(b) Identify which runs are accepting.
(c) Conclude whether $ab^3ab^\omega \in L_{\text{fin}}$.

Exercise 8 (Non-elementary complexity). Suppose an S1S formula has exactly 3 quantifier alternations (e.g., $\exists \dots \forall \dots \exists$) and size $n = 10$. Estimate the size of the NBA produced by the Büchi construction in the worst case (assume each complementation doubles the number of states). Compare this with n, 2^n, and $\exp(3, n)$.

CHAPTER 9

Deterministic Omega-Automata and Star-Free Languages

Chapter 8 closed on a slightly bitter note: deterministic Büchi automata (DBAs) are *strictly weaker* than their non-deterministic cousins, witnessed by the pumping-style argument showing that no DBA can recognise the language L_{fin} of words with only finitely many a's. If we want complementation to be a one-liner and reactive synthesis to be solvable by a real algorithm, that limitation must be overcome. This chapter does exactly that, in three movements.

First we revisit the DBA limitation through an *algebraic* lens, characterising DBA-recognisable languages as the *vectorial closures* of regular languages (section 9.2). The mismatch between this algebraic form and the standard ω-regular form $\bigcup_i U_i V_i^{\omega}$ tells us *why* the Büchi condition is too weak under determinism. We then repair the expressiveness gap by enriching the acceptance condition: **Deterministic Muller Automata (DMA)**, **Deterministic Rabin Automata (DRA)**, and the equivalent **parity** condition all restore the equivalence between determinism and ω-regularity (section 9.3, section 9.4). McNaughton's theorem (1966) is the landmark result, and Safra's construction (1988) makes it constructive.

Finally, we return to logic. Restricting S1S to its *first-order* fragment FO-S1S[<] strips away second-order quantification, and the corresponding class of languages is exactly the *star-free* regular languages of chapter 2 (section 9.5). The McNaughton-Papert theorem identifies star-free, first-order definable, counter-free, and aperiodic as four faces of one class. This circle of equivalences is the doorway to Linear Temporal Logic, which we will reach in chapter 11.

Determinism is essential for reactive synthesis (constructing a system from a specification by playing a game against the environment) and for trivialising language complementation.

9.1 The Quest for Determinism in Infinite Verification

In finite-state verification, we frequently transition between logical specifications, regular expressions, and automata. Over finite words, determinization is always possible via the standard subset construction of chapter 1: for every Non-deterministic Finite Automaton (NFA), we can construct an equivalent Deterministic Finite Automaton (DFA). Chapter 8 then exported the same intuition to infinite words and crashed into a wall: the language L_{fin} of words with finitely many a's is ω-regular and accepted by a 2-state NBA, but *no* DBA can recognise it.

Over infinite words, the symmetry between determinism and non-determinism is broken. As we will see, an NBA cannot always be determinised into a DBA, and section 9.2 below will tell us *algebraically* why. This is not just a mathematical curiosity; it has profound implications for two major areas of verification:

1. **Language Inclusion and Complementation**: To check whether the executions of a system M satisfy a property P, we verify that $L(M) \subseteq L(P)$, which is equivalent to checking that $L(M) \cap \overline{L(P)} = \emptyset$. If the property P is represented by a deterministic automaton, complementation is trivial (we merely swap the accepting and non-accepting states). If P is represented by an NBA, complementation requires a complex, non-elementary construction.
2. **Reactive Synthesis**: In synthesis, we seek to generate a system that reacts to an environment. The environment controls input variables x, and the system controls output variables y. The interaction is modeled as a game on a graph. To determine if the system has a winning strategy, the game solver must monitor the history of the play. A non-deterministic automaton cannot be used directly in this setting because the solver cannot "guess" the correct transition a posteriori; the tracking must be deterministic.

9.1.1 An Intuitive Example: Non-Deterministic Guessing vs. Real-Time Control

To illustrate the difference, consider a reactive system that receives requests (r) from the environment and must eventually grant them (g). The specification says: "every request must be granted."

An NBA can recognize this language by looping in an initial state, and whenever a request r is read, it non-deterministically spawns a branch to check that a grant g occurs. Since it runs offline, the NBA is allowed to look at the entire infinite execution and verify if there exists *some* run that successfully resolved all requests.

Now, imagine we want to automatically synthesize a physical controller (e.g., a hardware circuit) from this specification. The controller must produce outputs (grants) in real-time. It cannot non-deterministically "guess" whether the environment will issue another request in the future, nor can it backtrack if a guess is wrong. The controller needs to make deterministic decisions based *only* on the prefix of the execution seen so far. This is why synthesis requires compiling the specification into a deterministic automaton.

In a two-player game, the system must choose its move before *knowing the future environment inputs. It cannot resolve non-deterministic choices after seeing the entire infinite play.*

9.1.2 Why the Subset Construction Fails on Büchi Automata

To understand why determinization is not straightforward, let us try to apply the standard Rabin-Scott subset construction to a simple non-deterministic Büchi automaton.

Consider the language L_{fin_a} of infinite words over $\Sigma = \{a, b\}$ containing only finitely many a's (meaning they eventually consist of b's only). An NBA A for this language is shown in Figure 9.2 (Section 9.2.3).

Recall that in the subset construction, the states of the deterministic automaton are subsets of the states of the original automaton. The transition relation is defined by taking the union of the targets of all possible transitions:

$$\delta'_{sub}(S, c) = \bigcup_{q \in S} \delta(q, c)$$

If we apply this construction to the NBA A of Figure 9.2:

- The initial state is $S_0 = \{q_0\}$.
- On reading a from S_0, we can only transition to q_0, so the next state is $\{q_0\} = S_0$.
- On reading b from S_0, the NBA can either stay in q_0 or transition to q_1. Thus, the next state is $\{q_0, q_1\} = S_1$.
- From $S_1 = \{q_0, q_1\}$, if we read a, we transition from q_0 to q_0, but q_1 has no outgoing transition on a. Thus, the next state is $\{q_0\} = S_0$.
- From S_1, if we read b, we transition from q_0 to $\{q_0, q_1\}$ and from q_1 to $\{q_1\}$. Taking the union gives $\{q_0, q_1\} = S_1$.

This yields the deterministic transition structure shown in Figure 9.1.

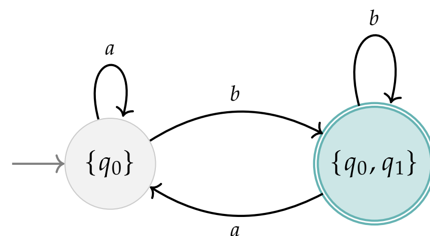


Figure 9.1: The deterministic automaton resulting from applying the subset construction to the NBA of Figure 9.2.

Now, how should we define acceptance for this deterministic automaton? In the Büchi setting, a run is accepting if it visits an accepting state infinitely often. Since the only accepting state of the original NBA is q_1, the natural choice for the DBA's accepting states is any subset containing q_1. In our case, this is $S_1 = \{q_0, q_1\}$.

Let us test this candidate DBA on the infinite word:

$$\alpha = (ab)^\omega = ababab \ldots$$

The run of the DBA on α is:

$$\sigma = S_0 \xrightarrow{a} S_0 \xrightarrow{b} S_1 \xrightarrow{a} S_0 \xrightarrow{b} S_1 \ldots$$

This run visits S_1 infinitely often, which means the DBA accepts the word. However, α contains infinitely many a's! It does not belong to the language L_{fin_a}, which requires only finitely many a's.

Why did the subset construction fail? The subset construction tracks *reachability* at each finite prefix. For any prefix ending in b (e.g., ab, $abab$), the state q_1 is indeed reachable in the NBA. However, these reachable states do not stitch together into a single, continuous infinite run that visits q_1 infinitely often. Every time an a is read, the path that had branched off to q_1 dies, forcing the NBA to backtrack (which is impossible for a deterministic machine). The subset construction "remembers" that q_1 was reachable, but it forgets that the path was a dead end when a arrived. This mismatch shows that tracking subset reachability is not sufficient for ω-acceptance. The next step is to make that mismatch exact: which languages can deterministic Büchi acceptance still capture, and where is the precise algebraic boundary?

9.2 Deterministic Büchi Automata (DBA) and the Limit of Vectorial Closure

Chapter 8 proved by a state-based pumping argument that no DBA recognises L_{fin}. The pumping argument is constructive but leaves a question open: *what* languages, exactly, can a DBA recognise? Answering that question algebraically is the goal of this section. We will characterise the DBA-recognisable languages as the *vectorial closures* of regular languages. With that characterisation in hand, the failure of L_{fin} becomes a clean diagonalisation argument at the level of *prefixes* rather than states—and it points the way to the repair (richer acceptance) that occupies the rest of the chapter.

We begin by formally characterising the languages recognizable by deterministic Büchi automata. In a standard DFA running over finite words, acceptance is determined by the state the automaton is in when it reaches the end of the word. However, since an infinite word has no end, we must evaluate the behavior of the automaton in the limit.

To see how we can characterize this, let us perform a thought experiment at the whiteboard. Suppose we have a DBA $A = (Q, \Sigma, \delta, q_0, F)$. How does A process an infinite word $\alpha = a_0 a_1 a_2 \dots$? Since A is deterministic and complete, there is a unique run $\sigma = s_0 s_1 s_2 \dots$ where $s_0 = q_0$ and $s_{i+1} = \delta(s_i, a_i)$. For each prefix of length n, $u_n = \alpha[0..n-1]$, the state reached by the automaton is exactly s_n. If we temporarily view A as a standard DFA A_{DFA} (with the exact same states, transitions, start state, and final states F), then A_{DFA} accepts the finite word u_n if and only if $s_n \in F$.

Now, let's look at the Büchi acceptance condition: A accepts the infinite word α if and only if the run σ visits the accepting set F infinitely often. Mathematically, this means:

$$\exists^{\infty} n \in \mathbb{N} \text{ such that } s_n \in F$$

Which is equivalent to:

$$\exists^{\infty} n \in \mathbb{N} \text{ such that } A_{DFA} \text{ accepts the prefix } u_n$$

In other words, the infinite word α is accepted by the DBA A if and only if *infinitely many of its prefixes are accepted by the corresponding DFA* A_{DFA}.

This beautiful correspondence motivates our first formal definition: the *vectorial closure* of a language of finite words. The vectorial closure (or limit) of a language $W \subseteq \Sigma^*$ is the set of all infinite words that have infinitely many prefixes in W.

9.2.1 Vectorial Closure

Definition 9.2.1 (Vectorial Closure)**.** Let $W \subseteq \Sigma^*$ be a language of finite words. The *vectorial closure* (or limit) of W, denoted by $W^{\top}$ or $W^{\delta} \subseteq \Sigma^{\omega}$, is the set of all infinite words that contain infinitely many prefixes in W:

$$W^{\delta} = \{\alpha \in \Sigma^{\omega} \mid \exists^{\infty} n \in \mathbb{N},\ \alpha[0..n-1] \in W\}$$

For example, if $\Sigma = \{a, b\}$ and $W = \Sigma^* a$, then W^{δ} consists of all infinite words that contain infinitely many occurrences of a, which is the language $(b^* a)^{\omega}$. Why? Any prefix in $W = \Sigma^* a$ must end with the letter a. An infinite word has infinitely many prefixes ending with a if and only if the letter a

appears infinitely often in the word. If a only appeared finitely many times, there would be some point after which no prefix could end in a, meaning the number of prefixes in W would be finite. Thus, $W^\delta = (b^*a)^\omega$.

To understand how the vectorial closure behaves, let us look at two more extreme examples:

Example 9.2.2 (Single-Word Limit)**.** Let $W = (ab)^*$. What is W^δ? An infinite word α belongs to W^δ iff it has infinitely many prefixes in $(ab)^*$. Let us trace this: the prefixes of $(ab)^\omega$ are $a, ab, aba, abab, ababa, \ldots$. The prefixes that belong to $W = (ab)^*$ are exactly those of even length: $ab, abab, ababab, \ldots$. There are infinitely many such prefixes, so $(ab)^\omega \in W^\delta$.

Can any other word β be in W^δ? Suppose $\beta \in W^\delta$. Then β must have infinitely many prefixes in $(ab)^*$. Since any two prefixes in $(ab)^*$ of different lengths must start with the same sequence, they must be prefixes of the same unique infinite word, which is $(ab)^\omega$. Thus, no other word can be in W^δ, so:

$$((ab)^*)^\delta = \{(ab)^\omega\}$$

This shows that the vectorial closure of an infinite regular language can collapse to a single infinite word.

Example 9.2.3 (Empty Limit)**.** Let $W = a^*b$. What is W^δ? A word $\alpha \in \Sigma^\omega$ belongs to W^δ iff it has infinitely many prefixes in a^*b. Let's see why: suppose an infinite word α has a prefix $u_1 \in a^*b$. This means $u_1 = a^k b$ for some $k \geq 0$. Now consider any longer prefix u_2 of α. Since u_2 contains u_1 as a prefix, the first $k+1$ letters of u_2 are $a^k b$. If u_2 were to also belong to a^*b, it would have to be of the form $a^m b$ for some $m \geq 0$. But since it starts with $a^k b$, and $b \neq a$, the only way it can be of the form $a^m b$ is if it does not contain b at position $k+1$, which is a contradiction.

More simply: any word in a^*b contains exactly one b, which is its last letter. Any prefix of α that is in a^*b must end at the first occurrence of b in α. Therefore, there can be at most one prefix of α in a^*b. Since a single prefix is not "infinitely many", no infinite word can have infinitely many prefixes in W. Thus:

$$(a^*b)^\delta = \emptyset$$

This demonstrates that the vectorial closure of a non-empty regular language can be completely empty.

9.2.2 The DBA Characterization Theorem

The relationship between DBAs and the vectorial closure is precise: the class of languages recognizable by DBAs is exactly the class of vectorial closures of regular languages.

Theorem 9.2.4 (DBA Characterization)**.** *An ω-language $L \subseteq \Sigma^\omega$ is recognized by a Deterministic Büchi Automaton (DBA) if and only if $L = W^\delta$ for some regular language $W \subseteq \Sigma^*$.*

Proof. We prove both directions of the equivalence.

($\Longrightarrow$) Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DBA recognizing L. We define the DFA $A' = (Q, \Sigma, \delta, q_0, F)$ having the same states, transition function, and final states, but running on finite words. Let $W = L(A') = \{u \in \Sigma^* \mid \delta^*(q_0, u) \in F\}$, which is a regular language by definition. We show $L(A) = W^\delta$:

- Let $\alpha \in L(A)$. The unique infinite run $\sigma = s_0 s_1 s_2 \ldots$ of A on α satisfies $\inf(\sigma) \cap F \neq \emptyset$. Let $i_0 < i_1 < i_2 < \ldots$ be the infinite sequence of indices where $s_{i_j} \in F$. Since A is deterministic, the state reached after reading the prefix $\alpha[0..i_j - 1]$ is exactly $\delta^*(q_0, \alpha[0..i_j - 1]) = s_{i_j} \in F$. Thus, by definition of W, we have $\alpha[0..i_j - 1] \in W$ for all $j \in \mathbb{N}$. Since there are infinitely many such prefixes, $\alpha \in W^\delta$.
- Conversely, let $\alpha \in W^\delta$. There exists an infinite sequence of indices $k_0 < k_1 < k_2 < \ldots$ such that the prefix $u_j = \alpha[0..k_j - 1] \in W$ for all $j \in \mathbb{N}$. By definition of W, $\delta^*(q_0, u_j) \in F$. Since the automaton is deterministic, the state at position k_j in the unique run σ of A on α is exactly $s_{k_j} = \delta^*(q_0, u_j) \in F$. Thus, the run σ visits F infinitely often, which implies $\alpha \in L(A)$.

($\Longleftarrow$) Let $L = W^\delta$ for a regular language $W \subseteq \Sigma^*$. Since W is regular, there exists a DFA $A' = (Q, \Sigma, \delta, q_0, F)$ recognizing W. We treat A' as a DBA $A = (Q, \Sigma, \delta, q_0, F)$. By the same reasoning as above, the unique run of A on an infinite word α visits F infinitely often if and only if α has infinitely many prefixes accepted by A', which means $\alpha \in W^\delta$. Thus, $L(A) = W^\delta$, and L is recognized by the DBA A.

□

■ **Formal details — Where Determinism is Crucial**

Observe that the proof of the converse ($\Longleftarrow$) relies entirely on the uniqueness of the run. In a non-deterministic Büchi automaton (NBA), different prefixes of an infinite word α can be accepted by different finite runs that do not compile into a single, infinite run.

Consider the NFA for L_{fin_a} (words with finitely many a's). Over finite words, this NFA recognizes $W = \Sigma^* b$ (any word ending in b, after making the transition to q_1). Let's evaluate the infinite word $(ab)^\omega$. This word has infinitely many prefixes in $\Sigma^* b$ (namely, ab, $abab$, $ababab$, etc.). So $(ab)^\omega \in W^\delta$. However, $(ab)^\omega$ has infinitely many a's, so $(ab)^\omega \notin L_{fin_a}$.

Why did W^δ incorrectly include $(ab)^\omega$? Because the non-deterministic automaton "guessed" the last a correctly for each finite prefix independently, but those different successful finite runs cannot be stitched together into a single valid infinite run. Determinism prevents this cheating.

9.2.3 The Limit of DBAs: The Finitely Many a's Language

We now reprove, algebraically, the theorem of section 8.11 in Chapter 8: no DBA recognises L_{fin}. There, the proof worked directly on DBA states via pumping. Here we use the DBA Characterization Theorem as a lever: we show that L_{fin_a} cannot be written as W^δ for any regular W, and therefore no DBA can recognise it. The two proofs are complementary—one operational, one algebraic—and both will be useful when we generalise to richer acceptance conditions.

Now we demonstrate that DBAs are strictly less expressive than NBAs by showing that a simple ω-regular language cannot be written as the vectorial closure of any regular language.

Let $\Sigma = \{a, b\}$. Consider the language:

$$L_{fin_a} = \{\alpha \in \Sigma^\omega \mid \alpha \text{ contains finitely many occurrences of } a\}$$

This language is easily recognized by the NBA shown in Figure 9.2.

Intuition behind the proof. Before diving into the formal proof, let us understand intuitively why a DBA cannot recognize L_{fin_a}. A DBA is deterministic: its transitions are completely determined by the prefix read so far. Suppose a DBA attempts to recognize L_{fin_a}. As it reads a sequence of b's, it must eventually decide that the a's have stopped and start visiting accepting states. However, because it cannot look into the future, the automaton can never be sure if the current a is the *last* one. If it starts accepting after a long sequence of b's, an adversary can "trick" the automaton by presenting another a, forcing it to stop accepting. By repeating this process infinitely often, the adversary can construct a word with infinitely many a's that the automaton is forced to accept. This "cat-and-mouse" game is formalized in the proof below.

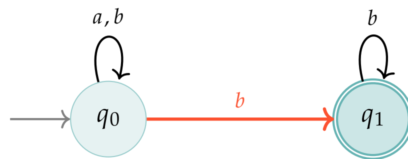


Figure 9.2: Non-deterministic Büchi automaton recognizing L_{fin_a}. The transition from q_0 to q_1 on b (highlighted in orange) requires the automaton to non-deterministically "guess" the last occurrence of the symbol a.

Theorem 9.2.5. *The language L_{fin_a} is not recognizable by any DBA.*

Proof. This is the algebraic counterpart of the state-based pumping proof of theorem 8.11.3 in Chapter 8. The proof is by contradiction. Assume that L_{fin_a} is recognizable by a DBA. By the DBA Characterization Theorem, there exists a regular language $W \subseteq \Sigma^*$ such that $L_{fin_a} = W^\delta$.

1. Consider the infinite word $\alpha_1 = b^\omega$. Since α_1 contains zero a's, it belongs to L_{fin_a}. Because $L_{fin_a} = W^\delta$, there must exist a prefix of α_1 in W. Thus, there exists a number $n_1 \geq 1$ such that:

$$w_1 = b^{n_1} \in W$$

2. Now consider the infinite word $\alpha_2 = b^{n_1}ab^\omega$. Since α_2 contains exactly one a, it belongs to $L_{fin_a} = W^\delta$. Therefore, α_2 must have infinitely many prefixes in W. We can choose one such prefix that is strictly longer than w_1 (meaning it must include the a and some suffix of b's). Thus, there exists $n_2 \geq 1$ such that:

$$w_2 = b^{n_1}ab^{n_2} \in W$$

3. We proceed inductively. Suppose we have constructed a finite word:

$$w_k = b^{n_1}ab^{n_2}a \dots ab^{n_k} \in W$$

   Consider the infinite word $\alpha_{k+1} = w_k ab^\omega$. Since α_{k+1} contains exactly k occurrences of a, it belongs to $L_{fin_a} = W^\delta$. Thus, it must have infinitely

many prefixes in W. We choose a prefix strictly longer than w_k, which forces the existence of $n_{k+1} \geq 1$ such that:

$$w_{k+1} = w_k a b^{n_{k+1}} = b^{n_1} a b^{n_2} a \dots a b^{n_k} a b^{n_{k+1}} \in W$$

4. Now, construct the infinite word:

$$\alpha = b^{n_1} a b^{n_2} a b^{n_3} a \dots$$

By construction, the infinite word α contains infinitely many prefixes of the form w_k, each of which belongs to W. By definition of the vectorial closure, this implies that:

$$\alpha \in W^{\delta} = L_{fin_a}$$

However, by construction, α contains infinitely many occurrences of the symbol a (since one a is placed between each segment b^{n_i}). This is a contradiction, since L_{fin_a} contains only words with finitely many a's.

Thus, our initial assumption is false: no DBA can recognize L_{fin_a}. □

Corollary 9.2.6. *The following properties hold:*

1. *Deterministic Büchi automata are strictly less expressive than non-deterministic Büchi automata:*

$$DBA \subset NBA$$

2. *DBAs are not closed under complementation.*

Proof. The language $L_{inf_a} = (b^* a)^\omega$ (words containing infinitely many a's) is recognizable by a DBA (a simple 2-state automaton that transitions to an accepting state whenever it reads a). The complement of this language is L_{fin_a}, which is not recognizable by any DBA. Hence, DBAs are not closed under complementation. □

The negative result now becomes a design brief. Büchi acceptance is too weak under determinism, so we keep the deterministic transition structure but enrich the acceptance condition.

9.3 Deterministic Muller Automata (DMA) and McNaughton's Theorem

We have just proved that DBAs are strictly less expressive than NBAs, and are not closed under complementation. This represents a serious bottleneck: reactive synthesis requires determinism, but we cannot compile general ω-regular specifications into DBAs.

To resolve this expressiveness gap, we must rethink the acceptance condition. The Büchi condition is fundamentally existential and local: it only asks whether *at least one* accepting state is visited infinitely often, completely ignoring the other states. If we are restricted to a deterministic transition function, this condition is too weak to separate complex languages like L_{fin_a}.

Instead of marking individual states as accepting, what if we specify the *exact set* of states that the unique infinite run is allowed to cycle through in the limit? By defining acceptance in terms of the set of states visited infinitely often, we can capture fine-grained properties. This is the core idea behind

the Muller condition, which successfully restores the equivalence between determinism and non-determinism.

Let's preview the idea on the recurring request/acknowledgement example. Suppose requests r and grants g arrive as separate symbols over $\Sigma = \{r, g, \text{idle}\}$. Consider the property "eventually, every request is followed by a grant"—more precisely, "from some point on, r and g alternate, with r preceding g". A deterministic 2-state machine that flips between q_r ("last event was a request, awaiting grant") and q_g ("last event was a grant, awaiting next request") reads every infinite word into a unique run. To accept the alternating property we want exactly the cycle $\{q_r, q_g\}$ to occur infinitely often; if the run eventually sticks to q_r alone, some request was never granted and we must reject. The Büchi condition "visit q_g infinitely often" would also do the job here, but the Muller condition lets us additionally insist that q_r is visited infinitely often—i.e., that the run keeps cycling, not sitting on a final state. That extra precision is exactly what we need to handle languages like L_{fin_a} that resist DBAs.

9.3.1 Muller Acceptance Condition

Definition 9.3.1 (Deterministic Muller Automaton)**.** A *Deterministic Muller Automaton* (DMA) is a tuple $A = (Q, \Sigma, \delta, q_0, \mathcal{F})$ where Q, Σ, δ, q_0 are defined as in a standard DFA (complete and deterministic), and $\mathcal{F} \subseteq \mathcal{P}(Q)$ is a family of subsets of states.

An infinite run σ of A on a word $\alpha \in \Sigma^\omega$ is successful if the set of states visited infinitely often is *exactly* one of the sets in the family $\mathcal{F}$:

$$\inf(\sigma) \in \mathcal{F}$$

The family $\mathcal{F}$ is the heart of the model. Where the Büchi condition $F \subseteq Q$ can only ask "is some accepting state visited infinitely often?", the Muller condition can pin down the *entire* recurrent set: which states are visited infinitely often, and (by omission) which states must be visited only finitely often. This is what lets us express "finitely many a's" deterministically: in the 2-state DMA with q_a reached on a and q_b reached on b, we accept iff $\inf(\sigma) = \{q_b\}$, ruling out q_a from the limit. Notice that the size of $\mathcal{F}$ can in principle be exponential in $|Q|$—a practical annoyance we will address in section 9.4 with the Rabin and parity conditions.

Unlike the Büchi condition, which only requires some *accepting state to be visited infinitely often, the Muller condition specifies the* exact set *of states that the run must cycle through in the limit.*

Example 9.3.2 (DMA for Finitely Many a's)**.** Recall the language L_{fin_a} (words with finitely many a's) that defeated the DBA. Let's see how the Muller condition effortlessly captures it deterministically. Consider a 2-state deterministic transition structure where reading a always goes to q_a, and reading b always goes to q_b. If a word has finitely many a's, it must eventually consist entirely of b's. This means the unique infinite run will eventually get stuck in q_b and never visit q_a again. Thus, the set of states visited infinitely often will be exactly $\{q_b\}$. We define the Muller acceptance family as:

$$\mathcal{F} = \{\{q_b\}\}$$

If a word has infinitely many a's, the run will visit q_a infinitely often, yielding $\inf(\sigma) = \{q_a, q_b\} \notin \mathcal{F}$, which is correctly rejected. The Muller condition

simply filters out the invalid runs by explicitly stating what the limit cycle must look like.

Example 9.3.3 (DMA for Infinitely Many a's and b's)**.** Let $\Sigma = \{a, b\}$. We want to recognize the language $L_{inf_a_and_b}$ of words containing infinitely many a's and infinitely many b's. This is a natural "both signals keep arriving" property of the request/acknowledgement pattern: a plays the role of a request, b the role of an acknowledgement, and we require that both keep happening forever. Consider the 2-state automaton with states $Q = \{q_0, q_1\}$, where reading a transitions to q_0 and reading b transitions to q_1. The run on any word containing infinitely many a's and b's must visit both q_0 and q_1 infinitely often. Thus, the set of states visited infinitely often is $\{q_0, q_1\}$. To accept this language, we set the Muller acceptance family to:

$$\mathcal{F} = \{\{q_0, q_1\}\}$$

If a word contains only finitely many a's, the run eventually stays in q_1, yielding $\inf(\sigma) = \{q_1\} \notin \mathcal{F}$. If it contains only finitely many b's, $\inf(\sigma) = \{q_0\} \notin \mathcal{F}$. Thus, the DMA correctly recognizes the language. Notice that a DBA cannot do this with a 2-state transition structure: insisting on *both* q_0 and q_1 being visited infinitely often is exactly the kind of constraint the Büchi condition cannot express.

9.3.2 Closure Properties of DMAs

The strength of the Muller condition lies in its robustness under Boolean operations.

Theorem 9.3.4. *DMA-recognizable languages are closed under complementation, union, and intersection.*

Proof. Let $A = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be a complete DMA recognizing L.

Complementation Since A is deterministic and complete, every infinite word α has a unique run σ. The set of states visited infinitely often $\inf(\sigma)$ is unique. To recognize the complement $\Sigma^\omega \setminus L$, we simply define:

$$A^C = (Q, \Sigma, \delta, q_0, \mathcal{P}(Q) \setminus \mathcal{F})$$

A run σ is accepting in $A^C \iff \inf(\sigma) \in \mathcal{P}(Q) \setminus \mathcal{F} \iff \inf(\sigma) \notin \mathcal{F} \iff \alpha \notin L$.

Union Let $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, \mathcal{F}_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, \mathcal{F}_2)$ be DMAs. We construct the product automaton:

$$A = (Q_1 \times Q_2, \Sigma, \delta, (q_{01}, q_{02}), \mathcal{F})$$

where $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$, and the acceptance family $\mathcal{F}$ is defined as:

$$\mathcal{F} = \{S \subseteq Q_1 \times Q_2 \mid \pi_1(S) \in \mathcal{F}_1 \vee \pi_2(S) \in \mathcal{F}_2\}$$

where $\pi_i(S)$ is the projection of the state set S onto the i-th component. Since the run of A on α is the parallel execution of the runs σ_1 and σ_2,

the projection of $\inf(\sigma)$ onto the components yields $\inf(\sigma_1)$ and $\inf(\sigma_2)$ respectively. Thus, the run is accepted iff the run of A_1 is accepted or the run of A_2 is accepted.

□

9.3.3 The DMA Characterization Lemma

We can now characterize DMA languages algebraically.

> **Lemma 9.3.5** (DMA Characterization). *An ω-language $L \subseteq \Sigma^\omega$ is recognized by a DMA if and only if it is a Boolean combination of vectorial closures of regular languages.*

Proof. We prove both directions of the lemma.

($\Longrightarrow$) Let $A = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be a DMA. We can partition the acceptance of A by the exact state set visited infinitely often:

$$L(A) = \bigcup_{F \in \mathcal{F}} \{\alpha \in \Sigma^\omega \mid \inf(\sigma) = F\}$$

The condition $\inf(\sigma) = F$ is equivalent to asserting that all states in F are visited infinitely often, and all states outside F are visited only finitely often:

$$\inf(\sigma) = F \iff \left(\bigwedge_{q \in F} q \text{ is visited infinitely often}\right) \wedge \left(\bigwedge_{q \notin F} q \text{ is visited finitely often}\right)$$

For each state $q \in Q$, we define the regular language of finite words:

$$W_q = \{u \in \Sigma^* \mid \delta^*(q_0, u) = q\}$$

A state q is visited infinitely often by the run σ if and only if the word has infinitely many prefixes in W_q. Thus, q is visited infinitely often iff $\alpha \in W_q^\delta$. Consequently, the language of words whose run cycles exactly on F is:

$$L_F = \bigcap_{q \in F} W_q^\delta \cap \bigcap_{q \notin F} (W_q^\delta)^C$$

Thus, $L(A) = \bigcup_{F \in \mathcal{F}} L_F$, which is a Boolean combination of vectorial closures of regular languages.

($\Longleftarrow$) Let L be a Boolean combination of W^δ for regular languages W. For each W, W^δ is DBA-recognizable by the DBA Characterization Theorem. Since every DBA is a DMA (by setting $\mathcal{F} = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$), each building block W^δ is DMA-recognizable. Since DMAs are closed under Boolean operations, the entire Boolean combination is DMA-recognizable.

□

9.3.4 McNaughton's Theorem and the Closure Bottleneck

We are finally ready to state the central theorem of this chapter. It closes the gap left open in Chapter 8: *determinism* and *ω-regularity* can coexist, provided we trade the Büchi condition for the Muller condition.

Theorem 9.3.6 (McNaughton, 1966)**.** *Deterministic Muller Automata are expressively equivalent to Non-deterministic Büchi Automata. That is, a language $L \subseteq \Sigma^\omega$ is ω-regular if and only if it is recognized by a DMA.*

To understand why the proof of McNaughton's theorem is highly non-trivial, we must look at the algebraic mismatch between the two models. What follows is a proof roadmap rather than a full derivation: it isolates the algebraic bottleneck and records the counterexamples that rule out the tempting shortcuts. An ω-regular language is naturally represented using the *ω-closure* of regular languages:

McNaughton's theorem is a landmark result because it proves that we do not lose expressiveness by forcing the transition structure to be deterministic, provided we enrich the acceptance condition.

$$L = \bigcup_{i=1}^{k} U_i \cdot V_i^\omega$$

In contrast, DMAs naturally characterize Boolean combinations of the *vectorial closure*:

$$L = \text{Boolean combination of } W_j^\delta$$

The bottleneck of the proof lies in converting the infinite iteration V_i^ω into combinations of prefix limits W_j^δ. Naive attempts to equate the two operators fail, as shown by the following counterexamples.

■ **Formal details — Proof roadmap for McNaughton's theorem**

The full proof is beyond this chapter (see W. Thomas, *Automata on Infinite Objects*, Handbook of Theoretical Computer Science vol. B, 1990), but the strategy can be summarised in three steps.

Step 1 (DMA Characterization Lemma, proved above) An ω-language is DMA-recognisable iff it is a Boolean combination of languages of the form W^δ with W regular. This turns the theorem into a purely algebraic statement about ω-regular languages.

Step 2 (the hard step) Every ω-regular language $L = \bigcup_{i=1}^{k} U_i \cdot V_i^\omega$ can be rewritten as a Boolean combination of vectorial closures of regular languages. This is the heart of McNaughton's argument and relies on a delicate combinatorial analysis (via $\mathcal{V}$-*witnesses* or, equivalently, ultimately-periodic decompositions of ω-semigroups). The two counterexamples below show that the most naive rewritings—$UV^\omega \stackrel{?}{=} UV^\delta$ and $(UV)^\delta \stackrel{?}{=} U \cdot V^\delta$—are simply *false*; the theorem holds, but only because we are allowed to swap U and V for cleverly re-engineered regular sets U', V' depending on L.

Step 3 (conclusion) Step 2 expresses L as a Boolean combination of W_j^δ, and Step 1 turns that Boolean combination into a DMA. Since DMAs are closed under complement, union, and intersection, no further machinery is needed.

Why McNaughton's proof is non-constructive The argument above tells us that *some* DMA exists for every NBA, but it does not give a direct algorithm to compute one. This is a serious problem for verification tools: given a 30-state NBA compiled from an LTL formula, we cannot afford to chase the proof's combinatorial core by hand. *Safra's construction* (section 9.4.3) is the constructive rescue: a single explicit procedure that turns any NBA into an equivalent DRA in time $2^{O(n \log n)}$.

■ **Formal details — Counterexample 1:** $U \cdot V^\omega \neq U \cdot V^\delta$

One might hope that if V is closed under concatenation ($V \cdot V \subseteq V$), then $V^\omega = V^\delta$. This is false. Let $\Sigma = \{a, b\}$ and $V = (ab^*)^*$. Consider the infinite word $\alpha = ab^\omega$. All finite prefixes of α are of the form ab^k, which belong to $ab^* \subseteq V$. Since there are infinitely many such prefixes, we have $\alpha \in V^\delta$. However, $\alpha \notin V^\omega$. The definition of V^ω requires α to be decomposed into an infinite concatenation of non-empty words $v_1 v_2 v_3 \ldots$ where each $v_j \in V = (ab^*)^*$. Since each non-empty word in $(ab^*)^*$ must contain at least one occurrence of the symbol a, any word in V^ω must contain infinitely many a's. Since $\alpha = ab^\omega$ contains exactly one a, it cannot belong to V^ω. Thus, $V^\delta \not\subseteq V^\omega$.

■ **Formal details — Counterexample 2:** $(U \cdot V)^\delta \neq U \cdot V^\delta$

Another attempt might be to distribute the prefix operator. This also fails. Let $U = b^*$ and $V = \{b\}$. Then $U \cdot V = b^* b = b^+$. Consider the word $\alpha = b^\omega$. Every prefix of α is of the form $b^k \in b^+$, so $\alpha \in (U \cdot V)^\delta$. However, the language $V^\delta = \{b\}^\delta = \emptyset$, because a finite set of words cannot have infinitely many prefixes in an infinite word (the length of words in $\{b\}$ is capped at 1). Thus, $U \cdot V^\delta = \emptyset$. Hence, $(U \cdot V)^\delta \not\subseteq U \cdot V^\delta$.

9.3.5 Corollary: Weak S1S (WS1S) = S1S

An elegant application of McNaughton's theorem is proving that restricting S1S variables to finite sets does not reduce the expressive power of the logic.

> **Theorem 9.3.7.** *Weak S1S (WS1S) is expressively equivalent to S1S.*

Proof. Clearly, WS1S ⊆ S1S because WS1S is syntactically a subset of S1S. To prove S1S ⊆ WS1S, let L be an S1S-definable language. By the Büchi Theorem, L is ω-regular. By McNaughton's Theorem, L is DMA-recognizable. By the DMA Characterization Lemma, L can be written as:

$$L = \text{Boolean combination of } W^\delta$$

for regular languages $W \subseteq \Sigma^*$. Since WS1S is closed under Boolean operations, we only need to show that W^δ is definable in WS1S for any regular language W. Since W is a regular language of finite words, there exists a WS1S formula $\phi(X_1, \ldots, X_k)$ that defines it. We introduce a new first-order variable y (representing a boundary). We relativize the formula ϕ to the finite interval $[0, y]$ by bounding all quantifiers:

- Replace $\exists x \psi$ with $\exists x (x \leq y \wedge \psi)$.
- Replace $\forall x \psi$ with $\forall x (x \leq y \rightarrow \psi)$.

Let this relativized formula be $\psi(X_1, \ldots, X_k, y)$. It asserts that the prefix of the word up to position y belongs to W. We can now assert that there are infinitely many such positions y:

$$\Phi(X_1, \ldots, X_k) = \forall x \exists y \left(y > x \wedge \psi(X_1, \ldots, X_k, y) \right)$$

Since x and y are first-order variables, they represent single positions (which are finite sets). Thus, Φ is a valid WS1S formula. □

Muller automata restore expressiveness, but they still leave open how to store and compute the acceptance condition efficiently. Rabin and parity acceptance address that implementation-facing side of determinisation.

9.4 Deterministic Rabin Automata, Parity, and Safra's Construction

McNaughton's theorem tells us that DMAs are expressively complete for ω-regular languages, but it leaves two practical problems on the table. First, the Muller family $\mathcal{F} \subseteq \mathcal{P}(Q)$ can have up to $2^{|Q|}$ elements, which is expensive to store and check. Second, the theorem is non-constructive: it tells us that a DMA exists without telling us how to compute one from an NBA. This section solves both problems.

We first replace the bulky Muller family with the more compact *Rabin pairs* (and the equivalent *parity ranking*). Then we give Safra's constructive determinisation procedure, which compiles any NBA into an equivalent DRA.

9.4.1 Rabin Acceptance Condition

The key idea, proposed by Michael Rabin, is to specify pairs of state sets. Each pair represents a constraint: we define a set of "bad" states L_i that should eventually be avoided, and a set of "good" states U_i that must be visited infinitely often. This allows us to define identical languages using a much more compact condition.

Definition 9.4.1 (Deterministic Rabin Automaton)**.** A *Deterministic Rabin Automaton* (DRA) is a tuple $A = (Q, \Sigma, \delta, q_0, \Omega)$ where Q, Σ, δ, q_0 are defined as in a DFA, and the acceptance condition is a set of pairs of state sets:

$$\Omega = \{(L_1, U_1), (L_2, U_2), \ldots, (L_n, U_n)\}$$

where $L_i, U_i \subseteq Q$ for all $1 \leq i \leq n$.

A run σ is successful if there exists an index $j \in \{1, \ldots, n\}$ such that:

$$\inf(\sigma) \cap L_j = \emptyset \quad \text{and} \quad \inf(\sigma) \cap U_j \neq \emptyset$$

This means that, for some pair j, the run eventually avoids all states in L_j while visiting at least one state in U_j infinitely often.

The Rabin condition is a hybrid: the L_j component is "Muller-flavoured" (it pins down what must *disappear* from the limit, like the $\mathcal{P}(Q) \setminus \mathcal{F}$ side of complementation), while the U_j component is "Büchi-flavoured" (it asks for something to occur infinitely often). The number of pairs n is typically $O(|Q|)$ rather than $2^{|Q|}$, which is the practical advantage over Muller. As an example, the request/acknowledgement property "every request is eventually followed by a grant" is naturally expressed by a single Rabin pair (L, U): let L contain all the "request-pending" states and U contain all the "grant-issued" states, so that the pair fires exactly when the run eventually leaves the pending region and keeps visiting grant states forever.

The Rabin condition combines a Muller-like restriction (L_j must not be visited infinitely often) with a Büchi-like trigger (U_j must be visited infinitely often).

Example 9.4.2 (Rabin Reset Automaton)**.** Consider a 3-state DRA over $\Sigma = \{A, B, C\}$ with states $Q = \{q_0, q_1, q_2\}$ where reading A resets the state to q_0, reading B resets to q_1, and reading C resets to q_2. We want to recognize the language where:

- either B is read finitely often and A is read infinitely often;
- or both A and B are read finitely often and C is read infinitely often.

We can define the Rabin condition with two pairs $\Omega = \{(L_1, U_1), (L_2, U_2)\}$:

Pair 1 $L_1 = \{q_1\}$ and $U_1 = \{q_0\}$. This requires the run to visit q_1 finitely often (finitely many B's) and q_0 infinitely often (infinitely many A's).

Pair 2 $L_2 = \{q_0, q_1\}$ and $U_2 = \{q_2\}$. This requires the run to visit q_0, q_1 finitely often (finitely many A's and B's) and q_2 infinitely often (infinitely many C's).

This DRA precisely captures the language.

9.4.2 Parity Acceptance and the Equivalence DMA ≡ DRA ≡ Parity

The Rabin condition is already a major practical improvement over Muller, but it can be simplified even further. A *parity condition* labels each state $q \in Q$ with an integer priority $\pi(q) \in \{0, 1, \ldots, 2k\}$ and accepts the run iff the *largest* priority occurring infinitely often is even.

Definition 9.4.3 (Deterministic Parity Automaton)**.** A *Deterministic Parity Automaton* (DPA) is a tuple $A = (Q, \Sigma, \delta, q_0, \pi)$ where $\pi : Q \to \{0, 1, \ldots, 2k\}$ is a priority function. A run σ is successful iff

$$\max\{\pi(q) \mid q \in \inf(\sigma)\}$$

is even.

The parity condition is the "minimal" deterministic acceptance condition for ω-regular languages: just one integer per state. Despite its simplicity, it is expressively complete.

Theorem 9.4.4 (Equivalence of deterministic conditions)**.** *For ω-languages over infinite words, the following deterministic models recognise exactly the same class—the ω-regular languages:*

$$DMA \equiv DRA \equiv DPA \equiv NBA.$$

Proof sketch. By McNaughton's theorem it suffices to show DMA ≡ DRA ≡ DPA.

DMA → DPA Given a DMA $(Q, \Sigma, \delta, q_0, \mathcal{F})$, the standard construction replaces $\mathcal{F}$ by an *appearance-record* (latest-appearance record, LAR) of the states. The LAR tracks, at every step, the permutation of Q ordered by most-recent visit. The states of the DPA are these permutations (so $|Q|!$ states), and the priority assigned to a permutation ρ is even iff the set of states appearing after the last "hit" equals some $F \in \mathcal{F}$. Since every infinite run induces a unique infinite sequence of LARs, the parity check coincides with the original Muller check.

DPA → DRA Given priorities $\pi : Q \to \{0, \ldots, 2k\}$, build one Rabin pair per even priority $2i$: set $L_{2i} = \{q \mid \pi(q) > 2i\}$ and $U_{2i} = \{q \mid \pi(q) = 2i\}$. The pair (L_{2i}, U_{2i}) fires iff $2i$ is the largest priority seen infinitely often—exactly the parity condition.

DRA → DMA Given pairs $\Omega = \{(L_i, U_i)\}$, the equivalent Muller family is

$$\mathcal{F} = \{S \subseteq Q \mid \exists i.\ S \cap L_i = \emptyset \ \wedge\ S \cap U_i \neq \emptyset\}.$$

Each $S \in \mathcal{F}$ certifies that some pair i is satisfied, and conversely.

□

The parity condition will return as the central acceptance condition for infinite games in synthesis, where it makes attractor computations polynomial (see chapter 5). For now, it confirms that we have a small, clean deterministic model that captures all ω-regular languages.

Over trees*, the equivalence between deterministic and non-deterministic automata breaks down: deterministic Rabin automata are strictly more expressive than NBAs. We will return to this in chapter 10.*

9.4.3 Safra's Determinization (Roadmap)

McNaughton's theorem tells us *that* every NBA admits an equivalent deterministic automaton, but it does not tell us *how* to compute one. Safra's 1988 construction is the constructive answer. We describe it at roadmap level—enough to understand why it works and what its complexity is, without the full case analysis or the full bookkeeping of node names.

■ **Formal details — Construction idea: Safra trees**

The problem with the subset construction Recall from section 9.1 that the Rabin-Scott subset construction fails on Büchi automata because it forgets *which* branches through the NBA are still alive in their search for accepting states. To recover determinism we need a data structure that simultaneously tracks subset reachability *and* remembers, for each candidate branch, whether it has visited an accepting state since the last checkpoint.

Safra's data structure A Safra tree is a finite ordered tree whose nodes are labelled with subsets of Q (states of the NBA). Each node also carries a Boolean flag "accepting hit since this node was created". The invariant is:

- the union of all node labels equals the current subset-construction state (so no NBA run is lost);
- the labels of sibling nodes are pairwise disjoint (so we never double-count an NBA state);
- a node's label is always a strict subset of its parent's label (so the tree is finite-depth);
- a child is created only when its parent has visited an NBA accepting state, so the child marks a "fresh post-accepting" run.

The transition on a single letter Reading letter a, the algorithm performs, in order:

1. **Power step.** Replace every node label S by $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$.
2. **Accepting hits.** For every node whose new label intersects F, set its "hit" flag and spawn a fresh child labelled with $S \cap F$.
3. **Horizontal sanitisation.** Walking siblings left-to-right, remove from each node's label any state already present in an elder sibling. (Order matters: this is how Safra remembers "which branch claimed this NBA state first".)
4. **Vertical merge.** If a node's label equals the union of its children's labels, delete the children and set the node's "hit" flag (the parent has now collected all the post-accepting runs its children were tracking).
5. **Cleanup.** Delete empty nodes.

The Rabin acceptance The Rabin pairs are indexed by the (finitely many possible) nodes of a Safra tree. For each possible node position v, the pair is

$$L_v = \{\text{configurations in which node } v \text{ is absent}\}, \quad U_v = \{\text{configurations in which node } v \text{ has its hit flag}$$

Intuition: if the run of the Safra automaton visits U_v infinitely often, then infinitely many accepting hits were collected at node v, so some NBA branch must have hit F infinitely often.

If additionally L_v is eventually avoided (node v stays alive forever), that branch survives forever—so the NBA accepts.

Complexity For an n-state NBA, the number of possible Safra trees is $2^{O(n \log n)}$ and the number of Rabin pairs is $O(n)$. This is asymptotically optimal: a lower bound of $n^{\Theta(n)}$ states is known for converting certain NBAs to deterministic automata. Modern variants (Piterman's parity version, Schewe's optimisation) trim constants but preserve the asymptotics.

Why we will not run Safra by hand Even a 4-state NBA produces a Safra DRA with hundreds of states. In practice, verification tools never run Safra directly; they use it as the theoretical guarantee that deterministic automata exist, while working with the original NBAs and quotient constructions in actual algorithms.

With the deterministic acceptance conditions in place, we now return to logic and ask what happens when S1S itself is syntactically restricted.

9.5 The First-Order Fragment of S1S and Star-Free Languages

Having characterized the deterministic variants of ω-automata on the operational side, we now return to the logical side. Full S1S is equivalent to the entire class of ω-regular languages (Chapter 8), but it suffers from non-elementary complexity. This complexity is driven by second-order quantification (the ability to quantify over infinite sets of positions).

What happens if we restrict S1S by forbidding second-order variables, allowing only first-order quantification over individual positions? This syntactic restriction brings us to a fundamental class in formal language theory: the *star-free* languages. The bridge back to chapter 2 is direct—there we met star-free regular expressions as extended regular expressions in which the Kleene star is forbidden, and we observed that, surprisingly, $(ab)^*$ is still star-free even though it is written with a star. The deep theorem of this section, due to McNaughton and Papert, says that the star-free class is exactly the first-order fragment of S1S over words—and that, under infinite words, this same class corresponds to Linear Temporal Logic (LTL). The chain of equivalences star-free ≡ first-order ≡ aperiodic ≡ counter-free ≡ LTL is one of the crown jewels of automata-theoretic verification, and we lay its foundation here.

9.5.1 The Signature FO-S1S[<]

In full S1S, the successor relation $+1$ and the ordering relation $<$ are interdefinable. But in the first-order fragment, this symmetry breaks. Why? Let's ask this at the whiteboard:

Can $+1$ **be defined by** $<$ **in FO?** Yes! We can say $y = x + 1$ by saying that y is strictly greater than x, and there is no element strictly between them:

$$y = x + 1 \iff x < y \land \neg \exists z (x < z \land z < y)$$

This formula uses only first-order position variables and is perfectly valid in FO-S1S[$<$].

Can $<$ **be defined by** $+1$ **in FO?** No! If we only have $+1$, defining $x < y$ means saying $y = x+1 \lor y = x+1+1 \lor \ldots$ To write this for an arbitrary distance, we need to compute the transitive closure of the $+1$ relation. Computing a transitive closure requires induction (saying "the set of all elements reachable by +1"), which inherently demands second-order quantification over sets.

Thus, if we restrict ourselves to first-order variables and only provide $+1$, the logic becomes too weak to even express $<$. To study the correct first-order fragment of S1S, we must explicitly include the ordering relation in the signature, yielding the logic FO-S1S[$<$].

9.5.2 Star-Free Languages

Definition 9.5.1 (Star-Free Language)**.** A language $W \subseteq \Sigma^*$ of finite words is *star-free* if it can be generated from finite languages using only Boolean operations (union, intersection, and complement relative to Σ^*) and concatenation.

This is the same definition we met in chapter 2, restated here for direct reference. The Kleene star is forbidden, but every other operation of extended regular expressions is allowed.

Notice that we can express the universal language Σ^* as the complement of the empty set:

$$\Sigma^* = \emptyset^C$$

Since $\emptyset$ is finite (hence star-free) and complementation is allowed, Σ^* is star-free. This allows us to write complex expressions.

Example 9.5.2 ($(ab)^*$ without Kleene star)**.** Let $\Sigma = \{a, b\}$. The language $(ab)^*$ consists of words that alternate a and b, starting with a and ending with b (including the empty word ϵ). We can define this language by ruling out all violations:

1. No two consecutive a's: $L_{aa} = \Sigma^* aa \Sigma^*$.
2. No two consecutive b's: $L_{bb} = \Sigma^* bb \Sigma^*$.
3. Does not start with b: $L_{start_b} = b\Sigma^*$.
4. Does not end with a: $L_{end_a} = \Sigma^* a$.

Since Σ^* is star-free, each of these violation languages is star-free (using concatenation and finite sets). The language $(ab)^*$ is the complement of their union:

$$(ab)^* = \emptyset^C \setminus (L_{aa} \cup L_{bb} \cup L_{start_b} \cup L_{end_a})$$

Thus, $(ab)^*$ is star-free.

9.5.3 Translation: Star-Free $\Longrightarrow$ FO-S1S[$<$]

We prove that every star-free language can be defined in first-order S1S.

Theorem 9.5.3. *For every star-free language $W \subseteq \Sigma^*$, there exists a first-order S1S formula $\phi(x, y)$ (with two free variables x and y) such that for every finite word u and positions $x \le y$, $u[x..y] \in W \iff u \models \phi(x, y)$.*

Proof. The proof is by induction on the structure of the star-free expression.

Base Case Let $W = \{a\}$ for $a \in \Sigma$.

$$\phi_{\{a\}}(x, y) = x = y \wedge x \in Q_a$$

Let $W = \{\epsilon\}$.

$$\phi_{\{\epsilon\}}(x, y) = y < x$$

Inductive Step Let U and V be star-free languages defined by $\phi_U(x,y)$ and $\phi_V(x,y)$.

Union $\phi_{U\cup V}(x,y) = \phi_U(x,y) \vee \phi_V(x,y)$.

Complement $\phi_{U^c}(x,y) = \neg\phi_U(x,y) \wedge x \leq y+1$ (where $y+1$ handles the empty word boundary).

Concatenation We assert the existence of a split point z:

$$\phi_{U\cdot V}(x,y) = \exists z \left(x \leq z \wedge z < y \wedge \phi_U(x,z) \wedge \phi_V(z+1,y)\right)$$

where $z+1$ is defined via the successor shortcut.

□

9.5.4 Translation: FO-S1S$[<]$ $\Longrightarrow$ Star-Free (McNaughton-Papert)

While translating star-free expressions to first-order formulas is straightforward (using split points and relativized subformulas), the converse translation is one of the most celebrated and difficult results in automata theory. The challenge lies in the fact that a first-order formula can use arbitrary quantification over positions, which has no direct local representation.

To bridge this gap, McNaughton and Papert introduced a technique to "separate" first-order formulas into past, present, and future intervals. The key tool for this analysis is a family of equivalence relations based on the nesting depth of quantifiers. If two words satisfy the same formulas up to a certain quantifier depth, they are indistinguishable to the logic. By showing that these equivalence relations have a finite number of classes, we can decompose any first-order formula into a finite concatenation of star-free components.

The converse is the landmark McNaughton-Papert Theorem.

Theorem 9.5.4 (McNaughton-Papert, 1971)**.** *A language of finite words $W \subseteq \Sigma^*$ is star-free if and only if it is definable in first-order S1S over finite intervals.*

The theorem statement is exact, but the notes below give only the proof roadmap for the difficult first-order-to-star-free direction. The missing formal details are the induction showing finite index and concatenation compatibility of the congruences; once those are established, the separation step explains how quantifiers become star-free concatenations.

To set up that roadmap, we introduce the concept of *quantifier depth* and equivalence relations.

Definition 9.5.5 (Quantifier Depth)**.** The quantifier depth $qd(\phi)$ of a formula ϕ is the maximum nesting level of quantifiers:

Atomic formulas $qd(\text{atomic}) = 0$.

Disjunction $qd(\phi \vee \psi) = \max(qd(\phi), qd(\psi))$.

Negation $qd(\neg\phi) = qd(\phi)$.

Existential quantification $qd(\exists x\phi) = qd(\phi) + 1$.

Think of quantifier depth as the number of nested "let me pick a position" moves available to the formula. A depth-0 formula only inspects a fixed finite window of positions; a depth-1 formula can name one position and reason about it; a depth-2 formula can play two such moves in sequence.

This intuition is what makes the equivalences $\equiv_n$ below work: two words are indistinguishable at depth n exactly when no depth-n formula can tell them apart.

Crucially, these equivalence classes have a *finite index*. Syntactically, you can write infinitely many formulas of depth n (for example, by repeating $\wedge$ over and over). But modulo logical equivalence, they collapse into finitely many distinct properties. This means that depth n formulas partition the infinite universe of all possible words into a finite number of buckets.

We define two equivalence relations of rank n:

Standard congruence $\equiv_n$ For $u, v \in \Sigma^*$, $u \equiv_n v$ iff they satisfy the same sentences of quantifier depth $\leq n$.

Extended congruence $\equiv_n^1$ For $(u, r), (v, s)$ where $u, v \in \Sigma^*$ and $r \in [1, |u|]$, $s \in [1, |v|]$ are distinguished positions, $(u, r) \equiv_n^1 (v, s)$ iff they satisfy the same formulas $\phi(x)$ with one free variable x of quantifier depth $\leq n$. Because we are evaluating formulas with one free variable $\phi(x)$, a plain word is not enough; we need a "pointer" (the distinguished position r) to indicate exactly where the variable x is instantiated.

Lemma 9.5.6 (Properties of $\equiv_n$ and $\equiv_n^1$). *The relations satisfy:*

Finite index *The number of equivalence classes of $\equiv_n$ and $\equiv_n^1$ is finite.*

Definability *Each equivalence class K of $\equiv_n$ can be defined by a sentence ϕ_K of quantifier depth n.*

Disjunction *Any formula $\phi(x)$ of quantifier depth n is equivalent to a finite disjunction of formulas representing classes of $\equiv_n^1$.*

Concatenation congruence *The relations are compatible with concatenation in the following two senses:*

- *If $u \equiv_n v$ and $u' \equiv_n v'$, then $uu' \equiv_n vv'$.*
- *If $u \equiv_n v$ and $u' \equiv_n v'$, then $(uau', |u| + 1) \equiv_{n+1}^1 (vav', |v| + 1)$.*

■ **Formal details — Proof roadmap for the separation step (FO → Star-Free)**

Let $\Phi = \exists x \phi(x)$ be a sentence of quantifier depth $n + 1$, where $qd(\phi) = n$. By the Disjunction property, the formula $\phi(x)$ is equivalent to a finite disjunction of class formulas of $\equiv_n^1$:

$$\phi(x) \equiv \bigvee_{W_j \models \phi} \phi_{W_j}(x)$$

Thus, the sentence is equivalent to:

$$\exists x \phi(x) \equiv \bigvee_j \exists x \phi_{W_j}(x)$$

We show that for each class W_j of $\equiv_n^1$, the language recognized by $\exists x \phi_{W_j}(x)$ is star-free.

Let W be a class of $\equiv_n^1$, and let $(u_0, r_0) \in W$ be a representative. We can decompose the word u_0 around the position r_0 as:

$$u_0 = u_L a u_R$$

where $a = u_0[r_0] \in \Sigma$ is the symbol at the distinguished position, $u_L \in \Sigma^*$ is the prefix of length $r_0 - 1$, and $u_R \in \Sigma^*$ is the suffix.

Let $U = [u_L]_{\equiv_{n-1}}$ and $V = [u_R]_{\equiv_{n-1}}$ be the equivalence classes of the prefix and suffix under the standard congruence of rank $n-1$. By the Concatenation Congruence property, for any $u \in U$ and $v \in V$, we have:

$$(uav, |u|+1) \equiv_n^1 (u_L a u_R, |u_L|+1) \in W$$

Thus, the set of words satisfying $\exists x \phi_W(x)$ is exactly the language:

$$L_W = U \cdot a \cdot V$$

By the induction hypothesis, since U and V are classes of $\equiv_{n-1}$ defined by sentences of quantifier depth $n-1$, they correspond to star-free languages. Therefore, the language $L_W = U \cdot a \cdot V$ is the concatenation of star-free languages, which is star-free. The finite union of such languages is also star-free.

Summary & Key Takeaways — First-Order Equivalence Class

The first-order fragment of S1S, star-free languages, counter-free automata (automata whose transition graphs contain no non-trivial cycles), and aperiodic monoids are all equivalent. Under infinite words, this exact class corresponds to Linear Temporal Logic (LTL).

The chapter closes with exercises that revisit the three main moves: DBA limits, richer deterministic acceptance, and first-order/star-free translations.

9.6 Exercises

Exercise 1 (DBA Characterization). Consider the language $L = (a^*b)^\omega$ over the alphabet $\Sigma = \{a, b\}$.

1. Determine whether L is recognizable by a DBA.
2. If yes, find a regular language $W \subseteq \Sigma^*$ such that $L = W^\delta$.
3. If no, prove it using the DBA characterization theorem.

Exercise 2 (Deterministic Muller Automaton). Let $\Sigma = \{a, b, c\}$. Construct a Deterministic Muller Automaton (DMA) recognizing the language of infinite words where the symbol c appears infinitely often, but the symbol a appears only finitely often. Write down the state-transition table and the family $\mathcal{F}$ of accepting subsets.

Exercise 3 (Star-Free Regular Expressions). Let $\Sigma = \{a, b\}$.

1. Write a star-free regular expression for the language of finite words that do not contain the subword ab.
2. Translate this language into an equivalent first-order S1S formula $\phi(x, y)$ over finite words.

Exercise 4 (Rabin Acceptance Pairs). Consider the 3-state deterministic transition structure over $\{a, b\}$ with states $Q = \{q_0, q_1, q_2\}$ and transition relation:

- $\delta(q_0, a) = q_1, \quad \delta(q_0, b) = q_0.$
- $\delta(q_1, a) = q_2, \quad \delta(q_1, b) = q_0.$
- $\delta(q_2, a) = q_2, \quad \delta(q_2, b) = q_0.$

Identify the Rabin accepting pairs $\Omega = \{(L_1, U_1), \ldots, (L_n, U_n)\}$ that recognize the language of words containing infinitely many occurrences of the subword aa.

CHAPTER 10

Tree Languages and Tree Automata

chapters 3 and 4 built the theory of ω-words: a single infinite word models one linear execution of a reactive system, which is the natural substrate for linear-time specifications such as LTL. Yet a reactive system is rarely committed to a single future. A non-deterministic transition relation, a concurrent scheduler, an environment that may inject inputs at any time — all of these mean that from a given configuration the system may evolve in *many* distinct ways. Verification questions like "does *every* possible run eventually grant the request?" or "is there *some* run that avoids the deadlock?" cannot be answered by inspecting one path in isolation; they quantify over the whole bundle of possible futures at once.

This chapter lifts the ω-word machinery to **ω-trees**. Just as an infinite word represents a single history, an infinite tree represents all histories of a system from a given initial configuration; the branching points are exactly the moments where the system (or its environment) makes a choice. The connection goes deeper: the parity and Muller games studied in chapter 5 are themselves played on the unfolding of an arena, which is a tree, and a strategy for either player carves a subtree out of that unfolding. So the tree model unifies three threads of the course so far — non-deterministic executions (chapter 7), monadic second-order specifications (chapters 8–9), and synthesis games (chapter 5).

To capture this branching behaviour we move from linear time to **branching time**, modelled by trees. We develop, in order: the algebraic theory of finite tree languages (regular tree expressions), top-down and bottom-up finite tree automata, Büchi and Rabin automata on infinite trees, and the logic S2S together with its weak fragment WS2S. The landmark result is Rabin's theorem: S2S is decidable, and its decision procedure rests on the closure of Rabin tree automata under complementation. The chapter closes the automata-theoretic core of Part II: the next chapter (chapter 11) will introduce Linear Temporal Logic as a tractable, syntactically restricted fragment of these very expressive — but high-complexity — monadic logics.

10.1 Introduction to Trees and Branching Time

To bridge the gap between reactive systems and trees, let us perform a thought experiment at the whiteboard. Suppose we have a non-deterministic transition system (represented as a directed graph) with states and transitions. If we

choose an initial state and "roll out" all possible executions step-by-step, we are performing a process called **unfolding** or **unraveling**.

10.1.1 Unfolding a Transition System

Consider the transition system shown on the left of fig. 10.1. It has two initial states, and the transitions from state A allow it to stay in A, go to B, or go to C. State C loops on itself deterministically, and state B transitions back to A.

If we start in the initial state A and trace all possible behaviors, we obtain the infinite computation tree shown on the right of fig. 10.1. Each branch in this tree represents a unique execution path, and the branching points represent the non-deterministic choices available to the system at that step.

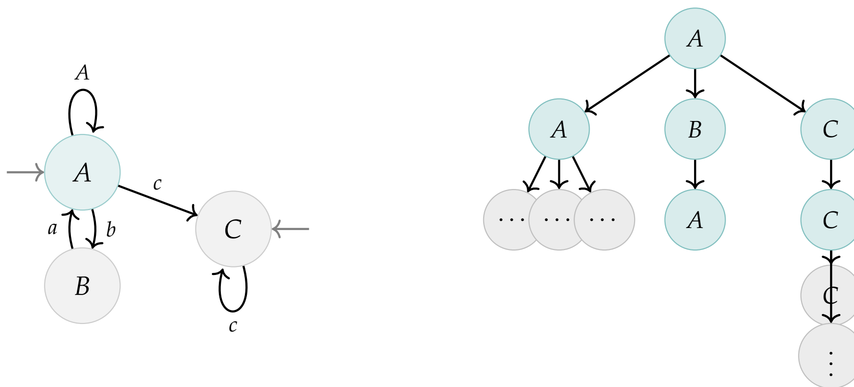


Figure 10.1: A transition system graph (left) and its infinite unfolding from initial state A into a computation tree (right).

While an LTL formula evaluates a single path in isolation, a CTL formula can reason about the tree structure directly, using quantifiers like "for all paths (∀)" or "there exists a path (∃)" at each branching point.

10.1.2 Formalizing Trees: Domains and Valuations

To study trees mathematically, we must formalize their structure. Let's think of the domain of a tree as the set of positions in the world. For words, we have the natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$ as linear time points, representing a single determined future. For trees, we replace $\mathbb{N}$ with words over branch indices, like $\mathbb{N}^*$ or $\{0, 1\}^*$. A position (or node) is represented by the sequence of choices made to reach it. For example, in a binary tree, the sequence 010 represents the time point reached by going left, then right, then left. This branching structure perfectly captures the idea of multiple possible futures.

This motivates the formal definition of a tree domain as a prefix-closed set of sequences.

> **Definition 10.1.1** (K-ary Labeled Tree)**.** Let A be a finite alphabet and let $K \geq 1$ be a natural number representing the maximum arity (branching factor) of the tree. A *K-ary A-valued tree t* is a labeling function:
>
> $$t : \text{dom}(t) \longrightarrow A$$
>
> where the set of nodes $\text{dom}(t) \subseteq \{0, \dots, K-1\}^*$ is called the *domain* of t. The domain must satisfy the following conditions:

1. **Non-emptiness**: $\mathrm{dom}(t) \neq \emptyset$, which implies that the root node ϵ (the empty word) always belongs to the domain.
2. **Prefix Closure**: For every word $w \in \{0, \ldots, K-1\}^*$ and symbol $j < K$, if $wj \in \mathrm{dom}(t)$, then $w \in \mathrm{dom}(t)$.
3. **Width Closure (Child Ordering)**: For every word $w \in \{0, \ldots, K-1\}^*$ and symbols $i, j < K$, if $wj \in \mathrm{dom}(t)$ and $i < j$, then $wi \in \mathrm{dom}(t)$.

Whiteboard Explanation of the Domain Conditions. Let us trace what these conditions mean operationally:

- **Prefix closure** ensures that we cannot have "floating" nodes. If a node exists in the tree, its parent must also exist. You cannot reach a position like 01 without first traversing through the root ϵ and the node 0.
- **Width closure** ensures that children are ordered and contiguous starting from index 0. An node cannot have a "second child" (labeled j) without also having all preceding children (labeled $i < j$). For instance, in a binary tree, we cannot have a right child ($w1$) unless the left child ($w0$) also exists.

10.1.3 Paths, Subtrees, and Frontiers

Now we define the key structural properties of a tree.

Definition 10.1.2 (Paths and Subtrees)**.** Let t be a K-ary A-valued tree:

- A *path* π through t is a maximal subset of $\mathrm{dom}(t)$ that is linearly ordered by the prefix relation.
- For any node $w \in \mathrm{dom}(t)$, the *subtree* of t at node w, denoted by $t|_w$, is the tree whose domain is:

$$\mathrm{dom}(t|_w) = \{v \in \{0, \ldots, K-1\}^* \mid wv \in \mathrm{dom}(t)\}$$

and whose labeling function is defined as $t|_w(v) = t(wv)$ for all $v \in \mathrm{dom}(t|_w)$.

Operational meaning of Subtrees. Notice how the definition of $\mathrm{dom}(t|_w)$ works operationally: it takes every node below w (which looks like wv) and strips away the w prefix. This mathematically "shifts" the entire subtree upwards so that its new root is always the empty word ϵ. The root of the new subtree $t|_w(\epsilon)$ gets the value $t(w)$, perfectly preserving the original labeling relative to the new origin.

To define tree automata, we need to specify where the run of the automaton terminates or accepts. For a word of length n, the run is of length $n+1$, terminating at the position after the last letter. For trees, we generalize this using the concepts of **frontier** (the set of leaves) and **outer frontier** (the set of positions immediately below the leaves).

Observe that the domain of a subtree $t|_w$ always resets to start at the empty word ϵ. The root of the subtree is labeled with the value that t had at position w.

Definition 10.1.3 (Frontiers)**.** Let t be a K-ary A-valued tree:

- The *frontier* of t (the set of leaves), denoted by $\mathrm{fr}(t)$, is the set:

$$\mathrm{fr}(t) = \{w \in \mathrm{dom}(t) \mid \forall j < K,\ wj \notin \mathrm{dom}(t)\}$$

- The *outer frontier* of t, denoted by $\mathrm{fr}^+(t)$, contains the positions immediately succeeding the leaves that are not part of the domain:

$$\mathrm{fr}^+(t) = \{wj \in \{0, \dots, K-1\}^* \setminus \mathrm{dom}(t) \mid w \in \mathrm{dom}(t) \wedge j < K\}$$

- The *extended domain* of t is defined as $\mathrm{dom}^+(t) = \mathrm{dom}(t) \cup \mathrm{fr}^+(t)$.

To visualize these definitions, let us look at the finite binary tree ($K = 2$) in fig. 10.2.

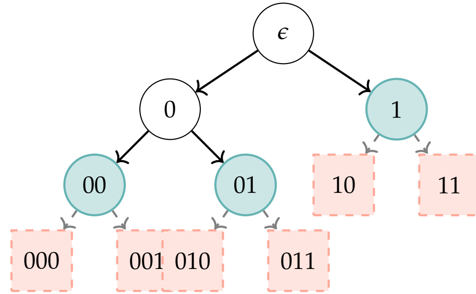


Figure 10.2: A finite binary tree. The domain consists of $\{\epsilon, 0, 1, 00, 01\}$. The frontier (leaves) consists of $\{1, 00, 01\}$ (highlighted in teal). The outer frontier consists of $\{10, 11, 000, 001, 010, 011\}$ (dashed squares, highlighted in orange).

10.1.4 Simplifying Focus: Binary Coding of K-ary Trees

Studying K-ary trees for arbitrary K complicates notations and proofs without adding conceptual value. Without loss of generality, we can restrict our study to **binary trees** ($K = 2$). Any K-ary tree can be encoded into a binary tree over an extended alphabet by introducing a dummy padding symbol Ω.

Definition 10.1.4 (Binary Coding of K-ary Trees)**.** Let t be a K-ary A-valued tree. Let Ω be a fresh symbol not in A. The binary coding of t is a binary tree $t_{bin} : \mathrm{dom}(t_{bin}) \longrightarrow A \cup \{\Omega\}$ constructed as follows:

1. We map each node $w = n_1 n_2 \dots n_r \in \mathrm{dom}(t)$ (where $n_i < K$) to a binary sequence using the injection $f : \{0, \dots, K-1\}^* \longrightarrow \{0, 1\}^*$:

$$f(n_1 n_2 \dots n_r) = 1^{n_1} 0 1^{n_2} 0 \dots 1^{n_r} 0$$

   where 1^k denotes k consecutive 1s.
2. The domain $\mathrm{dom}(t_{bin})$ is the prefix closure of the image $f(\mathrm{dom}(t))$.
3. The valuation t_{bin} is defined as:

$$t_{bin}(w') = \begin{cases} t(w) & \text{if } w' = f(w) \text{ for some } w \in \mathrm{dom}(t) \\ \Omega & \text{otherwise} \end{cases}$$

Example 10.1.5 (Encoding a Node)**.** Let $K = 3$ and consider a node $w = 201$ in the 3-ary tree. Its binary encoding is:

$$f(201) = 1^2 0 1^0 0 1^1 0 = 110010$$

At this position in the binary tree, $t_{bin}(110010) = t(201)$. The intermediate nodes along the path (such as 1, 11, 110, 1100, 11001) are labeled with the dummy symbol Ω.

From now on, we define:

- $T(A)$ as the set of all finite binary A-valued trees.
- $T_\omega(A)$ as the set of all infinite binary A-valued trees (whose domain is $\{0, 1\}^*$).
- $T_\infty(A) = T(A) \cup T_\omega(A)$ as the set of all binary A-valued trees.

This encoding is a prefix-preserving bijection between the original nodes and a subset of the binary tree. It can be extended to trees with countably infinite branching arity. Thus, we will assume all trees are binary ($K = 2$) unless specified otherwise.

A subset of $T(A)$ or $T_\omega(A)$ is called a **tree language**.

With domains, paths, subtrees, and frontiers fixed, we can now ask the same question that starts the theory of regular word languages: which tree languages have a finite description? The first answer is algebraic.

10.2 Regular Tree Languages and Concatenation

In the classical theory of formal languages over finite words, the Kleene theorem establishes a fundamental bridge between operational models (finite automata) and algebraic expressions (regular expressions). Regular expressions are built using three core operators: union, concatenation, and the Kleene star.

To generalize this framework to trees, we must redefine these algebraic operators. Union is straightforward: it is simply the set-theoretic union of tree languages. However, defining concatenation is not trivial. In words, a word has a single, unique end position where we can append another word. In contrast, a finite tree can have many leaves (its frontier). When we concatenate a tree t' to a tree t, we must decide *which* leaf of t to replace.

To solve this, we define tree concatenation as a **substitution** operation. We designate specific symbols from a set of placeholders C to represent "attachment points" on the leaves of the tree.

10.2.1 Tree Concatenation (Substitution)

Let A be an alphabet and let C be a set of placeholders disjoint from A. Trees are built over the alphabet $A \cup C$. Concatenation is defined with respect to a specific placeholder symbol $c \in C$.

Before we get to the dense mathematical formulation, let's understand the operation intuitively. To concatenate a tree t' to a tree t, we simply search the frontier (leaves) of t for any leaf labeled with the placeholder c. We then "graft" the entire tree t' onto t by replacing that leaf with the root of t'. If there are multiple leaves labeled c, we can graft a different tree from our target set onto each one.

Definition 10.2.1 (Tree Concatenation). Let $T \subseteq T(A \cup C)$ be a language of finite trees, and let $T' \subseteq T_\infty(A \cup C)$. The *concatenation of T' with T with respect to c*, denoted by $T \cdot_c T'$, is the set of all trees obtained by taking a tree $t \in T$ and, for every leaf $w \in \mathrm{fr}(t)$ labeled with c, substituting the leaf w with a tree $t'_w \in T'$.

Mathematically, for each leaf w labeled c, we can choose a different tree $t'_w \in T'$. The resulting tree t_{new} has the domain:

$$\mathrm{dom}(t_{new}) = (\mathrm{dom}(t) \setminus \{w \in \mathrm{fr}(t) \mid t(w) = c\}) \cup \bigcup_{w \in \mathrm{fr}(t),\, t(w)=c} \{wv \mid v \in \mathrm{dom}(t'_w)\}$$

and the valuation:

$$t_{new}(z) = \begin{cases} t(z) & \text{if } z \in \mathrm{dom}(t) \text{ and } z \text{ is not a descendant of any leaf labeled } c \\ t'_w(v) & \text{if } z = wv \text{ where } t(w) = c \text{ and } v \in \mathrm{dom}(t'_w) \end{cases}$$

Example 10.2.2 (Concatenation of Tree Languages). Let $A = \{a, b, f\}$ and $C = \{c\}$. Let T and T' be two tree languages defined as:

$$T = \{t_1, t_2\}, \quad T' = \{t_3, t_4\}$$

where:

- $t_1 = f(a, c)$ (a root f with left child a and right child c),
- $t_2 = b$ (a single leaf node labeled b),
- $t_3 = a$ (a single leaf node labeled a),
- $t_4 = f(b, b)$ (a root f with two children labeled b).

Let us compute the concatenation $T \cdot_c T'$. For each tree in T:

- For $t_2 = b$, there are no occurrences of the placeholder c on the frontier. Thus, substitution yields the tree itself: $t_2 \cdot_c T' = \{b\}$.
- For $t_1 = f(a, c)$, the right child is labeled with the placeholder c. We substitute this leaf with any tree from T'. Since T' contains two candidate trees, we obtain two possible trees:

$$t_{1a} = f(a, t_3) = f(a, a)$$
$$t_{1b} = f(a, t_4) = f(a, f(b, b))$$

Taking the union of all possible resulting trees, we get:

$$T \cdot_c T' = \{b,\ f(a, a),\ f(a, f(b, b))\}$$

This example demonstrates that when concatenating sets of trees, different occurrences of the placeholder can be substituted by different elements from the target set, and trees without placeholders are preserved unchanged.

An intuitive visualization of this substitution is shown in fig. 10.3.

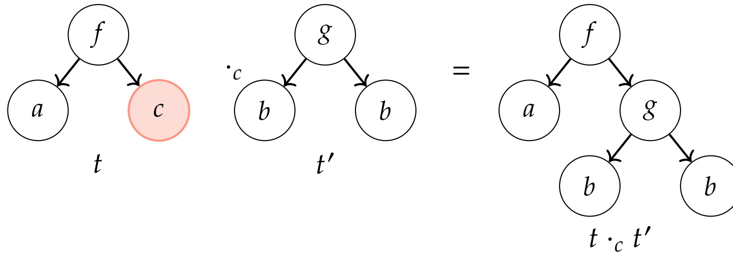


Figure 10.3: Tree concatenation as substitution: the leaf labeled with the placeholder c in tree t is replaced by the tree t'.

10.2.2 Tree Kleene Star (Iterated Substitution)

Having defined single substitution, we can now define iterated substitution. This corresponds to the Kleene star.

Imagine growing a tree from a seed. The seed is just a placeholder c. At step 1, you replace the seed with a small tree (say, $f(c,c)$). At step 2, you replace the new c leaves with more small trees, growing it larger. Because you take the union of all these steps, the language contains trees of all possible finite sizes grown this way.

Definition 10.2.3 (Tree Kleene Star)**.** Let $T \subseteq T(A \cup C)$ be a language of finite trees and let $c \in C$. The *Kleene star of* T *with respect to* c, denoted by $T^{*,c}$, is defined as the union:

$$T^{*,c} = \bigcup_{n \geq 0} T^{n,c}$$

where the sequence of approximations $T^{n,c}$ is defined inductively as:

$$T^{0,c} = \{c\}$$
$$T^{n+1,c} = T \cdot_c T^{n,c} \cup T^{n,c}$$

Example 10.2.4 (Traces of the Tree Kleene Star)**.** Let $A = \{a, f\}$ and $C = \{c\}$, and let $T = \{f(c,c), a\}$. Let us trace the approximations $T^{n,c}$ for the iterated substitution $T^{*,c}$:

- **Base Case (**$n = 0$**):**

  $$T^{0,c} = \{c\}$$

  This represents a single root node labeled with the placeholder c.
- **First Iteration (**$n = 1$**):**

  $$T^{1,c} = T \cdot_c T^{0,c} \cup T^{0,c} = \{f(c,c), a\} \cdot_c \{c\} \cup \{c\} = \{f(c,c), a, c\}$$

- **Second Iteration (**$n = 2$**):**

  $$T^{2,c} = T \cdot_c T^{1,c} \cup T^{1,c}$$

  To compute $T \cdot_c T^{1,c}$, we take each tree in T and replace its c-leaves with any choice from $T^{1,c} = \{f(c,c), a, c\}$:

- For $a \in T$, there are no c leaves, so we get $\{a\}$.
- For $f(c, c) \in T$, we have two leaves labeled c. We can replace them independently with any combination of trees from $\{f(c, c), a, c\}$. This yields:
  * Replacing both with a: $f(a, a)$
  * Replacing left with a, right with c: $f(a, c)$
  * Replacing left with c, right with a: $f(c, a)$
  * Replacing left with a, right with $f(c, c)$: $f(a, f(c, c))$
  * Replacing both with $f(c, c)$: $f(f(c, c), f(c, c))$
  * ... and so on, for all 9 possible combinations.

Taking the union with $T^{1,c}$, we see that $T^{2,c}$ contains trees of varying shapes and branching configurations (such as $f(a, c)$ where one branch stopped growing while the other was replaced by the placeholder c, or $f(a, a)$ where all placeholders have been resolved to terminals).

Formal details — Why is the Tree Kleene Star Cumulative?

Observe that in the inductive step, we define $T^{n+1,c}$ by taking the union with $T^{n,c}$ (making the sequence cumulative). This is in stark contrast to words, where the Kleene star is defined by simply concatenating $n + 1$ words, and the union is taken only at the very end.

Why is this cumulative union required for trees? In words, concatenation only occurs at the single end of the string, meaning all parts of the word grow uniformly in length. In trees, however, each step of the substitution replaces *all* occurrences of the placeholder c on the frontier. If we did not include the cumulative union $T^{n,c}$ at each step, we would be forced to substitute c at all leaves at the exact same iteration step. This would imply that all branches of the tree must grow to the exact same depth, preventing us from generating trees with branches of different lengths (e.g., one branch terminating early while another continues to branch). Including $T^{n,c}$ in the inductive step acts as a "stopping option" for any leaf labeled c, allowing different branches to decouple their growth and generate arbitrary irregular tree shapes.

10.2.3 Regular Tree Languages

With these definitions, we can now formalize regular tree languages over finite trees.

Definition 10.2.5 (Regular Tree Language)**.** A tree language $T \subseteq T(A)$ is *regular* if and only if there exists a finite set of placeholder symbols C disjoint from A such that T can be obtained from finite subsets of $T(A \cup C)$ by applying a finite number of times the operations of:

1. Union $(T_1 \cup T_2)$,
2. Concatenation with respect to $c \in C$ $(T_1 \cdot_c T_2)$,
3. Kleene star with respect to $c \in C$ $(T_1^{*,c})$.

Example 10.2.6 (Algebraic Expression for All Finite Binary Trees)**.** Suppose we want to define the language of all finite binary trees over the alphabet $A = \{a, b\}$. A tree in this language is either a single terminal leaf (labeled a or b) or a node (labeled a or b) with two child subtrees which are themselves finite binary trees.

Using a single placeholder c, we can define the finite set of base trees:

$$B = \{a,\ b,\ a(c, c),\ b(c, c)\}$$

The regular tree language $B^{*,c}$ contains exactly all finite binary trees over A. Let's see why:

- At each step of the Kleene star substitution, we can choose to replace the active placeholder leaves c either with terminals a or b (which terminates that branch), or with branching patterns $a(c, c)$ or $b(c, c)$ (which extends the branch and introduces new placeholders).
- For example, the tree $a(b, a(b, b))$ is generated by:

$$c \xrightarrow{B} a(c, c) \xrightarrow{B} a(b, c) \xrightarrow{B} a(b, a(c, c)) \xrightarrow{B} a(b, a(b, b))$$

  which belongs to $B^{4,c} \subseteq B^{*,c}$.
- Since all placeholders are eventually replaced by a or b, the final trees contain only symbols from A, making $B^{*,c}$ a valid regular tree language over A.

Since the final regular language T is a subset of $T(A)$, it consists only of trees labeled by A. This implies that all placeholder symbols from C must be completely substituted away during the algebraic construction.

10.2.4 Tuple Concatenation and Omega-Closure

In many constructions, we need to perform simultaneous substitution of multiple different placeholder symbols. This is modeled by **tuple concatenation**.

Definition 10.2.7 (Tuple Concatenation)**.** Let $C = \{c_1, \ldots, c_m\}$ be a set of placeholders. Let $T \subseteq T(A \cup C)$ and let $T_1, \ldots, T_m \subseteq T_\infty(A \cup C)$ be tree languages. The *tuple concatenation*, denoted by:

$$T \cdot_{\bar{c}} (T_1, \ldots, T_m)$$

is the tree language obtained by taking a tree $t \in T$ and, for each $i \in \{1, \ldots, m\}$, replacing every leaf labeled c_i on the frontier of t with some tree from T_i.

Using tuple concatenation, we can define the infinite iteration of tree substitution, leading to the ω**-closure** of tree languages, which is the algebraic representation of infinite trees.

Definition 10.2.8 (Tree Omega-Closure)**.** Let $C = \{c_1, \ldots, c_m\}$ be placeholders. Let $T_1, \ldots, T_m \subseteq T(A \cup C)$ be languages of finite trees. The *ω-closure* with respect to $\bar{c}$, denoted by:

$$(T_1, \ldots, T_m)^{\omega,\bar{c}}$$

is the set of all infinite trees obtained by applying the tuple concatenation infinitely many times starting from the initial set of placeholders.

Example 10.2.9 (Alternating Paths via Omega-Closure)**.** Suppose we want to generate the set of all infinite binary trees where every path alternates between the symbols a and b, starting with a at the root. We can define this language algebraically. Let $C = \{c_a, c_b\}$ be placeholders. We define the base sets:

$$T_a = \{a(c_b, c_b)\}$$
$$T_b = \{b(c_a, c_a)\}$$

The language of alternating trees is exactly the ω-closure:

$$T_{alt} = (T_a, T_b)^{\omega,(c_a,c_b)}$$

In this construction, we start with the placeholder c_a. It is replaced by $a(c_b, c_b)$ (which introduces c_b at both children). In the next step, both occurrences of c_b are replaced by $b(c_a, c_a)$, which introduces c_a at all four grandchildren. Repeating this infinitely yields an infinite binary tree where every path alternates a and b.

10.2.5 Concatenation of Infinite Trees

If a tree is infinite, it does not have a frontier (its set of leaves is empty). How can we define concatenation for infinite trees?

To solve this, we define concatenation by substituting the *first* occurrence of the placeholder c along each path.

Definition 10.2.10 (Infinite Tree Concatenation)**.** Let $t \in T_\omega(A \cup C)$ be an infinite tree, and let $T' \subseteq T_\omega(A \cup C)$. The concatenation $t \cdot_c T'$ is the set of trees obtained by replacing the subtree $t|_w$ with some tree in T' for each minimal node $w \in \text{dom}(t)$ (with respect to prefix ordering) such that $t(w) = c$.

This ensures that we perform substitution at the boundary where the placeholder c first appears along each infinite history.

These operations give finite recipes for building tree languages, but finite recipes are only half of the regular-language story. For words, Kleene's theorem says that regular expressions and finite automata define the same languages; the expression is the algebraic view, and the automaton is the recognizer that can be executed on an input. The same question is natural here. Once concatenation has become substitution, and Kleene star has become iterated substitution at many frontier leaves, what finite machine recognizes exactly the languages generated by those operations?

The answer is given by finite tree automata. Their runs are themselves trees: a state is placed at each node, and local transition constraints check that the state assignment is consistent with the input labels. The rest of this section develops the two possible directions of recognition and then returns to the Kleene-style equivalence between regular tree expressions and automata.

10.3 Tree Automata on Finite Trees

Having established the algebraic representation of tree languages (regular tree expressions), we now introduce the corresponding operational model: **tree automata**.

Just as a standard automaton processes a word by reading symbols sequentially and transitioning between states, a tree automaton processes a tree by traversing its branching structure. Over finite trees, we can process the structure in two directions:

1. **Top-Down**: We start at the root and work down towards the leaves. At each node, the automaton reads the label and splits its state, spawning two independent state processes to run on the left and right child subtrees.
2. **Bottom-Up**: We start at the leaves (outer frontier) and work up towards the root. At each node, the automaton reads the states reached by its children and combines them with the current node's label to produce a state for the parent.

As we will see, this choice of direction represents a crucial fork in the road: top-down deterministic automata are strictly weaker than their non-deterministic counterparts, whereas bottom-up deterministic and non-deterministic automata are expressively equivalent.

10.3.1 Top-Down Tree Automata

We begin by defining the top-down model.

Definition 10.3.1 (Non-deterministic Top-down Tree Automaton)**.** A *non-deterministic top-down tree automaton* (NTTA) over finite binary trees is a tuple:

$$A = (Q, \Sigma, Q_0, \Delta, F)$$

where:

- Q is a finite set of states,
- Σ is the input alphabet,
- $Q_0 \subseteq Q$ is the set of initial states (applied at the root),
- $F \subseteq Q$ is the set of final states (checked at the leaves),
- $\Delta \subseteq Q \times \Sigma \times Q \times Q$ is the transition relation.

A transition $(q, a, q_L, q_R) \in \Delta$ (often written $q \xrightarrow{a} (q_L, q_R)$) means that if the automaton is in state q at a node labeled a, it sends state q_L to the left child and state q_R to the right child.

Definition 10.3.2 (NTTA Run and Acceptance)**.** Let $t \in T(\Sigma)$ be a finite binary tree. A *run* of the NTTA A on t is a tree $R : \mathrm{dom}^+(t) \to Q$ labeled with states, which is defined over the extended domain of t, satisfying:

1. **Initiality**: The root of the run is labeled with an initial state:

$$R(\epsilon) \in Q_0$$

2. **Consequentiality**: For every node $w \in \mathrm{dom}(t)$ (excluding the outer frontier), the transitions must obey the transition relation:

$$(R(w), t(w), R(w0), R(w1)) \in \Delta$$

The run R is *successful* if all leaves of the run (which are the nodes in the outer frontier of t) are labeled with final states:

$$\forall w \in \mathrm{fr}^+(t),\ R(w) \in F$$

The tree t is accepted by A if there exists at least one successful run of A on t. The language recognized by A is denoted by $L(A)$.

Notice that the run R is defined on the extended domain $dom^+(t) = dom(t) \cup fr^+(t)$. This is because, like in the case of finite words, we need an extra "look-ahead" step at the leaves to check if we can transition into final states.

Example 10.3.3 (NTTA Run Visualization)**.** Let us visualize a run on the tree $t = f(a, b)$, which has the domain $\mathrm{dom}(t) = \{\epsilon, 0, 1\}$ and outer frontier $\mathrm{fr}^+(t) = \{00, 01, 10, 11\}$.

A run R of the automaton must assign states to all nodes in $\mathrm{dom}^+(t)$, as shown in fig. 10.4.

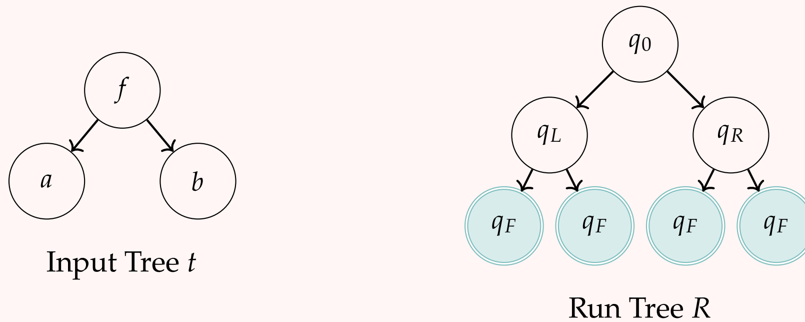


Figure 10.4: Side-by-side comparison of an input tree t of height 1 and its corresponding run tree R of height 2 over the extended domain.

10.3.2 The Expressive Limit of Deterministic Top-Down Automata

A top-down tree automaton is **deterministic** (DTTA) if:

1. There is exactly one initial state: $|Q_0| = 1$,
2. For every state $q \in Q$ and symbol $a \in \Sigma$, there is *at most one* pair $(q_L, q_R) \in Q \times Q$ such that $(q, a, q_L, q_R) \in \Delta$.

In the case of finite words, deterministic and non-deterministic finite automata (DFAs and NFAs) are equivalent. Surprisingly, this equivalence breaks for top-down tree automata. We first demonstrate a language that is easily recognizable deterministically, and then a language that highlights the limit of top-down determinism.

Example 10.3.4 (Path Property: Deterministic Top-Down Tree Automaton)**.** Let $\Sigma = \{a, b\}$ and consider the language $T_{\geq 1,a}$ of all finite binary trees where *every* path from the root to a leaf contains at least one node labeled a. We can design a DTTA for $T_{\geq 1,a}$ as follows:

- States: $Q = \{q_0, q_a\}$, where q_0 represents the state before seeing an a on the path, and q_a represents the state after seeing an a.
- Initial state: $Q_0 = \{q_0\}$.
- Final states: $F = \{q_a\}$ (meaning we must have seen an a before reaching the leaf).
- Transitions:

$$q_0 \xrightarrow{a} (q_a, q_a) \quad \text{and} \quad q_0 \xrightarrow{b} (q_0, q_0)$$
$$q_a \xrightarrow{a} (q_a, q_a) \quad \text{and} \quad q_a \xrightarrow{b} (q_a, q_a)$$

Let us trace this automaton on a tree. At the root, we start in q_0. If we see b, we propagate q_0 down to both children. If we see a, we transition to q_a and propagate q_a down. Once a branch transitions to q_a, it stays in q_a regardless of subsequent labels. Since we check that all leaves in the outer frontier are in $F = \{q_a\}$, the tree is accepted if and only if every single path contains at least one a. This automaton is deterministic because for each state and label, there is exactly one target pair.

Theorem 10.3.5 (Top-Down Deterministic Limit)**.** *Deterministic top-down tree automata (DTTA) are strictly less expressive than non-deterministic top-down tree automata (NTTA):*

$$DTTA \subset NTTA$$

Proof. Clearly, DTTA $\subseteq$ NTTA by definition. To prove that the inclusion is strict, we show that the finite tree language:

$$T_{xor} = \{f(a, b),\ f(b, a)\}$$

is recognizable by an NTTA but cannot be recognized by any DTTA.

1. **NTTA Construction**: We construct an NTTA with states $Q = \{q_0, q_a, q_b, q_F\}$, initial state $Q_0 = \{q_0\}$, final states $F = \{q_F\}$, and transitions:

$$q_0 \xrightarrow{f} (q_a, q_b) \quad \text{and} \quad q_0 \xrightarrow{f} (q_b, q_a)$$
$$q_a \xrightarrow{a} (q_F, q_F)$$
$$q_b \xrightarrow{b} (q_F, q_F)$$

   This automaton easily accepts $f(a, b)$ by choosing the first transition, and $f(b, a)$ by choosing the second transition. All other runs get stuck and reject. Thus, $L(A) = T_{xor}$.

2. **Proof of Unrecognizability by DTTA**: Assume for contradiction that there exists a DTTA $A' = (Q', \Sigma, \{q_0'\}, \delta', F')$ recognizing T_{xor}. Since $f(a, b) \in L(A')$, there must exist a unique transition for the root:

$$q_0' \xrightarrow{f} (q_L', q_R')$$

   such that q_L' accepts the left child leaf a and q_R' accepts the right child leaf b. This means:

$$(q_L', a, q_F', q_F') \in \delta' \quad \text{and} \quad (q_R', b, q_F', q_F') \in \delta'$$

for some final states in F'.
Similarly, since $f(b,a) \in L(A')$, the same unique root transition $q'_0 \xrightarrow{f} (q'_L, q'_R)$ must be used (as A' is deterministic). This implies that q'_L must accept the leaf b and q'_R must accept the leaf a:

$$(q'_L, b, q'_F, q'_F) \in \delta' \quad \text{and} \quad (q'_R, a, q'_F, q'_F) \in \delta'$$

Now, consider the trees $t_{aa} = f(a,a)$ and $t_{bb} = f(b,b)$. If we run A' on t_{aa}, the root transitions to (q'_L, q'_R). The state q'_L accepts the left leaf a (since q'_L accepts a), and the state q'_R accepts the right leaf a (since q'_R accepts a). Thus, A' accepts t_{aa}. By symmetric reasoning, A' also accepts t_{bb}. However, $t_{aa}, t_{bb} \notin T_{xor}$. This is a contradiction, since $L(A') = T_{xor}$.

Thus, no DTTA can recognize T_{xor}, proving DTTA ⊂ NTTA. □

Why does determinism fail top-down? A top-down automaton is forced to split its state at a node before *it reads the leaves below. A deterministic automaton cannot relate the choices made on the left branch with the choices made on the right branch; once the split happens, the two branches run in complete isolation. Non-determinism bypasses this by allowing the automaton to guess the global correlation (e.g., guessing whether the tree is $f(a,b)$ or $f(b,a)$) at the root.*

10.3.3 Bottom-Up Tree Automata

To restore the equivalence between determinism and non-determinism, we turn to the **bottom-up** model. A bottom-up automaton reads the tree starting from the leaves and moves up to the root.

Operationally, think of a bottom-up automaton like evaluating a syntax tree for an arithmetic expression. You cannot evaluate the root "+" until you have evaluated its left and right subtrees. The automaton starts at the outer frontier (the leaves), assigns initial states, and then works upwards. At every parent node, it reads the states passed up from its left and right children, looks at the symbol at the current node, and "merges" this information into a single new state that it passes further up to its own parent.

Definition 10.3.6 (Non-deterministic Bottom-up Tree Automaton)**.** A *non-deterministic bottom-up tree automaton* (NBTA) over finite binary trees is a tuple:

$$A = (Q, \Sigma, Q_0, \Delta, F)$$

where:

- Q is a finite set of states,
- Σ is the input alphabet,
- $Q_0 \subseteq Q$ is the set of initial states (applied at the outer frontier),
- $F \subseteq Q$ is the set of final states (checked at the root),
- $\Delta \subseteq Q \times Q \times \Sigma \times Q$ is the transition relation.

A transition $(q_L, q_R, a, q) \in \Delta$ (written $(q_L, q_R) \xrightarrow{a} q$) means that if the left child is in state q_L, the right child is in state q_R, and the parent node is labeled a, the parent transitions to state q.

Definition 10.3.7 (NBTA Run and Acceptance)**.** Let $t \in T(\Sigma)$ be a finite binary tree. A *run* of the NBTA A on t is a tree $R : \text{dom}^+(t) \to Q$ satisfying:

1. **Initiality**: All leaf nodes of the run (the outer frontier $\text{fr}^+(t)$) are labeled with initial states:

$$\forall w \in \text{fr}^+(t),\ R(w) \in Q_0$$

2. **Consequentiality**: For every node $w \in \text{dom}(t)$, the parent state $R(w)$ is determined by the children's states $R(w0)$, $R(w1)$ and the node label $t(w)$:
$$(R(w0), R(w1), t(w), R(w)) \in \Delta$$

The run R is *successful* if the root node of the run is labeled with a final state:
$$R(\epsilon) \in F$$

A bottom-up tree automaton is **deterministic** (DBTA) if $|Q_0| = 1$ and for every $q_L, q_R \in Q$ and $a \in \Sigma$, there is *at most one* state $q \in Q$ such that $(q_L, q_R, a, q) \in \Delta$.

10.3.4 Bottom-Up Determinization (Subset Construction)

Unlike the top-down case, bottom-up tree automata can always be determinized. This is because a bottom-up transition can collect and merge information from the children as it propagates upwards.

Theorem 10.3.8 (Bottom-Up Determinization)**.** *Deterministic bottom-up tree automata (DBTA) are expressively equivalent to non-deterministic bottom-up tree automata (NBTA):*
$$DBTA \equiv NBTA$$

Proof. Clearly, DBTA ⊆ NBTA. To prove the converse, let $A = (Q, \Sigma, Q_0, \Delta, F)$ be an NBTA. We construct an equivalent DBTA $A' = (Q', \Sigma, Q'_0, \delta', F')$ using a generalization of the subset construction:

- **States**: $Q' = \mathcal{P}(Q)$ (the power set of Q).
- **Initial State**: $Q'_0 = \{Q_0\}$ (the single initial state of A' is the set of all initial states of A).
- **Transition Function**: For any subset of states $S_L, S_R \subseteq Q$ and symbol $a \in \Sigma$:
$$\delta'(S_L, S_R, a) = \{q \in Q \mid \exists q_L \in S_L,\ q_R \in S_R \text{ such that } (q_L, q_R, a, q) \in \Delta\}$$
- **Final States**: $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$.

We prove by induction on the height of the tree t that the state reached at the root of t in the unique run of A' is exactly the set of all states that can be reached at the root of t in some run of A.

- **Base Case**: Let t be a single leaf labeled a (height 0). The domain is $\{\epsilon\}$ and the outer frontier is $\{0, 1\}$. In the DBTA A', the children 0 and 1 are labeled with Q_0. The root transitions to:
$$S' = \delta'(Q_0, Q_0, a) = \{q \in Q \mid \exists q_L, q_R \in Q_0,\ (q_L, q_R, a, q) \in \Delta\}$$
This is precisely the set of states that the NBTA A can reach at the root of t starting from initial states in the outer frontier.
- **Inductive Step**: Let $t = a(t_L, t_R)$ be a tree where the subtrees t_L, t_R have height at most h. By the induction hypothesis, the unique run of A' on t_L (resp. t_R) reaches the subset of states S_L (resp. S_R), which contains exactly the states reachable by A at the root of t_L (resp. t_R). When evaluating the root of t, the DBTA A' computes:
$$S' = \delta'(S_L, S_R, a)$$

By definition of δ', $q \in S'$ if and only if there exist $q_L \in S_L$ and $q_R \in S_R$ such that $(q_L, q_R, a, q) \in \Delta$. Since S_L and S_R are the exact sets of reachable states for t_L and t_R, S' is the exact set of states that A can reach at the root of t.

Finally, A' accepts $t \iff S' \in F' \iff S' \cap F \neq \emptyset$, which is true if and only if there exists a successful run of A on t. Thus, $L(A') = L(A)$. □

Example 10.3.9 (Determinizing the NBTA for T_{xor})**.** Let us trace the subset construction on the NBTA for $T_{xor} = \{f(a,b), f(b,a)\}$. Recall the NBTA states are $Q = \{q_0, q_a, q_b, q_{acc}\}$, with $Q_0 = \{q_F\}$, $F = \{q_{acc}\}$, and transitions:

$$(q_F, q_F) \xrightarrow{a} q_a$$
$$(q_F, q_F) \xrightarrow{b} q_b$$
$$(q_a, q_b) \xrightarrow{f} q_{acc}$$
$$(q_b, q_a) \xrightarrow{f} q_{acc}$$

We construct the DBTA A':

- The initial state of A' is the set $S_0 = \{q_F\}$.
- If we evaluate a leaf node labeled a, its children (in the outer frontier) are in state $S_0 = \{q_F\}$. The DBTA transitions to:

$$\delta'(\{q_F\}, \{q_F\}, a) = \{q \in Q \mid (q_F, q_F) \xrightarrow{a} q\} = \{q_a\}$$

- Similarly, for a leaf node labeled b, the DBTA transitions to:

$$\delta'(\{q_F\}, \{q_F\}, b) = \{q \in Q \mid (q_F, q_F) \xrightarrow{b} q\} = \{q_b\}$$

- Now, consider the root node labeled f of the tree $t = f(a,b)$. The left subtree (leaf a) reaches state $\{q_a\}$, and the right subtree (leaf b) reaches state $\{q_b\}$. The DBTA transitions to:

$$\delta'(\{q_a\}, \{q_b\}, f) = \{q \in Q \mid \exists q_L \in \{q_a\}, q_R \in \{q_b\} \text{ s.t. } (q_L, q_R) \xrightarrow{f} q\} = \{q_{acc}\}$$

- For the tree $t' = f(b,a)$, the left subtree reaches state $\{q_b\}$, and the right subtree reaches state $\{q_a\}$. The DBTA transitions to:

$$\delta'(\{q_b\}, \{q_a\}, f) = \{q_{acc}\}$$

- If we have $t_{aa} = f(a,a)$, both children reach state $\{q_a\}$. The transition evaluates to:

$$\delta'(\{q_a\}, \{q_a\}, f) = \emptyset$$

  which is not an accepting state because $\emptyset \cap \{q_{acc}\} = \emptyset$.

Since $\{q_{acc}\} \cap F \neq \emptyset$, the DBTA accepts $f(a,b)$ and $f(b,a)$ but rejects $f(a,a)$ and $f(b,b)$ as desired, successfully determinizing the system.

10.3.5 Decision Problems and Closure Properties

We conclude this section by stating the algorithmic and algebraic properties of finite tree languages.

Theorem 10.3.10 (Decidability of Emptiness). *The emptiness problem for tree automata (top-down or bottom-up) is decidable in polynomial time.*

Proof. We can solve the emptiness problem by computing the set of reachable states in a bottom-up fashion. Let $A = (Q, \Sigma, Q_0, \Delta, F)$ be an NBTA. We define the sequence of sets of reachable states $Reach_i \subseteq Q$:

$$Reach_0 = Q_0$$
$$Reach_{i+1} = Reach_i \cup \{q \in Q \mid \exists q_L, q_R \in Reach_i,\ a \in \Sigma \text{ such that } (q_L, q_R, a, q) \in \Delta\}$$

Since Q is finite, this sequence must reach a fixed point in at most $|Q|$ steps, i.e., $Reach_k = Reach_{k+1}$ for some $k \leq |Q|$. The language $L(A)$ is non-empty if and only if:

$$Reach_k \cap F \neq \emptyset$$

This fixed-point computation can be performed in time polynomial in the size of the transition relation. □

Summary & Key Takeaways — Small Model Property

The fixed-point reachability algorithm implies a **small model property** for finite trees: if a tree language recognized by a tree automaton is non-empty, then it contains a tree of height at most $|Q|$.

Finally, we state the equivalence between the algebraic and operational models, which is the tree-counterpart to Kleene's theorem.

Theorem 10.3.11 (Kleene's Theorem for Trees). *A tree language $T \subseteq T(A)$ is regular if and only if it is recognized by a finite tree automaton.*

Theorem 10.3.12 (Boolean Closure). *The class of regular tree languages is closed under union, intersection, complementation, and projection.*

We prove closure under union and intersection by explicit product constructions; complementation then follows by first determinizing the bottom-up automaton (theorem 10.3.8 would inline the same construction at the automaton level) and swapping the final set, and projection corresponds on automata to the usual hiding of the projected alphabet coordinate.

Closure under union and intersection by product construction. Let $A_1 = (Q_1, \Sigma, Q_0^{(1)}, \Delta_1, F_1)$ and $A_2 = (Q_2, \Sigma, Q_0^{(2)}, \Delta_2, F_2)$ be two NBTA over the same alphabet. We build the product NBTA $A_\times = (Q_1 \times Q_2,\ \Sigma,\ Q_0^{(1)} \times Q_0^{(2)},\ \Delta_\times,\ F_\times)$ whose transition relation contains

$$((q_1^L, q_2^L),\ (q_1^R, q_2^R),\ a,\ (q_1, q_2)) \in \Delta_\times$$

whenever $(q_1^L, q_1^R, a, q_1) \in \Delta_1$ and $(q_2^L, q_2^R, a, q_2) \in \Delta_2$.

The product runs A_1 and A_2 synchronously on the same input tree: a run R of $A_\times$ projects to a run $R_1 = \pi_1 \circ R$ of A_1 and a run $R_2 = \pi_2 \circ R$ of A_2, and conversely any pair of runs (R_1, R_2) lifts to a unique run of $A_\times$. Choosing

$$F_\times^\cup = (F_1 \times Q_2) \cup (Q_1 \times F_2) \quad \text{and} \quad F_\times^\cap = F_1 \times F_2$$

yields $L(A_\times, F_\times^\cup) = L(A_1) \cup L(A_2)$ and $L(A_\times, F_\times^\cap) = L(A_1) \cap L(A_2)$, since acceptance is declared at the root and the projection correspondence above is bijective.

Complementation. Given an NBTA A, first determinize it bottom-up via the subset construction to obtain an equivalent DBTA $A' = (Q', \Sigma, \{S_0\}, \delta', F')$. Then the DBTA $A^C = (Q', \Sigma, \{S_0\}, \delta', \mathcal{P}(Q) \setminus F')$ recognizes the complement: every input tree has a *unique* run on a DBTA, so swapping the final set flips the verdict at the root. □

■ **Formal details — Pumping lemma for regular tree languages**

The small model property is the tree analogue of the pumping lemma for regular word languages. Suppose L is recognized by an NBTA with n states and let $t \in L$ have a path longer than n. Along that path the unique bottom-up run visits more nodes than there are states, so two ancestors $u \subsetneq v$ carry the same state. Splicing out the segment between u and v — replacing the subtree at v by the subtree at u — produces a strictly smaller tree still in L, and *pumping* the segment (repeating v/u any number of times) yields infinitely many trees in L. As for words, this gives the standard tool to prove that a tree language is *not* regular.

The finite case is now complete: algebraic expressions, top-down nondeterminism, bottom-up nondeterminism, and bottom-up determinism all meet at the same class of regular tree languages. Infinite trees preserve the branching structure but remove the leaves, so the next step changes the direction and the acceptance condition of the automaton.

10.4 Automata on Infinite Trees

In section 10.3, we analyzed tree automata on finite trees and established that bottom-up automata can be determinized using a subset construction, making them equivalent to regular tree expressions. However, when we transition to infinite trees, we encounter a fundamental structural barrier: **infinite trees have no leaves or frontier** from which a bottom-up automaton could begin its computation.

Consequently, on infinite trees, we are forced to process the tree in a **top-down** direction, starting at the root and moving down the branches. But as we proved in theorem 10.3.5, top-down determinism is strictly weaker than non-determinism, meaning we cannot determinize top-down automata using standard subset techniques. To build a robust theory of infinite tree languages, we must enrich the acceptance conditions of top-down automata to reason about behavior in the limit (along the infinite paths of the tree). This leads us to Büchi and Rabin tree automata.

10.4.1 Büchi Tree Automata

A Büchi tree automaton checks whether every infinite path in the run tree satisfies a Büchi acceptance condition.

Definition 10.4.1 (Büchi Tree Automaton)**.** A *non-deterministic Büchi tree automaton* (BTA) over infinite binary trees is a tuple:

$$A = (Q, \Sigma, q_0, \Delta, F)$$

where:

- Q is a finite set of states,
- Σ is the input alphabet,
- $q_0 \in Q$ is the single initial state,
- $\Delta \subseteq Q \times \Sigma \times Q \times Q$ is the transition relation,
- $F \subseteq Q$ is the set of accepting states.

Definition 10.4.2 (BTA Run and Acceptance)**.** Let $t : \{0,1\}^* \to \Sigma$ be an infinite binary tree. A *run* of the BTA A on t is an infinite tree $R : \{0,1\}^* \to Q$ labeled with states such that:

1. **Initiality**: The root of the run is labeled with the initial state:

$$R(\epsilon) = q_0$$

2. **Consequentiality**: For every node $w \in \{0,1\}^*$, the transitions must obey the transition relation:

$$(R(w), t(w), R(w0), R(w1)) \in \Delta$$

An infinite path $\pi = d_1 d_2 d_3 \cdots \in \{0,1\}^\omega$ (where each $d_i \in \{0,1\}$) induces an infinite sequence of states in the run:

$$R|_\pi = R(\epsilon),\ R(d_1),\ R(d_1 d_2),\ R(d_1 d_2 d_3),\ \ldots$$

The run R is *successful* (or accepting) if for *every* infinite path π through the tree, the sequence of states $R|_\pi$ visits the set of accepting states F infinitely often:

$$\forall \pi \in \{0,1\}^\omega, \quad \inf(R|_\pi) \cap F \neq \emptyset$$

The tree t is accepted by A if there exists a successful run of A on t. The language recognized by A is denoted by $L(A)$.

To understand the interaction between non-determinism and path quantification, let us look at two classical examples at the blackboard.

Notice the universal path quantification in the acceptance condition: every path π must satisfy the Büchi condition. If even a single path fails to visit F infinitely often, the entire run is rejected.

Example 10.4.3 (Universal Property: Every Path Infinitely Many a's)**.** Let $\Sigma = \{a, b\}$. We want to recognize the tree language $T_{\forall,\infty,a}$ of all infinite binary trees where every path contains infinitely many nodes labeled a. We can construct a deterministic BTA for this language as follows:

- States: $Q = \{q_0, q_a\}$, with initial state q_0.

- Accepting states: $F = \{q_a\}$.
- Transitions: For each state $q \in Q$:

$$(q, a, q_a, q_a) \in \Delta$$
$$(q, b, q_0, q_0) \in \Delta$$

Since the transitions are deterministic, for any input tree t, there is a unique run R. Along any path π, the state transitions to q_a if and only if the current node is labeled a, and transitions to q_0 if and only if the node is labeled b. Thus, the run visits q_a infinitely often along a path π if and only if the path contains infinitely many a's. Since the acceptance condition requires this for all paths, the tree is accepted if and only if every path contains infinitely many a's.

Example 10.4.4 (Existential Property: At Least One Path Infinitely Many a's)**.** Let $\Sigma = \{a, b\}$. We want to recognize the tree language $T_{\exists,\infty,a}$ of all infinite binary trees where *there exists* at least one path containing infinitely many nodes labeled a.

Notice the fundamental mismatch here: our language asks an *existential* question ("at least one path"), but the Büchi acceptance condition is inherently *universal* ("every path"). How can a universal machine check an existential property?

The solution relies on non-deterministic guessing. The automaton guesses which path is the "good" one and sends the real state machine down that path. But what about all the other paths? We cannot just ignore them; the Büchi condition will check them anyway. So we route all those uninteresting paths into a dummy "sink" state q_T which trivially accepts everything. In this way, the universal check on the dummy paths automatically passes, leaving only the guessed path to do the real work.

Let us build the BTA for $T_{\exists,\infty,a}$:

- States: $Q = \{q_0, q_a, q_T\}$, with initial state q_0.
- Accepting states: $F = \{q_a, q_T\}$.
- Transitions:
  - For the dummy state q_T, we accept everything:

$$(q_T, \sigma, q_T, q_T) \in \Delta \quad \text{for all } \sigma \in \{a, b\}$$

  - For $q \in \{q_0, q_a\}$:

$$\begin{aligned} \text{If label is } a: \quad & (q, a, q_a, q_T) \in \Delta \quad \text{(guess left is good)} \\ & (q, a, q_T, q_a) \in \Delta \quad \text{(guess right is good)} \\ \text{If label is } b: \quad & (q, b, q_0, q_T) \in \Delta \quad \text{(guess left is good)} \\ & (q, b, q_T, q_0) \in \Delta \quad \text{(guess right is good)} \end{aligned}$$

Whiteboard Trace of the Guessing Mechanism. Let us trace how this automaton evaluates a run on a tree t:

- The automaton starts at the root in state q_0. At each step, it chooses one child to receive a state in $\{q_0, q_a\}$ (continuing the search for the path with infinitely many a's), and sends the other child to state q_T.

- Any path routed to q_T stays in q_T forever. Since $q_T \in F$, these paths trivially satisfy the Büchi condition $\inf(R|_\pi) \cap F \neq \emptyset$.
- The single path that is checked will visit state q_a (which is in F) if and only if it encounters nodes labeled a.
- If there exists a path with infinitely many a's, the automaton can non-deterministically guess this path. The check along this path will succeed (visiting q_a infinitely often), and all other paths will succeed trivially in q_T. Hence, the run is successful.
- If no such path exists, then along any run, the checked path will only visit q_a finitely many times, failing the Büchi condition. Thus, no successful run exists, and the tree is rejected.

10.4.2 The Complementation Barrier for Büchi Tree Automata

In the case of infinite words, non-deterministic Büchi automata (NBAs) are closed under complementation. However, this closure property fails dramatically for tree automata.

> **Theorem 10.4.5** (Büchi Non-closure under Complementation). *The class of languages recognized by Büchi tree automata is not closed under complementation.*

Proof. To prove this, we consider the complement of the existential language $T_{\exists,\infty,a}$ from theorem 10.4.4:

$$T_{co\text{-}\exists} = \{t \in T_\omega(\{a, b\}) \mid \text{every path in } t \text{ contains finitely many } a\text{'s}\}$$

We show that while $T_{\exists,\infty,a}$ is recognizable by a BTA, its complement $T_{co\text{-}\exists}$ is not recognizable by any BTA.

Assume for contradiction that there exists a BTA $A = (Q, \Sigma, q_0, \Delta, F)$ such that $L(A) = T_{co\text{-}\exists}$. Consider the tree t_b where all nodes are labeled with b. Since t_b has zero a's, every path in t_b contains only finitely many a's, so $t_b \in T_{co\text{-}\exists}$. Thus, there exists a successful run R of A on t_b. By definition of a successful run, for every path $\pi \in \{0, 1\}^\omega$, we have $\inf(R|_\pi) \cap F \neq \emptyset$.

Let $n = |Q|$ be the number of states in A. We can trace a path in the run tree R to construct a contradiction. Starting from the root ϵ, since every path in the run visits F infinitely often, we can find a node $w_1 = d_1 \dots d_{k_1}$ (with $k_1 > 0$) such that the state sequence from the root to w_1 visits at least one state in F. Similarly, from the subtree root w_1, we can find a node $w_1 w_2$ (where $w_2 = d'_1 \dots d'_{k_2}$, $k_2 > 0$) such that the state sequence from w_1 to $w_1 w_2$ visits a state in F. Repeating this construction, we obtain an infinite path:

$$\pi = w_1 w_2 w_3 \dots$$

such that for each segment w_j, the run visits a state in F. Since Q is finite, by the pigeonhole principle, if we make these segments sufficiently long, we can find two nodes u and uv along this path such that:

1. $R(u) = R(uv) = q$ for some state $q \in Q$,
2. The path segment from u to uv visits at least one state in F.

Now, since $R(u) = R(uv) = q$, the state at the boundary repeats. Because the transitions of the top-down automaton only depend on the current state and the current node label, we can "graft" this repeating transition sequence. Specifically, let us construct a new input tree t':

- Along the path π, we repeat the segment v infinitely many times.
- At the end of each segment (i.e., at nodes of the form uv^m), we label the node with a.
- For all other nodes in t' not along this repeating path, we label them with b.

We can construct a run R' on t' by copying the transitions of R for all branches that branch off the main path, and repeating the state sequence from u to uv along the main path. Because the state repeats at the boundary ($q \xrightarrow{b} \ldots \xrightarrow{a} q$), this run is valid. Let us evaluate if R' is successful:

- For any path that branches off the main path π, it eventually enters a subtree labeled entirely with b, where the run behaves exactly like the successful run R on t_b. Thus, all these paths satisfy the Büchi condition.
- For the main path π itself, the state sequence is periodic and repeats the segment from u to uv infinitely often. Since this segment contains a state in F, the path π visits F infinitely often, satisfying the Büchi condition.

Thus, the run R' is successful, meaning A accepts t'. However, the path π in t' contains infinitely many nodes labeled a (since we placed a at the end of each repeated segment v). Thus, $t' \notin T_{co\text{-}\exists}$. This is a contradiction, since $L(A) = T_{co\text{-}\exists}$. Therefore, no BTA can recognize $T_{co\text{-}\exists}$, proving that Büchi tree automata are not closed under complementation. □

Why does complementation fail for Büchi tree automata? In words, an NBA can be complemented by converting it to a deterministic Rabin automaton (using Safra's construction) and then complement the Rabin condition. In trees, we cannot determinize top-down, and the lack of coordination between paths prevents us from using the word complementation techniques directly. The universal path quantification of BTA makes it impossible to check that a path property is violated existential-style on all paths simultaneously.

10.4.3 Rabin Tree Automata

To restore closure under complementation and build a complete logic-automata connection on infinite trees, Michael Rabin introduced **Rabin Tree Automata** in his landmark 1969 paper. Rabin tree automata generalize the acceptance condition by using pairs of state sets.

Think of the Rabin condition as a combination of Muller and Büchi. A Büchi condition (F) says "visit this infinitely often". A Muller condition ($\mathcal{F}$) says "visit exactly these states infinitely often". A Rabin pair (L, U) splits the requirement: L acts as a negative condition ("you must visit L only finitely many times"), while U acts as a positive Büchi condition ("you must visit U infinitely many times").

Definition 10.4.6 (Rabin Tree Automaton)**.** A *non-deterministic Rabin tree automaton* (RTA) over infinite binary trees is a tuple:

$$A = (Q, \Sigma, q_0, \Delta, \Omega)$$

where Q, Σ, q_0, Δ are defined as in BTA, and:

$$\Omega = \{(L_1, U_1), (L_2, U_2), \ldots, (L_k, U_k)\}$$

is a finite set of k Rabin pairs of states, where $L_i, U_i \subseteq Q$.

Definition 10.4.7 (RTA Acceptance). A run $R : \{0,1\}^* \to Q$ of the RTA A is *successful* if for *every* infinite path $\pi \in \{0,1\}^\omega$, the set of states visited infinitely often along π satisfies the Rabin condition with respect to Ω:

$$\forall \pi \in \{0,1\}^\omega, \quad \exists i \in \{1,\dots,k\} \text{ s.t. } \inf(R|_\pi) \cap L_i = \emptyset \wedge \inf(R|_\pi) \cap U_i \neq \emptyset$$

In words, for every path π, there must exist some pair (L_i, U_i) such that the path visits states in L_i only finitely many times, and visits states in U_i infinitely many times.

Let us see how Rabin pairs allow us to bypass the Büchi complementation barrier by recognizing the language $T_{co\text{-}\exists}$ of infinite binary trees where every path contains only finitely many a's.

Example 10.4.8 (RTA for "Every Path Finitely Many a's"). Let $\Sigma = \{a, b\}$. We want to design a deterministic Rabin tree automaton (DRTA) for the language $T_{co\text{-}\exists}$ where every path contains only finitely many a's. We construct the DRTA as follows:

- States: $Q = \{q_0, q_a\}$, with initial state q_0.
- Transitions: For each state $q \in Q$:

$$(q, a, q_a, q_a) \in \Delta$$
$$(q, b, q_0, q_0) \in \Delta$$

- Rabin pairs: $\Omega = \{(L_1, U_1)\}$ where $L_1 = \{q_a\}$ and $U_1 = \{q_0\}$.

Let us trace the Rabin condition on any path π of the unique run tree:

- The state is q_a at a node if and only if the node label is a.
- If the path contains only finitely many a's, then the state q_a is visited only finitely many times, meaning $\inf(R|_\pi) \cap L_1 = \emptyset$. Since the path is infinite, it must contain infinitely many b's, which means the state q_0 is visited infinitely often: $\inf(R|_\pi) \cap U_1 \neq \emptyset$. The Rabin condition is satisfied.
- If the path contains infinitely many a's, then q_a is visited infinitely often, meaning $\inf(R|_\pi) \cap L_1 \neq \emptyset$, violating the Rabin condition.

Since the condition is checked universally for all paths, the automaton accepts the tree if and only if every path contains only finitely many a's. This deterministic RTA recognizes $T_{co\text{-}\exists}$ using a single Rabin pair.

10.4.4 Rabin's Decidability Theorem and S2S

We have spent the whole chapter building an operational model on infinite trees. The pay-off is logical: just as Büchi's theorem connected S1S to NBA on words, Rabin's theorem connects the Monadic Second-Order Logic of Two Successors — S2S — to Rabin tree automata. S2S is the natural language for branching-time specifications: its first-order variables range over nodes of the computation tree, its second-order variables over arbitrary sets of nodes, and its two successor relations let us step down the left or right branch. Any property expressible in S2S — including subtle ones that relate different branches, like "whenever a request appears on some branch, every branch through that node eventually sees a grant" — is captured by a finite automaton.

Definition 10.4.9 (S2S Logic)**.** The logic *S2S* is the monadic second-order theory of two successors. The language of S2S is defined over the domain $\{0, 1\}^*$ (the infinite binary tree). Formally, S2S formulas are built from:

- First-order variables x, y, z (representing nodes in the binary tree),
- Second-order variables X, Y, Z (representing sets of nodes),
- Relations:
  - $y = x0$ (left successor relation),
  - $y = x1$ (right successor relation),
  - $x \in X$ (set membership).
- Standard logical connectives $(\vee, \wedge, \neg)$ and quantifiers $(\exists x, \forall x, \exists X, \forall X)$.

S2S is extremely powerful. We can express ordering relations ($x < y$, meaning x is a prefix of y), check if a set of nodes is a path, and reason about properties of infinite branching computations.

In 1969, Michael Rabin proved one of the most celebrated results in mathematical logic and computer science:

Theorem 10.4.10 (Rabin's Decidability Theorem)**.** *The Monadic Second-Order Theory of Two Successors (S2S) is decidable.*

The proof of Rabin's theorem follows the same automata-theoretic pipeline we saw for S1S:

1. We establish a correspondence: for every S2S formula $\phi(X_1, \ldots, X_n)$ with free set variables, we can construct an equivalent Rabin tree automaton A_ϕ over the alphabet $\Sigma = \{0, 1\}^n$ such that $L(A_\phi)$ is the set of all tree valuations satisfying ϕ.
2. The logical connectives map to operations on automata:
   - Disjunction $(\vee)$ maps to automaton union,
   - Existential second-order quantification $(\exists X)$ maps to automaton projection,
   - Negation $(\neg)$ maps to automaton complementation.
3. Once the formula is compiled into a sentence (no free variables), checking the validity of the formula reduces to checking the emptiness of the compiled Rabin tree automaton.

■ **Formal details — Regular witnesses for non-emptiness**

The emptiness step for infinite tree automata has its own tree analogue of the finite small-model property. Rabin's basis theorem says that if a Rabin-recognizable language of infinite trees is non-empty, then it contains a *regular tree*: an infinite tree with only finitely many distinct subtrees, equivalently the unfolding of a finite directed graph with labels. Intuitively, an accepting run that succeeds somewhere can be regularized because the automaton has finitely many states and the Rabin condition depends only on which states occur infinitely often along paths. In the game view, a winning strategy can be chosen with finite memory; unfolding the finite strategy graph gives the required regular witness. Thus non-emptiness never requires searching arbitrary infinite branching objects one by one: if there is a witness, there is one with a finite description.

To make this pipeline work, the key mathematical hurdle is proving that Rabin tree automata are closed under complementation. Unlike word automata, where this is relatively simple, tree complementation is incredibly complex and represents the core of Rabin's paper:

Theorem 10.4.11 (Rabin Complementation Theorem)**.** *The class of languages recognized by Rabin tree automata is closed under complementation.*

Proof idea and road map. Let A be an RTA. To construct an automaton recognizing $L(A)^C$, we must verify that no successful run of A exists on an input tree t. Rabin formulated the search for a successful run as a game played between two players on an infinite arena:

- **Player 1 (Automaton)** tries to choose transitions that build a successful run,
- **Player 2 (Path-Finder)** tries to select a path π that violates the Rabin acceptance condition.

The input tree t is accepted by A if and only if Player 1 has a winning strategy in this run-construction game. Consequently, t is rejected (belongs to the complement) if and only if Player 1 does not have a winning strategy. By the **Borel Determinacy Theorem** (proved by Donald A. Martin), since the winning condition of the game is Borel, the game is determined: one of the two players must have a winning strategy. Therefore, t is in the complement if and only if Player 2 has a winning strategy. The construction of A^C then follows this road map: enrich the input with a finite description of Player 2's choices, let another Rabin tree automaton check locally that the description is a valid strategy, and use the Rabin condition to verify that every play compatible with it defeats every attempted run of A. This paragraph deliberately suppresses the technical encoding of strategies and ranks; it should be read as the proof idea, not as the full complementation construction. □

10.4.5 Weak S2S Characterization

The jump from S1S to S2S is enormous — we went from a single linear future to an entire binary bundle of futures. A natural temptation is to ask whether we can recover some of the algorithmic tameness of the linear world by restricting the second-order quantification. This is precisely what the *weak* fragment does. Just as for S1S, the weak version of S2S is denoted **WS2S**, and its second-order variables range over finite sets of nodes only.

Definition 10.4.12 (Weak S2S Logic)**.** The logic *WS2S* is the monadic second-order theory of two successors where second-order variables X, Y, Z are restricted to range only over *finite* sets of nodes.

In the linear-time case, WS1S and S1S are expressively equivalent: both correspond to regular languages of infinite words. In the branching-time case, this equivalence breaks. WS2S is strictly weaker than S2S.

We can characterize this expressiveness gap using Büchi tree automata:

Theorem 10.4.13 (Weak S2S Characterization)**.** *A tree language $T \subseteq T_\omega(A)$ is definable in WS2S if and only if both T and its complement T^C are recognizable by Büchi tree automata.*

Since $T_{\exists,\infty,a}$ is recognizable by a BTA but its complement $T_{co\text{-}\exists}$ is not, we immediately obtain:

- The language $T_{\exists,\infty,a}$ ("there exists a path with infinitely many a's") is definable in S2S,
- The language $T_{\exists,\infty,a}$ is *not* definable in WS2S.

This illustrates that restricting second-order quantification to finite sets limits our ability to reason about properties that must hold infinitely often along branching paths.

10.4.6 From Tree Automata to Temporal Logics

We have reached the end of the automata-theoretic core of Part II. Looking back, the chapter tells a single story: to analyse *all* executions of a reactive system at once, we replaced ω-words by ω-trees; we rebuilt for trees the algebra of regular expressions (substitution in place of concatenation), the operational model (top-down vs. bottom-up automata, with the determinism asymmetry between them), the limit-behaviour acceptance hierarchy (Büchi, then Rabin), and finally the logic–automata correspondence S2S ≡ Rabin tree automata.

The correspondence S2S ≡ RTA is mathematically beautiful but practically crippling. S2S has non-elementary decision complexity — a tower of exponentials whose height grows with the quantifier alternation depth of the formula — and writing a specification directly in monadic second-order logic is error-prone in the same way we already saw for S1S. Industrial verification needs logics that engineers can read, write, and discharge efficiently.

The way out, pioneered by Amir Pnueli in 1977, is to identify *tractable fragments* of S1S and S2S with syntax tuned to the temporal vocabulary of verification: "eventually", "always", "until", "next". Two such fragments will occupy the rest of Part II:

- **Linear Temporal Logic (LTL)**, the subject of chapter 11, is the fragment of S1S that quantifies over a single linear execution. Every LTL formula compiles to a Büchi automaton, and model checking reduces to the emptiness/inclusion problems of chapter 3. LTL cannot express branching properties like "there exists a run that. . . " — it lives entirely on a single branch of the computation tree.
- **Computation Tree Logic (CTL)** and its cousin CTL* live on the tree itself. They are fragments of S2S in which path quantifiers (∀ paths / ∃ path) alternate with temporal modalities, exactly mirroring the universal-vs-existential dichotomy we exploited in theorem 10.4.4 when guessing the "good" path of a Büchi tree automaton.

There is one more pleasant consequence of moving to trees that is worth recording before we leave the subject. For pure *model checking* (as opposed to satisfiability or synthesis), the tree setting is actually friendlier than the word setting: the unfolding of a finite-state system into a computation tree is a regular tree — it has only finitely many distinct subtrees, one per reachable state — and a tree automaton can be run on this unfolding in lock-step without ever materialising the whole infinite object. This is the algorithmic reason why CTL model checking is polynomial in the size of the system, while LTL model checking is exponential in the size of the formula. We will return to this contrast in chapter 11 and in the symbolic model-checking chapter.

Notice the clean methodological loop: temporal logics are presented to engineers as readable modal notations, but the verification algorithms that decide them compile every formula back into the automata we have built in chapters 3–10. The theory you have just finished is the engine that makes model checking work.

Before turning to exercises, we compress the main equivalences and separations into one reference box.

Summary & Key Takeaways — Chapter 10 in one screen

A regular tree language over finite trees is equivalently: a language defined by regular tree expressions (union, $\cdot_c$ substitution, $*, c$ iteration), a language recognised by a non-deterministic top-down or bottom-up tree automaton, or a language recognised by a deterministic bottom-up tree automaton. Determinism matters: deterministic top-down recognises only path properties and is strictly weaker than non-deterministic top-down.

On infinite trees there is no bottom-up model, so we move to top-down Büchi and then Rabin tree automata. Büchi tree automata are *not* closed under complementation; Rabin tree automata are. This closure result is the technical heart of Rabin's theorem: S2S ≡ Rabin tree automata, and S2S is decidable. The weak fragment WS2S (quantification over finite sets only) is strictly smaller: it corresponds to languages recognisable by Büchi tree automata together with their complements.

This closes the automata-theoretic core of the book. The next chapter (chapter 11) introduces Linear Temporal Logic as a practical, low-complexity fragment of S1S, and CTL (later in Part II) will play the same role for S2S.

The exercises below check both sides of the chapter: the basic tree definitions and the automata constructions that recognize the resulting languages.

Exercises

Exercise 1 (Tree domain properties). Verify whether the following sets of binary strings satisfy the prefix-closure and child-ordering properties to form a valid binary tree domain:
(a) $D_1 = \{\epsilon, 0, 1, 00, 01, 11\}$
(b) $D_2 = \{\epsilon, 0, 1, 00, 01, 10\}$

Exercise 2 (Binary coding of 3-ary trees). Let t be a 3-ary tree with domain $\mathrm{dom}(t) = \{\epsilon, 0, 1, 2\}$ and labeling $t(\epsilon) = f$, $t(0) = a$, $t(1) = b$, $t(2) = a$. Compute the domain and labeling of its binary coded tree t_{bin} according to theorem 10.1.4.

Exercise 3 (Bottom-up automaton construction). Construct a deterministic bottom-up tree automaton (DBTA) that recognizes the language $T_{xor} = \{f(a,b), f(b,a)\}$. Write down the transition table and explain how the automaton processes the tree $f(a,b)$.

Exercise 4 (Büchi tree automaton guessing). Trace a successful run of the BTA from theorem 10.4.4 on the tree t where the leftmost path $\pi = 0^\omega$ has infinitely many a's, and all other paths contain only b's. Show the state labels assigned to nodes of depth up to 2.

Exercise 5 (Rabin vs. Büchi tree automata). Let $A = (Q, \Sigma, q_0, \Delta, \Omega)$ be a Rabin tree automaton with a single Rabin pair $\Omega = \{(L_1, U_1)\}$ and $L_1 = \emptyset$. Write down the equivalent Büchi tree automaton and explain why the constructions are equivalent.

CHAPTER 11

Linear Temporal Logic

In Part I and the opening chapters of Part II, we established the mathematical theory of automata over infinite words, specifically non-deterministic Büchi automata (NBA), and logical systems like $S1S$ (Monadic Second-Order Logic of One Successor). We proved Büchi's Theorem: $S1S$ and Büchi automata are expressively equivalent, defining the class of ω-regular languages.

However, compiling specifications written in $S1S$ is practically intractable due to its non-elementary decision complexity (characterized by a tower of exponentials of unbounded height). Furthermore, writing specifications in $S1S$ is highly error-prone.

To bridge the gap between declarative convenience and algorithmic efficiency, we introduce *Linear Temporal Logic (LTL)*, proposed by Amir Pnueli in 1977. LTL is a modal logic designed to specify properties over linear, infinite paths. It serves as the standard specification language in industrial model checking. In this chapter, we develop the syntax and semantics of LTL, analyze its expressive boundaries (star-free languages), prove the exponential succinctness of past-time operators, and detail the graph-theoretic tableau decision procedure for LTL satisfiability.

Where we are. S1S and Büchi automata gave a complete theory of ω-regular specifications, and chapter 9 showed how determinism and first-order restrictions shape that theory.

What this chapter adds. LTL keeps the infinite-trace viewpoint but replaces second-order quantification with temporal operators such as "next", "until", "eventually", and "always". The result is a language that engineers can write and model checkers can compile.

Where it leads. The tableau and automata translation here are the input to the LTL model checking algorithm in chapter 12.

Chapter map.

- Sections 11.1 and 11.2 introduce linear time and the syntax/semantics of LTL.
- Sections 11.3 and 11.4 develop useful equivalences and locate LTL's expressive boundary.
- Section 11.5 studies past operators and their succinctness.
- Sections 11.6 and 11.7 build the satisfiability tableau and reduce eventualities to graph search.
- Section 11.8 turns the tableau view into an automata translation.

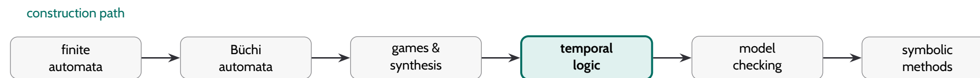


Figure 11.1: Where this chapter sits in the construction path of the book. Each step reuses the previous one as a representation or an algorithmic primitive.

Running example — request/acknowledgement specifications

The running liveness requirement is now written directly as a formula: $\mathbf{G}(r \to \mathbf{F}g)$. It says that every request must be followed by some future grant along the same execution trace.

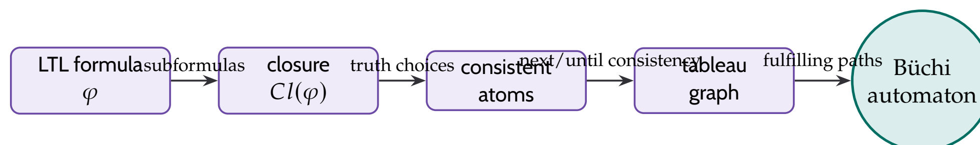


Figure 11.2: The tableau construction for LTL satisfiability. The formula is decomposed into closure formulas, atoms choose locally consistent truth values, tableau edges enforce next-step consistency, and Büchi acceptance checks eventualities.

11.1 The Logic of Linear Time

A temporal logic is characterized by how it represents the flow of time. In LTL, we assume a *linear-time* model: at each moment, there is a single, unique future. This corresponds to evaluating formulas along a single execution path of a system.

Before formalizing the syntax and semantics of LTL, we establish its place in the broader context of software and hardware verification.

11.1.1 Modal Logic Foundations

Before we define the syntax of LTL, it is helpful to look at its roots. LTL is a specific flavor of *modal logic*. Let's step up to the whiteboard and draw a Kripke structure: a graph of "multiple worlds." Imagine a 3-node graph with states s_0, s_1, s_2 connected by directed edges representing an "accessibility relation" (which worlds are reachable in one time-step from another). In standard modal logic, we evaluate formulas at a specific state. We have two key operators:

- **Necessity ($\Box P$):** P is true in *all* immediately accessible neighboring worlds.
- **Possibility ($\Diamond P$):** P is true in *at least one* immediately accessible neighboring world.

This branching-world view is very expressive, but sometimes we want to study just a single execution trace of a system. What happens if we restrict our "multiple worlds" graph to be just an infinite linear chain of states ($s_0 \to s_1 \to s_2 \to \ldots$)? On a linear chain, "all accessible worlds" and "some accessible world" collapse into exactly one world: the unique next state! This restriction gives birth to Linear Temporal Logic. The $\Box$ (always) and $\Diamond$ (eventually) operators

are extended to evaluate properties over the entire infinite sequence rather than just the immediate neighbors.

11.1.2 Verification vs. Validation

In formal methods, we distinguish between two fundamental activities:

- **Verification** ("Are we building the product right?"): The process of proving that a system meets its formal specification. It compares the system model against a set of mathematically defined properties.
- **Validation** ("Are we building the right product?"): The process of ensuring that the system meets the actual needs and expectations of the user. It checks the specifications themselves against the informal, real-world requirements.

11.1.3 Historical Motivation: Classic Software and Hardware Bugs

The necessity of formal verification is underscored by high-profile software and hardware failures where testing failed to uncover critical, edge-case bugs:

1. **Therac-25 (1985–1987)**: A computerized radiation therapy machine that, due to software race conditions and a lack of hardware interlocks, administered massive radiation overdoses to patients, resulting in severe injuries and deaths.
2. **AT&T Network Outage (1990)**: A single buggy line of C code in a switch recovery routine caused a cascading crash, bringing down AT&T's long-distance telephone network for 9 hours and disrupting 70 million calls.
3. **Ariane 5 Flight 501 (1996)**: The rocket self-destructed 37 seconds after launch due to an unhandled integer overflow. A 64-bit floating-point variable representing horizontal velocity was cast into a 16-bit signed integer, causing the guidance software to fail.

11.1.4 Broader AI Applications of LTL

While software verification is the primary driver for LTL, the logic has deep roots in Artificial Intelligence. Because LTL can concisely describe sequences of events and eventual goals, it is widely used in:

- **Automated Planning**: Finding a sequence of robot actions that satisfies an LTL goal specification.
- **Automated Synthesis**: Compiling an LTL specification directly into a correct-by-construction reactive system.
- **Reinforcement Learning**: Using LTL formulas as reward functions to train agents, or as "shields" to prevent agents from taking unsafe actions during exploration.

11.1.5 Dimensions and Classification of Temporal Logics

Temporal logics can be classified along five main dimensions:

1. **Qualitative Time vs. Real-Time**: *Qualitative* logics focus on the relative order of events (e.g., "event P happens before Q" or "P eventually holds"). LTL is qualitative. *Real-time* logics associate a numerical timestamp $t_i \in \mathbb{R}$ with each state s_i. To be physically meaningful, timestamps must satisfy *monotonicity* ($t_i \leq t_{i+1}$) and *progress* (ruling out Zeno behaviors where infinitely many

events occur in a finite interval of time, e.g., timestamps $4.3, 4.33, 4.333, \ldots$ converging to $4.\overline{3}$ seconds). Formally, progress is enforced by the non-Zeno condition: $\lim_{i\to\infty} t_i = \infty$.

2. **Discrete Time vs. Dense Time**: In *discrete* time, the state sequence is isomorphic to $\mathbb{N}$: every point has a unique, immediate successor (+1), making the Next operator (X) well-defined. In *dense* time (isomorphic to $\mathbb{R}$ or $\mathbb{Q}$), the concept of a successor is meaningless. LTL assumes discrete time.
3. **Point-Based vs. Interval-Based**: *Point-based* logics evaluate formulas at individual states. LTL is point-based. *Interval-based* logics evaluate formulas over durations (e.g., "property P holds throughout interval I").
4. **Pure Future vs. Past and Future**: *Pure future* logics only refer to states ahead of the current evaluation point. Logics with past operators can look backward in time.
5. **Propositional vs. First-Order**: *Propositional* logics evaluate formulas over sets of boolean variables. LTL is propositional.

The ω-Regular Connection. By Kamp's Theorem (1968), LTL corresponds precisely to the *first-order fragment of S1S* ($FO(S1S)$). This fragment defines exactly the *star-free* ω-regular languages (languages definable by ω-regular expressions without using the Kleene star).

Fairness preview. When LTL is used to describe executions of concurrent systems, it is common to talk about fair paths. The two standard terms are *justice* or weak fairness, for actions that stay enabled and should eventually occur, and *compassion* or strong fairness, for actions that are enabled infinitely often and should also occur infinitely often. Chapter 12 uses that terminology in the model-checking setting.

11.2 Formal Syntax and Semantics of LTL

Let AP be a fixed, finite set of atomic propositions. The alphabet of our traces is $\Sigma = 2^{AP}$.

> **Definition 11.2.1** (LTL Syntax)**.** The formulas of Linear Temporal Logic (LTL) are defined inductively by the grammar:
>
> $$\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid X\phi \mid \phi_1 U \phi_2$$
>
> where $p \in AP$.

Other boolean connectives are derived as standard shortcuts: $\top \equiv p \vee \neg p$, $\bot \equiv \neg\top$, $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$, and $\phi_1 \implies \phi_2 \equiv \neg\phi_1 \vee \phi_2$.

The temporal operators are:

- $X\phi$ ("Next" or "Tomorrow"): ϕ must hold in the immediate successor state.
- $\phi_1 U \phi_2$ ("Until"): ϕ_2 must eventually hold, and ϕ_1 must hold continuously up to that point.

An LTL formula is interpreted over an infinite sequence of states, called a *trace*. A trace σ is an infinite word over Σ:

$$\sigma = s_0 s_1 s_2 \cdots \in (2^{AP})^\omega$$

where $s_i \subseteq AP$ is the set of propositions true at step i. We denote the suffix $s_i s_{i+1} s_{i+2} \ldots$ as $\sigma[i..]$.

Definition 11.2.2 (LTL Semantics). The satisfaction relation $\sigma, i \models \phi$ (trace σ satisfies ϕ at position $i \geq 0$) is defined inductively:

$$
\begin{aligned}
\sigma, i \models p &\iff p \in s_i \quad (\text{for } p \in AP) \\
\sigma, i \models \neg\phi &\iff \sigma, i \not\models \phi \\
\sigma, i \models \phi_1 \wedge \phi_2 &\iff \sigma, i \models \phi_1 \text{ and } \sigma, i \models \phi_2 \\
\sigma, i \models X\phi &\iff \sigma, i+1 \models \phi \\
\sigma, i \models \phi_1 U \phi_2 &\iff \exists j \geq i \text{ such that } \sigma, j \models \phi_2 \text{ and } \forall k \in [i, j-1], \sigma, k \models \phi_1
\end{aligned}
$$

Operational meaning of Semantics. Let's decode this operationally. Evaluating p, $\neg\phi$, or $\phi_1 \wedge \phi_2$ at position i just means looking at the state s_i in isolation. The temporal magic happens with X and U. The $X\phi$ operator simply shifts the evaluation pointer one step into the future to $i+1$. The $\phi_1 U \phi_2$ operator is a search loop: it looks for a future position j where the goal ϕ_2 is met, while simultaneously checking that the condition ϕ_1 is unbroken for every step between i and $j-1$.

11.2.1 Derived Temporal Operators

We define three crucial temporal operators as syntactic sugar:

- **Eventually** (F): $F\phi \equiv \top U \phi$.

$$\sigma, i \models F\phi \iff \exists j \geq i \text{ such that } \sigma, j \models \phi$$

- **Globally** (G): $G\phi \equiv \neg F \neg \phi$.

$$\sigma, i \models G\phi \iff \forall j \geq i, \sigma, j \models \phi$$

- **Release** (R): The dual of Until. $\phi_1 R \phi_2 \equiv \neg(\neg\phi_1 U \neg\phi_2)$.

$$\sigma, i \models \phi_1 R \phi_2 \iff \forall j \geq i, \big(\forall k \in [i, j], \sigma, k \not\models \phi_1\big) \implies \sigma, j \models \phi_2$$

  Release requires that ϕ_2 holds up to and including the point where ϕ_1 becomes true; if ϕ_1 never becomes true, ϕ_2 must hold forever.

A trace σ satisfies ϕ, written $\sigma \models \phi$, if and only if $\sigma, 0 \models \phi$. The language of ϕ is $\mathcal{L}(\phi) = \{\sigma \in (2^{AP})^\omega \mid \sigma \models \phi\}$. A formula ϕ is *satisfiable* if $\mathcal{L}(\phi) \neq \emptyset$, and *valid* (a tautology) if $\mathcal{L}(\phi) = (2^{AP})^\omega$.

11.3 Equivalences and Temporal Expansions

LTL formulas satisfy several algebraic dualities and expansion rules.

11.3.1 Duality of Next and Eventually

The operators F and X commute:

$$FX\phi \equiv XF\phi$$

Proof. Let $\sigma \in \Sigma^\omega$ and $i \geq 0$.

$$\begin{aligned} \sigma, i \models FX\phi &\iff \exists j \geq i \text{ s.t. } \sigma, j \models X\phi \\ &\iff \exists j \geq i \text{ s.t. } \sigma, j+1 \models \phi \\ &\iff \exists j' \geq i+1 \text{ s.t. } \sigma, j' \models \phi \quad (\text{letting } j' = j+1) \\ &\iff \sigma, i+1 \models F\phi \\ &\iff \sigma, i \models XF\phi. \end{aligned}$$

□

11.3.2 Expansion (Unrolling) Rules

The expansion rules decompose a temporal operator into its present constraint and its postponed future constraint. These rules are central to the tableau construction:

$$\begin{aligned} G\phi &\equiv \phi \wedge XG\phi \\ F\phi &\equiv \phi \vee XF\phi \\ \phi_1 U \phi_2 &\equiv \phi_2 \vee (\phi_1 \wedge X(\phi_1 U \phi_2)) \end{aligned}$$

11.3.3 Concrete Trace Evaluations

Let $AP = \{P, Q, R\}$. We evaluate LTL formulas over concrete traces.

Example 11.3.1 (Until Evaluation)**.** Let $\phi = (P \vee Q)UR$. Consider the trace $\sigma = \{P\}\{Q\}\{P\}\{Q,R\}\emptyset^\omega$, illustrated in Figure 11.3.

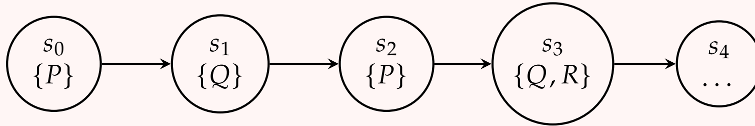


Figure 11.3: Trace σ satisfying $(P \vee Q)UR$ with the existential witness at step 3.

- For $j = 3$, $R \in s_3$, satisfying the existential part of Until.
- For $k \in \{0,1,2\}$, we have $P \in s_0$ (so $P \vee Q$ holds), $Q \in s_1$ ($P \vee Q$ holds), and $P \in s_2$ ($P \vee Q$ holds).
- Thus, $\sigma \models (P \vee Q)UR$ is **True**.

If we modify the trace to $\sigma' = \{P\}\emptyset\{R\}\{P,R\}^\omega$, evaluating at $j = 2$ requires checking $k = 1$. Since $s'_1 = \emptyset$, $P \vee Q$ is false. Thus, $\sigma' \not\models (P \vee Q)UR$ due to a universal violation at step 1.

Example 11.3.2 (Reactivity / Response)**.** Let $\phi = G(r \implies Fg)$, specifying that every request r is eventually followed by a grant g (where $AP = \{r, g\}$). Consider the trace $\sigma = \{r\}\{r\}\{r\}\{g\}\emptyset^\omega$, shown in Figure 11.4.

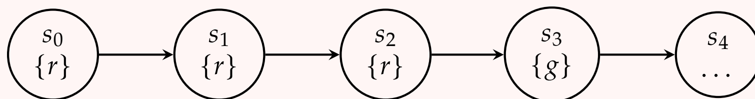


Figure 11.4: Trace σ satisfying $G(r \implies Fg)$.

For $i \in \{0, 1, 2\}$, $r \in s_i$ and a grant occurs at s_3, so Fg holds. For $i \geq 3$, $r \notin s_i$, making the implication vacuously true. Hence, the trace satisfies ϕ.

Example 11.3.3 (Next-Step Formula). Let $\phi = P \wedge (XQ \vee XX\neg R)$ over $AP = \{P, Q, R\}$. This formula states that at the initial position 0, P must be true, and in the next step Q must hold, or two steps later R must be false. We evaluate this formula over four distinct traces:

1. **Trace** $\sigma_A = \{P\}^\omega$ (illustrated in Figure 11.5):

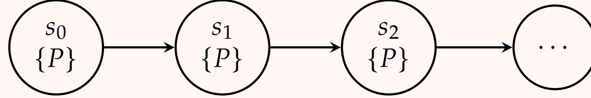


Figure 11.5: Trace σ_A satisfying the next-step formula.

   Evaluation at position 0:
   - $P \in s_0 \implies$ True.
   - $\sigma_A, 1 \models Q \implies$ False (since $Q \notin s_1$). Thus, $\sigma_A, 0 \models XQ$ is False.
   - $\sigma_A, 2 \models \neg R \implies$ True (since $R \notin s_2$). Thus, $\sigma_A, 0 \models XX\neg R$ is True.
   - The disjunction is True, so $\sigma_A \models \phi$ is **True**.
2. **Trace** $\sigma_B = \{P, Q\}^\omega$: Evaluation at position 0:
   - $P \in s_0 \implies$ True.
   - $\sigma_B, 1 \models Q \implies$ True. Thus, $\sigma_B, 0 \models XQ$ is True.
   - The disjunction is True, so $\sigma_B \models \phi$ is **True**.
3. **Trace** $\sigma_C = \{P, R\}^\omega$: Evaluation at position 0:
   - $P \in s_0 \implies$ True.
   - $\sigma_C, 1 \models Q \implies$ False, so $\sigma_C, 0 \models XQ$ is False.
   - $\sigma_C, 2 \models \neg R \implies$ False (since $R \in s_2$), so $\sigma_C, 0 \models XX\neg R$ is False.
   - The disjunction is False, so $\sigma_C \not\models \phi$ (**False**).
4. **Trace** $\sigma_D = \{Q, R\}^\omega$: Evaluation at position 0:
   - $P \in s_0 \implies$ False (since $P \notin s_0$).
   - Thus, $\sigma_D \not\models \phi$ (**False**).

Example 11.3.4 (Nested Eventually). Let $\phi = F(P \wedge FQ)$ over $AP = \{P, Q\}$. This formula states that there is some state in the future (including the present) where P holds, and from that state forward (including the same state), Q eventually holds. Essentially, P must be followed by Q. We evaluate this over four traces:

1. **Trace** $\sigma_1 = \{Q\}\{Q\}\{Q\}\{P\}\{P\}^\omega$: At index 3, P holds. However, for any $j \geq 3$, Q is never true. Thus, FQ is false at index 3, and the formula is **False**.
2. **Trace** $\sigma_2 = \{P\}\{P\}\{P\}\{Q\}\{Q\}^\omega$: At index 0, P is true, and there is a future index (index 3) where Q is true. Thus, the formula is **True**.
3. **Trace** $\sigma_3 = \emptyset\emptyset\{P, Q\}\emptyset^\omega$: At index 2, both P and Q hold. Since the present is part of the future under the semantics of Eventually, FQ is true at

index 2, making $P \wedge FQ$ true at index 2. Since index 2 is reachable from index 0, the formula is **True**.

4. **Trace** $\sigma_4 = \{P\}^\omega$: Since Q never holds at any index, FQ is always false. The formula is **False**.

Example 11.3.5 (Stabilization / Persistence)**.** Let $\phi = FGQ$ over $AP = \{P, Q\}$. This specifies that eventually Q becomes true and remains true forever. This represents system stabilization.

1. **Trace** $\sigma_A = \{P\}\{Q\}\{P\}\{Q\}^\omega$: At index 3, GQ holds because Q is true at all $j \geq 3$. Thus, FGQ holds at index 0. The formula is **True**.
2. **Trace** $\sigma_B = \{P, Q\}^\omega$: Since Q is true globally, GQ holds at index 0, so FGQ is **True**.
3. **Trace** $\sigma_C = (\{Q\}\{Q\}\{P\})^\omega$: No matter how far we go, there is always a future state containing $\{P\}$ (where Q is false). Thus, GQ never holds at any index, so FGQ is **False**.

Example 11.3.6 (Liveness / Infinitely Many Times)**.** Let $\phi = GFP$. This formula requires P to hold infinitely many times.

- **Connection to S1S**: In Monadic Second-Order Logic, this infinite visitation is written as:
$$\forall x\, \exists y \geq x : P(y)$$
In LTL, the combination GF mirrors this: the outer G acts as the universal quantifier $\forall x$, and the inner F acts as the existential quantifier $\exists y \geq x$.
- **Trace** $\sigma_A = \{P\}\{Q\}\{P\}\{Q\}^\omega$: P only holds at indices 0 and 2. It does not hold infinitely many times. The formula is **False**.
- **Trace** $\sigma_B = (\{Q\}\{Q\}\{P\})^\omega$: P holds at indices $2, 5, 8, \ldots$ which are infinitely many times. The formula is **True**.

11.3.4 Declarative vs. Operational Ordering Blow-Up

A key benefit of LTL is its declarative nature: it specifies *what* properties must hold without describing the operational tracking of states. This allows for extremely compact formulas where ordering is left implicit. However, compiling these formulas to operational representations (such as automata) forces the explicit representation of all possible state interleavings, leading to exponential size blow-ups.

Consider the formula:
$$\psi = \bigwedge_{i=1}^{n} FP_i$$
which requires that each atomic proposition P_i (for $i \in \{1, \ldots, n\}$) must eventually hold.

- **LTL representation**: The size of ψ is linear in n ($O(n)$).
- **Automata representation**: A Büchi automaton recognizing the language of ψ must explicitly keep track of which propositions have already been visited. It must branch to accommodate all $n!$ possible arrival orders of the events P_i. Thus, the state space of the automaton is of size $O(2^n)$, representing the power set of $\{P_1, \ldots, P_n\}$. This illustrates the declarative-to-operational gap.

11.4 Expressive Power and Boundaries

LTL is a powerful specification language, but it has strict expressive boundaries.

11.4.1 Kamp's Theorem

Kamp's landmark theorem (1968) establishes the expressive equivalence of linear-time formalisms:

Theorem 11.4.1 (Kamp). *Let $L \subseteq \Sigma^\omega$ be an ω-language. The following are equivalent:*

1. *L is definable by an LTL formula.*
2. *L is definable by a first-order formula over the signature* $(\mathbb{N}, 0, +1, <, \{Q_a\}_{a \in \Sigma})$.
3. *L is a star-free ω-regular language.*

11.4.2 The Modulo Counting Barrier

A key consequence of Kamp's Theorem is that LTL cannot perform modulo counting.

Proposition 11.4.2 (Non-LTL-Definable Language). *The language* $L = \{\sigma \in (2^{\{P\}})^\omega \mid \forall i \geq 0, (i \text{ is even}) \implies P \in s_i\}$ *is not definable in LTL.*

Intuition. To recognize L, an automaton must count modulo 2 (alternating between even and odd positions) while accepting any valuation on odd steps. This unconstrained modulo cycle prevents the language from being counter-free, placing it outside the star-free regular languages. □

Conversely, if the states at odd positions are also constrained, we can express the alternation locally:

Proposition 11.4.3 (LTL-Definable Alternation). *The language* $L' = \{\sigma \in (2^{\{P\}})^\omega \mid \forall i \geq 0, P \in s_i \iff i \text{ is even}\}$ *is definable in LTL by the formula:*

$$\phi_{L'} = P \wedge G(P \iff X\neg P)$$

11.5 Linear Temporal Logic with Past

To make specifications more compact, LTL can be extended with past-time operators, yielding LTL + P.

11.5.1 Syntax and Semantics

The past-time operators are Yesterday (Y_s, strong previous), Weak Yesterday (Y_w), and Since (S, past-time counterpart of Until):

$$\begin{aligned}
\sigma, i \models Y_s\phi &\iff i > 0 \text{ and } \sigma, i-1 \models \phi \\
\sigma, i \models Y_w\phi &\iff i = 0 \text{ or } (i > 0 \text{ and } \sigma, i-1 \models \phi) \\
\sigma, i \models \phi_1 S \phi_2 &\iff \exists j \in [0, i] \text{ s.t. } \sigma, j \models \phi_2 \text{ and } \forall k \in [j+1, i], \sigma, k \models \phi_1
\end{aligned}$$

We derive Once ($O\phi \equiv \top S\phi$) and Historically ($H\phi \equiv \neg O\neg\phi$).

Unrolling Rules for Past Operators. Just like future temporal operators, past operators satisfy expansion recurrence relations looking backward in time:

$$\begin{aligned}\phi_1 S \phi_2 &\equiv \phi_2 \vee (\phi_1 \wedge Y_s(\phi_1 S \phi_2))\\ O\phi &\equiv \phi \vee Y_s(O\phi)\\ H\phi &\equiv \phi \wedge Y_w(H\phi)\end{aligned}$$

The Initial-State Anchor. The weak yesterday operator applied to false acts as a unique filter identifying the start of time:

$$\sigma, i \models Y_w\bot \iff i = 0$$

This allows formulas evaluating in the past to detect when they have reached the initial state, serving as a boundary anchor during backward unrolling.

Extended Closure for LTL + P. To support the translation of formulas with past operators to automata, we define the *extended closure* $Cl_{ex}(\psi)$ of an LTL + P formula ψ as the smallest set containing ψ such that:

1. If $\chi \in Cl_{ex}(\psi)$, then $\neg\chi \in Cl_{ex}(\psi)$ (with $\neg\neg\alpha \equiv \alpha$).
2. If $\chi_1 \wedge \chi_2$ or $\chi_1 \vee \chi_2 \in Cl_{ex}(\psi)$, then $\chi_1, \chi_2 \in Cl_{ex}(\psi)$.
3. If $X\chi \in Cl_{ex}(\psi)$, then $\chi \in Cl_{ex}(\psi)$.
4. If $Y_s\chi$ or $Y_w\chi \in Cl_{ex}(\psi)$, then $\chi \in Cl_{ex}(\psi)$.
5. If $\chi_1 U \chi_2 \in Cl_{ex}(\psi)$, then $\chi_1, \chi_2 \in Cl_{ex}(\psi)$ and $X(\chi_1 U \chi_2) \in Cl_{ex}(\psi)$.
6. If $\chi_1 R \chi_2 \in Cl_{ex}(\psi)$, then $\chi_1, \chi_2 \in Cl_{ex}(\psi)$ and $X(\chi_1 R \chi_2) \in Cl_{ex}(\psi)$.
7. If $\chi_1 S \chi_2 \in Cl_{ex}(\psi)$, then $\chi_1, \chi_2 \in Cl_{ex}(\psi)$ and $Y_s(\chi_1 S \chi_2) \in Cl_{ex}(\psi)$.

11.5.2 Gabbay's Equivalence vs. Succinctness

Past operators do not add expressive power, but they make formulas exponentially smaller.

Theorem 11.5.1 (Gabbay's Separation Theorem)**.** *LTL + P is expressively equivalent to pure-future LTL.*

Theorem 11.5.2 (Succinctness of LTL + P)**.** *LTL + P is exponentially more succinct than pure-future LTL.*

Proof. For each $n \geq 1$, let $AP_n = \{P_0, \ldots, P_n\}$. We define the language family $A_n \subseteq (2^{AP_n})^\omega$:

$$A_n = \left\{ w \in (2^{AP_n})^\omega \;\middle|\; \forall i \geq 0 : \left(\left[\bigwedge_{j=1}^{n} (P_j \in w_i \iff P_j \in w_0) \right] \implies (P_0 \in w_i \iff P_0 \in w_0) \right) \right\}$$

That is, any state i that agrees with the initial state on $\{P_1, \ldots, P_n\}$ must also agree on P_0.

1. Linear LTL + P Formula. We can define A_n in LTL + P with a formula of size $O(n)$:

$$\phi_n = G\left(\bigwedge_{j=1}^{n} (P_j \iff O(Y_w\bot \wedge P_j)) \implies (P_0 \iff O(Y_w\bot \wedge P_0)) \right)$$

Since $Y_w\bot$ anchors the evaluation to index 0, $O(Y_w\bot \wedge P_j)$ retrieves the value of P_j at state 0.

2. Exponential Lower Bound for LTL. We generalize A_n to B_n, where any two matching states i, j must agree on P_0:

$$B_n = \left\{ w \in (2^{AP_n})^\omega \;\middle|\; \forall i, j \geq 0 : \left(\left[\bigwedge_{k=1}^{n} (P_k \in w_i \iff P_k \in w_j) \right] \implies (P_0 \in w_i \iff P_0 \in w_j) \right) \right\}$$

> **Lemma 11.5.3** (Connection Lemma). *If A_n is definable by a pure-future LTL formula ϕ_n of size $f(n)$, then B_n is definable by $\psi_n = G\phi_n$ of size $f(n) + 1$.*

Proof. $w \models G\phi_n \iff \forall k \geq 0, w^{(k)} \models \phi_n$. Since ϕ_n defines A_n, this requires every suffix $w^{(k)}$ to satisfy the A_n condition. This translates to verifying the implication for all pairs $k, k+i$, which is exactly the definition of B_n. □

> **Lemma 11.5.4.** *Any NBA recognizing B_n must have at least 2^{2^n} states.*

Proof. Let $L = 2^n$. Let $\bar{A} = S_0 S_1 \ldots S_{L-1}$ be a finite word enumerating all possible subsets of $\{P_1, \ldots, P_n\}$. For each subset $K \subseteq \{0, \ldots, L-1\}$, define the word $\bar{A}_K$ by adding P_0 to states at indices in K. The infinite word $w_K = (\bar{A}_K)^\omega$ belongs to B_n by construction.

Assume an NBA $\mathcal{B}$ recognizing B_n has fewer than 2^{2^n} states. Then, by pigeonhole principle, there must be two distinct subsets $K \neq K'$ such that the runs of $\mathcal{B}$ on w_K and $w_{K'}$ reach the same state $q_K = q_{K'}$ after reading the first block $\bar{A}_K$ and $\bar{A}_{K'}$. This allows the automaton to accept the hybrid word $w^* = \bar{A}_{K'} \cdot (\bar{A}_K)^\omega$ (since the state runs merge at q_K). However, because $K \neq K'$, there is some index r where $r \in K'$ and $r \notin K$. Comparing positions r and $L + r$ in w^*: they share the same $\{P_1, \ldots, P_n\}$ signature S_r, but differ on P_0 (P_0 is true in the first block, false in the second). Thus, $w^* \notin B_n$, contradicting that $\mathcal{B}$ recognizes B_n. Hence, $\mathcal{B}$ must have at least 2^{2^n} states. □

The run merging logic is represented visually in Figure 11.6.

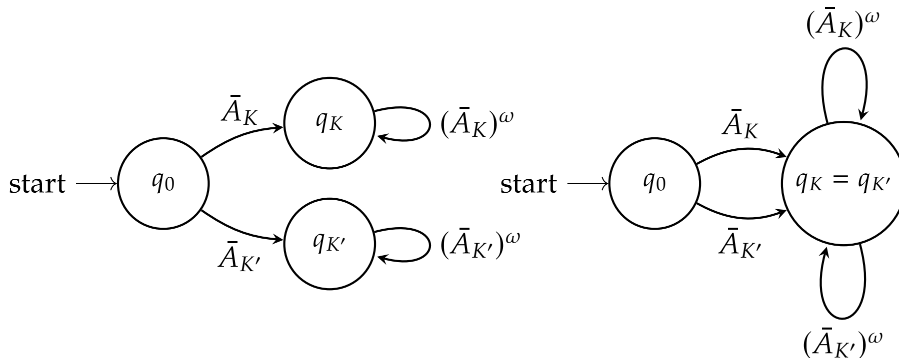


Figure 11.6: Run merging argument: if the NBA has fewer than 2^{2^n} states, two distinct runs on w_K and $w_{K'}$ must merge at some state, causing the automaton to incorrectly accept the hybrid trace w^*.

Combining the pieces: if A_n were definable by a sub-exponential pure-future LTL formula ϕ_n ($|\phi_n| < 2^{cn}$), then B_n would be definable by $\psi_n = G\phi_n$ of size $< 2^{cn} + 1$. Translating ψ_n to an NBA would yield an automaton of size $2^{O(|\psi_n|)} < 2^{2^n}$ for large n. This contradicts the 2^{2^n} lower bound. Thus, any pure-future LTL formula defining A_n must be at least exponential in n. □

11.6 LTL Satisfiability: The Tableau Construction

The LTL satisfiability problem is decided via a graph-based representation called the *tableau*.

Whiteboard intuition: What is a Tableau? Imagine we are handed an LTL formula ϕ and asked: "Does there exist an infinite trace that satisfies this?" To answer this, we will build a massive state-space graph called the tableau. Every node in this graph will be a "candidate state" (called an atom) that contains a consistent set of formulas it claims are true. Edges will represent valid single-step time transitions. If we can find an infinite path through this graph starting from a state that claims ϕ is true, we have found our model! However, there is a trap: to handle "eventually" properties (Beta formulas), nodes are allowed to postpone their obligations to the future. This means our graph will contain "spurious" paths that endlessly postpone promises and lie about the future. Our final job will be to filter out those lying paths.

11.6.1 Negation Normal Form (NNF)

In NNF, negations are pushed inward to literals. To negate Until formulas, we introduce the *Release* operator (R), the dual of Until:

$$\phi_1 R \phi_2 \equiv \neg(\neg\phi_1 U \neg\phi_2)$$

Semantically, $\phi_1 R \phi_2$ requires ϕ_2 to hold up to and including the point where ϕ_1 becomes true; if ϕ_1 never holds, ϕ_2 must hold forever. The unrolling rule for Release is:

$$\phi_1 R \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee X(\phi_1 R \phi_2))$$

11.6.2 Classification of Formulas: Alpha and Beta Formulas

To systematically analyze formulas and build the tableau, we classify temporal and logical formulas in NNF into two categories based on their branching behavior: **Alpha** (universal/conjunctive) and **Beta** (existential/disjunctive) formulas.

Alpha Formulas (α)

An alpha formula behaves like a conjunction: if it is true, all of its constituents must be true in the same state. For each alpha formula $\alpha \in Cl(\phi)$, we define its constituent set $K(\alpha)$:

Formula α	Constituent Set $K(\alpha)$
$\psi_1 \wedge \psi_2$	$\{\psi_1, \psi_2\}$
$G\psi$	$\{\psi, XG\psi\}$

The semantic rule is that a state sequence σ satisfies α at position j if and only if it satisfies all formulas in $K(\alpha)$:

$$\sigma, j \models \alpha \iff \forall \chi \in K(\alpha),\ \sigma, j \models \chi$$

Operational meaning of Alpha. Alpha formulas are deterministic enforcements. There is no choice here. If you claim α is true today, you are strictly forced to also claim everything in $K(\alpha)$ is true today.

Beta Formulas (β)

A beta formula behaves like a disjunction: it introduces branching (non-determinism). If a beta formula holds, at least one of its two constituent sets must hold. For each beta formula $\beta \in Cl(\phi)$, we define two constituent sets $K_1(\beta)$ and $K_2(\beta)$:

Formula β	First Set $K_1(\beta)$	Second Set $K_2(\beta)$
$\psi_1 \vee \psi_2$	$\{\psi_1\}$	$\{\psi_2\}$
$F\psi$	$\{\psi\}$	$\{XF\psi\}$
$\psi_1 U \psi_2$	$\{\psi_2\}$	$\{\psi_1, X(\psi_1 U \psi_2)\}$
$\psi_1 R \psi_2$	$\{\psi_1, \psi_2\}$	$\{\psi_2, X(\psi_1 R \psi_2)\}$

The semantic rule is that a state sequence σ satisfies β at position j if and only if it satisfies all formulas in $K_1(\beta)$ or all formulas in $K_2(\beta)$:

$$\sigma, j \models \beta \iff (\forall \chi \in K_1(\beta),\ \sigma, j \models \chi) \vee (\forall \chi \in K_2(\beta),\ \sigma, j \models \chi)$$

Operational meaning of Beta. Beta formulas represent non-deterministic choices. For example, if you claim $F\psi$ is true today, you must choose a path: either fulfill it today by claiming ψ is true now (First Set), or postpone the obligation by claiming $XF\psi$ is true, kicking the can to tomorrow (Second Set).

11.6.3 Closure and Atoms

Definition 11.6.1 (Closure). The closure $Cl(\phi)$ of an LTL formula ϕ in NNF is the smallest set of formulas containing ϕ such that:

1. If $\chi \in Cl(\phi)$, then $\neg\chi \in Cl(\phi)$ (with $\neg\neg\alpha \equiv \alpha$).
2. If $\chi_1 \wedge \chi_2$ or $\chi_1 \vee \chi_2 \in Cl(\phi)$, then $\chi_1, \chi_2 \in Cl(\phi)$.
3. If $X\chi \in Cl(\phi)$, then $\chi \in Cl(\phi)$.
4. If $\chi_1 U \chi_2 \in Cl(\phi)$, then $\chi_1, \chi_2 \in Cl(\phi)$ and $X(\chi_1 U \chi_2) \in Cl(\phi)$.
5. If $\chi_1 R \chi_2 \in Cl(\phi)$, then $\chi_1, \chi_2 \in Cl(\phi)$ and $X(\chi_1 R \chi_2) \in Cl(\phi)$.

The size of $Cl(\phi)$ satisfies $|Cl(\phi)| \leq 4|\phi|$.

Definition 11.6.2 (ϕ-Atom). A ϕ-atom is a subset $A \subseteq Cl(\phi)$ that satisfies the following three conditions:

1. **Propositional completeness and consistency**: For every formula $\psi \in Cl(\phi)$, exactly one of $\{\psi, \neg\psi\}$ is in A. Moreover, $\top \in A$ and $\bot \notin A$.

2. **Alpha consistency**: For every alpha formula $\alpha \in Cl(\phi)$, if $\alpha \in A$ then $K(\alpha) \subseteq A$. If $\alpha \notin A$, then $K(\alpha) \not\subseteq A$ (meaning at least one constituent is missing from A).
3. **Beta consistency**: For every beta formula $\beta \in Cl(\phi)$, if $\beta \in A$ then $K_1(\beta) \subseteq A$ or $K_2(\beta) \subseteq A$. If $\beta \notin A$, then both $K_1(\beta) \not\subseteq A$ and $K_2(\beta) \not\subseteq A$.

Basic Formulas. A formula is called *basic* if it is an atomic proposition $p \in AP$ or a next-step formula $X\psi$. An important algebraic property of the closure is that a ϕ-atom is uniquely determined by the truth values of its basic formulas. All non-basic formulas are reconstructed bottom-up by propagating the alpha and beta consistency rules. This reduces the search space of possible atoms from $2^{|Cl(\phi)|}$ to the combinations of truth assignments to the basic formulas (at most 2^k, where k is the number of basic formulas in the closure).

11.6.4 Tableau Graph

Definition 11.6.3 (Tableau)**.** The tableau $T(\phi)$ is a directed graph (V, E) where:

- V is the set of all ϕ-atoms.
- $E \subseteq V \times V$ is the transition relation, defined as:

$$(A, B) \in E \iff \forall X\chi \in Cl(\phi) : X\chi \in A \iff \chi \in B$$

Operational meaning of the Tableau. Operationally, the tableau is a massive graph where every node is a candidate state of our system (an atom), and edges represent valid single-step transitions forward in time. An edge exists from A to B if and only if all the "tomorrow" promises made in A ($X\chi$) are kept "today" in B (χ). Finding a model for our formula reduces to finding a valid infinite path through this graph starting from an atom containing ϕ.

11.7 Fulfilling Paths and SCS Reduction

An infinite path in $T(\phi)$ is a sequence of atoms $\pi = A_0A_1A_2\ldots$ with $(A_i, A_{i+1}) \in E$.

11.7.1 Induced Paths and the Infinite Postponement Problem

Does every path in the tableau correspond to a real LTL model? To answer this, we look at the concept of an *induced path.*

If we are given an actual, true LTL model $\sigma = s_0s_1s_2\ldots$ that satisfies ϕ, we can trace its exact footprint through the tableau. For each step i, we define an atom A_i containing exactly the formulas that are true at step i: $A_i = \{\psi \in Cl(\phi) \mid \sigma, i \models \psi\}$. Because σ is a real model, this sequence of atoms $A_0A_1A_2\ldots$ is guaranteed to be a valid path in $T(\phi)$. We say the model σ *induces* this path. This proves a critical property: **the tableau does not lose anything.** Any valid model lives inside the tableau as an induced path.

But the reverse is not true! There is an asymmetry. Consider the trace where P is always true: $\sigma = \{P\}^\omega$. Now, let's try to evaluate the formula $F\neg P$. In the tableau, an atom might claim $F\neg P$ is true by using the Beta non-determinism to choose $XF\neg P$ (postponing the obligation to tomorrow). At the

next step, it postpones again. And again. The tableau allows this infinite chain of postponements because it only enforces local step-by-step consistency. Thus, the tableau contains a "spurious" path that endlessly kicks the can down the road, claiming $F\neg P$ is true but never actually delivering $\neg P$. This is the *infinite postponement problem*.

Because of this asymmetry, we must filter the paths in the tableau. We only want paths that don't lie about the future.

11.7.2 Fulfilling Paths

Definition 11.7.1 (Promising Formulas and Fulfilling Paths)**.** A formula $\psi \in Cl(\phi)$ is *promising* if it is of the form $\alpha U \beta$. It promises β. An atom A *fulfills* ψ if $\neg\psi \in A$ or $\beta \in A$. A path π is *fulfilling* if for every promising formula $\psi \in Cl(\phi)$, π contains infinitely many atoms that fulfill ψ.

Theorem 11.7.2 (Fulfilling Path Theorem)**.** *An LTL formula ϕ is satisfiable if and only if there exists a fulfilling path $\pi = A_0 A_1 \dots$ in $T(\phi)$ such that $\phi \in A_0$.*

Proof. ($\Longrightarrow$) Let $\sigma = s_0 s_1 \dots$ be an LTL model satisfying ϕ. Define the path $\pi = A_0 A_1 \dots$ by setting $A_i = \{\chi \in Cl(\phi) \mid \sigma, i \models \chi\}$. Propositional, local, and temporal consistency follow from LTL semantics, so each A_i is an atom. Edge consistency holds because $X\chi \in A_i \iff \sigma, i \models X\chi \iff \sigma, i+1 \models \chi \iff \chi \in A_{i+1}$. If $\alpha U \beta \in A_i$, then $\sigma, i \models \alpha U \beta$, meaning there is some $j \geq i$ where $\sigma, j \models \beta$, so $\beta \in A_j$, which fulfills the promise. Thus, π is fulfilling.

($\Longleftarrow$) Let $\pi = A_0 A_1 \dots$ be a fulfilling path with $\phi \in A_0$. Define the model $\sigma = s_0 s_1 \dots$ by setting $s_i = A_i \cap AP$. We prove by induction on the structure of $\psi \in Cl(\phi)$ that $\sigma, i \models \psi \iff \psi \in A_i$. The base case ($\psi = p \in AP$), negation, and conjunction steps follow immediately from the definition of σ and propositional consistency of atoms.

- **Next Step** ($\psi = X\chi$): $\sigma, i \models X\chi \iff \sigma, i+1 \models \chi \iff \chi \in A_{i+1}$ (by induction hypothesis) $\iff X\chi \in A_i$ (by edge relation $A_i \longrightarrow A_{i+1}$).
- **Until Step** ($\psi = \alpha U \beta$):
  - If $\alpha U \beta \in A_i$: since π is fulfilling, there is a smallest index $j \geq i$ such that A_j fulfills the promise. If $j = i$, we must have $\beta \in A_i$ (otherwise $\neg(\alpha U \beta) \in A_i$, a contradiction). By induction hypothesis, $\sigma, i \models \beta$, so $\sigma, i \models \alpha U \beta$. If $j > i$, then for all $k \in [i, j-1]$, A_k does not fulfill the promise, so $\beta \notin A_k$, forcing $\alpha \in A_k$ and $X(\alpha U \beta) \in A_k$ by the temporal consistency rule. By edge transition, this propagates to $\alpha U \beta \in A_j$. Since A_j is fulfilling, $\beta \in A_j$, meaning $\sigma, j \models \beta$. By induction hypothesis, $\sigma, k \models \alpha$ for all $k \in [i, j-1]$. Thus, $\sigma, i \models \alpha U \beta$.
  - If $\alpha U \beta \notin A_i$: then $\neg(\alpha U \beta) \in A_i$. By temporal consistency, $\neg\beta \in A_i$ and $\neg X(\alpha U \beta) \in A_i$. Inductively, this propagates: if β were true at any future step $j \geq i$, we would have $\beta \in A_j$, contradicting the propagation of $\neg\beta$. Thus, $\sigma, i \not\models \alpha U \beta$. □

11.7.3 Reduction to Reachable Strongly Connected Subgraphs

Searching for infinite fulfilling paths on a finite graph reduces to strongly connected subgraphs (SCS).

Definition 11.7.3 (Fulfilling SCS)**.** A strongly connected subgraph $S \subseteq T(\phi)$ is non-transient if it contains at least one cycle. A non-transient SCS S is fulfilling if for every promising formula $\alpha U \beta \in Cl(\phi)$ and every node $u \in S$:

$$\alpha U \beta \in u \implies \exists v \in S \text{ such that } \beta \in v$$

Theorem 11.7.4. *An LTL formula ϕ is satisfiable if and only if the tableau $T(\phi)$ contains a fulfilling SCS S reachable from some initial atom A_0 containing ϕ.*

If such an SCS exists, we can construct an *ultimately periodic model* of the form $u \cdot v^{\omega}$.

11.7.4 Tableau Pruning Rules

Instead of checking all SCSs, we partition the graph into Maximal Strongly Connected Subgraphs (MSCS) and apply two pruning rules:

1. **Reachability Pruning**: Delete all states not reachable from a ϕ-atom.
2. **Terminal MSCS Pruning**: An MSCS S is terminal if it has no outgoing transitions to states outside S. If a terminal MSCS is not fulfilling, prune it entirely. Since it is terminal, a path cannot leave it, and since it is not fulfilling, it cannot contain any valid infinite path.

11.7.5 Detailed Case Study: $\phi = \Box P \land \Diamond \neg P$

We verify the unsatisfiability of $\phi = \Box P \land \Diamond \neg P$ (written in NNF). The closure is $Cl(\phi) = \{\phi, \neg\phi, \Box P, \neg\Box P, \Diamond\neg P, \neg\Diamond\neg P, P, \neg P, X\Box P, \neg X\Box P, X\Diamond\neg P, \neg X\Diamond\neg P\}$. We identify the 8 atoms based on the basic formulas P, $X\Box P$, and $X\Diamond\neg P$:

Atom	P	$X\Box P$	$X\Diamond\neg P$	$\Box P$	$\Diamond\neg P$	ϕ	Promises
A_0	0	0	0	0	1	0	$\Diamond\neg P, \neg\Box P$
A_1	0	0	1	0	1	0	$\Diamond\neg P, \neg\Box P$
A_2	0	1	0	0	1	0	$\Diamond\neg P, \neg\Box P$
A_3	0	1	1	0	1	0	$\Diamond\neg P, \neg\Box P$
A_4	1	0	0	0	0	0	$\neg\Box P$
A_5	1	0	1	0	1	0	$\Diamond\neg P, \neg\Box P$
A_6	1	1	0	1	0	0	None
A_7	1	1	1	1	1	1	$\Diamond\neg P$

Applying the transition relation:

- From $\{A_0, A_4\}$, where $X\Box P = 0$ and $X\Diamond\neg P = 0$, transitions go to $\{A_4\}$.
- From $\{A_1, A_5\}$, where $X\Box P = 0$ and $X\Diamond\neg P = 1$, transitions go to $\{A_0, A_1, A_2, A_3, A_5\}$.
- From $\{A_2, A_6\}$, where $X\Box P = 1$ and $X\Diamond\neg P = 0$, transitions go to $\{A_6\}$.
- From $\{A_3, A_7\}$, where $X\Box P = 1$ and $X\Diamond\neg P = 1$, transitions go to $\{A_7\}$.

The only state containing ϕ is A_7. The only path starting from A_7 is A_7^{ω}. However, A_7^{ω} contains the promising formula $\Diamond\neg P$, which is never fulfilled because $\neg P$ is false in A_7 (where $P = 1$). Thus, no fulfilling path starting from a ϕ-state exists: ϕ is unsatisfiable.

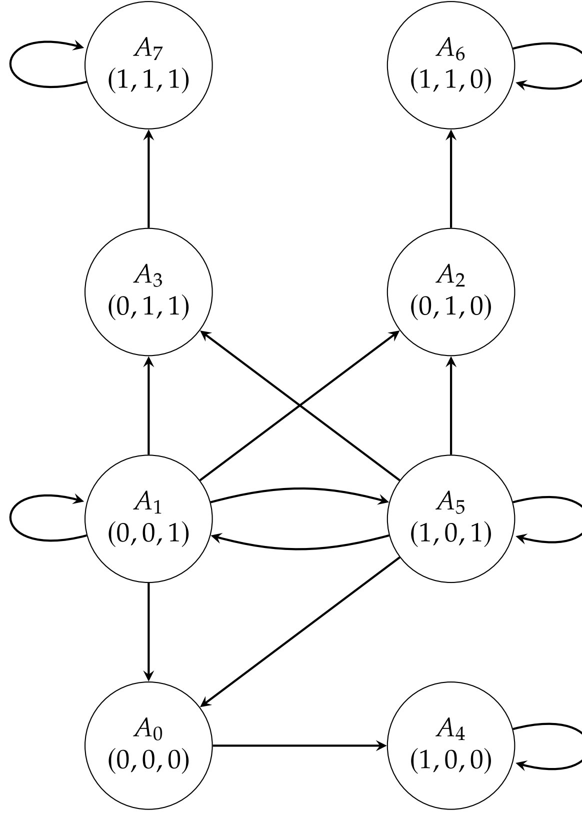


Figure 11.7: Tableau graph for $\phi = \Box P \wedge \Diamond\neg P$. The node labels represent the basic variables $(P, X\Box P, X\Diamond\neg P)$.

11.8 Automata-Theoretic Translation

We compile LTL with past formulas directly into Generalized Büchi Automata.

Definition 11.8.1 (GNBA for LTL + P)**.** Given an LTL + P formula ψ, we define the GNBA $\mathcal{A}_\psi = (Q, \Sigma, I, \Delta, \mathcal{F})$:

1. $\Sigma = 2^{AP}$.
2. Q is the set of all propositionally, locally, and temporally consistent atoms over $Cl_{ex}(\psi)$.
3. $I = \{S \in Q \mid \psi \in S \text{ and } \forall Y\alpha \in Cl_{ex}(\psi), Y\alpha \notin S\}$.
4. $(S, S') \in \Delta$ on symbol a iff:
   - $P \in a \iff P \in S$ (for all $p \in AP$).
   - $X\alpha \in S \iff \alpha \in S'$.
   - $Y\alpha \in S' \iff \alpha \in S$.
5. $\mathcal{F} = \{F_{\alpha U\beta} \mid \alpha U\beta \in Cl_{ex}(\psi)\}$ where $F_{\alpha U\beta} = \{S \in Q \mid \beta \in S \vee \neg(\alpha U\beta) \in S\}$.

To translate the GNBA with acceptance sets $\{F_1, \ldots, F_k\}$ to a standard NBA, we apply a counter construction:

$$
\begin{aligned}
Q' &= Q \times \{1, \ldots, k\} \\
I' &= I \times \{1\} \\
F' &= Q \times \{k\}
\end{aligned}
$$

The transition relation advances the counter from i to $(i \bmod k) + 1$ if and only if the current state belongs to F_i.

Bridge to Chapter 12. This chapter stops at the specification side: syntax, semantics, expressiveness, past operators, satisfiability, and the automata viewpoint. Chapter 12 takes the next step and uses those formulas for model checking over finite systems, with fairness handled there.

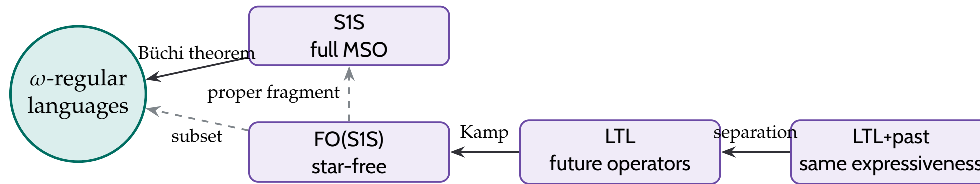


Figure 11.8: Expressiveness map for the temporal-logic part of the book. S1S captures all ω-regular languages; LTL corresponds to the first-order/star-free fragment; adding past operators improves succinctness but not expressiveness.

Summary & Key Takeaways

- *LTL* is evaluated on a single infinite path; its core operators are X (next) and U (until), from which F (eventually) and G (always) are derived.
- LTL corresponds to the *first-order* fragment of S1S and to star-free ω-regular languages. Some ω-regular properties (e.g., "a at every even position") are not LTL-expressible.
- Common specification patterns: $GF\,p$ (infinitely often), $FG\,p$ (eventually always), $G(p \to F\,q)$ (every p triggers a future q).
- Past-time operators (Y, S) add succinctness but not expressive power; every past LTL formula has a pure-future equivalent.
- LTL model checking is **PSPACE-complete** in the formula and polynomial in the model.

Exercises

Exercise 1 (Weak Yesterday Nesting). Let σ be a state sequence.

1. Prove that $\sigma, i \models Y_w\bot \iff i = 0$.
2. Show that $\sigma, i \models Y_w(Y_w(Y_w\bot)) \iff i \leq 2$.
3. Write an LTL + P formula that holds at position i if and only if $i > 2$.

Solution. 1. By the semantic definition of the weak yesterday operator:

$$\sigma, i \models Y_w\bot \iff i = 0 \vee (i > 0 \text{ and } \sigma, i-1 \models \bot)$$

Since $\sigma, i-1 \models \bot$ is false for any suffix position, the second disjunct is always false. Thus:

$$\sigma, i \models Y_w\bot \iff i = 0$$

2. Let $\psi_0 = \bot$ and $\psi_k = Y_w\psi_{k-1}$ for $k \geq 1$. We show by induction on k that $\sigma, i \models \psi_k \iff i \leq k-1$.

   - **Base Case** ($k = 1$): $\sigma, i \models Y_w\bot \iff i \leq 0 \iff i = 0$, which is true by part 1.

- **Inductive Step**: Assume the property holds for k, i.e., $\sigma, i \models \psi_k \iff i \leq k-1$. For $k+1$:

$$\sigma, i \models Y_w \psi_k \iff i = 0 \vee (i > 0 \text{ and } \sigma, i-1 \models \psi_k)$$

By the induction hypothesis, $\sigma, i-1 \models \psi_k \iff i-1 \leq k-1 \iff i \leq k$. Therefore, the satisfaction condition becomes:

$$i = 0 \vee (i > 0 \text{ and } i \leq k) \iff i \leq k$$

which matches $i \leq (k+1) - 1$. This completes the induction.

Setting $k = 3$, we obtain $\sigma, i \models Y_w(Y_w(Y_w \bot)) \iff i \leq 2$.

3. The condition $i > 2$ is the exact negation of the condition $i \leq 2$. Using the formula derived in part 2, the desired formula is:

$$\phi = \neg Y_w(Y_w(Y_w \bot))$$

□

Exercise 2 (Omega-Regular Conversion). Consider the LTL formula $\phi = PUGQ$ over $AP = \{P, Q\}$. Write the equivalent ω-regular expression over $\Sigma = 2^{AP}$.

Solution. The alphabet is $\Sigma = \{\emptyset, \{P\}, \{Q\}, \{P, Q\}\}$. We partition Σ as $\Sigma_P = \{\{P\}, \{P, Q\}\}$ and $\Sigma_Q = \{\{Q\}, \{P, Q\}\}$. The Until requires a suffix where GQ holds, meaning all states belong to Σ_Q. Before this suffix, P must hold, meaning all states belong to Σ_P. Thus, the equivalent expression is:

$$L(\phi) = (\Sigma_P)^*(\Sigma_Q)^\omega = (\{P\} + \{P, Q\})^*(\{Q\} + \{P, Q\})^\omega$$

□

Exercise 3 (LTL Rewrite). Rewrite $\phi = G(PUQ)$ using only boolean operators, X, F, and G, but without using U. Prove the equivalence.

Solution. The equivalent formula is $\phi' = G(P \vee Q) \wedge GFQ$.

- ($\Longrightarrow$) Let $\sigma \models G(PUQ)$. Let $i \geq 0$. Since $\sigma, i \models PUQ$, there is $j \geq i$ where $Q \in s_j$, so FQ holds. Since i is arbitrary, GFQ holds. By the unrolling rule, $\sigma, i \models Q \vee (P \wedge X(PUQ))$, which implies $P \vee Q$ holds. Thus, $G(P \vee Q)$ holds.
- ($\Longleftarrow$) Let $\sigma \models G(P \vee Q) \wedge GFQ$. Let $i \geq 0$. Since GFQ holds, let $j \geq i$ be the first step where Q holds. For all $k \in [i, j-1]$, Q is false. Since $P \vee Q$ holds globally, we must have P true at all $k \in [i, j-1]$. Thus, $\sigma, i \models PUQ$. Since i is arbitrary, $G(PUQ)$ holds.

□

CHAPTER 12

LTL Model Checking and Fairness

Chapter 11 gave LTL its syntax, semantics, and tableau-based satisfiability machinery. We now turn that logical language into a model checking procedure. The shift is small in notation but decisive in meaning: instead of asking whether *some* infinite trace satisfies a formula, we ask whether *all* executions of a finite system satisfy it. The answer is computed by negating the property, translating the negation into a Büchi automaton, taking the product with the system, and searching for an accepting lasso. If such a lasso exists, it is the counterexample. If it does not, the system satisfies the property.

An LTL formula becomes useful to an engineer only when it can be checked against a finite model, and a violated formula becomes useful only when the checker returns a concrete infinite witness.

The lecture material for this chapter has two layers. First comes the classical automata-theoretic model checking pipeline. Then comes the fairness refinement: justice and compassion are path filters that discard unrealistic scheduler behaviours before the property is judged. On a finite graph, fairness is handled by strongly connected component search, which is why the practical algorithms look so much like the emptiness test for Büchi automata from Part I.

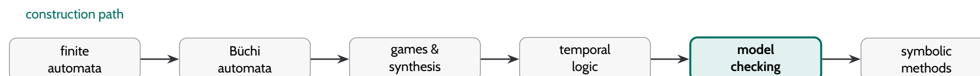


Figure 12.1: Where this chapter sits in the construction path of the book. Each step reuses the previous one as a representation or an algorithmic primitive.

> **■ Running example — bad request/grant lassos**
>
> For model checking, the request/grant property is most useful through its negation. A counterexample has a finite prefix that reaches a pending request, then a cycle where no grant ever appears.

12.1 P-Validity and P-Satisfiability

In Chapter 11, we studied *general satisfiability*: does there exist *any* infinite trace in the universe that satisfies formula φ? Model Checking is simply satisfiability constrained to the paths of a specific finite-state program P.

When we verify a program P against a specification φ, we want to ensure that *every* computation of P satisfies φ. This universal property is called **P-validity**, denoted $P \models \varphi$.

However, searching universally is algorithmically awkward. The key observation behind LTL model checking is to turn this universal verification question

into an existential search for a counterexample. Instead of proving that every execution satisfies φ, we ask: does there exist *at least one* computation of P that satisfies $\neg\varphi$? This existential property is called **P-satisfiability**, denoted $P \models_{sat} \neg\varphi$.

If $P \models_{sat} \neg\varphi$ is true, the satisfying trace is our counterexample, and $P \models \varphi$ is false. If $P \models_{sat} \neg\varphi$ is false, then $P \models \varphi$ is true.

Definition 12.1.1 (Kripke structure)**.** A *Kripke structure* over a set of atomic propositions AP is a tuple

$$\mathcal{M} = (Q, I, R, L)$$

where Q is a finite set of states, $I \subseteq Q$ is the set of initial states, $R \subseteq Q \times Q$ is a total transition relation, and $L : Q \to 2^{AP}$ labels each state with the propositions that hold there.

The LTL model checking problem is:

$$\mathcal{M} \models \varphi \quad ?$$

meaning that every infinite path starting from an initial state of $\mathcal{M}$ satisfies φ.

Theorem 12.1.2 (Automata-theoretic model checking)**.** *Let $\mathcal{M}$ be a Kripke structure and φ an LTL formula. Then*

$$\mathcal{M} \models \varphi \iff L(\mathcal{M}) \cap L(\neg\varphi) = \emptyset \iff \mathcal{M} \times \mathcal{A}_{\neg\varphi} \text{ has no accepting run}$$

where $\mathcal{A}_{\neg\varphi}$ is a Büchi automaton for the negated formula.

Idea. If $\mathcal{M} \not\models \varphi$, then some path of $\mathcal{M}$ satisfies $\neg\varphi$. The automaton $\mathcal{A}_{\neg\varphi}$ accepts exactly those paths, so the product has an accepting run. Conversely, any accepting run of the product projects to a path of $\mathcal{M}$ that satisfies $\neg\varphi$. □

Why negation helps. The universal verification question is hard only because it quantifies over every behaviour. By negating the property we move to an existential problem: find one bad trace. Büchi automata are already designed to recognise existential infinite witnesses, so this is the right target representation.

12.1.1 The Behavior Graph Construction

Instead of building a full Büchi automaton for $\neg\varphi$, the Manna-Pnueli direct algorithm constructs a **Behavior Graph**. The behavior graph is the synchronous product of the Kripke structure (the system) and the LTL Tableau (the logic) from Chapter 11.

Definition 12.1.3 (Consistent Atom)**.** Let s be a state of the system and A be a tableau atom. The pair (s, A) is *consistent* if the physical state s satisfies all the immediate propositional formulas promised by the logical atom A. Formally, for every atomic proposition $p \in AP$, $p \in A \iff p \in L(s)$.

Operationally, an atom represents a set of logical claims about the present and future. A system state represents the physical reality of the present. If the atom claims p is true today, but the system state says p is false, the pairing is inconsistent and discarded.

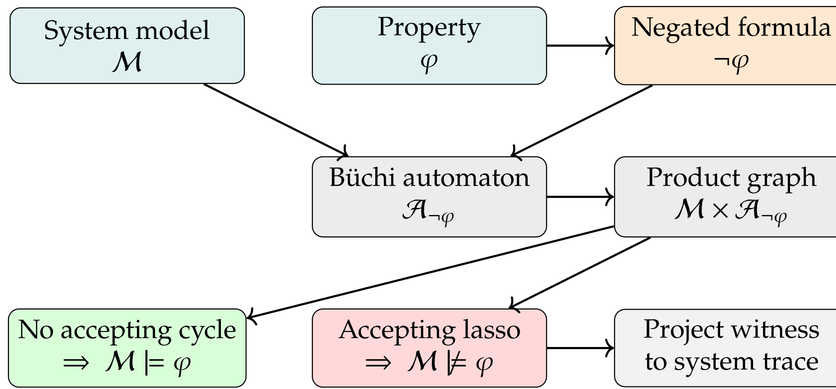


Figure 12.2: The LTL model-checking pipeline. Negate the property, compile the negation to a Büchi automaton, take the product with the system, and search for an accepting cycle.

Definition 12.1.4 (Behavior Graph)**.** Let $\mathcal{M} = (Q, I, R, L)$ be a Kripke structure and $T(\neg\varphi) = (V, E)$ be the tableau for $\neg\varphi$. The behavior graph $\mathcal{B} = \mathcal{M} \times T(\neg\varphi)$ is a directed graph where:

- Nodes are consistent pairs $(s, A) \in Q \times V$.
- Edges $(s, A) \to (s', A')$ exist if and only if $(s, s') \in R$ (a valid system transition) and $(A, A') \in E$ (a valid tableau transition).

Example 12.1.5 (The `loop` system)**.** To make this concrete, consider the `loop` system: a simple modulo-4 counter with states $\{s_0, s_1, s_2, s_3\}$. From any state s_i, the system can take a transition τ_{count} to $s_{(i+1) \bmod 4}$, or it can take an idling transition τ_{idle} to remain in s_i.

If we check the property $\varphi = \mathbf{F}(s_3)$, we build the behavior graph of `loop` $\times\, T(\mathbf{G}\neg s_3)$. A path in this behavior graph might enter s_0, and then the system repeatedly chooses the τ_{idle} transition forever. The corresponding tableau atoms easily accommodate this: they just claim $\neg s_3$ is true today and $\mathbf{G}\neg s_3$ is true tomorrow. This infinite sequence $(s_0, A) \to (s_0, A) \to \ldots$ is a valid, fulfilling path in the behavior graph! It proves P-satisfiability, showing the system can fail to reach s_3.

The intuition is straightforward. The system chooses the next concrete state; the tableau restricts the logical futures; the behavior graph records both constraints at once. A counterexample is therefore not just a path in the system, but a path annotated with the logical atoms that explain why the path violates the property.

Example 12.1.6 (A lasso counterexample)**.** Consider the property

$$\varphi = \mathbf{G}(r \to \mathbf{F}g)$$

over a system where a request r may become true and then remain pending forever if no grant g occurs. The negation is the statement that there is some point after which a request is never followed by a grant.

Suppose the system has a path

$$s_0 \to s_1 \to s_1 \to s_1 \to \cdots$$

where r becomes true in s_1 and g never appears again. The product with $\mathcal{A}_{\neg\varphi}$ contains a reachable accepting cycle on the pair corresponding to s_1. The checker reports the witness as the lasso

$$s_0 \to s_1 \to (s_1)^\omega.$$

The finite prefix gets us to the bad region; the cycle shows that we can stay there forever.

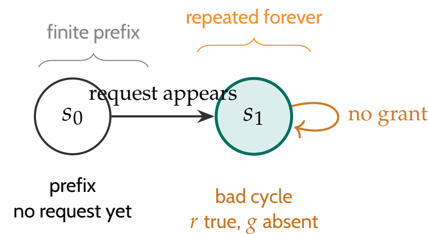


Figure 12.3: A lasso counterexample to $\mathbf{G}(r \to \mathbf{F}g)$. The prefix reaches a state with a pending request; the orange accepting cycle witnesses that no grant is forced afterwards.

Why the witness is a lasso. The product graph is finite. Any infinite path in a finite graph repeats a state, and once a state repeats we can separate the path into a finite prefix and a cycle. That is exactly the shape of a Büchi witness: a finite prefix that reaches an accepting cycle, followed by that cycle repeated forever.

12.2 Direct Algorithms and SCC Search

The product construction gives the conceptual answer, but real tools rarely build the whole product explicitly. They search it on the fly, because the product can be enormous even when the original model is small.

12.2.1 On-the-fly emptiness checking

The direct algorithm is the model-checking analogue of the Büchi emptiness test from Part I. It explores the reachable portion of the product graph and looks for a reachable accepting cycle.

1. Start from an initial product state.
2. Explore successors depth-first, generating product states only when they are needed.
3. When an accepting state is encountered, check whether it can reach itself again inside the reachable subgraph.
4. If it can, record the prefix and the cycle as a lasso counterexample.

This is why the algorithm is often described as *direct*. The checker never needs the full product in memory; it only needs enough information to continue the current search and to reconstruct a witness when one is found. In practice,

this is usually implemented with a depth-first traversal, sometimes with a nested search for the closing cycle.

12.2.2 Strongly connected components

The graph-theoretic view becomes cleaner if we compress the reachable part of the product into strongly connected components.

Definition 12.2.1 (Accepting SCC)**.** An SCC of $\mathcal{M} \times \mathcal{A}_{\neg\varphi}$ is *accepting* if it is reachable from an initial product state, contains at least one accepting automaton state, and contains a directed cycle.

Proposition 12.2.2 (SCC criterion for counterexamples)**.** *The model $\mathcal{M}$ violates φ if and only if the reachable part of $\mathcal{M} \times \mathcal{A}_{\neg\varphi}$ contains an accepting SCC.*

Sketch. Any accepting run eventually stays inside one SCC, because the product graph is finite. If the SCC contains an accepting state, the run visits it infinitely often, so the SCC yields a lasso. Conversely, every accepting lasso obviously sits inside a reachable SCC. □

The SCC criterion is the same structural idea that we used for Büchi emptiness in Part I and for LTL satisfiability in Chapter 11. The difference is only where the graph comes from: here it is a product of a system and an automaton rather than a standalone automaton.

12.2.3 Complexity

Proposition 12.2.3 (Complexity of LTL model checking)**.** *LTL model checking is PSPACE-complete.*

Proof sketch. The upper bound comes from the on-the-fly algorithm. A product state can be represented with one system state and one automaton state, both of which have polynomial descriptions. The depth-first search stores only the current search stack and a bounded amount of local information, so the procedure uses polynomial space.

The lower bound is the standard PSPACE-hardness of LTL satisfiability, which embeds into model checking by asking whether a suitable finite model admits a path satisfying the formula. In other words, model checking already contains the same hard infinite-word search problem that appears in satisfiability. □

The practical point is that explicit state counts can still explode exponentially in the size of the formula, even though the decision problem is only PSPACE-complete. That is the usual verification trade-off: the worst case is hard, but the graph search is structured enough to make many real instances tractable.

12.3 Fairness

Chapter 7 introduced fairness as a way to ignore unrealistic scheduler behaviours. Here we use the same idea algorithmically: fairness is a filter on the paths we are allowed to consider when searching for a counterexample.

The intuition is easiest to state on paths, and then to translate to SCCs. Suppose a system has several processes competing for the next transition. A path that permanently starves one process is usually not the execution we mean to reason about. Fairness says that such paths should not count against the specification.

12.3.1 Transition-Based Justice and Compassion

In the Manna-Pnueli formalism taught in this course, fairness is defined over *transitions* (τ) rather than states. We restate the two fairness notions at the level of strongly connected components (SCCs) in the behavior graph.

> **Definition 12.3.1** (Transition-based Justice and Compassion)**.** Let S be a Strongly Connected Component (SCC) in the behavior graph.
>
> - **Justice (Weak Fairness)**: A transition τ is *just* in S if either τ is taken in S (i.e., there is an edge in S corresponding to τ), or τ is *not enabled* in some node within S.
> - **Compassion (Strong Fairness)**: A transition τ is *compassionate* in S if either τ is taken in S, or τ is *not enabled* in **all** nodes within S.

Operational meaning of Justice. If you cycle inside S forever, a transition is just if you either eventually take it, or you at least visit some state where it's impossible to take it (giving you an excuse). If it is enabled *everywhere* in S but you *never* take it, that is an unfair starvation, and the SCC violates justice.

Operational meaning of Compassion. If you cycle inside S forever, a transition is compassionate if you either eventually take it, or you never even have the choice to take it (it's disabled everywhere in S). If it is enabled in *even one* state in S, then you see the opportunity infinitely often as you cycle. If you never take it, you are violating compassion.

12.3.2 The Adequate Subgraph

On a finite graph, every infinite path eventually stays inside one SCC. That is the point where fairness and tableau fulfillment become easy to test together.

> **Definition 12.3.2** (Adequate Subgraph)**.** A reachable SCC S of the behavior graph is an *adequate subgraph* if it satisfies three conditions:
>
> 1. **Fulfilling**: For every promising formula $\alpha U \beta$ present in any atom in S, there exists a node in S whose atom fulfills the promise (β or $\neg(\alpha U \beta)$).
> 2. **Just**: Every transition τ in the system is just in S.
> 3. **Compassionate**: Every transition τ in the system is compassionate in S.

This 3-part criterion is the core of the direct algorithm. It combines the logical requirement (the path must be a real model, not a spurious postponed path) with the scheduler requirements (the path must be realistic).

The practical algorithm therefore builds the adequate subgraph: the reachable part of the behavior graph after pruning every SCC that fails any of the three adequacy conditions. Once the pruning is done, the remaining graph contains exactly the fair, fulfilling counterexamples.

Example 12.3.3 (The `loop+` system). Building on Example 12.1.5, consider the `loop+` system, which adds a transition τ_3 from s_0 directly to s_3. Suppose the fairness conditions require τ_3 to be compassionate. If we check $\varphi = \mathbf{F}(s_3)$ and get the same infinite cycle on s_0 using τ_{idle}, is it an adequate subgraph?

- It is Fulfilling (no unmet U promises in $\mathbf{G}\neg s_3$).
- It is Just (because τ_3 is enabled everywhere in the SCC but never taken, it fails Justice too!).

Actually, τ_3 being enabled at s_0 but never taken violates *both* justice and compassion in that single-node SCC. The SCC is pruned, and no adequate subgraph exists, correctly proving $P \models \mathbf{F}(s_3)$ under fairness!

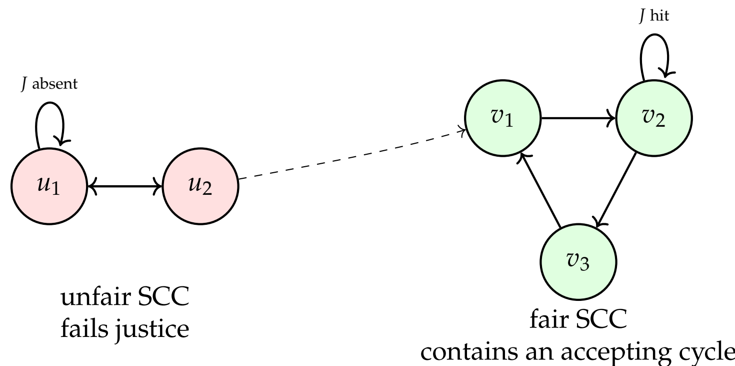


Figure 12.4: Fairness prunes SCCs before the accepting-cycle search. The left SCC is unreachable as a fair witness because it violates justice; the right SCC remains as a candidate fair cycle.

12.3.3 Reading a fair counterexample

The final witness returned by a fair model checker is still a lasso. The difference is that the loop must be fair. So the reading protocol is:

1. follow the prefix until the system enters the bad region;
2. check that the cycle is reachable inside a fair SCC;
3. verify that the cycle satisfies the fairness constraints;
4. project the witness back to the system and explain it in the model's vocabulary.

This is exactly why fairness matters in practice. Without it, a checker may hand you a technically valid but operationally meaningless lasso. With it, the returned witness matches the scheduler assumptions encoded in the model.

12.4 Summary

Summary & Key Takeaways

- LTL model checking asks whether every system execution satisfies a property.
- The standard reduction negates the property, translates the negation to a Büchi automaton, and searches the product for an accepting cycle.
- A counterexample is always a lasso: a finite prefix followed by a repeating cycle.
- Direct implementations search the implicit product on the fly and use SCC structure to find accepting cycles efficiently.
- LTL model checking is PSPACE-complete.
- Fairness filters unrealistic runs. Justice and compassion are checked by SCC pruning: a reachable SCC is fair only if it meets the justice constraints and respects every compassion pair.

Exercises

Exercise 1 (Negating the property). Let $\varphi = \mathbf{G}(r \to \mathbf{F}g)$. Write an LTL formula for $\neg\varphi$ that makes the counterexample pattern explicit. Then describe, in words, the shape of a trace accepted by an automaton for $\neg\varphi$.

Exercise 2 (Reading a lasso). Consider the infinite trace $s_0\, s_1\, (s_2\, s_3)^\omega$ with labels $L(s_0) = \emptyset$, $L(s_1) = \{r\}$, $L(s_2) = \{r\}$, $L(s_3) = \emptyset$. Does this trace violate $\mathbf{G}(r \to \mathbf{F}g)$? Explain which part of the lasso is the finite prefix and which part is the repeating witness.

Exercise 3 (Accepting SCC criterion). Suppose the reachable product graph contains an SCC with three nodes u, v, w, edges $u \to v$, $v \to w$, $w \to u$, and $v \to v$. The accepting automaton states occur only at v. Is this SCC enough to prove a counterexample? How would the answer change if v were not reachable from an initial product state?

Exercise 4 (Justice pruning). An SCC consists of a single node s with a self-loop labelled τ_{idle}. A transition τ_{go} is enabled at s but no edge in the SCC is labelled τ_{go}. Does this SCC satisfy justice for τ_{go} under the fairness definitions in section 12.3? Explain why the corresponding lasso should or should not survive.

Exercise 5 (Compassion pruning). An SCC has two nodes s and t. Transition τ is enabled at s, disabled at t, and never taken inside the SCC. Does the SCC satisfy justice for τ? Does it satisfy compassion for τ? Use the definitions to justify the difference.

Exercise 6 (Adequacy checklist). Build a three-row checklist for an SCC that might be returned as a fair LTL counterexample. The rows should correspond to fulfillment, justice, and compassion. For each row, write one concrete reason why the SCC could fail the check.

CHAPTER 13

CTL and Symbolic Model Checking

The last chapter completed the linear-time story: LTL formulas describe a single execution, automata turn the negated property into an emptiness problem, and fairness prunes away unreal schedulers. That pipeline is powerful, but it has one conceptual limitation: it can only talk about one path at a time.

Chapter 12 treated each execution as a single line and searched that line for bad infinite behaviours. This chapter turns the angle: now the interesting object is the branching future of a state, and the main question is whether a property holds along every branch, along some branch, or by a fixed point of both.

This chapter opens the branching-time side of the book. We introduce *Computation Tree Logic* (CTL), the temporal logic in which properties can quantify over the future itself. A CTL formula can say that something happens on *all* continuations from a state, or on *some* continuation, and it can do so locally at every state of a Kripke structure. That branching perspective leads naturally to symbolic model checking: once a system is too large to enumerate explicitly, we reason about sets of states with Boolean formulas, BDDs, and fixed points instead of with a concrete graph node by node.

The arc of the chapter is simple. First comes CTL syntax and semantics, with an example of a microwave oven whose safety depends on what every possible future can do. Then we position CTL inside CTL* and compare it with LTL. After that, explicit and symbolic model checking are contrasted directly, and we build the symbolic picture from Boolean formulas and transition relations. The last part introduces bounded model checking, OBDDs, and the fixed-point view that makes CTL model checking both elegant and implementable.

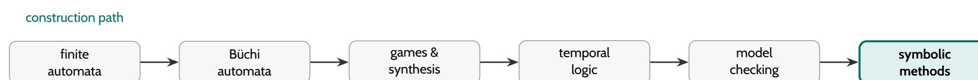


Figure 13.1: Where this chapter sits in the construction path of the book. Each step reuses the previous one as a representation or an algorithmic primitive.

13.1 Branching Time and CTL

13.1.1 Why branching time

LTL speaks about one path because that is the right abstraction for traces. CTL speaks about states because in a branching system a single state can have many possible futures. The choice matters. From one state, a controller may be able to force a good future only on some branches, while an environment may be able to spoil the property on others. CTL keeps those futures separate.

■ **Intermezzo — A state is a fork, not a line**

At a branching point, the question "what happens next?" is often not well-posed as a single answer. CTL therefore lets us say either "there exists a future where..." or "all futures satisfy..." before we look at the temporal evolution along that future.

13.1.2 Syntax

We keep the same notion of Kripke structure used in Chapter 12: a finite graph of states, total transitions, and a labelling of atomic propositions. CTL formulas are state formulas, but they are built by quantifying over paths and then applying temporal operators to the path.

Definition 13.1.1 (CTL syntax)**.** Let AP be a finite set of atomic propositions. CTL state formulas are generated by

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid EX\,\varphi \mid AX\,\varphi \mid E[\varphi_1\,U\,\varphi_2] \mid A[\varphi_1\,U\,\varphi_2]$$

where $p \in AP$.

The familiar derived operators are abbreviations:

$$EF\,\varphi \equiv E[\top U \varphi], \qquad AF\,\varphi \equiv A[\top U \varphi],$$

$$EG\,\varphi \equiv \neg AF\,\neg\varphi, \qquad AG\,\varphi \equiv \neg EF\,\neg\varphi.$$

The temporal operators are the same ones that appeared in LTL, but now they come with a path quantifier:

- $EX\,\varphi$: there exists a next state where φ holds;
- $AX\,\varphi$: in every next state, φ holds;
- $E[\varphi_1 U \varphi_2]$: there exists a path where φ_2 eventually holds and φ_1 holds until then;
- $A[\varphi_1 U \varphi_2]$: on every path, φ_2 eventually holds and φ_1 holds until then.

13.1.3 Semantics

CTL is interpreted over a Kripke structure $\mathcal{M}$ and a state s. Path quantifiers range over infinite paths starting from s.

Definition 13.1.2 (CTL semantics)**.** Let $\mathcal{M} = (Q, I, R, L)$ be a Kripke structure. We write $(\mathcal{M}, s) \models \varphi$ when the state formula φ holds in state s. The clauses for the propositional connectives are standard. The temporal clauses are:

- $(\mathcal{M}, s) \models EX\,\varphi$ iff there exists a successor s' with $(s, s') \in R$ and $(\mathcal{M}, s') \models \varphi$;
- $(\mathcal{M}, s) \models AX\,\varphi$ iff for every successor s' of s, we have $(\mathcal{M}, s') \models \varphi$;
- $(\mathcal{M}, s) \models E[\varphi_1 U \varphi_2]$ iff there exists a path $\pi = s_0 s_1 s_2 \cdots$ with $s_0 = s$ and some index j such that $(\mathcal{M}, s_j) \models \varphi_2$ and $(\mathcal{M}, s_i) \models \varphi_1$ for all $0 \le i < j$;
- $(\mathcal{M}, s) \models A[\varphi_1 U \varphi_2]$ iff for every path $\pi = s_0 s_1 s_2 \cdots$ with $s_0 = s$, the same condition holds.

The intuition. The quantifier chooses the branch, and the temporal operator inspects that chosen branch. CTL therefore evaluates a formula at a state, but it does so by ranging over the future trees rooted at that state. The order matters: first choose whether we mean “some future” or “all futures”, and only then interpret the temporal condition along the selected branch.

How the operators are read in practice. The abbreviations are meant to be spoken as ordinary engineering statements:

- $EX\,\varphi$: one step is enough to reach a state where φ holds;
- $AX\,\varphi$: every immediate successor must satisfy φ;
- $E[\varphi_1 U \varphi_2]$: there is some branch on which φ_2 eventually appears, while φ_1 keeps holding beforehand;
- $A[\varphi_1 U \varphi_2]$: on every branch, φ_2 must arrive eventually and φ_1 must hold until then;
- $EF\,\varphi$: some branch can eventually reach φ;
- $AF\,\varphi$: every branch must eventually reach φ;
- $EG\,\varphi$: some branch can stay in φ forever;
- $AG\,\varphi$: every branch must stay in φ forever.

13.1.4 The microwave oven example

The microwave controller is a good running example because it is small enough to draw and still rich enough to show the difference between “some future” and “all futures”. A few states are enough:

- `idle`: the oven is waiting for a command;
- `ready`: the door is closed and the oven can begin heating;
- `doorOpen`: the door is open, so heating must be disabled;
- `heating`: the oven is running;
- `done`: the cooking cycle has completed.

Think of the state names as atomic propositions describing the current mode of the controller. From `idle`, there are at least two immediate futures: the door can be opened, or the cycle can move towards cooking. That is exactly the situation CTL is built for, because a single state does not determine a single future.

One safety requirement is that the magnetron never heats while the door is open. In CTL this is naturally a universal branching property:

$$AG(\texttt{doorOpen} \rightarrow \neg\texttt{heating})$$

Another requirement is that from every admissible start, there is a way to finish the cycle:

$$AG(\texttt{start} \rightarrow EF\,\texttt{done})$$

Here `start` is the label for states from which the controller accepts a new cooking command. These formulas are not equivalent. The first is a safety statement about every branch; the second is a reachability statement that only asks for one successful branch.

The more interesting part is how the short operators behave on the same controller. From `idle`, the formula EX `ready` is true because one step can move into the `ready` state. By contrast, AX `ready` is false there if one of the

immediate futures is `doorOpen`, since `doorOpen` is not `ready`. The reachability formula EF `done` is true from the initial state, because there is a branch

$$\texttt{idle} \rightarrow \texttt{ready} \rightarrow \texttt{heating} \rightarrow \texttt{done}$$

that reaches completion. But AF `done` is stronger and fails if the model allows the heating state to stutter forever: one branch can keep taking the `time` transition and never finish. That same looping branch makes EG `heating` true at `heating`, because there exists a future that can remain in the heating state indefinitely.

The until operators read in the same way. $E[\neg\texttt{doorOpen}\, U\, \texttt{done}]$ says that there exists a branch along which the door stays closed until the cooking cycle completes. $A[\neg\texttt{doorOpen}\, U\, \texttt{done}]$ demands that every branch behave that well, which is too strong if the controller also admits an early door-open branch. The universal formula $AG(\texttt{doorOpen} \rightarrow \neg\texttt{heating})$ then summarises the safety invariant: no branch may ever reach a state where the door is open and the heater is on at the same time.

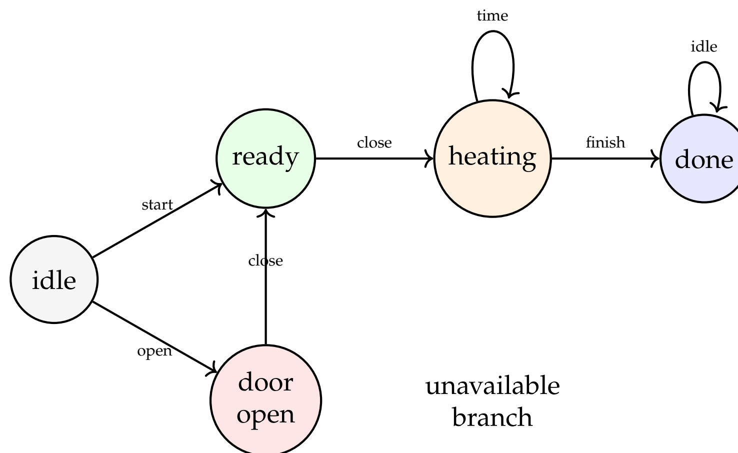


Figure 13.2: A branching view of a microwave oven. From the same state the future can split into safe, finishing, or blocked behaviours, and CTL can quantify over those branches explicitly.

13.1.5 The CTL restriction

CTL is a fragment of CTL* with a very specific shape. The restriction is what makes the logic decidable by the fixed-point algorithms we will use later.

> **Definition 13.1.3** (CTL restriction)**.** A CTL formula is a CTL* formula in which every temporal operator is immediately preceded by a path quantifier, and path quantifiers do not nest arbitrarily inside each other. Informally: temporal structure lives on the path, but each path must be selected first.

The restriction is not cosmetic. It prevents formulas from freely mixing branching and linear viewpoints inside the same temporal nest. That trade-off is exactly what we exploit later: the syntax is narrower than CTL*, but the resulting model checking problem is much easier to solve.

13.2 Explicit State CTL Model Checking

Before we jump into advanced symbolic representations (which operate on formulas and binary decision diagrams), it is critical to understand the classic, explicit-state algorithm for CTL. Imagine stepping up to a whiteboard drawing of a finite state graph and physically sticking subformula labels onto the nodes.

The explicit algorithm works *bottom-up* on the syntax tree of the CTL formula. For example, to check $E[p\ U\ q]$, we first find all nodes where p is true and stick a "p" label on them. We do the same for q. Then, we search the graph to figure out which nodes should get the "$E[p\ U\ q]$" label.

A key pedagogical insight is the direction of the search:

- **Backward search for EU/EF**: To find all states satisfying $E[p\ U\ q]$, we start at the target nodes (labeled q) and walk *backwards* along the edges. If we step backward into a node labeled p, we label it $E[p\ U\ q]$ and continue our backward search from there. This is efficient because we are growing a set of successful states from the target outwards.
- **Forward search for AU/AF**: It is intrinsically wrong to evaluate $A[p\ U\ q]$ by walking backwards! If we step backwards from q into p, we only know that *one* branch successfully reached q. But $A[p\ U\ q]$ demands that *all* branches reach q. To check this safely, we must use a *forward* recursive traversal, exploring every outgoing edge from a state to guarantee that no branch escapes the p-region without eventually hitting q.

Once the root formula is evaluated, the set of states containing the root label is exactly the set of states that satisfy the property. This physical labeling process builds the intuition for the symbolic set-manipulations (fixed points) we will see later.

13.3 CTL* and Expressiveness

CTL* sits above both CTL and LTL. It is the logic in which state formulas and path formulas coexist without the CTL restriction. A CTL* formula may say “there exists a path where eventually something happens” just as LTL can, but it can also say “on every branch, there is a way to reach a region from which every continuation satisfies a safety condition” just as CTL can.

13.3.1 Positioning CTL*

The simplest way to think about CTL* is as the common generalisation of the two logics already familiar from the book:

- LTL is linear-time and talks about one path at a time.
- CTL is branching-time and talks about a state by quantifying over paths before using temporal operators.
- CTL* allows arbitrary nesting of the two styles, so it contains both as fragments.

> *Remark* 13.3.1. The price of that extra freedom is not conceptual but algorithmic: CTL* model checking is still decidable and remains PSPACE-complete, but the formula no longer has the uniform fixed-point shape that makes CTL particularly pleasant to evaluate symbolically.

13.3.2 Why CTL and LTL are incomparable

CTL and LTL are not just different notations for the same class of properties. Each can express properties the other cannot.

Example 13.3.2 (A CTL property not expressible in LTL)**.** Consider the branching property

$$AG\,EF\,p.$$

It says that from every reachable state there exists a continuation that can eventually reach a state satisfying p. The key point is the alternation: from each state, a fresh existential choice may be made. LTL cannot express that kind of branching control because it sees only one path at a time.

Example 13.3.3 (An LTL property not expressible in CTL)**.** The eventual recurring property

$$GF\,p$$

is linear in spirit: along one execution, p must happen infinitely often. Its meaning depends on the shape of a single path, not on how many futures the current state branches into. CTL cannot isolate that purely linear behaviour without collapsing the branching structure it is meant to keep.

The practical lesson. CTL is excellent when specifications are branching and local, while LTL is excellent when specifications are trace based. In engineering work both are used, but they solve different kinds of questions.

13.3.3 The satisfiability boundary

CTL satisfiability is significantly harder than CTL model checking.

Proposition 13.3.4 (Complexity of CTL satisfiability)**.** *CTL satisfiability is EXPTIME-complete.*

The point is not only that satisfiability is hard in the worst case. The point is that model checking can exploit the finite system model, while satisfiability must search for any model at all. Once the model itself is no longer fixed, the branching possibilities explode much faster.

13.4 Explicit and Symbolic Model Checking

The first model checkers worked explicitly: a state space was built as a graph and traversed node by node. That works until the graph becomes too large to store. Symbolic model checking changes the representation, not the problem. States are no longer concrete nodes in memory; they are encoded by Boolean variables, and sets of states are manipulated as formulas.

13.4.1 The state-space explosion

Suppose a model has n Boolean state variables. In the worst case there are 2^n concrete states. Only a tiny fraction may be reachable, but explicit exploration

still has to discover them one by one. That is the state-space explosion problem in its cleanest form: the model is compact, the graph is not.

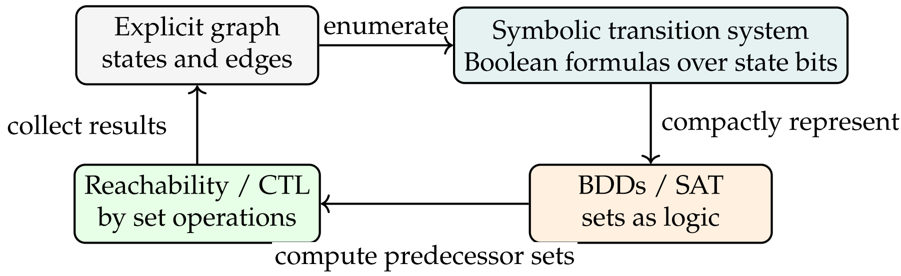


Figure 13.3: Explicit and symbolic model checking solve the same verification task, but symbolic methods replace graph storage by Boolean representation and Boolean operations.

13.4.2 Symbolic transition systems

An SMV module is best viewed as a symbolic transition system. The current state is not stored as a node in a graph; it is stored as a valuation of Boolean variables. If the model uses n Boolean variables, then one concrete state is just one assignment to

$$V = \{x_1, \ldots, x_n\}.$$

The next state uses a primed copy

$$V' = \{x'_1, \ldots, x'_n\},$$

and the transition relation is a Boolean formula over both sets of variables. That is the bridge from an explicit Kripke structure to the SMV style of representation.

> **Definition 13.4.1** (Symbolic transition system)**.** Let $V = \{x_1, \ldots, x_n\}$ be a finite set of Boolean variables and let $V' = \{x'_1, \ldots, x'_n\}$ be their next-state copies. A symbolic transition system consists of three Boolean formulas:
>
> - $Init(V)$, describing the initial states;
> - $T(V, V')$, describing the transition relation;
> - $L_p(V)$ for each atomic proposition p, describing where p holds.

To see the conversion explicitly, take a finite Kripke structure $\mathcal{M} = (Q, I, R, L)$. Choose enough Boolean variables to encode every state in Q, and fix an encoding function

$$enc : Q \longrightarrow \{0, 1\}^n.$$

Then the three symbolic formulas are obtained by collecting the encodings of the corresponding explicit sets:

$$Init(V) = \bigvee_{q \in I} enc(q), \qquad T(V, V') = \bigvee_{(q,q') \in R} enc(q) \wedge enc'(q'),$$

$$L_p(V) = \bigvee_{q \in L(p)} enc(q).$$

Here $enc'(q')$ is the same bit pattern as $enc(q')$, but written on the primed variables. In other words, an initial set becomes a disjunction of encoded states, a transition becomes a disjunction of encoded state pairs, and each atomic proposition becomes the disjunction of all states where that label is true.

Example 13.4.2 (A concrete SMV-style encoding)**.** Suppose the explicit model is a 2-bit modulo-4 counter with bits b_1, b_0. The initial state is 00, so

$$Init(b_1, b_0) = \neg b_1 \wedge \neg b_0.$$

The transition relation says that the counter increments by one modulo 4:

$$T(b_1, b_0, b_1', b_0') = (b_0' \leftrightarrow \neg b_0) \wedge (b_1' \leftrightarrow (b_1 \oplus b_0)).$$

If p means "the counter is at state 3", then

$$L_p(b_1, b_0) = b_1 \wedge b_0.$$

The explicit graph has four nodes and four edges, but the symbolic representation stores the whole model with one initial formula, one transition formula, and one label formula.

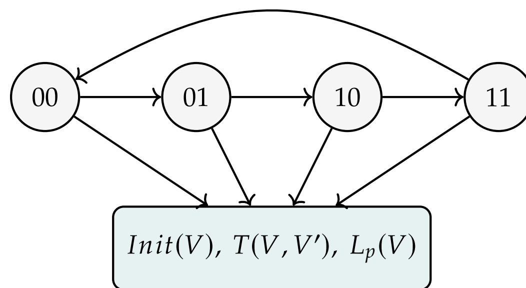


Figure 13.4: A finite explicit model is encoded by Boolean variables, then collapsed into an initial condition, a transition formula, and label formulas over those variables.

The SMV syntax from Chapter 7 is simply a structured way to write those formulas. A variable declaration becomes part of the state encoding; an assignment to `next` becomes a conjunct of T; a specification becomes a property over the reachable set.

Why this is powerful. A symbolic transition system can have an enormous implicit graph while still using only a short formula to describe it. If n Boolean variables are enough to encode the model, then the explicit state graph may have up to 2^n states and $2^n \cdot 2^n$ possible pairs of states, but the symbolic description still has size polynomial in n and in the structure of the formulas. This is the source of the exponential succinctness: the representation size grows with the number of bits, while the set of represented states grows with the number of valuations.

That is the main reason symbolic checking scales better than explicit enumeration on many real hardware models.

13.4.3 Symbolic reachability

Reachability is the core operation behind symbolic model checking. Instead of following one edge at a time, we compute sets of successors and predecessors. For a set of states Z, the existential predecessor operator is

$$Pre_E(Z) = \{\, s \mid \exists s'.\, T(s,s') \wedge s' \in Z \,\}.$$

Its universal dual is

$$Pre_A(Z) = \{\, s \mid \forall s'.\, T(s,s') \rightarrow s' \in Z \,\}.$$

With these two operators, the symbolic view of model checking becomes a game of set iteration.

Example 13.4.3 (A symbolic intuition)**.** Suppose we want to know which states can reach a safe shutdown state. In an explicit graph we would mark predecessors one edge at a time. In the symbolic view we keep a Boolean formula for the current set, compute its predecessor by one formula transformation, and repeat until the set stops growing.

13.5 Bounded Model Checking and OBDDs

13.5.1 Bounded model checking

Symbolic checking is still exhaustive. Bounded model checking is not, at least not by itself. It is a different use of the same symbolic machinery: instead of computing a whole fixed point, we ask whether there exists a short counterexample of length at most k.

Definition 13.5.1 (Bounded model checking)**.** Given a transition system $\mathcal{M}$ and a property φ, bounded model checking asks whether there exists a path of length at most k from an initial state of $\mathcal{M}$ that violates φ. The question is encoded as a propositional satisfiability problem.

LTL Bounded Semantics. When applying BMC to temporal formulas, the semantics rely on the shape of the finite prefix. There are two cases:

1. **k-loop (lasso)**: The finite path contains a back-edge to a previously visited state, proving it represents an infinite cycle.
2. **loop-free**: The finite path has no cycle; it simply ends at step k.

These shapes dictate conservative bounded semantics. Consider an LTL property on a loop-free prefix. An eventuality like $\mathbf{F}\, p$ can evaluate to True if p occurs anywhere in the finite prefix. However, a global property like $\mathbf{G}\, p$ *cannot* evaluate to True on a loop-free path—even if p is true at every state up to k, we do not know what happens at $k+1$. $\mathbf{G}\, p$ can only be proven true if the prefix forms a k-loop (proving p will hold forever). This is why BMC is naturally an asymmetric bug-finding tool: a finite loop-free prefix is often enough to decisively falsify a safety property, but not to prove it.

The unrolling is direct, but it helps to see one concrete instance all the way through. Consider a tiny two-bit transition system with state variables x_1, x_0. The initial state is

$$Init(x_1, x_0) \equiv \neg x_1 \wedge \neg x_0,$$

so the machine starts in state 00. Its next-state relation is the binary incrementer modulo four:

$$x_0' \leftrightarrow \neg x_0, \qquad x_1' \leftrightarrow (x_1 \oplus x_0).$$

Starting from 00, the unique execution is 00 → 01 → 10 → 11 → 00. Suppose that 11 is the bad state, and we want a counterexample to the safety property $AG\,\neg\mathtt{bad}$.

For a bound $k = 3$, we create four copies of the state variables:

$$(x_1^0, x_0^0),\ (x_1^1, x_0^1),\ (x_1^2, x_0^2),\ (x_1^3, x_0^3).$$

The SAT instance is then built in three steps:

1. **Initial state.** Fix the first copy to satisfy the initial condition:

$$Init(x_1^0, x_0^0).$$

2. **Transition relation.** Constrain every adjacent pair of copies to follow the transition relation:

$$\bigwedge_{i=0}^{2} \Big((x_0^{i+1} \leftrightarrow \neg x_0^i) \wedge (x_1^{i+1} \leftrightarrow (x_1^i \oplus x_0^i)) \Big).$$

3. **Bad prefix.** Ask that the bad state is reached at some point within the bound:

$$Bad_3 \equiv (x_1^0 \wedge x_0^0) \vee (x_1^1 \wedge x_0^1) \vee (x_1^2 \wedge x_0^2) \vee (x_1^3 \wedge x_0^3).$$

Putting the pieces together, the bounded counterexample formula is

$$\Phi_3 = Init(x_1^0, x_0^0) \wedge \bigwedge_{i=0}^{2} \Big((x_0^{i+1} \leftrightarrow \neg x_0^i) \wedge (x_1^{i+1} \leftrightarrow (x_1^i \oplus x_0^i)) \Big) \wedge Bad_3.$$

This formula is satisfiable, with the witness trace

$$00,\ 01,\ 10,\ 11,$$

which is exactly the counterexample we expected.

That is the general pattern. The state variables are replicated for each time frame, the initial condition anchors the first frame, the transition relation connects consecutive frames, and the negation of the property is asserted on the unrolled prefix. For safety properties this often means "some bad state appears within k steps"; for richer formulas, the bad condition is encoded by translating the temporal operators over the bounded prefix.

What BMC buys. SAT solvers are very good at proving the existence of short bad behaviours. That is why BMC is often used as a bug-finding mode: fast for shallow errors, incomplete unless a completeness threshold is reached.

13.5.2 From binary decision trees to OBDDs

The other symbolic workhorse is the ordered binary decision diagram, usually called an OBDD. It is a compressed representation of a Boolean function.

> **Definition 13.5.2** (OBDD)**.** An *ordered binary decision diagram* is a reduced binary decision graph in which every internal node is labelled by a Boolean variable, variables appear in the same fixed order along every path, and repeated subgraphs are shared.

The starting point is a binary decision tree. Every internal node asks one variable, every edge corresponds to the value 0 or 1, and every leaf is a Boolean constant. We minimize this tree into an OBDD by applying three strict reduction rules from the bottom up:

1. **Remove duplicate terminals**: If there are multiple leaf nodes representing 0, merge them into a single 0 node. Do the same for 1.
2. **Remove duplicate non-terminals**: If two internal nodes query the same variable and their 0-edges and 1-edges point to the exact same respective children, they are identical. Merge them into a single node.
3. **Remove redundant tests**: If an internal node has both its 0-edge and 1-edge pointing to the same child, the test is useless (the outcome is identical regardless of the variable's value). Remove the node and bypass it, routing incoming edges directly to the child.

At the algebraic level this is just Shannon expansion:

$$f = (\neg x \wedge f|_{x=0}) \vee (x \wedge f|_{x=1}).$$

An OBDD is the DAG form of that expansion, with one fixed variable order shared by every path.

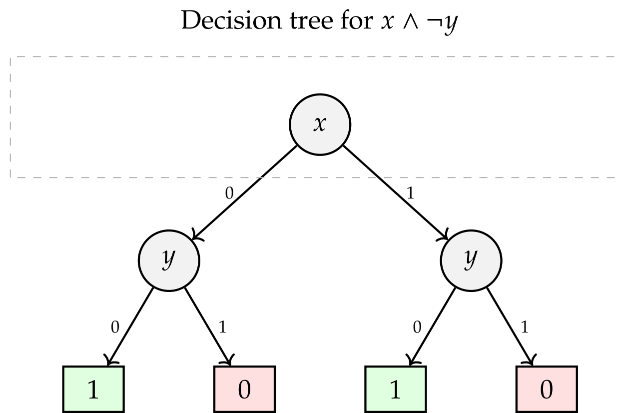


Figure 13.5: A small decision tree. The OBDD is obtained by fixing an order, removing duplicated tests, and merging equal subtrees. For this example, the reduced form is already close to the tree.

> **Example 13.5.3** (Ordering-sensitive equality: The n-bit Comparator)**.** Consider an n-bit equality comparator:
>
> $$eq(x_1 \dots x_n, y_1 \dots y_n) \equiv \bigwedge_{i=1}^{n} (x_i \leftrightarrow y_i)$$
>
> - **Interleaved Order ($x_1, y_1, x_2, y_2, \dots$)**: The diagram checks bit 1 of X against bit 1 of Y. If they mismatch, it routes immediately to the 0

terminal. If they match, it proceeds to check bit 2. Because it only ever needs to remember "are we still matching?", the resulting OBDD size is perfectly linear: exactly $3n + 2$ nodes.

- **Separated Order ($x_1, x_2, \ldots, x_n, y_1, y_2, \ldots$):** The diagram must read all n bits of X before seeing a single bit of Y. To remember the exact bit-pattern of X so it can compare it against Y later, the decision tree must branch into 2^n distinct intermediate nodes. The OBDD size explodes exponentially.

This dramatic difference gives rise to a crucial practical heuristic for choosing variable orderings in symbolic model checkers: **"Variables that are close together in the formula (strongly related) should be kept close together in the OBDD ordering."**

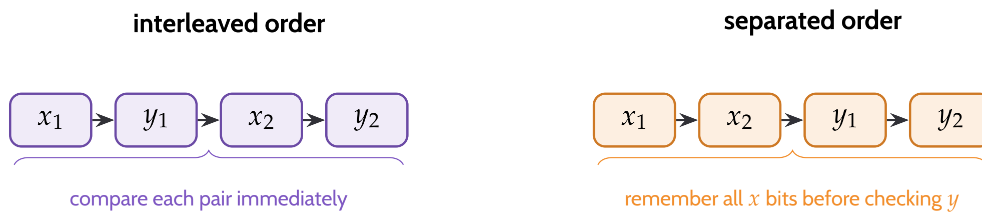


Figure 13.6: Variable order can dominate OBDD size. Interleaving related bits exposes sharing after each comparison; separating all x bits from all y bits forces the diagram to remember many partial patterns.

13.5.3 The restrict function

The recursive algorithms for OBDDs are built around variable restriction. For a Boolean function f and a variable x,

$$f|_{x=0} \quad \text{and} \quad f|_{x=1}$$

are the cofactors of f obtained by fixing x.

> **Definition 13.5.4** (Restrict)**.** The restrict operation returns the function obtained by assigning a Boolean value to one variable and simplifying the result.

The meaning is easiest to see on a formula such as

$$f(x, y, z) = (x \wedge y) \vee z.$$

Then

$$f|_{x=0} = z \qquad \text{and} \qquad f|_{x=1} = y \vee z.$$

The point of restriction is that it turns one Boolean function into the two smaller functions that appear in its Shannon expansion. With that operation, an OBDD can be built recursively: pick the first variable in the global order, compute the two cofactors, build the two sub-OBDDs, and merge them if they are identical. The same idea underlies the standard Boolean operations on OBDDs: to compute $f \circ g$, recursively combine the cofactors of the two inputs and memoise the intermediate results.

Negation is the easiest case. If an OBDD already represents f, the graph for $\neg f$ has the same internal tests and the same sharing pattern; only the terminal

values change. Intuitively, every path that used to end in 0 now ends in 1, and vice versa. In a reduced OBDD this is why complementation is so cheap: the variable order and the structure do not need to be rebuilt from scratch.

13.5.4 Why ordering matters

Example 13.5.5 (Ordering sensitivity)**.** Two equivalent Boolean functions may have OBDDs of wildly different sizes under different variable orders. The reason is structural: an OBDD is not a free graph but a graph constrained by the order. A good order exposes shared subfunctions; a bad order hides them.

This sensitivity is the main practical caveat of OBDDs. In return, when the order is good, Boolean operations become efficient and the representation can be dramatically smaller than the full truth table. That is why BDD packages remain central to symbolic verification tools.

13.6 CTL by Predicate Transformers and Fixed Points

The deepest algorithmic view of CTL is not graph-theoretic but algebraic. Instead of asking "which path satisfies this formula?" we ask "which set of states satisfies it?" That set is obtained by applying monotone predicate transformers until a fixed point is reached.

This viewpoint is useful because it matches the shape of the properties themselves. Reachability grows outward from a base case: first we know where the goal already holds, then we add states that can step into that set. That is exactly the behaviour of a least fixed point. Invariance works in the opposite direction: we begin with all states and remove the ones that can escape the safe region, so the approximation shrinks toward a greatest fixed point.

13.6.1 Predecessor transformers

The two basic symbolic transformers were already introduced in the reachability section:

$$Pre_E(Z) = \{ s \mid \exists s'.\, T(s,s') \wedge s' \in Z \}$$

$$Pre_A(Z) = \{ s \mid \forall s'.\, T(s,s') \rightarrow s' \in Z \}$$

They are the existential and universal predecessor operators. The first asks whether there is at least one successor in Z; the second asks whether all successors stay in Z.

Note 13.6.1. In symbolic implementations these operators are formula transformations. With state formulas encoded as Boolean formulas, Pre_E is typically computed by existentially quantifying next-state variables, while Pre_A is derived by duality.

13.6.2 Fixed-point characterisations

CTL model checking becomes a matter of computing least and greatest fixed points over sets of states.

Proposition 13.6.2 (CTL as fixed points)**.** *Let* $[\![\varphi]\!]$ *denote the set of states satisfying the CTL formula* φ*. Then:*

$$[\![EF\,\varphi]\!] = \mu Z.\ [\![\varphi]\!] \cup Pre_E(Z)$$

$$[\![EG\,\varphi]\!] = \nu Z.\ [\![\varphi]\!] \cap Pre_E(Z)$$
$$[\![AF\,\varphi]\!] = \mu Z.\ [\![\varphi]\!] \cup Pre_A(Z)$$
$$[\![AG\,\varphi]\!] = \nu Z.\ [\![\varphi]\!] \cap Pre_A(Z)$$
$$[\![E[\varphi_1 U \varphi_2]]\!] = \mu Z.\ [\![\varphi_2]\!] \cup \big([\![\varphi_1]\!] \cap Pre_E(Z)\big)$$
$$[\![A[\varphi_1 U \varphi_2]]\!] = \mu Z.\ [\![\varphi_2]\!] \cup \big([\![\varphi_1]\!] \cap Pre_A(Z)\big)$$

The meaning of the notation is straightforward, but the algorithmic intuition is worth spelling out. A least fixed point $\mu Z.\,F(Z)$ is the smallest set that is closed under the rule encoded by F. We discover it by starting from the empty set and repeatedly adding states that are justified by the rule. That is why least fixed points describe reachability: a state belongs to the result only if there is a finite chain of justifications that leads back to the base case.

A greatest fixed point $\nu Z.\,F(Z)$ is the largest set that survives the rule. We discover it by starting from the whole state space and repeatedly removing states that cannot justify themselves. That is why greatest fixed points describe invariance or “always” properties: a state belongs to the result only if every step can stay inside the candidate set forever.

13.6.3 The iteration view

These equations turn into simple loops. The loop variable is not just a technical artefact: it is the current approximation to the answer.

- For a least fixed point, initialise $Z_0 = \emptyset$ and repeatedly set $Z_{i+1} = F(Z_i)$. At each pass, Z_i contains the states whose correctness has already been justified by a path of length at most i.
- For a greatest fixed point, initialise Z_0 to all states and repeatedly set $Z_{i+1} = F(Z_i)$. At each pass, Z_i contains the states that have not yet been disproved as safe or persistent.

■ **Formal details — Why the iteration is correct**

The critical fact is monotonicity: if $Z \subseteq Z'$, then $F(Z) \subseteq F(Z')$. That single property gives the loop its invariant.

For a least fixed point, every iterate is an under-approximation: $Z_0 \subseteq Z_1 \subseteq \cdots$, and each new set only adds states whose membership is already supported by the previous approximation. Nothing enters the set “too early”, because a state can only be added when one of its predecessors is already known to be good, or the base case already holds.

For a greatest fixed point, every iterate is an over-approximation: $Z_0 \supseteq Z_1 \supseteq \cdots$, and each new set only removes states that fail the preservation check. Nothing is removed “too soon”, because a state stays in the candidate set exactly when all of its successors still lie inside the current approximation.

Since the state space is finite, the chain cannot keep changing forever. The least-fixed-point loop eventually reaches the first set that is closed under the rule, and the greatest-fixed-point

loop eventually reaches the largest set that survives it. At that moment the current approximation is a fixed point, and the loop invariant tells us it is the least or greatest one we wanted.

How this reads operationally. A CTL checker does not "search the tree" in the naive sense. It repeatedly computes preimages of sets until the sets stop changing. The current set is the worklist of states already proved good, and the predecessor transformer is the step that tries to extend that proof by one transition. That is the algebraic core of the standard CTL algorithm.

13.6.4 A worked reading of the formulas

The fixed-point view explains the usual CTL operators very cleanly:

- $EF\,\varphi$ is the set of states from which φ is reachable along some path.
- $AF\,\varphi$ is the set of states from which every path is eventually forced into φ.
- $EG\,\varphi$ is the set of states that can stay inside φ forever by following some branch.
- $AG\,\varphi$ is the set of states from which no branch can escape φ.

13.6.5 Worked derivation: AF as a least fixed point

The operator $AF\,\varphi$ is the cleanest place to see the algorithm at work. Intuitively, a state satisfies $AF\,\varphi$ when every path from that state is forced to encounter a φ-state eventually. The fixed-point equation

$$[\![AF\,\varphi]\!] = \mu Z.\ [\![\varphi]\!] \cup Pre_A(Z)$$

says exactly that a state is good for one of two reasons: either φ is already true there, or every successor is already known to be good.

Write $F(Z) = [\![\varphi]\!] \cup Pre_A(Z)$. Then the iteration is:

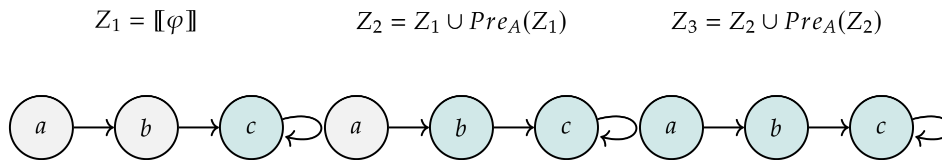
$$Z_0 = \emptyset, \qquad Z_{i+1} = F(Z_i).$$

Now read the first few rounds as a proof search.

- $Z_1 = [\![\varphi]\!]$. In one step we discover the states where the goal is already true.
- Z_2 adds every state whose successors are all in Z_1. These are states from which φ is guaranteed within one transition.
- Z_3 adds states whose successors are all in Z_2. These are the states that are guaranteed to reach φ within two transitions.
- And so on. At stage i, Z_i contains precisely the states from which φ is forced within at most $i-1$ steps.

The limit of that chain is the set of states from which no path can avoid eventually hitting φ. If a state were missing from the limit, then there would be a branch that keeps escaping the candidate set forever, and the universal predecessor test would never be able to certify it. If a state is in the limit, then some finite round has already established the bound on when φ must appear.

The same style of argument works for $EG\,\varphi$, but with the quantifiers reversed. There the loop starts from all states, keeps only those that have a successor still inside the candidate set, and converges to the largest region from which one can stay in φ forever. So the fixed point is not just an abstract denotation: it is the exact form of the algorithm.



For $AF\ \varphi$, teal grows backward only through states whose successors are already teal.

Figure 13.7: Least-fixed-point growth for $AF\ \varphi$. The first approximation marks states where φ already holds. Each later round adds states whose every successor is already marked, until no new state can be added.

13.7 Summary

Summary & Key Takeaways

- CTL is a branching-time logic: each temporal operator is preceded by either an existential or a universal path quantifier.
- CTL* is the larger logic that contains both CTL and LTL. CTL and LTL are incomparable in expressive power.
- CTL satisfiability is EXPTIME-complete, while model checking becomes efficient because the model is fixed and finite.
- Symbolic model checking replaces explicit graph traversal by Boolean formulas over state variables, transition relations, and predecessor set computations.
- Bounded model checking searches for short counterexamples by turning a bounded unrolling into a SAT instance.
- OBDDs compactly represent Boolean functions once a variable order is fixed, but the size can depend strongly on that order.
- CTL model checking can be written as a collection of least and greatest fixed points over predecessor transformers.

Exercises

Exercise 1 (One-step CTL operators). In the microwave model of fig. 13.2, decide whether each formula holds at `idle`: EX `ready`, AX `ready`, EF `done`, AF `done`. For each answer, name the branch or successor that justifies it.

Exercise 2 (Labelling $E(p\ U\ q)$). Consider a graph with edges $a \to b$, $a \to c$, $b \to d$, $c \to c$, and $d \to d$. Let q hold only at d, and let p hold at a and b. Compute the set of states satisfying $E[p\ U\ q]$ by the backward labelling algorithm.

Exercise 3 (Why universal until is stricter). On the same graph, compute the set of states satisfying $A[p\ U\ q]$. Explain why state a should be treated differently for the existential and universal formulas.

Exercise 4 (Symbolic encoding). Encode the four-state counter $00 \to 01 \to 10 \to 11 \to 00$ with Boolean variables b_1, b_0. Write formulas for $Init(V)$, $T(V, V')$, and the proposition $even$ that holds at states 00 and 10.

Exercise 5 (Bounded counterexample). Using the counter from the previous exercise, write the bounded model checking formula for a bad state 11 with bound $k = 2$. Is it satisfiable? Repeat the answer for $k = 3$ without rewriting the whole formula.

Exercise 6 (OBDD ordering). For the equality function $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$, compare the variable orders x_1, y_1, x_2, y_2 and x_1, x_2, y_1, y_2. Which order should share more substructure, and why?

Exercise 7 (Fixed-point iteration). For the graph $a \to b$, $b \to c$, $c \to c$, with φ true only at c, compute the approximants for $AF\, \varphi = \mu Z.\, [\![\varphi]\!] \cup Pre_A(Z)$. List Z_0, Z_1, Z_2, Z_3 and identify the fixed point.

CHAPTER 14

Learning Regular Languages

Up to this point, our verification algorithms assumed that the model of the system was given. We had a Kripke structure, an SMV file, or a set of multiset rewriting rules, and our task was to verify that it met a temporal logic specification or trace property. In practice, however, a complete model of a complex software system is rarely available. Often, we are confronted with a legacy implementation, a black-box hardware controller, or a third-party protocol whose internal state transitions are hidden.

Chapter 13 ended our exploration of verification over known state sets. This chapter turns the question around: what if the state machine is a black box, and we must construct its model from external observations? We show how automata theory and machine learning meet via active learning.

This chapter introduces the technique of *learning regular languages* (or automata learning). We bridge the gap between machine learning and formal language theory. Instead of learning continuous functions over fixed-dimensional real vectors (as in modern neural networks), our goal is to reconstruct the discrete structure of a finite-state machine by observing its input-output behavior. While this chapter focuses on deterministic finite automata, these active learning techniques have been successfully extended to probabilistic automata (e.g., for user profiling in systems like Amazon), tree automata for XML schema extraction, and tree transformations for modeling Linux package managers.

We follow the progression of the lecture block. We begin with the mathematical foundations of the Myhill–Nerode theorem, which guarantees that a regular language has a unique minimal representation. Then we transition to the active learning paradigm, explaining how a learner can reconstruct this minimal representation by asking targeted questions to a teacher. We describe Angluin's L^* active-learning algorithm in detail, complete with a step-by-step trace on a concrete language and a formal proof of its correctness and complexity.

Where we are. The model-checking chapters assumed that the model was available: as an automaton, a Kripke structure, an SMV program, or a symbolic transition relation.

What this chapter adds. Automata learning asks how to recover a finite-state model from observations. Myhill–Nerode provides the semantic target, and active learning provides the algorithmic protocol for finding it.

Where it leads. This changes verification from analysing a known model to constructing one. That perspective is useful whenever the system is legacy, black-box, or too poorly documented to model by hand.

Chapter map.

- Section 14.1 recalls the equivalence classes that define the minimal DFA.
- Section 14.2 introduces membership and equivalence queries.
- Section 14.3 gives Angluin's L^* algorithm.

- Section 14.4 traces the algorithm on a concrete language.
- Section 14.5 proves correctness, termination, and complexity bounds.

14.1 The Myhill–Nerode Foundation

The mathematical bridge that enables regular language learning is the Myhill–Nerode theorem. It provides a purely language-theoretic characterization of regular languages, independent of any automaton structure. This characterization is essential because a learner does not initially have access to states or transitions; it only observes whether specific strings belong to the target language.

14.1.1 The equivalence relation

Given a language $L \subseteq \Sigma^*$ over a finite alphabet Σ, we want to determine when two prefix strings x and y are "equivalent" from the perspective of L. Two prefixes are equivalent if they cannot be distinguished by any suffix extension.

Definition 14.1.1 (Myhill–Nerode equivalence). Let $L \subseteq \Sigma^*$ be a language. The Myhill–Nerode equivalence relation $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ is defined as:

$$x \equiv_L y \iff \forall z \in \Sigma^*, (xz \in L \iff yz \in L)$$

If $x \equiv_L y$, we say that x and y behave identically with respect to completions in L. We denote the equivalence class of a string x by $[x]_L = \{y \in \Sigma^* \mid x \equiv_L y\}$.

To simplify notation, we write $xz =_L yz$ to denote that xz and yz have the same membership status in L (both inside or both outside). The relation $\equiv_L$ is an equivalence relation: it is reflexive, symmetric, and transitive.

■ Intermezzo — The Puzzle Piece Analogy

Imagine the universe of all strings Σ^* painted on a canvas. The simplest classification is the trivial two-class equivalence: a blue region (strings inside L) and a gray region (strings outside L). The Myhill–Nerode equivalence classes act as *puzzle pieces* that tile this canvas.

Two critical properties govern these puzzle pieces:

1. **Saturation**: A single puzzle piece cannot cross the blue/gray boundary. Every string in a piece must share the same membership status.
2. **Right-invariance**: If you take all the strings in one puzzle piece and append the same letter a, they must all land squarely inside a *single target puzzle piece*. You cannot split a puzzle piece by taking a step.

Crucially, $\equiv_L$ satisfies the property of being a *right congruence* (or being right invariant), perfectly matching our puzzle piece intuition.

Lemma 14.1.2 (Right invariance). *The equivalence relation $\equiv_L$ is right invariant. That is, for all $x, y \in \Sigma^*$ and all $a \in \Sigma$:*

$$x \equiv_L y \implies xa \equiv_L ya$$

Proof. Assume $x \equiv_L y$. By Definition 14.1.1, this means that for all $z' \in \Sigma^*$, we have $xz' \in L \iff yz' \in L$. To show $xa \equiv_L ya$, we must show that for any suffix $z \in \Sigma^*$, $xaz \in L \iff yaz \in L$. This follows immediately by choosing $z' = az$ in the assumption. □

14.1.2 Characterizing regular languages

The Myhill–Nerode theorem states that the regularity of a language is completely determined by the number of equivalence classes (the index) of the relation $\equiv_L$.

Theorem 14.1.3 (Myhill–Nerode)**.** *A language $L \subseteq \Sigma^*$ is regular if and only if the equivalence relation $\equiv_L$ has a finite index (i.e., the quotient set $\Sigma^*/\equiv_L$ is finite).*

Furthermore, if the index of $\equiv_L$ is finite and equal to n, then n is exactly the number of states in the unique minimal deterministic finite automaton (DFA) recognizing L.

■ **Formal details — Minimal DFA construction**

If $\equiv_L$ has a finite index, we can construct the canonical minimal DFA $A = (Q, \Sigma, \delta, q_0, F)$ recognizing L directly from the equivalence classes:

- **States**: $Q = \Sigma^*/\equiv_L = \{[w]_L \mid w \in \Sigma^*\}$. Since the index is finite, Q is finite.
- **Initial State**: $q_0 = [\epsilon]_L$.
- **Transition Function**: $\delta([w]_L, a) = [wa]_L$ for all $w \in \Sigma^*$ and $a \in \Sigma$. (This is well-defined because if $[u]_L = [v]_L$, then $u \equiv_L v$, which implies $ua \equiv_L va$ by right invariance, meaning $[ua]_L = [va]_L$).
- **Final States**: $F = \{[w]_L \mid w \in L\}$. (This is well-defined because if $u \equiv_L v$ and $u \in L$, then by choosing the empty suffix $z = \epsilon$, we have $u\epsilon \in L \iff v\epsilon \in L$, hence $v \in L$).

14.1.3 Examples of equivalence classes

To clarify the intuition behind the Myhill–Nerode relation, we look at a non-regular language and a regular language.

Example 14.1.4 (A non-regular language: Palindromes)**.** Let $\Sigma = \{a, b\}$ and let $L_{\text{pal}} = \{w \in \Sigma^* \mid w = w^R\}$ be the language of palindromes (strings that read the same backward and forward). We show that $\equiv_{L_{\text{pal}}}$ has an infinite index.

Consider two distinct strings $u, v \in \Sigma^*$ of the same length, say $u = ab$ and $v = bb$. To prove they belong to different equivalence classes, we must find a suffix z that distinguishes them. Choose $z = a$ (the reverse of u).

- $uz = aba \in L_{\text{pal}}$ (a palindrome).
- $vz = bba \notin L_{\text{pal}}$ (not a palindrome).

Since $uz \in L_{\text{pal}}$ but $vz \notin L_{\text{pal}}$, we have $u \not\equiv_{L_{\text{pal}}} v$.

In general, for any two distinct strings $u \neq v$, if we choose the suffix $z = u^R$ (the reverse of u), then $uz = uu^R$ is always a palindrome. However, if u and v have different structures, vu^R is not guaranteed to be a palindrome. In fact, each string $w \in \Sigma^*$ belongs to its own distinct equivalence class. Since

Σ^* is infinite, $\equiv_{L_{\text{pal}}}$ has an infinite index, which proves that the language of palindromes is not regular.

Example 14.1.5 (A regular language: $L = ab^*$)**.** Let $\Sigma = \{a, b\}$ and $L = ab^* = \{a, ab, abb, abbb, \ldots\}$. We determine the equivalence classes of $\equiv_L$:

1. **Class 1:** $[\epsilon]_L$. Contains the empty string ϵ. If we extend ϵ with a suffix z, the resulting string z is in L iff $z \in ab^*$.
2. **Class 2:** $[a]_L$. Contains strings that have already matched the prefix a and can be completed by any number of b's. This class contains $\{a, ab, abb, \ldots\}$, which is exactly the language L itself. For any $w \in [a]_L$, extending it with b keeps it in the class: $wb \in L$, whereas extending it with a goes outside the language: $wa \notin L$.
3. **Class 3:** $[b]_L$. Contains strings that start with the incorrect symbol b, making it impossible to ever complete them to form a string in L. This is a "sink" class containing $\{b, ba, bb, aa, aab, \ldots\}$. For any string w in this class and any suffix z, $wz \notin L$.

Thus, there are exactly three equivalence classes: $[\epsilon]_L$, $[a]_L$, and $[b]_L$. Since the index is 3, the minimal DFA recognizing ab^* has exactly 3 states.

14.2 Passive vs. Active Learning

In machine learning, we distinguish between different paradigms depending on how the learner interacts with the data source. The two primary paradigms are *passive learning* and *active learning*.

14.2.1 Passive learning and its limitations

In the classical passive learning model (such as Gold's identification in the limit), the learner is presented with a fixed set of training samples:

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$$

where each $x_i \in \Sigma^*$ is a string and $y_i \in \{0, 1\}$ indicates whether $x_i \in L$. The learner's task is to find a DFA A that is consistent with the dataset D (i.e., $L(A)$ accepts x_i iff $y_i = 1$).

Passive learning of regular languages has severe computational limitations:

- **NP-completeness**: Finding the minimal DFA that is consistent with a given set of positive and negative examples is NP-complete (Gold, 1978).
- **Hardness of approximation**: It is computationally hard to approximate the minimal DFA within any polynomial factor unless P = NP.

This hardness arises because the learner is a passive recipient of the data. It cannot choose which parts of the state space to explore, leading to an exponential search space to resolve ambiguities.

14.2.2 Active learning: The MAT framework

To bypass these negative results, Dana Angluin (1987) proposed the *active learning* paradigm. In this model, the information flow is two-way: the learner

is not passive but can actively interact with an oracle called the **Minimally Adequate Teacher** (MAT).

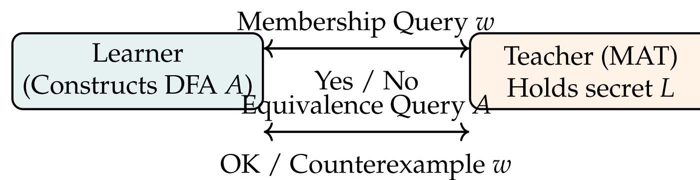


Figure 14.1: The Minimally Adequate Teacher (MAT) framework. The learner interacts with the teacher using two types of queries to iteratively refine its candidate DFA.

The active learning setup is often framed as a game between a **Learner** and a **Teacher**. The teacher holds a secret target regular language L (represented internally as a DFA, regular expression, or black-box program). The learner only knows the alphabet Σ and must reconstruct the minimal DFA for L. To do this, the learner can ask two types of queries.

Could the learner win the game using only one type of query?

- **Only Membership Queries?** No. If the learner only asks "is string w in L?", it can only gather a finite set of labeled examples. There are always infinitely many valid DFAs that fit any finite set of points. The learner could never be mathematically certain it found the right one.
- **Only Equivalence Queries?** Yes! Since the set of all DFAs is countable, the learner could simply enumerate them ($A_1, A_2, A_3 \dots$) and ask "Is it A_1?", "Is it A_2?". The teacher will eventually say yes. However, this brute-force approach takes *exponential time*.

By combining both queries, the MAT framework allows the learner to win the game in polynomial time.

Definition 14.2.1 (Membership Query)**.** In a **Membership Query (MQ)**, the learner presents a string $w \in \Sigma^*$ to the teacher. The teacher answers:

$$\mathrm{MQ}(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{if } w \notin L \end{cases}$$

Definition 14.2.2 (Equivalence Query)**.** In an **Equivalence Query (EQ)**, the learner presents a candidate DFA A to the teacher. The teacher checks whether $L(A) = L$.

- If $L(A) = L$, the teacher replies **Yes** (or OK), and the learning process terminates successfully.
- If $L(A) \neq L$, the teacher replies **No** and provides a **counterexample** $w \in \Sigma^*$. The counterexample is in the symmetric difference of the two languages:

$$w \in (L(A) \setminus L) \cup (L \setminus L(A))$$

■ **Intermezzo — The shortest counterexample requirement**

To prevent an adversarial teacher from stalling the algorithm by returning arbitrarily long counterexamples, we assume the teacher is honest and returns the *shortest* possible counterexample. In practice, the shortest counterexample is polynomial in the size of the minimal DFA.

By combining MQs and EQs, active learning makes it possible to reconstruct the minimal DFA in polynomial time. The learner uses MQs to explore local state transitions and EQs to detect global discrepancies between the candidate model and the true language.

Approximating EQs in Practice. The MAT framework is a theoretical ideal. In real-world system identification (often called "model learning" or "black-box checking"), the teacher is not a mathematical oracle but a physical protocol or software system. While MQs are easy to execute (just send a test sequence to the system and observe if it accepts or crashes), EQs are impossible to answer perfectly. In practice, EQs are approximated via **fuzzing**: the learner launches large batches of random test sequences against both the physical system and the candidate DFA. If the outputs match for thousands of random tests, the learner assumes the candidate DFA is equivalent to the system. If a discrepancy is found, that specific test sequence serves as the counterexample.

14.3 Angluin's L^* Algorithm

Angluin's L^* algorithm organizes the learner's knowledge in a structured tabular format called the **observation table**. The algorithm maintains and updates two finite sets of strings over Σ:

- $S \subset \Sigma^*$: The set of **state representatives**. These represent prefixes that lead to distinct states in the target DFA.
- $T \subset \Sigma^*$: The **test set** of suffixes. These are used to distinguish different state representatives.

Initially, $S = \{\epsilon\}$ and $T = \{\epsilon\}$.

14.3.1 The observation table structure

The observation table is a matrix where:

- The rows are indexed by elements of the set $S \cup (S \cdot \Sigma)$, where $S \cdot \Sigma = \{s \cdot a \mid s \in S, a \in \Sigma\}$ represents the one-step extensions of our state representatives.
- The columns are indexed by elements of the test set T.
- The cell at row $s \in S \cup (S \cdot \Sigma)$ and column $t \in T$ is filled by querying the teacher via a Membership Query:

$$\text{Table}(s, t) = \text{MQ}(s \cdot t) \in \{0, 1\}$$

Before defining the row profile, we must formally introduce the T**-restricted equivalence** ($\equiv_{L,T}$). This relation sits strictly between the trivial 2-class equivalence (equal-L) and the true Myhill-Nerode equivalence ($\equiv_L$). Two strings are T-equivalent if they cannot be distinguished by any suffix from the set T:

In practice, the observation table also serves as a memoization cache: if a string's membership has already been queried, its result is stored, preventing redundant queries to the teacher.

$$x \equiv_{L,T} y \iff \forall t \in T, (xt \in L \iff yt \in L)$$

Intermezzo — The Test Set as a Knob

Think of the test set T as a *tuning knob*. When T is small (e.g., just $\{\epsilon\}$), $\equiv_{L,T}$ is a very coarse approximation, yielding large, chunky puzzle pieces. As the learner discovers counterexamples, it adds strings to T, effectively "turning the knob." This refines the equivalence, shattering the large puzzle pieces into smaller, more precise ones, until the approximation perfectly aligns with the true Myhill-Nerode equivalence classes.

For any string $s \in S \cup (S \cdot \Sigma)$, we define its **row profile** $\text{row}(s)$ as the finite vector of boolean values indexed by T:

$$\text{row}(s) = \langle \text{Table}(s, t_1), \text{Table}(s, t_2), \ldots, \text{Table}(s, t_{|T|}) \rangle$$

The row profile is simply the characteristic function (a vector of 0s and 1s) representing the $\equiv_{L,T}$ equivalence class of s. If two strings have the same row profile, they currently reside in the same approximated puzzle piece.

14.3.2 Table properties: Minimality and Completeness

The algorithm relies on two structural properties of the observation table to decide whether a candidate DFA can be constructed:

Definition 14.3.1 (T-minimality)**.** An observation table is T**-minimal** (or consistent) if no two distinct state representatives in S have the same row profile:

$$\forall s_1, s_2 \in S, (\text{row}(s_1) = \text{row}(s_2) \implies s_1 = s_2)$$

In puzzle piece terms, minimality means we have selected *at most one representative string per puzzle piece*. There are no redundant states.

Definition 14.3.2 (T-completeness)**.** An observation table is T**-complete** if for every one-step extension $s \cdot a \in S \cdot \Sigma$ (where $s \in S$ and $a \in \Sigma$), there exists a state representative $s' \in S$ with the same row profile:

$$\forall s \in S, \forall a \in \Sigma, \exists s' \in S, (\text{row}(s \cdot a) = \text{row}(s'))$$

In puzzle piece terms, completeness means that if any representative takes a step by one letter and lands in a new puzzle piece, *that target puzzle piece must already have a designated representative in* S. If it doesn't, the table is incomplete, and we must promote the stray string to be the new representative for that piece.

14.3.3 DFA Construction

When the observation table is both T-minimal and T-complete, the learner has enough information to construct a candidate DFA $A = (Q, \Sigma, \delta, q_0, F)$:

- **States**: $Q = S$ (the state representatives themselves).
- **Initial State**: $q_0 = \epsilon$ (which is always in S).
- **Transition Function**: For any $s \in S$ and $a \in \Sigma$, we define $\delta(s, a) = s'$, where $s' \in S$ is the unique state representative such that:

$$\text{row}(s \cdot a) = \text{row}(s')$$

(T-completeness guarantees that such an s' exists, and T-minimality guarantees that it is unique).

- **Final States**: $F = \{s \in S \mid \text{Table}(s, \epsilon) = 1\}$. (Since $\epsilon \in T$, we know whether $s \cdot \epsilon = s \in L$ directly from the table's first column).

14.3.4 The L^* loop

The algorithm runs in a loop by executing three phases:

1. **Phase 1: Make Complete**. If the table is not complete, find some $s \in S$ and $a \in \Sigma$ such that $\text{row}(s \cdot a) \neq \text{row}(s')$ for all $s' \in S$. Add $s \cdot a$ to S. Fill the new table rows using MQs. Repeat until the table is complete. (Note: adding $s \cdot a$ to S preserves T-minimality because $s \cdot a$ has a row profile that was not present in S).
2. **Phase 2: Propose Candidate**. Once the table is complete, construct the candidate DFA A and submit it in an Equivalence Query (EQ) to the teacher.
3. **Phase 3: Refine via Suffixes**. If the teacher returns a counterexample w, the learner refines its approximation by adding w and all its suffixes to the test set T. Adding columns to the table splits equivalence classes, making the table incomplete again. The learner loops back to Phase 1.

Algorithm 1: Angluin's L^* active-learning algorithm

```
1  S ← {ε};   T ← {ε}
2  Initialize observation table using MQs for S ∪ (S · Σ) × T
3  while true do
4  |  while table is not complete do
5  |  |  Find s ∈ S, a ∈ Σ such that ∀s' ∈ S, row(s · a) ≠ row(s')
6  |  |  S ← S ∪ {s · a}
7  |  |  Update observation table using MQs for the new rows
8  |  end
9  |  Construct candidate DFA A from S and T
10 |  if EQ(A) returns OK then
11 |  |  return A (Learning completed successfully)
12 |  else
13 |  |  Receive counterexample w from the teacher
14 |  |  T ← T ∪ suffixes(w)
15 |  |  Update observation table using MQs for the new columns
16 |  end
17 end
```

14.4 Walkthrough with a Concrete Example

We trace the execution of the L^* algorithm on a concrete language to show how the observation table evolves, how completeness is checked, and how counterexamples refine the test set.

Let the alphabet be $\Sigma = \{a, b\}$ and the target language be:

$$L = (ab)^* = \{\epsilon, ab, abab, ababab, \dots\}$$

We start with $S = \{\epsilon\}$ and $T = \{\epsilon\}$.

14.4.1 Iteration 1

We initialize the table with $S = \{\epsilon\}$ and $S \cdot \Sigma = \{a, b\}$. We query the teacher with Membership Queries for the strings $\epsilon \cdot \epsilon = \epsilon$, $a \cdot \epsilon = a$, and $b \cdot \epsilon = b$.

$S \cup (S \cdot \Sigma)$		$T = \{\epsilon\}$
ϵ	(S)	1
a		0
b		0

Table 14.1: Observation table in Iteration 1.

We check T-completeness:

- The row profile of the representative is $\text{row}(\epsilon) = \langle 1 \rangle$.
- The row profiles of the extensions are $\text{row}(a) = \langle 0 \rangle$ and $\text{row}(b) = \langle 0 \rangle$.
- Neither $\text{row}(a)$ nor $\text{row}(b)$ matches any row profile in S. The table is **not complete**.

To resolve the incompleteness, we select the prefix a and move it to the set of representatives. Our sets become:

$$S = \{\epsilon, a\}, \quad S \cdot \Sigma = \{b, aa, ab\}$$

14.4.2 Iteration 2

We update the table and fill the new rows using MQs. Note that $aa \notin L$ and $ab \in L$.

$S \cup (S \cdot \Sigma)$		$T = \{\epsilon\}$
ϵ	(S)	1
a	(S)	0
b		0
aa		0
ab		1

Table 14.2: Observation table in Iteration 2.

We check T-completeness:

- The representative profiles are $\text{row}(\epsilon) = \langle 1 \rangle$ and $\text{row}(a) = \langle 0 \rangle$.
- The extension profiles are:
  - $\text{row}(b) = \langle 0 \rangle = \text{row}(a)$ (matched)
  - $\text{row}(aa) = \langle 0 \rangle = \text{row}(a)$ (matched)
  - $\text{row}(ab) = \langle 1 \rangle = \text{row}(\epsilon)$ (matched)

  All extensions have a matching row profile in S. The table is **complete**.

Since the table is complete and minimal, we construct the first candidate DFA A_1:

- **States**: $Q = \{q_\epsilon, q_a\}$.
- **Initial State**: q_ϵ.

- **Final States**: $F = \{q_\epsilon\}$ (since Table$(\epsilon, \epsilon) = 1$).
- **Transitions**:

$$
\begin{aligned}
\delta(q_\epsilon, a) &= q_a && \text{(since } \mathrm{row}(\epsilon \cdot a) = \mathrm{row}(a))\\
\delta(q_\epsilon, b) &= q_a && \text{(since } \mathrm{row}(\epsilon \cdot b) = \mathrm{row}(b) = \mathrm{row}(a))\\
\delta(q_a, a) &= q_a && \text{(since } \mathrm{row}(a \cdot a) = \mathrm{row}(aa) = \mathrm{row}(a))\\
\delta(q_a, b) &= q_\epsilon && \text{(since } \mathrm{row}(a \cdot b) = \mathrm{row}(ab) = \mathrm{row}(\epsilon))
\end{aligned}
$$

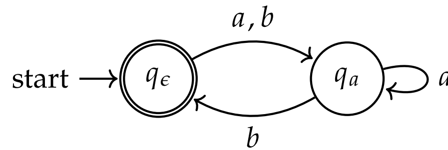


Figure 14.2: Candidate DFA A_1 proposed at the end of Iteration 2.

The learner submits A_1 in an Equivalence Query. The teacher checks if $L(A_1) = (ab)^*$. It finds that A_1 accepts the string bb (since $\delta(q_\epsilon, b) = q_a$ and $\delta(q_a, b) = q_\epsilon$, which is accepting). However, $bb \notin L$. The teacher replies **No** and returns the counterexample:

$$w = bb$$

The suffixes of w are $\{\epsilon, b, bb\}$. Since ϵ is already in T, we add $\{b, bb\}$ to the test set:

$$T = \{\epsilon, b, bb\}$$

Adding these columns will split the row profiles and distinguish states that were previously merged.

14.4.3 Iteration 3

We reconstruct the observation table with the new column suffixes. The sets are $S = \{\epsilon, a\}$, and extensions $S \cdot \Sigma = \{b, aa, ab\}$. We query MQs to fill the cells.

$S \cup (S \cdot \Sigma)$		ϵ	b	bb
ϵ	(S)	1	0	0
a	(S)	0	1	0
b		0	0	0
aa		0	0	0
ab		1	0	0

Table 14.3: Observation table in Iteration 3.

We check T-completeness:

- The representative profiles are $\mathrm{row}(\epsilon) = \langle 1, 0, 0 \rangle$ and $\mathrm{row}(a) = \langle 0, 1, 0 \rangle$.
- The extension profiles are:
  - $\mathrm{row}(b) = \langle 0, 0, 0 \rangle$ (no match in S)
  - $\mathrm{row}(aa) = \langle 0, 0, 0 \rangle$ (no match in S)
  - $\mathrm{row}(ab) = \langle 1, 0, 0 \rangle = \mathrm{row}(\epsilon)$ (matched)

The extension b has a row profile $\langle 0, 0, 0 \rangle$ that is not present in S. The table is **not complete**.

We add the prefix b to the set of representatives:

$$S = \{\epsilon, a, b\}, \quad S \cdot \Sigma = \{aa, ab, ba, bb\}$$

14.4.4 Iteration 4

We fill the new rows for ba and bb. Since they both start with b, any extension of them will also start with b and can never belong to $L = (ab)^*$. Thus, all their suffix extensions are 0.

$S \cup (S \cdot \Sigma)$		ϵ	b	bb
ϵ	(S)	1	0	0
a	(S)	0	1	0
b	(S)	0	0	0
aa		0	0	0
ab		1	0	0
ba		0	0	0
bb		0	0	0

Table 14.4: Observation table in Iteration 4.

We check T-completeness:

- The representative profiles are $\text{row}(\epsilon) = \langle 1, 0, 0 \rangle$, $\text{row}(a) = \langle 0, 1, 0 \rangle$, and $\text{row}(b) = \langle 0, 0, 0 \rangle$.
- The extension profiles are:
  - $\text{row}(aa) = \langle 0, 0, 0 \rangle = \text{row}(b)$ (matched)
  - $\text{row}(ab) = \langle 1, 0, 0 \rangle = \text{row}(\epsilon)$ (matched)
  - $\text{row}(ba) = \langle 0, 0, 0 \rangle = \text{row}(b)$ (matched)
  - $\text{row}(bb) = \langle 0, 0, 0 \rangle = \text{row}(b)$ (matched)

All extensions have a matching row profile in S. The table is **complete**.

We construct the second candidate DFA A_2:

- **States**: $Q = \{q_\epsilon, q_a, q_b\}$.
- **Initial State**: q_ϵ.
- **Final States**: $F = \{q_\epsilon\}$.
- **Transitions**:

$$\begin{aligned}
\delta(q_\epsilon, a) &= q_a && (\text{since row}(\epsilon \cdot a) = \text{row}(a)) \\
\delta(q_\epsilon, b) &= q_b && (\text{since row}(\epsilon \cdot b) = \text{row}(b)) \\
\delta(q_a, a) &= q_b && (\text{since row}(a \cdot a) = \text{row}(b)) \\
\delta(q_a, b) &= q_\epsilon && (\text{since row}(a \cdot b) = \text{row}(\epsilon)) \\
\delta(q_b, a) &= q_b && (\text{since row}(b \cdot a) = \text{row}(b)) \\
\delta(q_b, b) &= q_b && (\text{since row}(b \cdot b) = \text{row}(b))
\end{aligned}$$

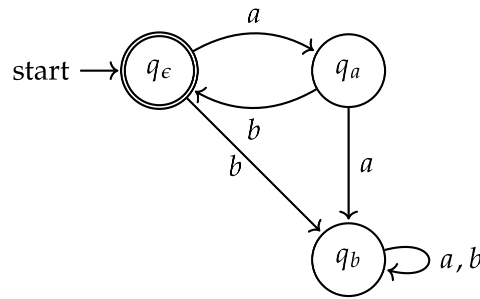


Figure 14.3: Candidate DFA A_2 proposed at the end of Iteration 4.

The learner submits A_2 to the teacher in an Equivalence Query. The teacher checks if $L(A_2) = (ab)^*$. Since A_2 is isomorphic to the minimal DFA for $(ab)^*$ (with q_b as the sink state), the teacher replies **Yes**. The algorithm terminates successfully.

14.5 Correctness, Termination, and Complexity

To establish the utility of the L^* algorithm, we must prove that it is guaranteed to terminate with the correct minimal DFA, and that it does so using a polynomial number of queries.

14.5.1 Complexity bounds

Let n be the number of states of the unique minimal DFA recognizing the target language L, and let m be the maximum length of any counterexample returned by the teacher during equivalence queries.

- **Size of S**: Since the observation table is maintained T-minimal, the set of representatives S contains at most one string for each equivalence class. By Theorem 14.1.3, the number of equivalence classes is n. Therefore, $|S| \leq n$.
- **Number of Equivalence Queries**: Each time we fail an equivalence query, we add suffixes of the counterexample to T. We will prove below that this forces the addition of at least one new representative to S in the next iteration. Since $|S|$ starts at 1 and cannot exceed n, the algorithm can perform at most $n - 1$ unsuccessful equivalence queries. Thus, the total number of equivalence queries is at most n.
- **Size of T**: We start with $T = \{\epsilon\}$. Each of the at most $n - 1$ counterexamples has length at most m, contributing at most m suffixes. Thus, $|T| \leq 1 + (n-1)m$.
- **Number of Membership Queries**: The size of the observation table is $(|S| + |S| \cdot |\Sigma|) \times |T|$. Since $|S| \leq n$ and $|T| \leq nm$, the table has at most $n(1 + |\Sigma|) \cdot nm = O(|\Sigma| n^2 m)$ cells. Each cell requires one Membership Query, so the total number of MQs is polynomial in n, m, and $|\Sigma|$.

14.5.2 Proof of correctness and termination

The key to the termination proof is showing that a counterexample really does force a state split. If we add the suffixes of a counterexample w to T, the table cannot remain complete without introducing a new state representative.

■ **Formal details — Proof roadmap: Counterexample state split**

Let A be the candidate DFA constructed from a complete and minimal table (S, T) in Phase 2. Suppose A is incorrect, and the teacher returns a counterexample $w = a_1 a_2 \ldots a_k$ of length $k \leq m$.

Let $T' = T \cup \text{suffixes}(w)$ be the updated test set, where $\text{suffixes}(w) = \{t_0, t_1, \ldots, t_k\}$ and each suffix is defined as:

$$t_i = a_{i+1} a_{i+2} \ldots a_k \quad (\text{with } t_k = \epsilon,\ t_0 = w)$$

We define the sequence of states $s_0, s_1, \ldots, s_k \in S$ reached by the candidate DFA A when processing the prefixes of w. Specifically, for each $i \in \{0, \ldots, k\}$, let:

$$s_i = \delta^*(q_\epsilon, a_1 \ldots a_i)$$

where $s_0 = \epsilon$ and s_i is the representative of the state reached after consuming prefix $a_1 \ldots a_i$.

We assume, for the sake of contradiction, that the table remains T'-complete with the same set of representatives S. That is, no new representatives are added.

The most unintuitive—yet brilliant—step of this proof involves the string $s_i t_i$. Why this string? Think of it operationally: we feed the counterexample w into the automaton, but we *pause* it after reading the prefix $a_1 \ldots a_i$. At this point, the automaton is in state s_i. We then *swap out* the prefix we just read for the canonical string representative s_i, and *play* the remaining suffix t_i. By tracking the membership of this hybrid string $s_i \cdot t_i$ as i goes from 0 to k, we can pinpoint exactly where the candidate automaton made a faulty transition. Under the assumption of completeness, we prove by induction that for all $i \in \{0, \ldots, k\}$, the string $s_i \cdot t_i$ has the same membership status in L:

$$s_i t_i \in L \iff s_{i+1} t_{i+1} \in L$$

For the inductive step, consider any $i \in \{0, \ldots, k-1\}$. By definition, the transition of A from state s_i on symbol a_{i+1} goes to state s_{i+1}:

$$\delta(s_i, a_{i+1}) = s_{i+1}$$

By the construction of the DFA's transition function (Section 14.3), this transition is well-defined because:

$$\text{row}_T(s_i \cdot a_{i+1}) = \text{row}_T(s_{i+1})$$

Since we assumed the table remains complete under T', we must have:

$$\text{row}_{T'}(s_i \cdot a_{i+1}) = \text{row}_{T'}(s_{i+1})$$

This equality implies that for all suffixes $t \in T'$:

$$(s_i a_{i+1}) t \in L \iff s_{i+1} t \in L$$

Since t_{i+1} is a suffix of the counterexample w, we have $t_{i+1} \in T'$. Choosing $t = t_{i+1}$ yields:

$$s_i a_{i+1} t_{i+1} \in L \iff s_{i+1} t_{i+1} \in L$$

Since $a_{i+1} t_{i+1} = t_i$, we obtain:

$$s_i t_i \in L \iff s_{i+1} t_{i+1} \in L$$

By induction from $i = 0$ to k, we obtain:

$$s_0 t_0 \in L \iff s_1 t_1 \in L \iff \cdots \iff s_k t_k \in L$$

Now we evaluate the endpoints of this chain:

- **Left-hand side**: $s_0 t_0 = \epsilon \cdot w = w$.
- **Right-hand side**: $s_k t_k = s_k \cdot \epsilon = s_k$.

Thus, we have:

$$w \in L \iff s_k \in L$$

However, s_k is the final state reached by A when reading w. By the construction of final states in A, s_k is a final state ($s_k \in F$) if and only if $s_k \in L$. Therefore, A accepts w if and only if $s_k \in L$, which is true if and only if $w \in L$.

This implies that A accepts w if and only if $w \in L$, which means A correctly classifies the counterexample w. This directly contradicts the fact that w is a counterexample ($w \in L(A) \iff w \notin L$).

This contradiction shows that the assumption of T'-completeness with the same set S must be false. Therefore, adding the suffixes of w to T must break the completeness of the table, forcing the algorithm to add at least one new state representative to S in the next iteration. Since the size of S is bounded by n, the algorithm must terminate.

Summary & Key Takeaways

- The *Myhill–Nerode theorem* characterises regular languages via right-congruences of finite index; the number of equivalence classes equals the number of states of the minimal DFA.
- *Passive learning* (from a fixed sample) cannot guarantee correctness; *active learning* uses a Minimally Adequate Teacher (MAT) with membership and equivalence queries.
- *Angluin's L^* algorithm* maintains an observation table, ensures closedness and consistency, extracts a candidate DFA, and refines on counterexamples.
- L^* terminates after at most n counterexample rounds (where n is the size of the target DFA) and uses $O(n^2 m + n^3)$ membership queries, where m is the longest counterexample.

Exercises

Exercise 1 (Myhill–Nerode classes). Let $L = \{w \in \{a, b\}^* \mid w \text{ contains } ab\}$. List the equivalence classes of the Myhill–Nerode relation $\equiv_L$ and draw the minimal DFA.

Exercise 2 (Observation table step). Consider the target language $L = \{w \in \{a, b\}^* \mid |w| \text{ is even}\}$. Starting from $S = \{\varepsilon\}$ and $T = \{\varepsilon\}$, fill the initial observation table, check closedness, and perform one round of the L^* algorithm until the table is closed and consistent.

Exercise 3 (Counterexample processing). Suppose the current candidate DFA has two states and accepts $L' = \{a, b\}^*$ but the target language is $L = \{w \mid w \text{ ends in } a\}$. The teacher returns the counterexample b. Show how the suffixes of b are added to T and which new state representative emerges.

Exercise 4 (Query complexity). A target DFA has $n = 5$ states and the longest counterexample has $m = 8$. How many membership queries does the L^* complexity bound predict? Compare this to the number of queries needed to test all words up to length m exhaustively.

CHAPTER 15

Security Protocol Verification with Tamarin

The last chapter ended with symbolic verification over Boolean state sets. Tamarin moves that symbolic idea into a message-passing world. Instead of a transition relation over bits, we use multiset rewriting over facts; instead of model states, we track protocol roles, network messages, and attacker knowledge; instead of a single execution graph, we reason about traces generated by rule instances. The point is the same: compress an enormous search space into a representation we can still manipulate mechanically. The difference is the object being represented.

Chapter 13 reasoned about sets of states. Security protocols need a different state model: not a control bit-vector, but a live multiset of facts describing who knows what, which role instance is active, and which messages have moved across the network.

Tamarin is not just a theoretical toy. It has had massive real-world impact. During the standardisation of the 5G mobile protocol, researchers used Tamarin to formally analyze the authentication handshakes. They discovered critical flaws that would have allowed attackers to break user anonymity and track mobile devices as they moved between antennas. The standard was subsequently patched based on Tamarin's proofs. It has also been used to find flaws in TLS 1.3 draft specifications.

The chapter follows the core ideas that recur in the Tamarin lecture block. We begin with the intuition behind protocol verification and the Dolev-Yao attacker model. Then we build the syntax and semantics of multiset rewriting from the ground up: terms, facts, fresh names, and rewrite rules. After that we shift from individual rules to protocol roles and traces, which is where a local step description becomes a global execution history. The closing section shows how Tamarin states properties over traces and searches for proofs or counterexamples.

Where we are. Symbolic model checking compressed large Boolean state spaces into formulas and decision diagrams. Security protocols need the same kind of compression, but their states are not fixed bit-vectors.

What this chapter adds. Tamarin represents protocol executions with terms, facts, multiset rewrite rules, and traces. This lets us reason about fresh names, adversary knowledge, role instances, and long-lived secrets in a single symbolic framework.

Where it leads. The chapter closes the notes by showing how the verification ideas developed for automata and transition systems become a tool-supported workflow for real security protocols.

Chapter map.

- Section 15.1 moves from state machines to protocol histories.
- Section 15.2 defines terms, facts, and multiset rewriting rules.
- Section 15.3 connects local protocol roles to global traces.

- Section 15.4 explains equational theories and unification.
- Section 15.5 states secrecy, authentication, and trace properties.
- Section 15.6 distils the modelling discipline.

15.1 From State Machines to Protocol Histories

Security protocols are not ordinary transition systems. A controller model usually has a small, fixed set of control locations and a neat transition graph. A protocol model has messages, nonces, long-term keys, sessions, replays, and an adversary who can store, forward, combine, and fabricate messages. The interesting question is therefore not just "which control state comes next?" but "which facts are currently in scope, which messages can be derived, and which role instances can continue?"

Intermezzo — Why protocol verification looks stateful

The state of a protocol run is not only the local control state of the honest participants. It also includes the messages in flight, the facts the adversary has learned, and the fresh values that were created earlier in the run. Tamarin keeps all of that in a single symbolic object: a multiset of facts.

The standard abstraction behind Tamarin is the Dolev-Yao model. Messages are symbolic terms, not bitstrings. Cryptography is idealised by equations and by the intruder's deduction power: if a message is sent on the network, the attacker can see it; if the attacker can derive a message from what it knows, it can send that message back on the network. Honest participants do not get special protection from the model. They survive only if the protocol rules force the attacker to miss a crucial piece of information.

That model is deliberately austere, but it is exactly what we need for protocol analysis. It lets us describe the protocol once, then quantify over all possible attacker behaviours. In the lecture material, this is the move from "one run we can simulate" to "all runs we must inspect".

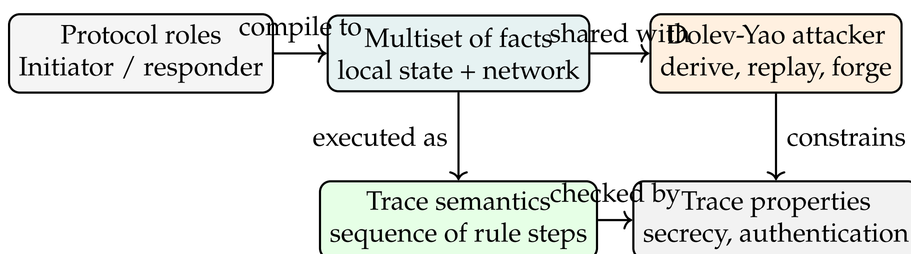


Figure 15.1: Tamarin keeps the protocol, the attacker, and the property in one symbolic picture. Roles generate facts, facts generate traces, and traces are the object on which properties are checked.

15.1.1 Case study: Designing a secure key exchange

To understand why security protocol verification requires specialized tools, we trace the step-by-step design of a protocol to establish a shared session key K_{AB} between Alice (A) and Bob (B) with the help of a trusted server S. This case study highlights how easily subtle vulnerabilities can slip into seemingly simple designs.

Attempt 1: Plaintext exchange

We assume Alice wants to communicate with Bob. She contacts the server, requesting a session key:

$$A \to S : A, B$$

The server generates a fresh session key K_{AB} and sends it to Alice:

$$S \to A : K_{AB}$$

Alice then forwards the key to Bob:

$$A \to B : K_{AB}$$

Vulnerability (Eavesdropping): Since the network is insecure and controlled by the Dolev-Yao attacker, the intruder can read all messages in transit. The attacker learns K_{AB} immediately from the server's reply or from Alice's forward. Secrecy is completely lost.

Attempt 2: Encrypted key distribution (no identities)

To protect the session key, we strengthen our assumptions. We assume that Alice shares a long-term secret symmetric key K_{AS} with the server, and Bob shares a key K_{BS} with the server. Now, the server can encrypt the session key:

$$\begin{aligned} &A \to S : A, B \\ &S \to A : \{K_{AB}\}_{K_{AS}}, \{K_{AB}\}_{K_{BS}} \\ &A \to B : \{K_{AB}\}_{K_{BS}} \end{aligned}$$

Alice decrypts $\{K_{AB}\}_{K_{AS}}$ to learn the session key, and forwards the second encrypted package to Bob. Bob decrypts it using K_{BS} to learn K_{AB}.

Vulnerability (Man-in-the-Middle): An intruder I, who is a registered user of the system and shares a key K_{IS} with the server, can intercept Alice's request and launch a Man-in-the-Middle (MITM) attack:

1. Alice initiates communication with Bob: $A \to S : A, B$.
2. The intruder intercepts this message and tampers with Bob's identity, replacing it with his own: $I(A) \to S : A, I$.
3. The server thinks Alice wants to communicate with the intruder. It generates a session key K_{AI} and replies:

$$S \to A : \{K_{AI}\}_{K_{AS}}, \{K_{AI}\}_{K_{IS}}$$

4. The intruder intercepts this reply. Since he knows K_{IS}, he decrypts $\{K_{AI}\}_{K_{IS}}$ and learns K_{AI}.
5. The intruder forwards $\{K_{AI}\}_{K_{AS}}$ to Alice.
6. Alice decrypts it. Because the encrypted payload does not contain the identity of her intended partner, she assumes that K_{AI} is the session key shared with Bob.

Alice now encrypts her messages for Bob using K_{AI}. The intruder intercepts them, decrypts them (since he knows K_{AI}), and can read or modify them before optionally re-encrypting them and forwarding them to Bob.

Attempt 3: Encrypted key with identities

To prevent the MITM attack, the server must explicitly bind the session key to the identities of the intended participants inside the encrypted payloads:

$$A \to S : A, B$$
$$S \to A : \{K_{AB}, B\}_{K_{AS}}, \{K_{AB}, A\}_{K_{BS}}$$
$$A \to B : \{K_{AB}, A\}_{K_{BS}}$$

If the intruder attempts the same tampering ($I(A) \to S : A, I$), the server replies with $\{K_{AI}, I\}_{K_{AS}}$. When Alice decrypts this, she sees that the key is bound to the intruder's identity I rather than Bob's, detects the discrepancy, and aborts the protocol.

Vulnerability (Replay attack): Suppose Alice and Bob successfully run the protocol and establish a session key K_{AB}. The intruder eavesdrops and stores the message $A \to B : \{K_{AB}, A\}_{K_{BS}}$ on his hard drive.

Now, imagine that five years later, the session key K_{AB} is compromised (e.g., through brute force or cryptanalysis). The intruder can now spoof Alice and replay the old encrypted message to Bob:

$$I(A) \to B : \{K_{AB}, A\}_{K_{BS}}$$

Bob decrypts the message, sees that it is a valid ciphertext encrypted by the server containing Alice's identity, and accepts K_{AB} as a fresh, secure session key. The intruder can now impersonate Alice to Bob or decrypt Bob's messages. This attack succeeds because Bob has no way of verifying the *freshness* of the message, highlighting the need for nonces or timestamps in security protocol design.

The bridge from Chapter 13. Symbolic model checking represented sets of states by Boolean formulas. Tamarin represents reachable protocol situations by multisets of facts and searches the induced trace space. The level of abstraction is comparable; the underlying object is simply different.

15.2 Multiset Rewriting: Terms, Facts, and Rules

15.2.1 Terms

At the base of the formalism are symbolic messages, which Tamarin calls *terms*. We stay in the free term algebra for the core presentation.

> **Definition 15.2.1** (Terms)**.** Let $\mathcal{N}$ be a set of names, $\mathcal{V}$ a set of variables, and $\mathcal{F}$ a set of function symbols with fixed arities. The set of terms is generated by
>
> $$t ::= n \mid x \mid f(t_1, \ldots, t_k)$$
>
> where $n \in \mathcal{N}$, $x \in \mathcal{V}$, and $f \in \mathcal{F}$ has arity k. Fresh names are usually written with a tilde, such as $\tilde{n}$, to indicate that they are generated during a protocol run.

Idealized Cryptography and Probabilities. In reality, when a protocol generates a 256-bit random nonce $\tilde{n}$, an attacker has a tiny, non-zero probability of guessing it correctly. Tamarin explicitly idealizes this. Fresh values are

modeled symbolically so that they absolutely *cannot* be guessed by the attacker. The tiny cryptographic probability is abstracted into a rigorous zero, allowing the tool to focus entirely on logical flaws rather than statistical ones.

Terms are the symbolic payloads of the protocol. A nonce, a key, a pair of values, or an encrypted message are all just terms. The point is not that the model knows how to perform real cryptography. The point is that the model can talk about who created which symbolic value and who later saw it.

Example 15.2.2 (A message term)**.** If an initiator sends its identity together with a fresh nonce, the network message can be represented by the term

$$\langle A, \tilde{n} \rangle.$$

If the protocol later nests this value inside a larger structure, the term tree records that structure explicitly. There is no hidden state inside the message. Every piece of information that matters is present in the term.

15.2.2 Facts

Facts are the protocol state. They are the resources, memories, and events that a rule can consume or produce.

Definition 15.2.3 (Facts)**.** A fact is an atom of the form $F(t_1, \ldots, t_k)$, where F is a predicate symbol and the t_i are terms. Facts may be linear or persistent. A persistent fact is written with a leading bang, as in $!F(t_1, \ldots, t_k)$.

Linear facts are consumed when a rule fires. Persistent facts remain available after use. This distinction is one of the most useful parts of the formalism:

- linear facts model one-shot resources, local control states, and ephemeral protocol state;
- persistent facts model long-term keys, static configuration, or any fact that should survive across multiple rule applications.

The most common special facts are the network facts:

- $\mathsf{Out}(m)$ publishes a message to the network;
- $\mathsf{In}(m)$ consumes a message from the network.

The adversary sees every $\mathsf{Out}(m)$ and may later supply any message that can be built from its current knowledge. In other words, the attacker is not an extra agent bolted onto the model afterwards. It is part of the semantics of how network facts are interpreted.

15.2.3 Rewrite rules

Protocols evolve by rewriting one multiset of facts into another.

Definition 15.2.4 (Multiset rewriting rule)**.** A rewrite rule has the shape

$$L \xrightarrow{A} R$$

where L and R are multisets of facts and A is a multiset of action facts. Informally, L is the rule's premise, R is its conclusion, and A records the observable events generated by the step.

An instance of a rule is obtained by choosing a substitution σ for the variables in the rule so that the left-hand side matches a submultiset of the current state. Then the matching facts are consumed, the right-hand side is added, and the action facts are appended to the trace. Fresh names are created at the moment the rule fires, so a rule using a fresh variable really does create a new symbolic value.

Definition 15.2.5 (One rewriting step)**.** Let M be a multiset of facts and let $L \xrightarrow{A} R$ be a rule. If there exists a substitution σ such that $L\sigma \subseteq M$, then one step of rewriting produces

$$M' = (M \setminus L\sigma) \uplus R\sigma.$$

The corresponding action facts $A\sigma$ are recorded in the trace.

Intuition: Sticky notes on a fridge. Imagine a large whiteboard or a fridge. A multiset of facts is just a collection of sticky notes placed on the fridge. When a rule fires, you remove specific sticky notes (linear facts) and slap on new ones. Persistent facts are like sticky notes glued to the fridge permanently. The trace is a ledger remembering every sticky note you ever moved.

Why do we need a *multi*set instead of a normal set? Because a single protocol can be instantiated multiple times concurrently. If Alice starts two simultaneous sessions, there will be two identical `start_Alice` sticky notes on the fridge. Set semantics would collapse them into one, breaking the ability to track independent parallel sessions. Multisets allow multiple identical sticky notes.

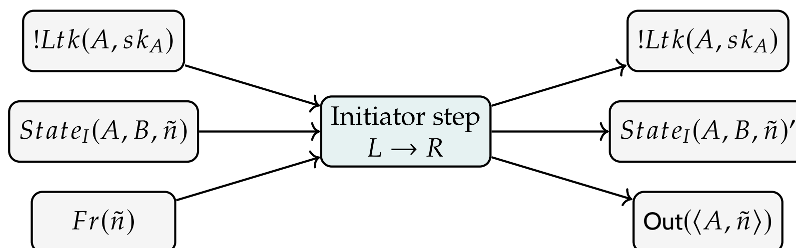


Figure 15.2: A rewrite rule consumes linear facts, preserves persistent facts, and emits new facts and action facts. The trace stores the rule application even when the consumed facts disappear from the state.

15.2.4 A concrete toy protocol

The simplest way to see the machinery is to write down a tiny handshake. The protocol is intentionally small; the goal is to show the shape of the rules, not to claim cryptographic strength.

```
1 rule Initiate:
2   [ Fr(~na), !Ltk(A, skA) ]
3   --[ Begin(A, B, ~na) ]->
```

```
4     [ StateI(A, B, ~na), Out(<A, ~na>) ]
5
6   rule Respond:
7     [ In(<A, na>), Fr(~nb), !Ltk(B, skB) ]
8     --[ End(B, A, na, ~nb) ]->
9     [ StateR(B, A, na, ~nb), Out(<na, ~nb>) ]
```

The first rule models an initiator session. It creates a fresh nonce, records that the session started, stores a local state fact, and sends the first message. The second rule models a responder session. It receives a message from the network, creates its own fresh nonce, records a completion event, stores its local state, and sends a reply.

This is the key modelling pattern in Tamarin:

- a *role instance* is represented by a family of rules that keep track of local state facts;
- a *fresh value* is created by a rule with a fresh premise;
- a *network message* is represented by In and Out facts;
- long-term secrets stay persistent, because they should survive across many sessions.

The protocol above is deliberately minimal, but the modelling style scales to real protocols. A TLS handshake, a key exchange, or an authentication API is written the same way: local state becomes facts, communication becomes network facts, and security-relevant steps become action facts.

15.2.5 The Dolev-Yao Attacker Rules

In Tamarin, the network is not a passive pipe; it is completely controlled by an active adversary modeled after Dolev and Yao. The attacker is not an external entity defined by custom protocol rules. Instead, the attacker is baked directly into the semantics of the network facts Out and In through built-in rewriting rules.

To track what the attacker has learned, Tamarin introduces a special persistent fact, $\mathsf{K}(x)$, which indicates that the attacker knows the term x. Since K is persistent, once the attacker learns a message, they never forget it.

The behavior of the Dolev-Yao attacker is defined by three main classes of built-in rewrite rules:

1. **Eavesdropping (Network Listening)**: Whenever a protocol rule outputs a message x to the network, the attacker immediately intercepts and learns it:

$$\mathsf{Out}(x) \longrightarrow \mathsf{K}(x)$$

2. **Injection (Message Spoofing)**: The attacker can send any message x they know to the network, allowing honest participants to consume it:

$$\mathsf{K}(x) \longrightarrow \mathsf{In}(x)$$

3. **Fresh Value Generation**: The attacker can generate fresh values (e.g., to model their own nonces or keys):

$$\longrightarrow \mathsf{Fr}(x)$$

4. **Term Derivation (Composition and Decomposition)**: For every function symbol f of arity k available in the protocol's signature, the attacker can apply f to terms they already know to construct a new term:

$$\mathsf{K}(x_1), \ldots, \mathsf{K}(x_k) \longrightarrow \mathsf{K}(f(x_1, \ldots, x_k))$$

   If the signature includes cryptographic functions (e.g., encryption, hashing, decryption), the attacker can apply them. However, they can only decrypt an encrypted term if they also know the corresponding key term, which is enforced by the equational theory.

■ **Intermezzo — Syntactic Constraints in Tamarin**

Because the attacker rules and value generation are built-in, Tamarin enforces strict syntactic constraints on how the user writes protocol rules:

- **Network Asymmetry**: The protocol designer must only use $\mathsf{In}(x)$ on the left-hand side (consuming from the network) and $\mathsf{Out}(x)$ on the right-hand side (sending to the network).
- **Freshness Asymmetry**: The user can only use the $\mathsf{Fr}(\tilde{n})$ fact on the *left-hand side* of a rule to bind a newly generated fresh value. The engine handles the infinite supply of fresh values implicitly.

This ensures that all messages sent by honest participants are routed through the attacker's knowledge pool, and that new nonces are truly fresh and unpredictable.

15.3 Protocol Roles and Traces

A protocol specification is easier to read when its rules are grouped into *roles*. A role is a parameterised template for one participant's view of the protocol. The same role may be instantiated many times, because a real protocol runs many sessions concurrently.

> **Definition 15.3.1** (Protocol role)**.** A protocol role is a parameterised collection of rewrite rules that share a local state discipline. Different instantiations of the same role may use different agent names, fresh values, and long-term keys, but they obey the same rule skeleton.

Think of a role as the protocol's control-flow skeleton. It says which local fact comes next, which message is sent or received, and which events are recorded. The role itself does not choose the actual attacker behaviour; it only states what an honest participant would do if the expected message is available.

15.3.1 A role-level view of the toy handshake

The handshake from the previous section can be read as two roles:

- an initiator role that creates a fresh nonce and sends the first message;
- a responder role that consumes that message, creates its own fresh nonce, and answers.

The pairing of those roles is what makes the protocol meaningful. If one side never answers, the trace stops. If the attacker forges a message, the trace records that the responder accepted it. If a fresh value is reused, that reuse is visible in the fact history.

Example 15.3.2 (Why role state matters)**.** Suppose a server accepts a request only once per session key. That condition cannot be expressed only by the network messages themselves, because the same message may appear multiple times. The server must therefore carry a local state fact such as `SeenKey(k)` or `StateS(k)`. The role updates that fact when a key is used, and later rules consult it to block replay.

15.3.2 Traces

Tamarin's properties are not checked against a single rewrite step. They are checked against traces: sequences of rule applications with their action facts.

Definition 15.3.3 (Trace)**.** A trace is a finite sequence of rule instances

$$(r_1, \sigma_1), (r_2, \sigma_2), \ldots, (r_n, \sigma_n)$$

starting from the initial multiset of facts. Each step rewrites the current multiset and appends the instantiated action facts to the observable history.

Internal State vs. Observable Events. A critical pedagogical point is the difference between state facts (inside the brackets `[ ]`) and action facts (above the arrow `-[ ]->`). The first-order logic formulas we will write next *cannot see* the internal state facts on the fridge. They only see the action facts appended to the trace. Action facts act as the deliberate API or "logging statements" that expose relevant internal steps to the verification logic.

In the tool, the observable history is usually written with timepoints. If an action fact $F(t)$ occurs at the i-th step, we write $F(t)@\#i$. The timepoints let the logic talk about ordering:

$$\#i < \#j$$

means that the event at $\#i$ happened strictly before the event at $\#j$. This is how Tamarin expresses correspondence arguments such as "every acceptance must have a prior send".

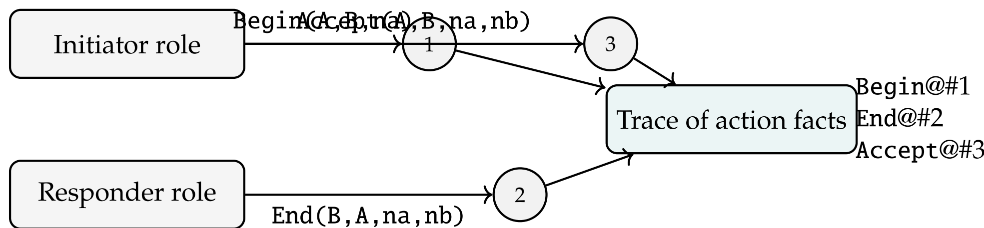


Figure 15.3: A trace is the observable history of the run. Action facts are attached to timepoints, which lets the logic express ordering and causality between protocol events.

Why traces are the right object. Security properties are rarely about one isolated state. They are about causality over time: a secret was created before it was leaked; a responder accepted only after some initiator really started; a key agreement happened only if both parties saw the same fresh values. The trace records exactly that causal history.

15.3.3 A classical protocol pattern

The Needham-Schroeder family of protocols is a classic teaching example for Tamarin because it highlights the same ingredients in a slightly richer form: fresh nonces, long-term keys, and correspondence between sends and receives. One role starts with a fresh challenge, another role answers, and the security question becomes whether the attacker can force a trace where the answer is accepted without the expected prior event.

The important lesson is not the exact message format. It is the structure of the specification:

1. each honest role is described by a small number of rules;
2. each important protocol step emits an action fact;
3. the trace becomes the evidence for or against the security claim.

15.4 Equational Theories and Unification

In a free term algebra, two terms are equal if and only if they are syntactically identical. For example, the term $enc(m, k)$ and the variable x do not match unless x is bound to the exact syntax $enc(m, k)$.

However, real security protocols rely on cryptographic primitives where terms have algebraic relationships. For instance, decrypting a message with the correct key should yield the plaintext:

$$dec(enc(m, k), k) = m$$

Here, the terms $dec(enc(m, k), k)$ and m are syntactically different, but we want the system to treat them as semantically equivalent. To achieve this, Tamarin interprets terms modulo an *equational theory* E (a set of equations defining equivalence classes over terms).

15.4.1 Semantic Equivalence in Attacker Knowledge

The equational theory is the mechanism through which the Dolev-Yao attacker decrypts intercepted messages. Rather than deconstructing a term $enc(m, k)$ directly (which would violate the black-box assumption of ideal cryptography), the attacker operates by applying functions to terms they already know:

1. The attacker intercepts $enc(m, k)$, learning the term: $\mathsf{K}(enc(m, k))$.
2. Later, the attacker intercepts the key k, learning: $\mathsf{K}(k)$.
3. Using the built-in term derivation rule, the attacker applies the decryption function symbol dec to construct a new term:

$$\mathsf{K}(dec(enc(m, k), k))$$

4. Because the equational theory states that $dec(enc(m, k), k) \equiv_E m$, the term in the attacker's knowledge base is semantically equivalent to m. The attacker now knows the plaintext m.

15.4.2 The Unification Problem

To apply a rewrite rule $L \xrightarrow{A} R$ to a concrete multiset of facts M, Tamarin must find a substitution σ such that the facts in $L\sigma$ match a subset of M. This matching process requires solving equations between terms:

- In the *free term algebra* (syntactic equality), matching is a special case of unification. The classical **Martelli-Montanari** algorithm (1982) solves unification in linear time by repeatedly transforming equations into solved forms.
- Modulo an *equational theory* E (E-unification), we must find a substitution σ such that $t_1\sigma \equiv_E t_2\sigma$. In full generality, E-unification is undecidable, and even when decidable, it can yield infinitely many minimal unifiers.

15.4.3 Subterm-Convergent Theories

To maintain decidability and efficiency, Tamarin permits the user to define custom equations, but restricts them to a class known as **subterm-convergent equational theories**.

> **Definition 15.4.1** (Subterm-Convergent Equation)**.** An equation $L = R$ is subterm-convergent if:
>
> 1. It can be oriented as a rewrite rule $L \rightarrow R$ from left to right.
> 2. The set of oriented rules is *terminating* (there are no infinite chains of rewrites) and *confluent* (rewriting a term in different orders always yields the same final result).
> 3. The right-hand side R is either a strict subterm of L or a constant.

For example, $dec(enc(m, k), k) = m$ is subterm-convergent because the right-hand side m is a subterm of the left-hand side.

Because subterm-convergent equations simplify terms (reducing their size or complexity), they guarantee that any term can be rewritten to a unique, irreducible *normal form* in linear time. Tamarin can then perform unification on these normal forms using the standard Martelli-Montanari style algorithm.

15.4.4 Non-Subterm-Convergent Theories and Undecidability

Many critical cryptographic primitives cannot be modeled with subterm-convergent equations. For example:

- **Associativity**: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
- **Commutativity**: $x \cdot y = y \cdot x$
- **Diffie-Hellman exponentiation**: $(g^x)^y = (g^y)^x$

These equations do not reduce the size of terms and have no clear direction of simplification. If a user were to write $x \cdot y = y \cdot x$ as a rewriting rule, the engine would enter an infinite loop:

$$x \cdot y \longrightarrow y \cdot x \longrightarrow x \cdot y \longrightarrow \ldots$$

To handle these complex algebraic structures without sacrificing termination, Tamarin does not allow users to define arbitrary non-subterm-convergent equations. Instead, it provides built-in equational theories (such as `diffie-hellman`, `bilinear-pairing`, or `associative-commutative`) that are hard-coded into the core prover. These built-in theories use specialized algebraic unification algorithms that reason about equivalence classes directly rather than relying on naive term rewriting.

15.5 Property Verification in Tamarin

Once a protocol is written as rewrite rules, Tamarin checks properties stated over traces. The property language is a first-order logic whose variables range over messages and timepoints. That is enough to express the standard verification questions in protocol analysis.

Definition 15.5.1 (Trace formula)**.** A trace formula may quantify over terms and timepoints, refer to action facts with the notation $F(t)@\#i$, and compare timepoints using relations such as $\#i < \#j$. Formulas are built from these atoms using the usual logical connectives and quantifiers.

Intermezzo — First-Order Logic Timepoints

Why the strange `@#i` notation? In standard First-Order Logic, to reason about an event occurring at a specific time, you would add a time parameter to a binary predicate, e.g., $\text{Event}(msg, t)$. Tamarin uses the `@` syntax to cleanly separate the protocol term (the message payload) from the temporal dimension (the trace index $\#i$). This separates the implicit timing of the rewriting rules from the explicit temporal logic of the properties.

The point of the logic is that it can state both local and temporal claims. For example, a secrecy statement says that a message should never appear in the intruder's knowledge; an authentication statement says that one event must be preceded by another; and a reachability statement says that there exists a trace satisfying a desired scenario.

15.5.1 Secrecy

Secrecy is the most basic property. In Tamarin it is usually phrased as the absence of intruder knowledge for a secret value.

Example 15.5.2 (Secrecy lemma)**.** If s is a session secret, then a typical secrecy lemma has the form

$$\forall \#i.\ \mathsf{Secret}(s)@\#i \longrightarrow \neg\exists \#j.\ \mathsf{K}(s)@\#j.$$

The intended reading is: whenever the protocol marks a value as secret, the attacker should never learn that value on any trace.

This is a good example of why traces matter. The property is not asking whether the secret is present in the current local state. It is asking whether the attacker can derive it at any point in the execution history.

15.5.2 Authentication and correspondence

Authentication is usually expressed as a correspondence between events. One event claims that a participant has finished a session; another event shows that the peer really started the same session earlier.

Example 15.5.3 (Correspondence lemma)**.** Suppose a responder records `End(B,A,na,nb)` when it accepts a session. A standard agreement property says that every such acceptance must have a prior initiator event:

$$\forall \#i.\ \mathsf{End}(B, A, na, nb)@\#i \longrightarrow \exists \#j.\ \mathsf{Begin}(A, B, na)@\#j \wedge \#j < \#i.$$

If the protocol is vulnerable to replay or impersonation, Tamarin will search for a trace where `End` occurs without a matching earlier `Begin`. That trace is the attack witness.

The exact names of the action facts are up to the modeler. What matters is that the events are chosen to expose the security-relevant milestones of the protocol. Once those milestones are in place, the property reads like an engineering statement about causality.

15.5.3 Reachability and existence

Not every property is universal. Sometimes we want to know whether a desired scenario is possible at all. Tamarin supports existential trace queries for that purpose.

Example 15.5.4 (Existence of a good run)**.** If a protocol has a successful completion event `Done`, one may ask whether there exists a trace that reaches it:

$$\exists \#i.\ \mathsf{Done}@\#i.$$

This is useful during model construction. Before proving that the protocol is secure, we often first check that the model can actually execute the intended good run.

What Tamarin returns. If the lemma is true, Tamarin produces a proof. If the lemma is false, it can return a counterexample trace showing an attack. If the search space is large, the proof may require user guidance in the form of hints, induction, or careful lemma shaping. The lecture point is that the verification task is still trace-based: the prover is trying either to establish a property of all traces or to expose one trace that violates it.

■ **Intermezzo — How to read a Tamarin result**

The output is not just “proved” or “failed”. A good proof tells you that all trace patterns allowed by the rules satisfy the lemma. A counterexample trace tells you exactly which sequence of rule applications breaks the claim. That trace is usually the most valuable debugging artefact in the whole analysis.

15.5.4 Forward Secrecy

A protocol that guarantees secrecy under normal runs might fail completely if long-term secrets are compromised in the future. For example, if an attacker records encrypted traffic today and steals the private key five years later, can they decrypt the old traffic?

A protocol achieves **forward secrecy** (or perfect forward secrecy) if the compromise of long-term keys does not compromise past session keys.

To verify forward secrecy in Tamarin, we must explicitly model the capability of the attacker to compromise long-term keys. We do this by adding a *key reveal rule* to our model:

```
rule Reveal_Ltk:
  [ !Ltk(A, ltk) ]
  --[ Reveal(A) ]->
  [ Out(ltk) ]
```

This rule consumes the persistent long-term key fact $\mathsf{!Ltk}(A, ltk)$ and sends it on the network via Out, which the Dolev-Yao attacker immediately learns. The action fact $\mathsf{Reveal}(A)$ is recorded on the trace to mark the exact moment of compromise.

We then write a forward secrecy lemma. It states that if a session key K is established between Alice (A) and Bob (B) at timepoint $\#i$, the attacker can only learn K if they compromised A or B's long-term key *before* the session key was used:

$$\forall A, B, K, \#i.\ \mathsf{SessionKey}(A, B, K)@\#i \longrightarrow (\neg(\exists \#j.\ \mathsf{K}(K)@\#j) \ \lor\ (\exists \#r.\ \mathsf{Reveal}(A)@\#r \ \land\ \#r < \#i) \ \lor\ (\exists \#r.\ \mathsf{Reveal}(B)@$$

If the attacker compromises a key at a future timepoint $\#r$ such that $\#i < \#r$, the formula requires that $\neg(\exists \#j.\ \mathsf{K}(K)@\#j)$ still holds. If Tamarin successfully proves this lemma, the protocol guarantees forward secrecy.

15.5.5 Anonymity and Observational Equivalence

Trace properties (like secrecy or agreement) are first-order formulas checked on individual traces. However, properties like *anonymity*, *privacy*, or *untrackability* cannot be expressed as properties of a single trace. They are properties of the relations between different executions: can the attacker distinguish between two different system configurations?

To verify these properties, Tamarin uses **observational equivalence**. Instead of verifying a first-order logic formula on a trace, Tamarin checks if the attacker can distinguish between two slightly different processes.

In Tamarin, this is modeled by writing a single protocol specification that contains a special built-in function $\mathsf{diff}(x, y)$. This function instructs the compiler to generate two versions of the system:

- System L, where the function evaluates to the left term x.
- System R, where the function evaluates to the right term y.

The prover then tries to show that for every execution in System L, there is a transition-by-transition equivalent execution in System R that produces the exact same observable outputs to the attacker, and vice versa.

Example 15.5.5 (Mobile Device Tracking and Anonymity in 5G)**.** Consider a mobile device D that connects to multiple cell antennas as the user moves (a scenario critical to the 5G standard analysis mentioned earlier). We want to check whether an eavesdropper listening to the antenna traffic can track the user's location. We set up a model where a device connects to Antenna 1 and then Antenna 2:

- In System L, the same device ID D connects to both antennas (simulating one user moving).

- In System R, device ID D_1 connects to Antenna 1, and a different device ID D_2 connects to Antenna 2 (simulating two different users).

We model this connection rule using the diff operator:

```
rule Connect_Antenna_2:
  [ State_Device(D), In(request) ]
  -->
  [ Out(diff(D, D_prime)) ]
```

If Tamarin proves observational equivalence, it guarantees that the attacker cannot distinguish a single user moving between antennas from two distinct users. Thus, the protocol guarantees anonymity and untrackability. If the proof fails, Tamarin returns a counterexample showing how the attacker can track the device (e.g., because the device reused a static cookie, certificate, or unencrypted identifier).

15.5.6 The modelling discipline

Tamarin rewards disciplined models. The rules should isolate the protocol state you care about; action facts should mark the events you want to reason about; and the properties should mention those events directly. If the model mixes too many concerns into one rule, the trace logic becomes hard to read. If the action facts are too coarse, the proof loses the causal structure that the property needs. Good models are therefore not just correct; they are observable.

This is the same engineering lesson that appeared in previous chapters in a different guise: symbolic verification works best when the model separates state, transition, and observation cleanly. Tamarin just applies that lesson to protocol histories.

15.6 Summary

Tamarin verifies security protocols by turning them into multiset rewriting systems and then reasoning about the traces those rules generate. Terms model symbolic messages. Facts model local state, network events, and persistent resources. Protocol roles package rules into reusable participant templates. Traces record the observable history of a run, and the property language states secrecy, authentication, and reachability claims over those traces.

The chapter's main idea is that protocol verification is not graph checking in the usual sense. It is history checking. Once that shift is clear, the rest of the formalism becomes natural: fresh names explain new values, persistent facts model long-term configuration, action facts expose relevant events, and timepoints let the logic talk about order and causality. That is the Tamarin viewpoint in one sentence: specify the protocol as rewrite rules, and specify the security claim as a property of the traces they produce.

Summary & Key Takeaways

- Security protocols are modelled as *multiset rewriting systems*: terms represent symbolic messages, facts represent local and network state, and rules describe protocol steps.
- The *Dolev–Yao attacker* controls the network: it can intercept, replay, and forge messages using any cryptographic operation whose key it knows.

- A *trace* is the observable history of a protocol run; properties (secrecy, authentication, forward secrecy) are stated as first-order formulas over traces.
- *Equational theories* capture algebraic properties of cryptographic primitives; subterm-convergent theories keep unification decidable.
- *Tamarin* automates this pipeline: protocol rules in, proof or counterexample trace out.

Exercises

Exercise 1 (Dolev–Yao deduction). The attacker knows the terms $\mathtt{aenc}(m, \mathtt{pk}(k))$ and k. List the sequence of Dolev–Yao rules the attacker applies to recover the plaintext m.

Exercise 2 (Trace reading). A Tamarin trace contains the action facts $\mathtt{Secret}(m)$ at timepoint $\#i$ and $\mathtt{K}(m)$ at timepoint $\#j$ with $\#i < \#j$. Does this trace satisfy the secrecy lemma $\forall m, \#i.\ \mathtt{Secret}(m) @ \#i \Rightarrow \neg(\exists \#j.\ \mathtt{K}(m) @ \#j)$? Explain your answer.

Exercise 3 (Authentication encoding). Write a Tamarin-style correspondence lemma expressing *non-injective agreement*: whenever Bob completes a session with Alice on a shared key k, Alice previously initiated a session with the same k. Then explain what additional quantifier structure is needed for *injective* agreement.

Exercise 4 (Forward secrecy). Consider a protocol that uses a long-term key $\mathtt{ltk}(A)$ to encrypt a session key k_s. After the session, the attacker learns $\mathtt{ltk}(A)$. Can the attacker recover k_s? What protocol mechanism would provide forward secrecy, and how would it change the Tamarin model?