Automata, Logic and Games – Lecture Notes

May 6, 2024 Shaull Almagor

1 Motivation

In this course, we will study the theoretical foundations of formal methods: automata, logic, and games.

1.1 Politically Correct Motivation

Formal methods in CS aim to give mathematical and precise models of systems and specification, so that we can reason about them automatically, without relying on e.g., human testing, and without sacrificing correctness.

What kinds of systems and specifications do we allow? This is a huge question, and we'll attempt to see some answers along the way. One immediate thing to keep in mind, is that if we allow systems or specifications that are too "rich", then things become undecidable, and we lose the algorithmic motivation. For example, if we model systems using Turing machines, and we allow the specification "the machine halts", then the reasoning reduces to the halting problem.

Let's start with an example of the type of reasoning we use.

Example 1.1. Consider an elevator working in an 8 floor building. The elevator can be in any of the 8 floors at any given time. Inside, the elevator has a button for each floor, and on each floor there is a call button.

We want to (formally) describe the operation of the elevator. That is, the algorithm, or control, that dictates its behaviour. Let's start by describing the state of the elevator at a certain point. We represent it using a vector in $S = \{1, \ldots, 8\} \times \{0, 1\}^8$, representing which floor the elevator is on, and which floors have been called.

LIE ALERT!

In the above we assume that the elevator state only relies on the current floor and the pressed floors, and does not store any additional information. But why should this be the case? It could well be that we want to remember whether the elevator was recently going up or down, etc.

At each time step, the algorithm reads which floors are being called, and then moves the elevator to some floor (not necessarily one up or one down, it may send it without stopping to a certain floor). Thus, the inputs for the system are vectors in $I = \{0, 1\}^8$.

We are now ready to model the system: we build a sort of automaton, whose states are S and its alphabet is the set of inputs I. Thus, for example, if we have the transition

$$\delta((3, (0, 0, 0, 1, 0, 0, 0, 1)), (1, 0, 0, 0, 0, 0, 0, 0)) = (4, (1, 0, 0, 0, 0, 0, 0, 1))$$

it means that if the elevator was on floor 3, and floors 4 and 8 were requested, and then a new request for floor 1 arrived, the control would take the elevator to floor 4, erase the 4 request, and put in a request for floor 1.

LIE ALERT!

We now assume that the algorithm for the elevator only relies on the state of the elevator, and nothing more. As we will see, there is a reason to assume something like that (namely the existence of memoryless strategies in parity games), but we're getting ahead of ourselves.

So now we have a full elevator control. What do we do with it? Well, we want to know if it's correct. To do that, we need to formalize *specifications*, and then check whether the control satisfies the specification. This is called *model checking*.

What kinds of specifications would we like to reason about? Here are some examples.

- Whenever floor i is pressed, the elevator eventually reaches it.
- If floor $i < j$ are pressed, and the elevator is at floor $k < i$, then it will go to floor i before floor j.
- Suppose an important person lives at floor 8. We can require that whenever floor 8 is called, the elevator immediately goes there.

In this course we will deal with questions such as:

- How can we formalize such specifications?
- Can we algorithmically check whether they hold?
- Can we algorithmically build a controller that automatically satisfies all the specifications?

△

1.2 Actual Motivation

Automata are perhaps the simplest models of computation one can think of. Nonetheless, they posses two very appealing properties: they are relatively tractable to reason about, and yet they are powerful enough to model many computing systems, or abstractions thereof.

An even more appealing property of automata is that they are fun to study: their underlying mathematical foundation is extremely poor (in that there is no rich theory underneath them, such as calculus, probability theory, group theory, etc.), and yet they give rise to very intricate problems. Since there are not many mathematical foundations to use, all we can use is our brain. Hence – fun!

2 Languages and ω-Languages

alphabet
letters

Let Σ be a finite set, which we call an *alphabet*. We refer to the elements of Σ as *letters*.

Finite words We define

$$\Sigma^\star = \bigcup_{n=0}^{\infty} \Sigma^n$$

words

The elements of $\Sigma^\star$ are sequences of any *finite* length of letters from Σ, and we refer to those as *words*. ϵ is called the *empty word*. For convenience, we don't write words as (a, b, b, b, a), but rather as *abbba*.

language

A *language* over the alphabet Σ is a set $L \subseteq \Sigma^\star$. That is, a set of words.

Infinite words Recall that ω is the first infinite ordinal (sometimes identified with the set of natural numbers $\mathbb{N}$). Then, Σ^ω is the set of infinite words over Σ. The elements of Σ^ω are called *ω-words*, or just words, if the context is clear (you can think of a word as a function $w : \mathbb{N} \to \Sigma$, if you want). An *$\omega$-language* is a subset $L \subseteq \Sigma^\omega$. Again, when the context is clear we will simply refer to them as languages.

ω-words
ω-language

L^ω
$L \cdot K$

For a $*$-language $L \subseteq \Sigma^\star$ and an ω-language $K \subseteq \Sigma^\omega$, let $L^\omega = \{v_1 v_2 \ldots : \forall i \in \mathbb{N}\, v_i \in L\}$ and $L \cdot K = \{v \cdot w : v \in L \text{ and } w \in K\}$.

For a word $w \in \Sigma^\omega$, we often denote its letters as $w = w_0 w_1 \ldots$.

3 Büchi Automata

Our goal is to be able to reason about ω-languages. To do so, we need a suitable computational model – one that defines certain ω-languages. For our purposes, the model of choice is an adaptation of the finite automata model for finite words. Before defining our model of automata, let's recall the model of (nondeterministic) automata over finite words:

NFA

Definition 3.1. *A nondeterministic finite automaton (*NFA*) is a tuple $\mathcal{A} = \langle Q, \Sigma, Q_0, \delta, F\rangle$ where*

- *Q is a finite set of states*

- *Σ is a finite alphabet*
- *Q_0 is a set of initial states*
- *$\delta : Q \times \Sigma \to 2^Q$ is a transition function*
- *$F \subseteq Q$ is a set of accepting states*

We will not recap the semantics of NFAs, but recall that an NFA accepts a word $w \in \Sigma^\star$ if it has a run on w that starts in Q_0 and ends in F.

Remark 3.2. Perhaps you have seen the definition of NFA with a transition relation $R \subseteq Q \times \Sigma \times Q$ instead of $\delta : Q \times \Sigma \to 2^Q$. They are of course interchangeable, and if you're more comfortable with one over the other - by all means use whichever. △

Example 3.3. Consider the DFA in Figure 1. It's language is $\{w \in \{a, b\}^* : w \text{ ends with } a\}$.

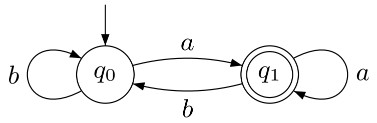


Figure 1: A deterministic automaton

△

3.1 Büchi Automata

When we want to define a model for automata that handle ω-words, we clearly cannot use the phrase "ends in F", as a run does not end. Thus, we need to propose some other *acceptance condition.* As we shall see, there are several interesting acceptance conditions. The most well-studied is called the Büchi acceptance condition, named after the Swiss mathematician Richard Büchi, who proposed and studied them in the 60's [2].

Intuitively, the Büchi acceptance condition dictates that a run is accepting if it visits an accepting state infinitely often.

Example 3.4. Consider the automaton in Figure 1 again, this time treated as a Büchi automaton. It's language in this case is $\{w \in \{a, b\}^\omega : w \text{ has infinitely many } a\text{'s}\}$. △

We now turn to formally define Büchi automata.

NBW **Definition 3.5.** *A nondeterministic Büchi automaton over words (*NBW*) is a tuple $\mathcal{A} = \langle Q, \Sigma, Q_0, \delta, \alpha \rangle$ where*

- *Q is a finite set of states*
- *Σ is a finite alphabet*
- *Q_0 is a set of initial states*
- *$\delta : Q \times \Sigma \to 2^Q$ is a transition function*
- *$\alpha \subseteq Q$ is a set of accepting states*

As you can see, the syntax of an NBW is identical to that of an NFA. In particular, in a *deterministic* `deterministic` Büchi automaton (DBW), we have a deterministic transition function (i.e., $|\delta(q, \sigma)| = 1$ for all q, σ) and a single initial state q_0.

The difference between NBWs and NFAs is of course in the semantics. Intuitively, an NBW accepts a word $w \in \Sigma^\omega$ if it has a run on w that visits α infinitely often. We now formalize this.

`run` Let $w = w_0 w_1 \cdots \in \Sigma^\omega$. A *run* of $\mathcal{A}$ on w is a sequence $\rho = q_0, q_1, \ldots \in Q^\omega$ such that:

- $q_0 \in Q_0$,
- for every $i \geq 0$, $q_{i+1} \in \delta(q_i, w_i)$.

That is, a run of $\mathcal{A}$ on w is an infinite sequence of states that progresses according to the transition function.

inf For a run $\rho = q_0, q_1, \ldots$, let $\inf(\rho) = \{q : \forall i\ \exists j \geq i\ :\ q_j = q\}$. That is, $\inf(\rho)$ is the set of states that repeat infinitely often in ρ. Observe that since Q is finite, we have that $\inf(\rho) \neq \emptyset$.

accepting run In the Büchi acceptance condition, we say that the run ρ is *accepting* if $\inf(\rho) \cap \alpha \neq \emptyset$. That is, if ρ visits α infinitely often.

language We say that $\mathcal{A}$ *accepts* w if $\mathcal{A}$ has an accepting run on w. Finally, we define the *language* of $\mathcal{A}$ to be $L(\mathcal{A}) = \{w \in \Sigma^\omega\ :\ \mathcal{A} \text{ accepts } w\}$.

3.2 Closure Properties (excluding complementation)

We now turn to study closure properties of NBWs. Perhaps the simplest closure property for nondeterministic automata is union:

Theorem 3.6. *Let $\mathcal{A}, \mathcal{B}$ be two* NBWs*, then there exists an* NBW *$\mathcal{C}$ such that $L(\mathcal{C}) = L(\mathcal{A}) \cup L(\mathcal{B})$.*

Proof sketch. Basically, put the NBWs "side by side". □

The next natural property to reason about is intersection. One way to prove closure under intersection is to prove closure under complementation, and then use de-Morgan rules. As we shall see, complementation poses a significant challenge, and therefore we tackle intersection directly first.

For NFAs, closure under intersection is typically shown using the *product construction.* We start by showing that naively, this does not work for NBWs.

Example 3.7. Consider the NBW $\mathcal{A}$ in Figure 1, and let $\mathcal{B}$ be a similar NBW with q_0 accepting instead of q_1.

What are $L(\mathcal{A})$ and $L(\mathcal{B})$? What happens when we take their product construction? △

The problem demonstrated in Example 3.7 is that the two automata may visit their accepting states in different times, and we need some way to keep track of that. We do this using an "indexed" product construction.

Theorem 3.8. *Let $\mathcal{A}, \mathcal{B}$ be two* NBWs*, then there exists an* NBW *$\mathcal{C}$ such that $L(\mathcal{C}) = L(\mathcal{A}) \cap L(\mathcal{B})$.*

Proof. Let $\mathcal{A} = \langle Q^1, \Sigma, Q_0^1, \delta^1, \alpha^1 \rangle$ and $\mathcal{B} = \langle Q^2, \Sigma, Q_0^2, \delta^2, \alpha^2 \rangle$, we construct a NBW $\mathcal{C} = \langle Q^1 \times Q^2 \times \{0,1\}, \Sigma, Q_0^1 \times Q_0^2 \times \{0\}, \delta', \alpha^1 \times Q^2 \times \{0\} \rangle$, where the transition function is as follows. Let $q_1 \in Q^1$ and $q_2 \in Q^2$

$$\delta((q_1, q_2, b), \sigma) = \begin{cases} \delta(q_1, \sigma) \times \delta(q_2, \sigma) \times \{0\} & b = 0 \wedge q_1 \notin \alpha^1 \\ \delta(q_1, \sigma) \times \delta(q_2, \sigma) \times \{1\} & b = 0 \wedge q_1 \in \alpha^1 \\ \delta(q_1, \sigma) \times \delta(q_2, \sigma) \times \{1\} & b = 1 \wedge q_2 \notin \alpha^2 \\ \delta(q_1, \sigma) \times \delta(q_2, \sigma) \times \{0\} & b = 1 \wedge q_2 \in \alpha^2 \end{cases}$$

Intuitively, we hold two copies of the product automaton, and we jump between them whenever we get to an accepting state of the corresponding automaton.

To prove correctness, consider a word $w \in \Sigma^\omega$ and runs of $\mathcal{A}$ and $\mathcal{B}$ on w. Observe that by induction, it is easy to prove that there is a run of $\mathcal{C}$ on w that is in state (q_1, q_2, b) after reading a prefix x of w iff the respective runs of $\mathcal{A}$ and $\mathcal{B}$ are in states q_1 and q_2 after reading x.

Next, observe that $\mathcal{C}$ stays in copy 0 until α_1 is seen in the first component, and moves to copy 1 otherwise, and similarly for copy 1 and α_2. It thus follows that there exist accepting runs of both $\mathcal{A}$ and $\mathcal{B}$ on w iff there exist such runs where both $\mathcal{A}$ and $\mathcal{B}$ visit their accepting states infinitely often, and this happens iff the "product run" in $\mathcal{C}$ visits $\alpha^1 \times Q^2 \times \{0\}$ infinitely often.

Thus, $\mathcal{C}$ accepts w iff both $\mathcal{A}$ and $\mathcal{B}$ accept w, and we are done. □

The next properties are somewhat more esoteric, but will turn out to be important for both complementation (Section 3.4) and for ω-regular expressions (which you will see in Exercise 1).

Theorem 3.9. *Let $\mathcal{A}$ be an* NFA*, then there exists an* NBW *$\mathcal{B}$ such that $L(\mathcal{B}) = L(\mathcal{A})^\omega$.*

Proof. The intuitive idea is that whenever $\mathcal{A}$ reaches an accepting state, $\mathcal{B}$ can "reset" the run back to Q_0, in which case we also mark a visit to an accepting state. There is a small technical nuisance here, which is that the "acceptance" mark should conceptually be on a transition, instead of a state. Since this is not allowed in our model, we need a small workaround, namely copying the initial states (or the accepting states).

Formally, we proceed as follows. Let $\mathcal{A} = \langle Q, \Sigma, \delta, Q_0, F\rangle$. We define the NBW $\mathcal{B} = \langle Q', \Sigma, \delta', Q_0, \alpha\rangle$ where:

- $Q' = Q \cup (Q_0 \times \{1\})$ where $Q_0 \times \{1\}$ is a fresh copy of Q_0.
- $\alpha = Q_0 \times \{1\}$ (the new copy of Q_0 are the accepting states).
- δ' is defined as follows: for every $q \in Q$ and $\sigma \in \Sigma$, let $S = \delta(q, \sigma)$, then

  $$\delta'(q,\sigma) = \begin{cases} S & S \cap F = \emptyset \\ S \cup (Q_0 \times \{1\}) & S \cap F \neq \emptyset \end{cases}$$

  and for every $q_0 \in Q_0$ we have $\delta'((q_0, 1), \sigma) = \delta(q_0, \sigma)$.

  Thus, $\mathcal{B}$ acts similarly to $\mathcal{A}$, but whenever it is possible to reach an accepting state, $\mathcal{B}$ sends a run to start again from Q_0, and marks a visit to an accepting state. Thus, $\mathcal{B}$ accepts a word w iff w can be written as $u_1 \cdot u_2 \cdots$ where $u_i \in L(\mathcal{A})$ for every i.

□

Theorem 3.10. *Let $\mathcal{A}$ be an* NFA *and let $\mathcal{B}$ be an* NBW*, then there exists an* NBW *$\mathcal{C}$ such that $L(\mathcal{C}) = L(\mathcal{A}) \cdot L(\mathcal{B})$.*

Proof Sketch. A bit similarly to Theorem 3.9, the idea is to add a transition from the accepting states of $\mathcal{A}$ to the initial states of $\mathcal{B}$ (but here we don't need to duplicate any state). □

We conclude this section by giving a name for the class of languages defined by regular expressions:

ω-regular **Definition 3.11.** *A language $L \subseteq \Sigma^\omega$ is* ω-regular *if it is recognizable by an* NBW*.*

3.3 Decision Problems for NBWs (part I)

In this section we study decision problems for NBWs. The first problem we consider is *emptiness.*

NBW Emptiness:

Given: NBW $\mathcal{A}$.

Question: Is $L(\mathcal{A}) = \emptyset$?

Theorem 3.12. NBW *emptiness is* **NL-Complete***.*

Proof. We'll start with the upper bound. Let $\mathcal{A} = \langle Q, \Sigma, \delta, Q_0, \alpha\rangle$. How can we decide whether $\mathcal{A}$ accepts some word? Well, it's clearly enough to determine whether $\mathcal{A}$ has some accepting *run.* Now, let's observe what an accepting run looks like: it first needs to reach α from Q_0, and then needs to reach α infinitely often. Since α is finite, we have that an accepting run in fact reaches some state $q \in \alpha$ infinitely often.

Thus, it is enough to decide the following: are there states $q_0 \in Q_0$ and $q \in \alpha$ such that q is reachable (in the underlying graph of $\mathcal{A}$) from q_0, and from itself?

We now add another simple observation: if a state q is reachable from another state s, then it is also reachable from s using a *simple* path (i.e., without loops).

Finally, note that the length of a simple path is at most $|Q|$.

So an algorithm for deciding non-emptiness in **NL** is the following: first, nondeterministically "guess" $q_0 \in Q_0$ and $q \in \alpha$. Then, guess "step by step" paths of length at most $|Q|$ from q_0 to q and then from q to q. If such paths are found, the language is non-empty, so accept.

Implementing this in logarithmic space is standard, by keeping on the tape only the current states, and the guesses for q_0 and q, as well as a step counter with $\lceil \log |Q| \rceil$ bits.

We now turn to show hardness. The canonical **NL-Complete** problem is directed graph connectivity:

$$STCONN = \{\langle G,s,t\rangle : G=(V,E) \text{ is a directed graph } s,t\in V \text{and there is a path from } s \text{ to } t \text{ in } G\}$$

A reduction from $STCONN$ to NBW **non**-emptiness is almost trivial: given $\langle G,s,t\rangle$ where $G=\langle V,E\rangle$, we define an NBW $\mathcal{B}=\langle Q,\Sigma,\delta,Q_0,\alpha\rangle$ where $Q=V$, $\Sigma=\{a\}$ is singleton. $Q_0=\{s\}$, $\alpha=\{t\}$, and δ is induced by the edges, with the addition of self-loops: for every $v\in V$, $\delta(v,a)=\{u\in V : (v,u)\in E\}\cup\{v\}$.

It is easy to see that the reduction can be implemented in logspace.

We now prove correctness. If there is a path from s to t in G, then t is reachable from s in $\mathcal{B}$, and it is reachable from itself via the self loop, so $\mathcal{B}$ accepts the word a^ω, and its language is non-empty.

Conversely, if $L(\mathcal{B})\neq\emptyset$, then the single accepting state t must be reachable from the single initial state s, so t is reachable from s also in G, and we are done.

Notice that since **NL** = **coNL** (by Immerman's theorem), then NBW emptiness is also **NL-Complete**.

□

There are two other natural decision problems for NBWs, namely *universality* and *containment*:

NBW Universality:

Given:	NBW $\mathcal{A}$.
Question:	Is $L(\mathcal{A})=\Sigma^\omega$?

NBW Containment:

Given:	NBWs $\mathcal{A},\mathcal{B}$.
Question:	Is $L(\mathcal{A})\subseteq L(\mathcal{B})$?

For NFAs, solving universality and containment is done using complementation. Since we do not know whether Büchi automata can be complemented yet, both NBW universality and containment seem a bit out of reach at the moment.

3.4 Complementation of NBWs

How can we complement NBWs? Let's try to draw some intuition from the case of finite words. In order to complement an NFA, we determinize it to a DFA, and then complement by flipping the accepting states.

Unfortunately, this approach breaks down for NBWs (in both steps), as we shall now show.

We start by showing that NBWs are not determinizable.

Theorem 3.13 (Landweber '69 [12]). *There exists an ω-language L that is recognizable by an* NBW*, but not by a* DBW.

Proof. Consider the language $L=\{w\in\{0,1\}^\omega : w \text{ has finitely many 1's}\}$. Observe that L is recognizable by the NBW in Figure 2.

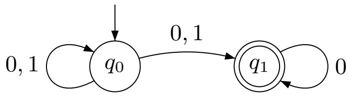


Figure 2: NBW for "finitely many 1's."

We claim that L cannot be recognized by a DBW. To this end, assume by way of contradiction that $\mathcal{A}=\langle Q,\{0,1\},\delta,\alpha\rangle$ is a DBW with $L(\mathcal{A})=L$, and let $n=|Q|$.

We're going to derive a contradiction using a sort of pumping argument, as follows.

Consider the word $0^\omega\in L$. The (unique) run of $\mathcal{A}$ on it is accepting, and thus visits α infinitely often. Let m_1 be an index such that the run of $\mathcal{A}$ on 0^{m_1} reaches α.

Now consider the word $0^{m_1}10^\omega\in L$. Following the same reasoning, we can find m_2 such that the run of $\mathcal{A}$ on $0^{m_1}10^{m_2}$ reaches α.

Proceeding in this fashion (or, formally, by induction) we can construct a sequence $m_1, m_2, m_3, \ldots$... such that the run of $\mathcal{A}$ on $0^{m_1}10^{m_2}1\cdots10^{m_k}$ reaches α for every k. Note that all the runs we considered are prefixes of each other, and that after every prefix of the form above, the run reaches α.

We can now complete the proof in two ways:

The "mathematically-confident" approach is to consider the infinite word $0^{m_1}10^{m_2}1\cdots$ obtained from our construction, and observe that the run of $\mathcal{A}$ on it visits α infinitely often, which is a contradiction.

The "safe" approach, for those who shy away from infinite constructions (as well they should), is the following: recall that $\mathcal{A}$ has n states. Consider the first $n+1$ prefixes $0^{m_1}10^{m_2}1\cdots10^{m_k}$, $k \in \{1,\ldots,n+1\}$. For every k, the run of $\mathcal{A}$ on the prefix ends in α. Thus, there exist $k_1 < k_2$ such that both $0^{m_1}10^{m_2}1\cdots10^{m_{k_1}}$ and $0^{m_1}10^{m_2}1\cdots10^{m_{k_2}}$ end in the same (accepting) state q.

We can now "pump" this accepting cycle infinitely. That is, the run of $\mathcal{A}$ on the word

$$0^{m_1}10^{m_2}1\cdots10^{m_{k_1}}(10^{m_{k_1+1}}\cdots10^{m_{k_2-1}})^\omega$$

reaches and stays in an accepting cycle. But this word has infinitely many 1's, so we have reached a contradiction.

□

In addition, it is not hard to see that swapping the accepting and non-accepting states in a DBW does not result in the complement of its language.

As it turns out, complementing NBWs is highly nontrivial. Büchi was the first to give a proof that they are closed under complementation [2]. Our proof below follows that of Thomas [27]. We start with some definitions.

Consider an NBW $\mathcal{A} = \langle Q, \Sigma, \delta, Q_0, \alpha\rangle$. Let $w \in \Sigma^\star$ be a *finite* word, and consider two states $q, q' \in Q$. There are three possible "fates" for the (partial) runs of $\mathcal{A}$ on w:

1. There is a run of $\mathcal{A}$ on w from q to q' that does not go through α (denoted $q \xrightarrow{w} q'$).
2. There is a run of $\mathcal{A}$ on w from q to q' that passes through α (denoted $q \xrightarrow{w}_\alpha q'$).
3. There is no run of $\mathcal{A}$ on w from q to q' (denoted $q \overset{w}{\not\longrightarrow} q'$).

Note that cases 1 and 2 are not mutually exclusive.[1]

type We now define the *type* of w to be an edge-colored bipartite directed (from left to right) graph whose vertices are two copies of Q, with the following edges: for vertices q, q' on the left and right, respectively, there is a red edge (q, q') if $q \xrightarrow{w} q'$, there is a blue edge (q, q') if $q \xrightarrow{w}_\alpha q'$, and otherwise there is no edge from q to q'.

Example 3.14. Consider the NBW in Figure 3. Let's examine the type of the word aba in this NBW.

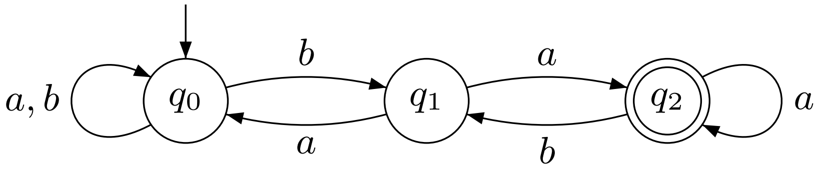


Figure 3: An NBW.

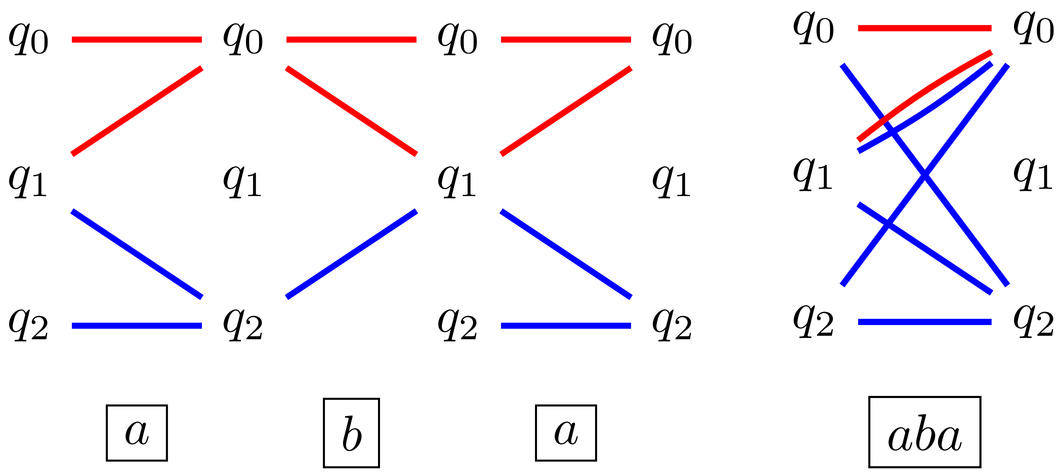
To do so, we can first examine the type of each letter, as depicted in Figure 4(a), and then concatenate the graphs, to get the resulting type in Figure 4(b).

△

We now make a simple observation:

Observation 3.15. There are finitely many different types for a given NBW $\mathcal{A}$.

[1] We could have defined condition 1 to also require that there is no run that goes through α. This makes some parts cleaner and some parts messier, but does not effect the general lines of the proof.



(a) The types of a, b, a, concatenated

(b) The type of aba

Figure 4: Finding the type of aba.

type graph

Given a finite sequence of types, we can concatenate them in the natural way to obtain a type graph. Similarly, for an infinite word w, decomposed as $w = u_0u_1\cdots$, its *type graph* is the infinite graph obtained by concatenating the types of the sequence $u_0, u_1, \ldots$. Note that for infinite concatenations, the type graph is no longer bipartite, but rather consists of an infinite concatenation of bipartite graphs.

We now formalize how type graphs are connected to membership in the language.

Lemma 3.16. *Consider a word $w \in \Sigma^*$ and a decomposition $w = u_0u_1\cdots$ with $u_i \in \Sigma^*$ for all $i \geq 0$. Let τ_i be the type of u_i, and $\tau = \tau_0\tau_1\cdots$ the corresponding type graph.*

We have that $w \in L(\mathcal{A})$ iff τ does not have a one-way path that begins in an initial state and passes infinitely often through blue edges.

Proof. This lemma is almost trivial, and is more of an observation.

Assume the right-hand condition holds, and consider a run of $\mathcal{A}$ on w. This run induces a one-way path through the graph τ from an initial state, and therefore does not pass through infinitely many blue edges. Thus, this run only passes through finitely many accepting state, so it is not accepting. Since this holds for every run, we have $w \notin L(\mathcal{A})$.

Conversely, assume $w \notin L(\mathcal{A})$, and consider an infinite path from an initial state in τ. Such a path again induces a run (actually, several runs) of $\mathcal{A}$ on w, and the corresponding runs are not accepting, and therefore visit α only finitely often. Thus, this path has only finitely many blue edges. □

Observe that in particular, the lemma shows that the condition on the type graph holds regardless of the decomposition $w = u_0u_1\cdots$, which naturally makes sense.

The next step in the proof is to invoke Ramsey theory. For a set S, write $(S)_n = \{S' \subseteq S : |S'| = n\}$ (i.e. the set of subsets of S of size n). We will use the following theorem:

Theorem 3.17 (Ramsey, 1930)**.** *Let $f : (\mathbb{N})_n \to \{1, \ldots k\}$, then there exists an infinite set $S \subseteq \mathbb{N}$ on which f is homogeneous, i.e., f is constant on $(S)_n$.*

We now use Theorem 3.17 to find special "lasso-shaped" type graphs for every word. This observation is at the heart of Büchi's work.

Lemma 3.18. *For every word $w \in \Sigma^\omega$ there exists types τ_0, τ such that w can be written as $w = u_0u_1\cdots$ where the type of u_0 is τ_0, and for every $i \geq 1$ the type of u_i is τ.*

Thus, the type graph of w is $\tau_0\tau^\omega$.

Proof. Let $w \in \Sigma^\omega$, and let T be the (finite) set of types in $\mathcal{A}$. We define a function $f : (\mathbb{N})_2 \to T$ that maps a set $\{i, j\} \subseteq \mathbb{N}$ with $i < j$ to the type of the infix $w_i \ldots w_{j-1}$.

By Theorem 3.17, there exists an infinite set $H = \{j_1, j_2, \ldots\}$, with $j_1 < j_2 < \ldots$ such that f is homogeneous on H.

Denote by τ the type such that for all $j, j' \in H$, $f(j, j') = \tau$, and denote by τ_0 the type of the prefix $w_0 \cdots w_{j_1 - 1}$.

Set $u_0 = w_0 \cdots w_{j_1-1}$ and for all $i \geq 1$ set $u_i = w_{j_i} \cdots w_{j_{i+1}-1}$, then the condition clearly holds. □

Remark 3.19. Note that the proof of Lemma 3.18 shows that in fact, not only does every u_i has the same type τ, but in fact $u_i \cdot u_{i+1}$ also has the same type. In other words, the type τ is *idempotent* under concatenation of types. △

Now, consider a type τ. We define the language

$$L_\tau = \{w \in \Sigma^+ \,:\, w \text{ has type } \tau\}.$$

Next, let τ_0, τ be two types (c.f. Lemma 3.18), we define the language

$$L_{\tau_0,\tau} = L_{\tau_0} \cdot L_\tau^\omega = \{w \in \Sigma^\omega \,:\, w \text{ has a decomposition with types } \tau_0 \tau^\omega\}.$$

We now come to the main characterization of the complement of $L(\mathcal{A})$.

Lemma 3.20. *Let* $B = \{(\tau_0, \tau) \,:\, L_{\tau_0,\tau} \cap L(\mathcal{A}) = \emptyset)\}$*. Then* $\overline{L(\mathcal{A})} = \bigcup_{(\tau_0,\tau)\in B} L_{\tau_0,\tau}$*.*

Proof. One direction is trivial: if $w \in \bigcup_{(\tau_0,\tau)\in B} L_{\tau_0,\tau}$, then $w \notin L(\mathcal{A})$ (since each set in the union is disjoint from $L(\mathcal{A})$).

For the converse, let $w \notin L(\mathcal{A})$. By Lemma 3.18, there are τ_0, τ such that $w \in L_{\tau_0,\tau}$. We claim that $L_{\tau_0,\tau} \cap L(\mathcal{A}) = \emptyset$.

Assume by way of contradiction that there exists some $w' \in L_{\tau_0,\tau} \cap L(\mathcal{A})$. Then, the type graph of w' that corresponds to the decomposition into types $\tau_0\tau^\omega$ has a path from an initial state that has infinitely many blue edges. However, this path also exists in the type graph of w (as it is identical), which contradicts Lemma 3.18. Thus, we are done. □

We are now done giving a characterization of $\overline{L(\mathcal{A})}$ using languages (defined by types). We now turn to show that we can actually construct an NBW that corresponds to this characterization. We start by showing that all the languages defined above are in fact ω-regular (or regular, for finite words).

Proposition 3.21. *The language* L_τ *is* $*$*-regular, and can be effectively obtained from* $\mathcal{A}$*.*

Proof Sketch. Recall that $\mathcal{A} = \langle Q, \Sigma, \delta, Q_0, \alpha\rangle$. We wish to construct an NFA $\mathcal{N}$ for L_τ. The idea is as follows: when reading a word $w \in \Sigma^+$, $\mathcal{N}$ needs to verify that for every pair of states $q, q' \in Q$, the runs of $\mathcal{A}$ between q and q' admit edges of the correct color prescribed by τ. Consider such states q, q'. Suppose that in τ there is a red edge from q to q', then $\mathcal{N}$ must check that there exists a run from q to q' that does not go through an accepting state of $\mathcal{A}$. We do this as follows: take a copy of $\mathcal{A}$ where q is the initial state, q' is the only accepting state, and remove all states in α from Q. The obtained NFA accepts w iff there exists a run from q to q' that does not go through an accepting state.

Now, if there is a blue edge from q to q' in τ, we construct an NFA that comprises two copies of $\mathcal{A}$: first, we run $\mathcal{A}$ from q until an accepting state is seen, and then we continue running in a new copy until we reach q'. It is not hard to see why this captures a blue edge.

We can then describe the properties "no red edge" and "no blue edge" by complementing as needed.

Finally, we take the conjunction over all q, q', to obtain the overall NFA. □

Exercise 3.22. What is the (worst case) size of $\mathcal{N}$ described above?

By the closure of ω-regular languages under concatenation and under ω (Theorems 3.9 and 3.10) we have that the language

$$L_{\tau_0,\tau} = L_{\tau_0} \cdot L_\tau^\omega = \{w \in \Sigma^\omega \,:\, w \text{ has a decomposition with types } \tau_0 \tau^\omega\}$$

is Büchi recognizable.

Finally, we are ready to show the main theorem.

Theorem 3.23. *Let* $\mathcal{A}$ *be an* NBW*, we can effectively*[2] *compute an* NBW $\mathcal{B}$ *such that* $L(\mathcal{B}) = \overline{L(\mathcal{A})}$*.*

[2] "Effectively" means that the construction is computable (not to be confused with "efficiently").

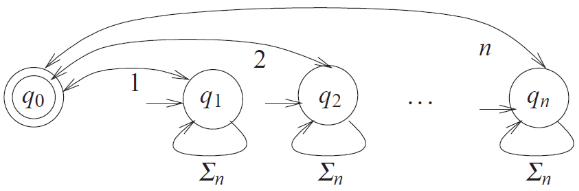


Figure 5: An NBW for L_n in Theorem 3.26.

Proof. Given $\mathcal{A}$, we start by constructing the NBWs $\mathcal{A}_{\tau_0,\tau}$ for every pair of types τ_0, τ of $\mathcal{A}$ (using Proposition 3.21 and Theorems 3.9 and 3.10). Then, for each such NBW, we check whether $L(\mathcal{A}_{\tau_0,\tau}) \cap L(\mathcal{A}) = \emptyset$, using Theorem 3.8, and the non-emptiness algorithm described in Theorem 3.12.

Finally, using Theorem 3.6, we construct an NBW for $\bigcup_{(\tau_0,\tau)\in B} L_{\tau_0,\tau}$, which by Lemma 3.20 is exactly $\overline{L(\mathcal{A})}$. □

Observe that the proof of Theorem 3.23 is constructive – all the steps involve constructions and algorithms we know how to implement.

Finally, let's give a bound on the size of the complement of an NBW:

Theorem 3.24. *Let $\mathcal{A}$ be an* NBW *with n states, then there exists an* NBW $\overline{\mathcal{A}}$ *for the complement language with $2^{O(n^2)}$ states.*

Proof. Exercise. □

Remark 3.25. The construction in Theorem 3.24 is not optimal. In fact, one can achieve a $2^{O(n \log n)}$ construction (see e.g., [11]). △

Theorem 3.24 shows that NBWs are closed under complementation, but gives a horrible blowup in terms of the size of the resulting complement NBW. A natural question is whether we can do better, or whether there is a lower bound.

Theorem 3.26 ([17])**.** *There is a family*[3] *of languages $L_1, L_2, \ldots$ such that $L_n \subseteq \Sigma_n^\omega$ can be recognized by an* NBW *with $n+1$ states, but every* NBW *for $\overline{L_n}$ requires at least $n!$ states.*

Proof. As with many lower bounds, the proof involves a bit of "magic". It can be frustrating to read, though, so for a first reading, I recommend embracing the magic, and following the proof steps without asking (too much) *why* and *how* it works. It just does.

We'll start by defining the family of languages we work with. For $n \in \mathbb{N}$, let $\Sigma_n = \{1, \ldots, n, \#\}$. Consider a word $w \in \Sigma_n^\omega$. We associate with w a directed graph $G_w = \langle V, E \rangle$ where $V = \{1, \ldots, n\}$ and $(i, j) \in E$ iff ij appears infinitely often in w.

For example, the word $21(123)^\omega \in \Sigma_3^\omega$ defines a directed triangle, with edges $(1, 2), (2, 3)$, and $(3, 1)$. Now, we define

$$L_n = \{w \in \Sigma_n^\omega : G_w \text{ contains a cycle}\}$$

Isn't that weird? Remember, it's magic.

We now present two claims: first, that L_n is recognizable by an $n+1$ state NBW, and second, that every NBW for $\overline{L_n}$ has at least $n!$ states. Naturally, we'll start with the easy part, showing an NBW for L_n.

Consider the NBW $\mathcal{A}_n$ in Figure 5. Clearly $\mathcal{A}_n$ has $n+1$ states. Now, it is not very difficult to prove that $L(\mathcal{A}_n) = L_n$, but we're not going to do that. The reason is that in order to prove the theorem, it's actually enough to *define* L_n as $L(\mathcal{A}_n)$, and regard the whole graph idea as an intuition, which we do henceforth.

So all that remains is to prove that any NBW for $\overline{L(\mathcal{A}_n)}$ has at least $n!$ states. Let $\mathcal{B}_n$ be such an NBW. How shall we get the $n!$ bound? by using permutations, of course. Let $\pi = (\pi_1, \ldots, \pi_n)$ be a permutation of $\{1 \ldots, n\}$ (i.e., $\pi(i) = \pi_i$), and consider the word $w_\pi = (\pi_1 \cdots \pi_n \cdot \#)^\omega$. Note that $w_\pi \notin L_n$. Indeed, in every run of $\mathcal{A}_n$ on w, after every visit to q_0, we must either see # (and there are

[3] If you're not sure why we need a family, and not a single language, please ponder it, and come see me.

no runs), or we proceed to the state corresponding to the next letter of the permutation. That is, from state q_i, a transition to q_0 can only be taken with letter i, and therefore must be followed by a transition on letter j, where j "follows" i in π (formally, $j = \pi(\pi^{-1}(i) + 1)$). Therefore, eventually the run reaches q_k for $k = \pi(n)$, which we cannot leave, since the only occurrences of k in the remainder of the word are followed by #, and # cannot be read from q_0. Thus, $w_\pi \in L(\mathcal{B}_n)$.

Now, consider two permutations $\pi \neq \tau$, and let S_π and S_τ be the sets of states that $\mathcal{B}_n$ visits infinitely often during some accepting runs r_π and r_τ on π and τ, respectively. We claim that $S_\pi \cap S_\tau = \emptyset$. Observe that this would allow us to conclude the claim, since for every permutation we would need at least one unique state.

Assume by way of contradiction that $S_\pi \cap S_\tau \neq \emptyset$, and let $q \in S_\pi \cap S_\tau$. Thus, we can find three finite words $h, u, v \in (\Sigma^n)^*$ with the following properties:

- h is a prefix of w_π, with which r_π moves from an initial state to q.
- u is an infix of w_π that contains π, with which r_π moves from q back to q, while visiting an accepting state of $\mathcal{B}_n$.
- v is an infix of w_τ that contains τ, with which r_τ moves from q back to q, while visiting an accepting state of $\mathcal{B}_n$.

Since r_π and r_τ are accepting, these words exist. Now, it is easy to see that $\mathcal{B}_n$ accepts $h(uv)^\omega$. We show, however, that $h(uv)^\omega \in L(\mathcal{A}_n)$, thus reaching a contradiction.

Write $\pi = (\pi_1, \ldots, \pi_n)$ and $\tau = (\tau_1, \ldots, \tau_n)$. Let j be the minimal index such that $\pi_j \neq \tau_j$. Where do π and τ send j? They cannot assign it to something that was previously assigned, so there must exist $j < k, l \leq n$ (possibly $k = l$) such that $\pi_j = \tau_k$ and $\tau_j = \pi_l$ (make sure you understand this point, it's slightly confusing).

Now, since u contains π and v contains τ, then $(uv)^\omega$ contains the pairs

$$\pi_j\pi_{j+1}, \pi_{j+1}\pi_{j+2}, \ldots, \pi_{l-1}\pi_l, \underbrace{\pi_l\tau_{j+1}}_{=\tau_j\tau_{j+1}} \tau_{j+1}\tau_{j+2}, \ldots \underbrace{\tau_k\pi_{j+1}}_{=\pi_j\pi_{j+1}}$$

and these pairs appear infinitely often. Thus, the graph defined by $(uv)^\omega$ has a cycle, so $\mathcal{A}_n$ accepts $h(uv)^\omega$, and we are done. □

To read more on Büchi complementation, and matching the lower and upper bounds, see [24, 10].

In contrast with NBW complementation, if we merely want to complement a DBW to an NBW, things are much easier:

Exercise 3.27. Let $\mathcal{A}$ be a DBW with n states. Show that there exists an NBW $\mathcal{B}$ such that $L(\mathcal{B}) = \overline{\mathcal{A}}$ and $\mathcal{B}$ has $O(n)$ states.

3.5 Decision Problems for NBWs (part II)

We are now fully equipped to tackle some decision problems for NBWs. Let's start with universality.

Theorem 3.28. *The universality problem for* NBWs *is* **PSPACE-Complete**

Proof. The proof has two parts, showing membership in **PSPACE**, and showing **PSPACE** hardness. Before proceeding, we recall that by Savitch's Theorem, we have **NPSPACE** = **PSPACE** = **coPSPACE** = **coNPSPACE**. Thus, in order to prove that a problem is in **PSPACE**, it suffices to prove that it's *complement* is in **NPSPACE**. Often, this is hugely helpful.

The complement of the universality problem, called the *non-universality* problem, asks whether $L(\mathcal{A}) \neq \Sigma^\omega$. We work with non-universality to establish membership in **PSPACE**.

Non-universality is in NPSPACE Consider an NBW $\mathcal{A}$. Clearly $L(\mathcal{A}) \neq \Sigma^\omega$ iff $\overline{L(\mathcal{A})} \neq \emptyset$, and the latter holds iff $L(\mathcal{A}') \neq \emptyset$, where $\mathcal{A}'$ is an NBW that complements $\mathcal{A}$. Thus, it is enough to decide whether $\mathcal{A}'$ accepts some word. Recall from Theorem 3.12 that $\mathcal{A}'$ accepts some word iff there exists a self-reachable accepting state that is reachable from an initial state. Now, if we had $\mathcal{A}'$ written down, then this is easy to check. Unfortunately, we don't. Moreover, computing it entirely may require exponential space.

There are two ways to overcome this. The standard approach uses *on-they-fly* construction, and is useful in many scenarios. We can also directly show an algorithm. In the following, we take both approaches.

Construction On-The-Fly: The trick to overcome this is to construct $\mathcal{A}'$ *on-the-fly*, and guess the lasso through it. That is, we first guess the initial state, then we compute the possible transitions from the initial state, and nondeterministically guess a successor. We proceed in this fashion for up to the size of $\mathcal{A}'$, namely $2^{O(n^2)}$, and observe that we can store a counter for up to that value in polynomial space.

LIE ALERT!

This on-the-fly construction requires careful examining, and is not completely trivial. This is in part due to the proof of complementation we used. There are other complementation constructions that are more clearly doable on-the-fly (see [10]).

Thus, non-universality is in **NPSPACE**, and by Savitch's Theorem, universality is in **PSPACE**.

Direct Algorithm: We show how to check whether $L(\mathcal{A}') \neq \emptyset$ in **NPSPACE**. Recall that by Theorem 3.24, we can assume that $\mathcal{A}'$ has at most $2^{O(n^2)}$ states. Thus, if there is some word that is accepted by $\mathcal{A}'$, there is also such a lasso word $w = uv^\omega$ where $|u|, |v| \leq 2^{O(n^2)}$.

Furthermore, by Lemmas 3.16 and 3.18, we know that there is such a word w iff w has a type-graph of the form $\tau_0\tau^\omega$ that does not have a directed path from an initial state that goes through infinitely many blue edges. Also, from Remark 3.19, we can assume that τ is idempotent.

However, it is not necessarily the case that τ_0 and τ are the types of u and v, respectively. That is, the type of v might not be idempotent. In order to circumvent this problem, we slightly modify u and v, as follows: observe that since there are only 4^{n^2} types, then if we consider the words $v, v^2, v^3, \ldots, v^{4n^2+1}$, there must exist $i < j \leq 4n^2 + 1$ for which the types of v^i and v^j coincide. Thus, $\tau_i \cdot \tau_{j-i} = \tau_j = \tau_i$, where e.g., τ_i is the type of v^i, etc. But then we get that $\tau_i \cdot \tau_{j-i}^k = \tau_i$ for all $k \geq 1$. Strictly speaking, this does not mean that τ_{j-i} is idempotent, since τ_i might cause this equation to hold. But it does mean that τ_{j-i} is idempotent when concatenated to τ_i. For our purposes, this is enough.

We can now write $u' = u \cdot v^i$ and $v' = v^{j-i}$, and we have that $u'v'^\omega$ is a lasso word with an "idempotent enough" type graph. Also, observe that $|u'|, |v'| \leq 2^{O(n^2)}$. We therefore assume in the following that u, v already satisfy the conditions above, with τ and τ_0 being the types of u, v, and τ idempotent.

So all we need to do is nondeterministically guess u and v (letter by letter, of course), and keep track of their types. Once we've guessed them, we have the two types τ_0 and τ, and it remains to check that $\tau_0\tau^\omega$ does not have a path with infinitely many blue edges.

To check that, we use the idempotency of τ again: observe that if there is a path with infinitely many blue edges, then it must traverse one of those blue edges infinitely often, denote it (q_i, q_j). but then, it also needs to go from q_j back to q_i, so there is also a path from q_j to q_i (path in the type-graph!), and hence there is also a path from q_i back to itself, that goes through a blue edge. But since τ is idempotent[4], then there must also be an *edge* (q_i, q_i), and moreover - a blue edge. It follows that τ^ω induces an infinite path with infinitely many blue edges iff it has a blue edge from a state to itself.

Thus, it's enough to check that for every state q reachable by τ_0 from Q_0, the type graph of τ does not have a blue edge (q, q).

All of this can clearly be done in polynomial space. So non-universality is in **NPSPACE**, and by Savitch's Theorem, universality is in **PSPACE**.

Universality is PSPACE-Hard There are two approaches we can take here. The first is to give a proper reduction from the generic word problem for space-bounded TMs. This is taken in [10], and I recommend reading it.

We'll take a different approach, and reduce from the following problem:

$$ALL_{\text{NFA}} = \{\langle \mathcal{N} \rangle : \mathcal{N} \text{ is an NFA and } L(\mathcal{N}) = \Sigma^\star\}$$

[4] Basically, idempotency means that we take the edge closure, and thus each path is represented by an edge.

which is well-known to be **PSPACE**-complete.

The reduction is simple, but not completely trivial. Given $\mathcal{N} = \langle Q, \Sigma, \delta, Q_0, F\rangle$, we construct an NBW $\mathcal{A}$. For didactic purposes, let's start with an incorrect (but almost correct) attempt:

Incorrect attempt: We want to somehow augment $\mathcal{N}$ to work on infinite words. However, there are intricate behaviours that use the alphabet of $\mathcal{N}$, so adding and removing transitions is dangerous. Instead, we introduce a new letter # and a new state $q_\#$ that allow us to "leave" $\mathcal{N}$, and define $\mathcal{A} = \langle Q', \Sigma', \delta', Q_0, \{q_\#\}\rangle$ with the following:

- The states are $Q' = Q \cup \{q_\#\}$.
- The alphabet is $\Sigma' = \Sigma \cup \{\#\}$.
- The initial states remain Q_0.
- The single accepting state is $\{q_\#\}$ (but don't worry, we retain the information on F in the transition function).
- The transition function is defined as follows: for $q \in Q$ and $\sigma \in \Sigma$ we have $\delta'(q, \sigma) = \delta(q, \sigma)$. In addition, if $q \in F$ then $\delta'(q, \#) = \{q_\#\}$ (and otherwise $\delta'(q, \#) = \emptyset$). Finally, $\delta'(q_\#, \sigma) = \{q_\#\}$ for every $\sigma \in \Sigma'$.

What does this construction give us? Well, if $L(\mathcal{N}) = \Sigma^*$, then every word $w = u\#v$ with $u \in \Sigma^\star$ and $v \in (\Sigma')^\omega$ is accepted by $\mathcal{A}$. So the language of $\mathcal{A}$ is almost $(\Sigma')^\omega$, it just might miss words that don't have #. But this is easy to fix, as we shall now see.

Correct solution: we take the construction of $\mathcal{A}$ in the incorrect attempt, and add to it a new state q_Σ which is initial, accepting, and has a self loop for every $\sigma \in \Sigma$. This makes sure all words without # are accepted, and we are done. □

Exercise 3.29. In the construction above, why not just keep the accepting states of $\mathcal{N}$, instead of adding the new sink state?

Theorem 3.30. *The containment problem for* NBWs *is* **PSPACE-Complete**.

Proof. This is almost a corollary of Theorem 3.28.

For the upper bound, observe that $L(\mathcal{A}) \subseteq L(\mathcal{B})$ iff $L(\mathcal{A}) \cap \overline{L(\mathcal{B})} = \emptyset$. Thus, following the same approach of on-the-fly construction, we only need to check the emptiness of $L(\mathcal{A}) \cap \overline{L(\mathcal{B})}$, which we can do on-the-fly in polynomial space.

For **PSPACE** hardness, observe that $L(\mathcal{A}) = \Sigma^\omega$ iff $\Sigma^\omega \subseteq L(\mathcal{A})$, and we can easily construct an NBW for Σ^ω, so we have a simple reduction from universality. □

4 Additional Acceptance Conditions

"If you never did you should. These things are fun and fun is good" [Dr. Seuss].

Recall that in the Büchi acceptance condition, we said that a run is accepting if it visits the accepting states infinitely often. While natural, this was somewhat an arbitrary decision.

It turned out to be quite useful, and we ended up with a fairly rich model, with interesting properties.

However, this model has some drawbacks. In particular, it is not determinizable (Theorem 3.13). This is one reason to introduce new acceptance conditions. Another reason is that for modelling purposes, often natural properties are easily captured by certain acceptance conditions and not by others, so this allows some flexibility, especially when we have tractable algorithms that work on various acceptance conditions.

Finally, the best reason to study other condition is that it is a lot of fun to study them and the translations between them. So let's get to it.

We define the following acceptance conditions, where each acceptance condition consists of the "type" of α, and a definition of accepting run:

Definition 4.1.

co-Büchi
- co-Büchi, *where* $\alpha \subseteq Q$ *and a run* r *is accepting if* $\inf(r) \subseteq \alpha$.

Rabin
- Rabin, *where* $\alpha = \{(G_1, B_1), (G_2, B_2), \ldots, (G_k, B_k)\}$ *and a run* r *is accepting if there exists* $1 \leq i \leq k$ *such that* $\inf(r) \cap G_i \neq \emptyset$ *and* $\inf(r) \cap B_i = \emptyset$.

Streett
- Streett, *where* $\alpha = \{(G_1, B_1), (G_2, B_2), \ldots, (G_k, B_k)\}$ *and a run* r *is accepting if for every* $1 \leq i \leq k$ *we have that* $\inf(r) \cap G_i = \emptyset$ *or* $\inf(r) \cap B_i \neq \emptyset$.

Parity
- Parity, *where* $\alpha : Q \to \{0, \ldots, k\}$ *is a function, and a run* r *is accepting if* $\min\{\alpha(q) : q \in \inf(r)\}$ *is even.*

Muller
- Muller, *where* $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ *with* $\alpha_i \subseteq Q$, *and a run* r *is accepting if there exists* $1 \leq i \leq k$ *such that* $\inf(r) = \alpha_i$.

That is a lot to take in. We'll make some sense of this mess as we go along.[5]

Remark 4.2 (Naming convention)**.** The acronym NBW stands for "Nondeterministic Büchi automaton over Words". We can now plug in the names of the new acceptance conditions instead of the "B" (i.e., C,R,S,P, and M).

In addition, there are other structures on which automata may be ran, such as Trees or Graphs (although we will not see this during the course), replacing the W with T or G. finally, there are other "flavours" of nondeterminism, called Universality and Alternation, replacing N with U or A.

Thus, in general, automata models are acronymed *XYZ* where X ∈ {D,N,U,A} Y ∈ {B,C,R,S,P,M}, and Z ∈ {W,T}.

For example, NPW stands for nondeterministic parity automata over words, UCT stands for universal co-Büchi automaton over trees, DSW for deterministic Streett automata, etc. △

Let's start with some examples.

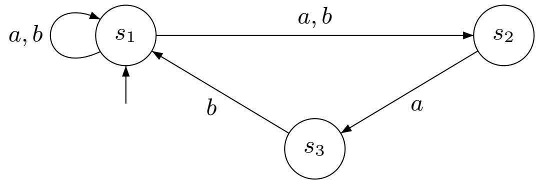


Figure 6: Example automaton

Example 4.3. Consider the automaton in Figure 6. What is the language of the automaton when viewed as:

1. An NBW with $\alpha = \{s_2\}$ (Ans: words containing infinitely many *ab*)
2. An NRW with $\alpha = \{(\{s_1, s_3\}, \{s_2\})\}$ (Ans: $\{a, b\}^\omega$)
3. An NSW with $\alpha = \{(\{s_2\}, \{s_3\}), (\{s_3\}, \{s_1\})\}$ (Ans: $\{a, b\}^\omega$)
4. An NCW with $\alpha = \{s_2\}$ (Ans: $\emptyset$)
5. An NPW with $\alpha(s_1) = \alpha(s_3) = 1$, $\alpha(s_2) = 0$ (Ans: words containing infinitely many *ab*)
6. An NPW with $\alpha(s_1) = \alpha(s_3) = 1$, $\alpha(s_2) = 2$ (Ans: $\emptyset$).

△

index

In the Rabin, Streett, parity, and Muller conditions, the number k is called the *index* of the automaton.

[5] Conditions are named after: Julius Richard Büchi [1924–1984], Michael O. Rabin [1931–], Robert S. Streett [56–], and David E. Muller [1924–2008] (also parity, brought to you by the Number 2).

4.1 Expressive Power and Translations

Our main interest now is to see whether we gain anything with all those new conditions. In particular, we wish to examine the *expressive power* of the classes. Consider two classes of automata, denoted γ and κ. We say that γ is *at least as expressive as* κ, if every language that is recognizable using a κ automaton, is also recognizable using a γ automaton. Let's examine some expressiveness results.

Consider the co-Büchi acceptance condition. It is not hard to see that a run r is accepting in the co-Büchi acceptance condition with set α iff it is *not* accepting in the Büchi condition with set $Q \setminus \alpha$. That is, co-Büchi is a "dual" condition to Büchi (hence the name). We can easily conclude the following.

Theorem 4.4. *Let $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, \alpha \rangle$ be a* DBW*, and consider the* DCW *$\mathcal{B} = \langle Q, \Sigma, \delta, q_0, Q \setminus \alpha \rangle$, then $L(\mathcal{B}) = \overline{L(\mathcal{A})}$.*

Remark 4.5. In most textbooks and papers, co-Büchi is defined by saying that a run r is accepting for condition α if $\inf(r) \cap \alpha = \emptyset$. This has the advantage of making Theorem 4.4 even cleaner – in order to complement a DBW, just "rename" it to a DCW.

However, this loses the flavour of "accepting", and makes α a "bad" set, and I just don't like it. △

What about a nondeterministic analogue of Theorem 4.4? Alas, if you take an NBW, flip the accepting states, and call it a DCW, you will not necessarily end up with the complement language (try to think of an example!).

In fact, an interesting result is that NCWs are equally expressive to DCWs. We might even have time to see it in this course.

Similarly to Theorem 4.4, the Rabin and Streett conditions are also dual (for deterministic automata):

Theorem 4.6. *Let $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, \alpha \rangle$ with $\alpha = \{(G_1, B_1), (G_2, B_2), \ldots, (G_k, B_k)\}$ be a* DRW*, and consider the* DSW *$\mathcal{B} = \langle Q, \Sigma, \delta, q_0, \alpha \rangle$ (syntactically identical to $\mathcal{A}$), then $L(\mathcal{B}) = \overline{L(\mathcal{A})}$.*

Proof. A word w is accepted by $\mathcal{A}$ iff its unique run r is accepting. That is, if there exists $1 \le i \le k$ such that r visits G_i infinitely often and B_i finitely often. This happens iff it is *not* the case that for all $1 \le i \le k$, the run r visits G_i finitely often or B_i infinitely often, which, in turn, happens iff $\mathcal{B}$ accepts w. □

Remark 4.7. A useful way to view the Streett condition is to say that a run is accepting if for every pair (G_i, B_i), if G_i is visited infinitely often, then so is B_i. This is just by viewing $\neg a \vee b$ as $a \to b$, but it often makes it easier to understand. △

We proceed by getting out of the way the "naive" translations: the cases where one acceptance condition simply captures another. The proof of the following theorem is an almost trivial exercise.

Theorem 4.8. *We have the following equivalences of acceptance conditions:*

- *A Büchi condition α is equivalent to the Rabin condition $\{(\alpha, \emptyset)\}$, to the Streett condition $\{(Q, \alpha)\}$, and to the parity condition $\alpha'(q) = \begin{cases} 0 & q \in \alpha \\ 1 & q \in Q \setminus \alpha \end{cases}$*
- *A co-Büchi condition α is equivalent to the Rabin condition $\{(Q, Q \setminus \alpha)\}$, to the Streett condition $\{(Q \setminus \alpha, \emptyset)\}$, and to the parity condition $\alpha'(q) = \begin{cases} 1 & q \in Q \setminus \alpha \\ 2 & q \in \alpha \end{cases}$*
- *A parity condition $\alpha : Q \to \{0, \ldots, k\}$ (w.l.o.g. k is even) is equivalent to the Rabin condition $\{(\alpha^{=0}, \emptyset), (\alpha^{=2}, \alpha^{\le 1}), \ldots, (\alpha^{=k}, \alpha^{\le k-1})\}$ and to the Streett condition $\{(\alpha^{=1}, \alpha^{\le 0}), (\alpha^{=3}, \alpha^{\le 2}) \ldots, (\alpha^{=k-1}, \alpha^{\le k})\}$ where $\alpha^{=i} = \{q : \alpha(q) = i\}$ and similarly $\alpha^{\le i} = \{q : \alpha(q) \le i\}$*
- *A Büchi, co-Büchi, Rabin, Streett or parity condition α is equivalent to the Muller condition $\{F : F \text{ satisfies } \alpha\}$.*

We now proceed to study (some of) the less-naive equivalences. There are many, many results about translating one type of automaton to another. Due to lack of time, we will focus on translations relating to Büchi acceptance.

Remark 4.9. When translating one class of automata to another, there are three possible kinds of translations: the easiest is when one acceptance condition can be rephrased as another, as we have done in Theorem 4.8. The next is when we can plug a new acceptance condition on top of the existing automaton, but in a way which may require reasoning about the structure of the automaton. The third case is when we actually need to construct a new automaton. △

We'll start with the translation from Rabin to Büchi.

Theorem 4.10. *Let $\mathcal{A}$ be an* NRW *with n states and index k. There is an* NBW *$\mathcal{B}$ with at most $2nk$ states such that $L(\mathcal{A}) = L(\mathcal{B})$.*

Proof. Let $\mathcal{A} = \langle Q, \Sigma, \delta, Q_0, \alpha\rangle$ with $\alpha = \{(G_1, B_1), \ldots, (G_k, B_k)\}$. Denote by $\mathcal{A}_i$ the NRW with index 1 obtained from $\mathcal{A}$ by including only the pair (G_i, B_i) in the acceptance condition. Clearly we have $L(\mathcal{A}) = \bigcup_{i=1}^{k} L(\mathcal{A}_i)$. By Theorem 3.6, NBW are closed under union. Thus, it's enough to show the translation for NRW with index 1 (however, we will need to take into account the size of the union NBW).

Let $\mathcal{A}' = \langle Q, \Sigma, \delta, Q_0, \{(G, B)\}\rangle$ be an NRW with index 1. Thus, a run is accepting if it visits B finitely often, and G infinitely often. We translate it to an NBW $\mathcal{B}$, by following a similar idea to Exercise 3.27: $\mathcal{B}$ will "guess" when states in B no longer occur, and check (using the Büchi acceptance condition) that states in G occur infinitely often.

Formally, let $\mathcal{B} = \langle (Q\times\{1\}) \cup ((Q\setminus B)\times\{2\}), \Sigma, \delta', Q_0\times\{1\}, G\times\{2\}\rangle$, where for all $q \in Q$ and $\sigma \in \Sigma$ we have $\delta'((q, 1), \sigma) = (\delta(q, \sigma)\times\{1\}) \cup ((\delta(q, \sigma)\setminus B)\times\{2\})$ and if $q \notin B$, then $\delta'((q, 2), \sigma) = (\delta(q, \sigma)\setminus B)\times\{2\}$.

That is, $\mathcal{B}$ holds two copies of $\mathcal{A}$, the second one without B. It simulates the run of $\mathcal{A}$ within the first copy, while nondeterministically guessing that states in B no longer occur, by moving into the second copy. Then, in the second copy, we are not allowed to visit B, and the run is accepting if it visits G (in the second copy) infinitely often.

Proving the correctness of this construction is a simple exercise.

Note that $\mathcal{B}$ has size $2n$. As mentioned above, by taking the union over the k Rabin pairs, we obtain an NBW of size $2nk$, that is equivalent to $\mathcal{A}$. □

Exercise 4.11. Improve the bound given in Theorem 4.10 from $2nk$ to $n(k + 1)$.

Combining the translation of parity to Rabin, given in Theorem 4.8, with Theorem 4.10, we have the following.

Theorem 4.12. *Let $\mathcal{A}$ be an* NPW *with n states and index k. There is an* NBW *$\mathcal{B}$ with at most $2nk$ states such that $L(\mathcal{A}) = L(\mathcal{B})$.*

So, we've managed to translate NRW to NBW by considering each Rabin pair separately. Can we do the same thing with Streett automata? Unfortunately, no. Technically, the "reason" is that even if a word is accepted under each Streett pair separately, the runs that witness this may be distinct, and there might not be a run that satisfies *all* pairs simultaneously.

More fundamentally, the underlying reason is that the Rabin condition places an *existential* quantifier on the Rabin pairs ('*'there exists* a pair such that..."), and this matches the flavour of nondeterminism: a word is accepted if *there exists* an accepting run. However, in the Streett condition, the quantifier is universal, and this clashes with the nondeterminism. Thus, in order to translate NSW to NBW, we must do so holistically.

Theorem 4.13. *Let $\mathcal{A}$ be an* NSW *with n states and index k. There is an* NBW *$\mathcal{B}$ with at most $n(k + 1)2^k$ states such that $L(\mathcal{A}) = L(\mathcal{B})$.*

Proof. Let $\mathcal{A} = \langle Q, \Sigma, \delta, Q_0, \alpha\rangle$ with $\alpha = \{(G_1, B_1), \ldots, (G_k, B_k)\}$, and recall that a run r is accepting if for every $1 \le i \le k$, either $\inf(r) \cap G_i = \emptyset$ or $\inf(r) \cap B_i \neq \emptyset$.

The idea behind the translation is to decompose the accepting runs according to the "way" they satisfy the Streett conditions – for each pair (G_i, B_i), we consider whether the run accepts by satisfying $\inf(r) \cap G_i = \emptyset$ or by satisfying $\inf(r) \cap B_i \neq \emptyset$.

More precisely, for a set $I \subseteq \{1, \ldots, k\}$, we construct an NBW $\mathcal{B}_I$ that accepts a word w iff there exists a run r of $\mathcal{A}$ on w such that $\inf(r) \cap G_i = \emptyset$ for all $i \in I$ and $\inf(r) \cap B_i \neq \emptyset$ for all $i \notin I$. Then, we have that $L(\mathcal{A}) = \bigcup_{I\subseteq\{1,\ldots,k\}} L(\mathcal{B}_I)$, and we are done.

Constructing $\mathcal{B}_I$ is fairly similar to the two-copies construction, augmented by a product construction: $\mathcal{B}_I$ first guesses when none of the states in $\bigcup_{i\in I} G_i$ will be seen, and then proceeds to verify that states in each B_i for $i \notin I$ are visited infinitely often. This is done as follows. For simplicity, assume $I = \{1, \ldots, m\}$, so $\overline{I} = \{m+1, \ldots, k\}$. Denote $G_I = \bigcup_{i\in I} G_i$.

Let $\mathcal{B}_I = \{Q', \Sigma, \delta', Q'_0, \alpha'\}$, where:

- $Q' = (Q \times \{0\}) \cup \bigcup_{i\notin I}(Q \setminus G_I) \times \{i\}$.
- $Q'_0 = Q_0 \times \{0\}$.
- $\alpha' = B_k \times \{k\}$ (i.e., the "last" copy contains accepting states).
- For $q \in Q$ and $\sigma \in \Sigma$ we have $\delta'((q,0),\sigma) = (\delta(q,\sigma) \times \{0\}) \cup ((\delta(q,\sigma) \setminus G_I) \times \{m+1\})$. For $i \notin I$ and $q \notin G_I$, if $q \notin B_i$ then $\delta'((q,i),\sigma) = (\delta(q,\sigma) \setminus G_I) \times \{i\}$, and if $q \in B_i$ then $\delta'((q,i),\sigma) = (\delta(q,\sigma) \setminus G_I) \times \{i+1\}$, where we identify $k+1$ with $m+1$ (that is, we cycle through copies $m+1, \ldots, k$ whenever we see the respective B_i).

As usual, the correctness of the construction is easy to verify.

Finally, the size of $\mathcal{B}_I$ is (at most) $n(k+1)$, and $\mathcal{B}$ is obtained as a union of 2^k such automata, so the overall size is $n(k+1)2^k$. □

Theorem 4.13 shows that translating an NSW to an NBW may incur an exponential blowup. As it turns out, this blowup is unavoidable (see [10] for a lower bound).

Remark 4.14. The translations between acceptance conditions described in Theorem 4.8 require only a naive change in the acceptance condition, and are essentially unrelated to the underlying automaton. In contrast, the translations in Theorems 4.10 and 4.13 change the structure of the automaton.

There are some translations "in between" these extremes: sometimes one acceptance condition can be translated to another without changing the structure, subject to conditions on the underlying automaton (e.g., determinism). When such a translation from class A to class B is possible, we say that the A is

typeness B-type. See [10] for more on *typeness*. △

Remark 4.15. Udi Boker from IDC maintains a website with all the known translations (and bounds) between types of automata. I strongly recommend checking it out: `http://www.faculty.idc.ac.il/udiboker/automata/wat.html` △

Exercise 4.16. Show that for every NMW (Nondeterministic Muller Automaton) $\mathcal{A}$ there exists an NBW $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$.

What is the size of $\mathcal{B}$ (with respect to the size and index of $\mathcal{A}$)?

From Theorems 4.10 and 4.13 and Exercise 4.16, we see that none of the acceptance conditions we suggest is more expressive than NBW. It thus raises the question of why define them? One possible answer is that sometimes, they are more *succinct* (i.e., they can capture the same language with less states).

However, as we shall see in Section 4.2, they offer a lot in the deterministic realm.

Finally, I heartily recommend reading [1] for a very detailed survey of the automata types, translations between them, and the reasons for their definitions.

4.2 Determinization of NBWs (and NCWs)

For many theoretical and practical uses, deterministic automata are more convenient to work with than nondetemrinistic ones. Unfortunately, as we have seen in Theorem 3.13, DBWs are strictly less expressive than NBWs. However, given the new acceptance conditions we introduced above, it is worth asking whether we can determinize an NBW to a deterministic Rabin/Streett/Parity/Muller automaton.

As it turns out, the answer is yes. The "quest" for NBW determinization has a long and interesting history. You can read all about it in [10] and references therein. The most notable parts for our purpose, leaving out the historical goodies, is that Safra [23] provided a $2^{O(n \log n)}$ translation from NBW to DRW, and Schewe [25] improved the construction, and made it from NBW to DPW (we will see later why this is an improvement).

4.2.1 NCW determinization

Safra's theorem is notoriously difficult, both to understand and to optimize in practice. In an attempt to make it clearer, we'll start by tackling a less daunting task, namely determinizing NCWs.

The first step is to admit you have a problem

Why is determinization so difficult?

Recall that for finite automata, determinization is done using the subset construction: we keep a record of all the reachable states, and when the word ends, if an accepting state is reachable, we accept. We've already seen that this approach doesn't work for NBWs (see Theorem 3.13), but let's try to think exactly what breaks down.

We'll focus for now on NCWs, but similar problems (worse ones, actually) arise for NBWs.

Consider the NCW $\mathcal{A}$ in Figure 7, for the language of words with finitely many a's.

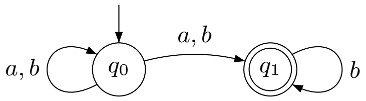


Figure 7: An NCW for the language of words with finitely many a's.

For a word $w \in \{a, b\}^*$, we can describe all the runs of $\mathcal{A}$ on w using an infinite *run DAG* (Directed Acyclic Graph), as demonstrated in Figure 8 (ignore the red line for now).

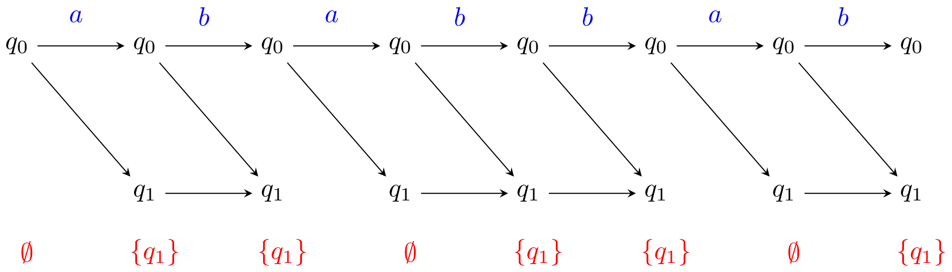


Figure 8: A run DAG with the breakpoint construction.

It's now easy to see what goes wrong with the subset construction: the accepting state q_1 is present in every reachable subset! This means we cannot possibly define a good acceptance condition on top of this structure. The underlying reason for this problem, is that the subset construction only gives us information about prefixes that end in an accepting state, not about whether an infinite path can be composed of these prefixes.

So, intuitively, we need to store some more information about the run DAG. One way to proceed is by storing the entire run DAG up to the current level. This is clearly enough to define a good acceptance condition, but it's also clearly unbounded, and we need the number of states to be finite.

The Breakpoint Construction

We will now use a clever, but simple, construction, called the *breakpoint construction*, which was proposed by Miyano and Hayashi [18] (in fact, their construction handles the more general setting of Alternating Automata).

Theorem 4.17. *Let $\mathcal{A}$ be an* NCW *with n states, then there exists an equivalent* DCW *$\mathcal{D}$ with at most 3^n states.*

Proof. The idea behind the construction is the following: start with the subset construction, but in every transition, keep track of runs that have not visited $Q \setminus \alpha$ recently (equivalently, we actually keep track of the runs that have seen only α recently). Once we find a subset such that all its incoming runs visit

$Q \setminus \alpha$ recently, we call this a breakpoint, and we "reset" this subset. Then, if we see infinitely many breakpoints, the run is rejecting.

The additional subset is depicted in the red line of Figure 8.

We now formally define it. Let $\mathcal{A} = \langle Q, \Sigma, \delta, Q_0, \alpha \rangle$, define $\mathcal{D} = \langle 2^Q \times 2^Q, \Sigma, \mu, (Q_0, \emptyset), \beta \rangle$ as follows. The set of states is $2^Q \times 2^Q$ (or rather a subset thereof). The transition function is defined as follows: let $(S, T) \in 2^Q \times 2^Q$ and let $\sigma \in \Sigma$. We define

$$\mu((S,T),\sigma) = \begin{cases} (\delta(S,\sigma), \delta(T,\sigma) \cap \alpha) & T \neq \emptyset \\ (\delta(S,\sigma), \delta(S,\sigma) \cap \alpha) & T = \emptyset \end{cases}$$

Thus, T starts by keeping track of all the accepting states, and with each transition it "loses" all the states that do not stay within α. When no states remain, T resets, and this is a breakpoint.

We capture that using the acceptance conditions: $\beta = \{(S,T) : S \in 2^Q \wedge T \neq \emptyset\}$.

We now claim that $L(\mathcal{A}) = L(\mathcal{D})$. For the first direction, let $w \in L(\mathcal{A})$, and let $\rho = q_0, q_1, \ldots$ be an accepting run of $\mathcal{A}$ on w. This means that there exists $N_0 \in \mathbb{N}$ such that for all $i \geq N_0$ we have $q_i \in \alpha$.

Consider the run of $\mathcal{D}$ on w, and assume it reaches a breakpoint in index $N_1 \geq N_0$ (if it doesn't then $\mathcal{D}$ accepts w and we are done). Then, at index $N_1 + 1$, the run of $\mathcal{D}$ on w is in state (S, T) with $q_{N_1} \in T$. Moreover, for every $i \geq N_1$ it holds that q_i is in the second component of the run of $\mathcal{D}$, by the definition of the transition function. Thus, the run of $\mathcal{D}$ does not see any more breakpoints, so $\mathcal{D}$ accepts w.

Conversely, assume $\mathcal{D}$ accepts w, then the run of $\mathcal{D}$ on w, denoted $\rho = (S_0, T_0), (S_1, T_1), \ldots$ is accepting. Thus, there exists $N_0 \in \mathbb{N}$ such that for all $i \geq N_0$, we have $T_i \neq \emptyset$. We now need to show that $\mathcal{A}$ has an accepting run on w. Consider the run DAG of $\mathcal{A}$ on w. It's enough to show that this graph has an infinite path from some state in Q_0 that visits $Q \setminus \alpha$ only finitely many times.

We refer to the run DAG as a graph with infinitely many *levels*, where at each level we have the set of reachable states in Q. Now, remove from the run DAG all the states in $Q \setminus \alpha$ after level N_0. Since $T_i \neq \emptyset$ for all $i \geq N_0$, it follows that for every level (every level, not just $i \geq N_0$), there exists a path from Q_0 that ends in some state in that level (this is basically just saying that the level is not empty). So we now have finite runs to every level, but we're still missing an infinite run.

As a rule, when you see an infinite tree with a finite branching factor, it's KÖNIG TIME!

Lemma 4.18 (König's lemma)**.** *A finitely branching infinite tree contains an infinite path.*

Now we can use König's Lemma, which tells us that the pruned run DAG has an infinite path starting from Q_0, as we wanted, so $w \in L(\mathcal{A})$.

Finally, note that the theorem says that the size of $\mathcal{D}$ is 3^n, whereas we only showed 4^n. Try to see why this is actually 3^n. □

4.2.2 NBW **determinization**

We are now mature enough to tackle NBW determinization.

Safra's theorem is stated as follows.

Theorem 4.19. *Let $\mathcal{A}$ be an* NBW *with n states, then there exists a* DRW *$\mathcal{D}$ with $2^{O(n \log n)}$ states and index n such that $L(\mathcal{D}) = L(\mathcal{A})$.*

and its DPW version is:

Theorem 4.20 ([25])**.** *Let $\mathcal{A}$ be an* NBW *with n states, then there exists a* DPW *$\mathcal{D}$ with $2^{O(n \log n)}$ states and index $2n$ such that $L(\mathcal{D}) = L(\mathcal{A})$.*

Before we proceed to see the construction, let us remark that we already know a lower bound. Indeed, in Theorem 3.26 we have seen a lower bound on NBW complementation: there is a family of languages $L(n)$ that have NBWs of size $n + 1$, but an NBW for $\overline{L(n)}$ requires $n!$ states. Now, suppose there was a DRW for $L(n)$ with less than $n!$ states, then by Theorem 4.6 there would be a DSW for $\overline{L(n)}$ with less than $n!$ states. However, the proof of Theorem 3.26 can easily be adapted to work for DSW (or for that matter, NSW), so that would yield a contradiction. Thus, we have the following.

Theorem 4.21. *There is a family of languages $L_1, L_2, \ldots$ that have an* NBW *of size $n + 1$, but every* DRW *for L_n requires at least $n!$ states.*

Finally, recall that $n! = 2^{\theta(n \log n)}$, so this gives a tight lower bound.

4.2.3 Proof of Theorem 4.19

Remark: These notes follow the course notes of Automata, Logic, and Games given at Oxford University by Prof. Luke Ong. They are based on the construction and examples from [25].

History Trees

The main idea behind NBW determinization is that of *history trees*. You can think of them as an extension of the subset construction and of the breakpoint construction, that contains much more information about the runs.

Γ-labelled tree

Let Γ be a finite alphabet. A Γ-*labelled tree* τ is a partial function $\tau : \mathbb{N}^* \to \Gamma$, which maps each "position" $x \in \mathbb{N}^*$ to its label $t(x) \in \Gamma$, such that the domain of τ, denoted $\mathrm{dom}(\tau)$ is prefix closed (that is, if $xd \in \mathrm{dom}(\tau)$ then $x \in \mathrm{dom}(\tau)$).

Intuitively, a Γ labeled tree is just a (possibly infinite) tree, whose nodes are labelled with elements from Γ. We also give explicit "names" to the nodes, by sequences of natural numbers (which means the tree has a possibly unbounded arity).

We say that τ is:

finite

- *finite* if $\mathrm{dom}(\tau)$ is finite.

ordered

- *ordered* if $xd \in \mathrm{dom}(t)$ implied that $xe \in \mathrm{dom}(\tau)$ for every $0 \le e \le d$.

See Figure 9 for an example. If xd and xe are two children of a node, with $e \ge d$, we say that xe is *younger* than xd. That is, the youngest child has a larger index (since it was "born" later)[6].

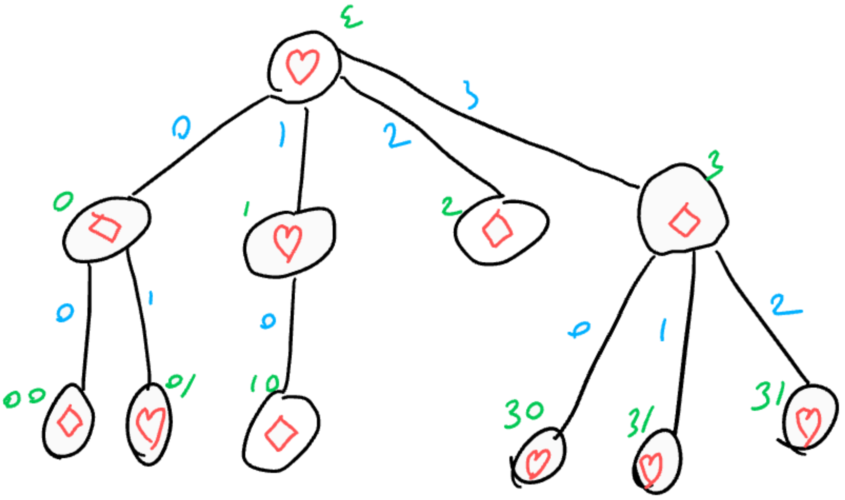


Figure 9: A Γ-labelled tree for $\Gamma = \{\heartsuit, \diamondsuit\}$. The names of the nodes are written in green, and the edges are marked with blue numbers that show how the name of the child node is obtained. The root's name is the empty word ϵ. For example, node 30 is the 0'th child of node 3. This tree is finite and ordered.

We can now get to the central definition.

history tree

Definition 4.22. *Let* $\mathcal{A} = \langle Q, \Sigma, Q_0, \delta, \alpha\rangle$ *be an* NBW. *A* history tree *for* $\mathcal{A}$ *is a* $2^Q \setminus \emptyset$*-labelled ordered finite tree* τ *such that:*

- *the label of each node is a* proper superset *of the union of labels of all its children, and*
- *the labels of the children of each node are disjoint.*

Observe that there are only finitely many possible history trees for a given NBW $\mathcal{A}$. These history trees can then serve as the basis for our state space.

enriched history tree

An *enriched history tree* is a history tree equipped with additional labels, that indicate for each node if it is unstable, or if it is a breakpoint (notions that we will define below).

[6] Had I known this earlier, my children would have definitely been named Shaull0 and Shaull1

Updating history trees

Suppose we somehow manage to get our history trees to serve as a state space. We now define a sort of transition function on them, by describing an "update" procedure.

Given a history tree τ for $\mathcal{A}$, and $\sigma \in \Sigma$, we construct a new enriched history tree τ' as follows:

1. **Update:** for every $x \in \text{dom}(\tau)$, let $\tau'(x) = \bigcup_{q \in \tau(x)} \delta(q, \sigma)$.

   That is, apply the transition function to all the states in all the nodes. Makes sense.

2. **Create children:** for every $x \in \text{dom}(\tau')$, create a new "youngest" child xd (i.e. with the largest index) with $\tau'(xd) = \tau'(x) \cap \alpha$.

   That is, the new child keeps track of the accepting states in its parent. Makes sense, just like in the breakpoint-construction.

3. **Horizontal merge:** for every $x \in \text{dom}(\tau')$ and every $q \in Q$, if q appears in an older sibling of x, then remove q from $\tau'(x)$ and all of its descendants.

   Intuitively, if q appears in an older sibling, then we're already "tracking" it. No need to track it in a newer copy. I wouldn't say this makes sense, but it sounds like something clever.

4. **Vertical merge:** for every $x \in \text{dom}(\tau')$:

   - if $\tau'(x) = \emptyset$, remove node x from τ' (an $\emptyset$ label could appear both in steps 2 and 3, and it serves no purpose.)
   - if $\tau'(x) = \bigcup_{d \in \mathbb{N}} \tau'(xd)$, then remove all nodes xd, and call this a breakpoint for x.
     Well, a node has to be a proper superset of its children. So if it isn't, we kill all the children (didn't think you'll see this sentence in a course on automata, did you?) The intuition about this is again, similar to the breakpoint construction.

5. **Repair order:** Rename the nodes to restore order to τ' (in case some siblings were removed). Call nodes that were renamed unstable.

   Restoring the order is always good. But keep an eye out for these unstable nodes. They'll play a role soon.

Example 4.23. Consider the NBW in Figure 10.

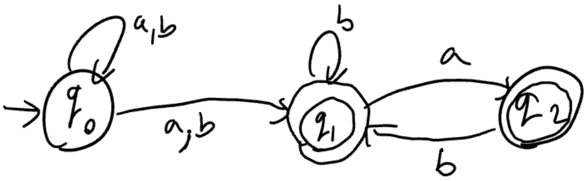


Figure 10: An NBW to be determinized.

We demonstrate the update of a history tree in Figure 11.

△

DRW construction

Now that we defined history trees, the definition of the equivalent DRW is fairly straightforward (if slightly inexplicable). We will then prove the correctness, and hopefully gain some intuition behind the construction.

Let $\mathcal{D} = \langle S, \Sigma, s_0, \mu, \beta \rangle$ be the following DRW:

- The state space S is the set of enriched history trees of $\mathcal{A}$.
- The alphabet is still Σ (if you don't understand this, maybe it's time for a break).

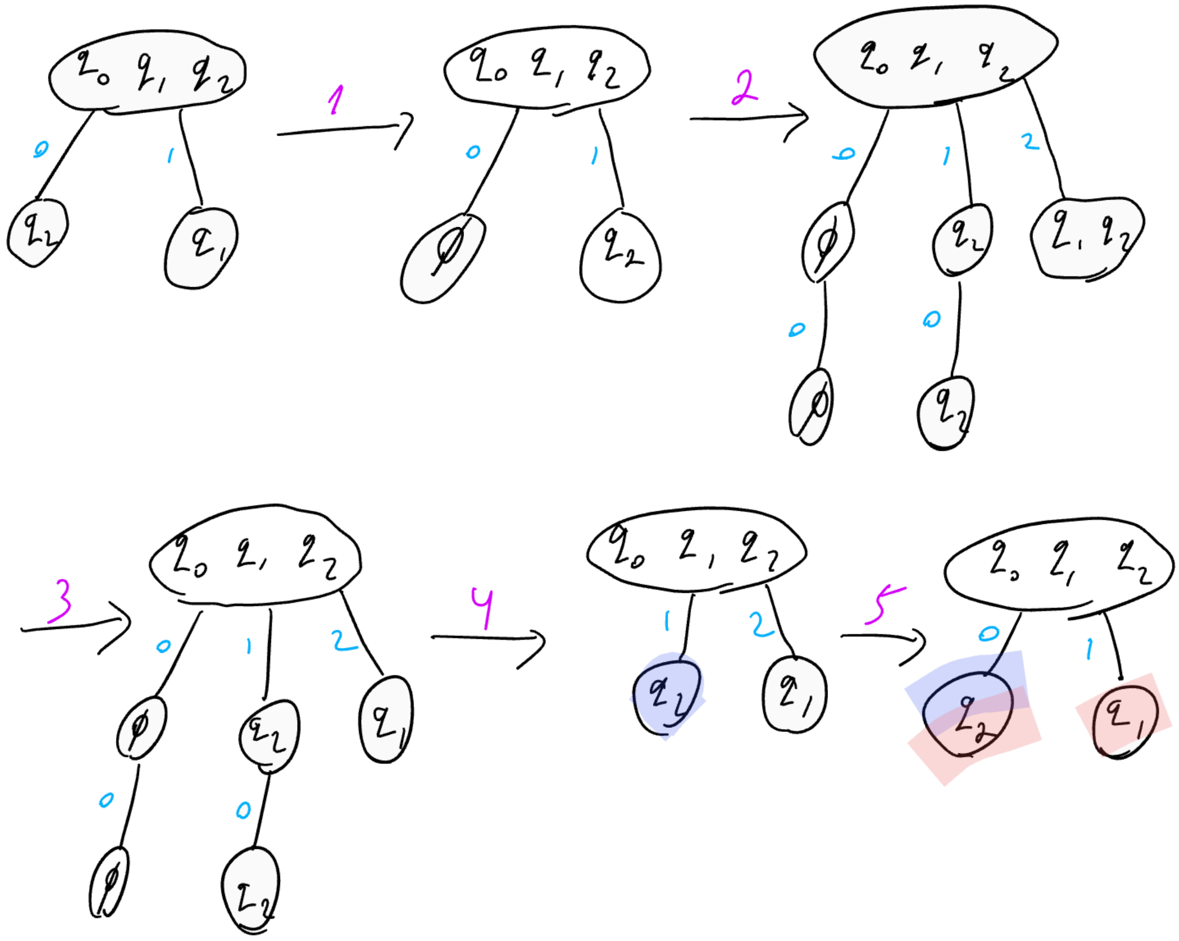


Figure 11: A history tree update for the NBW in Figure 10 with the letter a. The blue highlight represents a breakpoint, and the red highlight represents an unstable position.

- The initial state s_0 is the history tree that is just a root labelled by Q_0. That is, $\mathrm{dom}(s_0) = \{\epsilon\}$ and $s_0(\epsilon) = Q_0$ (recall that s_0 is itself a history tree).
- The transition function is defined by the tree update: for a history tree $s \in S$ and $\sigma \in \Sigma$, $\mu(s, \sigma) = s'$ is obtained by updating s according to the steps above, and marking the breakpoint and unstable parts of the result s' (the original enrichments on s are ignored for the transition purposes).

...drumroll...

- The Rabin acceptance condition is $\beta = \{(G_x, B_x) : x \in P\}$ where:
  - P is the set of positions in the history trees in S (sanity check: $P \subseteq \mathbb{N}^*$)
  - $G_x \subseteq S$ is the set of enriched history trees where x is a breakpoint.
  - $B_x \subseteq S$ is the set of enriched history trees where x is unstable, or not in the domain.

  That is, the acceptance condition says that a run is accepting if there is some position x that is eventually always stable (that is, present and not unstable) and is infinitely often a breakpoint.

In Figure 12 we show an equivalent DRW to the NBW in Figure 10 obtained by the construction.

Correctness proof

We want to prove that $L(\mathcal{A}) = L(\mathcal{D})$. There are two directions to the proof – a hard direction, and a harder direction.

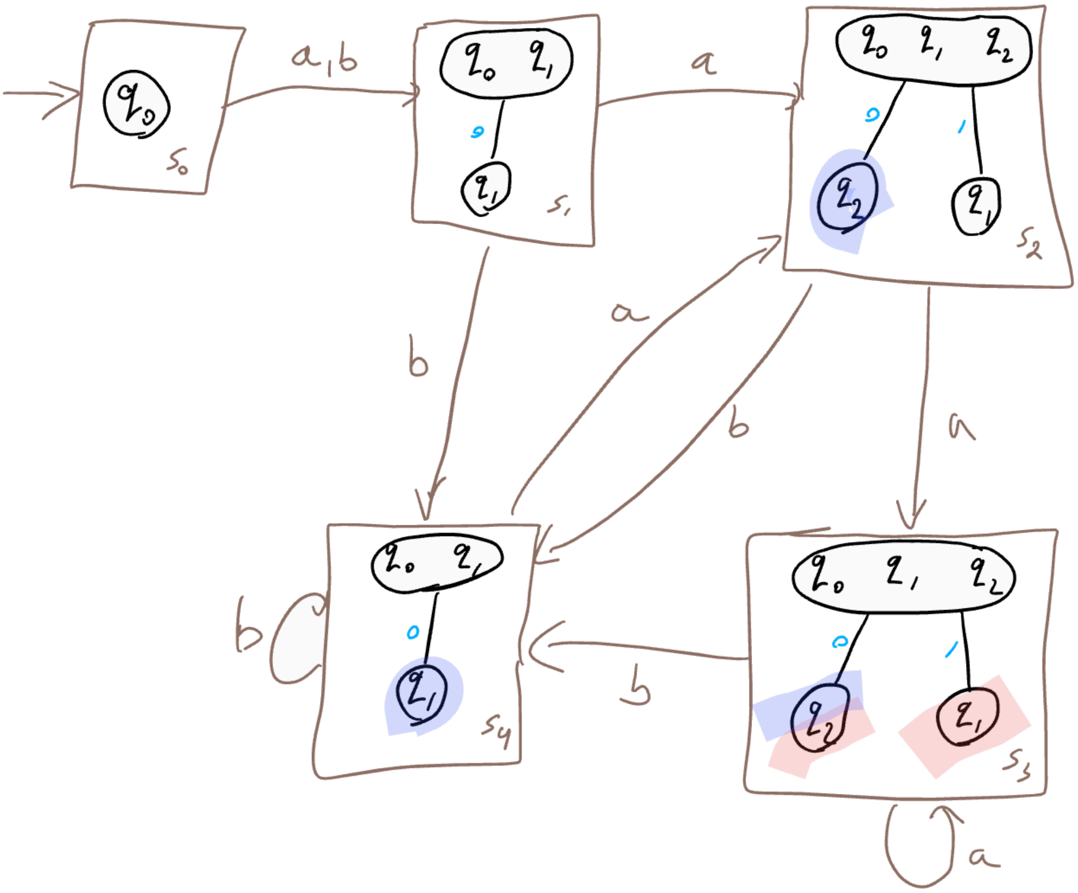


Figure 12: An equivalent DRW obtained by Safra's determinization. The acceptance condition is $\{(\{s_2, s_3, s_4\}, \{s_3\})\}$, since the only position that can potentially be a breakpoint is 0.

$\mathbf{L}(\mathcal{A}) \subseteq \mathbf{L}(\mathcal{D})$: Let $w \in L(\mathcal{A})$ and let $\rho_{\mathcal{A}} = q_0, q_1, \ldots$ be an accepting run of $\mathcal{A}$ on w. Let $\rho_{\mathcal{D}} = s_0, s_1, \ldots$ be the (single) run of $\mathcal{D}$ on w. We claim $\rho_{\mathcal{D}}$ is accepting.

First, observe that the roots of the states in $\rho_{\mathcal{D}}$ are always stable. Indeed, the root never has siblings, so it is never renamed. Thus, if the roots are infintely often a breakpoint, then $\rho_{\mathcal{D}}$ is accepting, and we are done. Otherwise, Let i_1 be the first index after the last breakpoint of the root, where $q_{i_1} \in \alpha$. By Step 2 (create children) of the tree update, it follows that for every $j \geq i_1$, the state q_j is in the label of a child of the root (since there are no more vertical merges to the root).

Moreover, from some point and on, all the states q_j are in the label of a stable child of the root. Indeed, horizontal merges can only transfer a state to an older sibling, so after some point either we reach sibling 0, or we get stuck at some higher sibling that is never renamed. Call this stable child x_1. If x_1 is infinitely often a breakpoint, then $\rho_{\mathcal{D}}$ is accepting and we are done. Otherwise, following the same logic, we get that eventually the states of $\rho_{\mathcal{A}}$ are in a label of a stable child x_2 of x_1. If x_2 is infinitely often a breakpoint, we're done, and so on.

Since the history trees are of depth at most $|Q|$, this must terminate at some point, at which the position is infinitely often a breakpoint (since it has no stable child that contains the states of the $\rho_{\mathcal{A}}$). So $w \in L(\mathcal{D})$.

$\mathbf{L}(\mathcal{D}) \subseteq \mathbf{L}(\mathcal{A})$: Let $w \in L(\mathcal{D})$, and let $\rho_{\mathcal{D}}$ be the accepting run of $\mathcal{D}$ on w. Thus, there is some position x that is eventually always stable, and infinitely often a breakpoint. Denote by R_i the label (i.e. the set of states of $\mathcal{A}$) at position x for the at the i-th time x is a breakpoint, after it is *stable*. Let u_i be the infix of w read between the i-th breakpoint and $(i+1)$-th breakpoint, and let u_0 be the prefix of w read before R_1.

What can we say about these R_i? First, for every $r \in R_1$, there is a (finite) run of $\mathcal{A}$ on u_0 from Q_0 that ends in r. Somewhat similarly, for every $r \in R_{i+1}$, there is a state $q \in R_i$ such that there is a finite run of $\mathcal{A}$ on u_i from q to r. But we can actually say something much stronger: for every $r \in R_{i+1}$, there is a state $q \in R_i$ such that there is a finite run of $\mathcal{A}$ on u_i from q to r that passes through α. Indeed, Since at R_{i+1} x is a breakpoint, then it was just vertically-merged, meaning that all its states appeared in its children. But the children are created (initially) only using states in α. So a vertical merge essentially means that all the states are reachable via a path that visited α. That is, if you think about the subset construction starting from R_i, if there was a state in R_{i+1} that is *not* reachable from R_i via α, then such a state would not appear in the children at R_{i+1}, so there would not be a breakpoint.

It seems like we're almost done. We have runs that visit α quite often. But we're still missing an infinite run. Observe that in the argument above, runs go "backward": we know that $r \in R_{i+1}$ is reachable from $q \in R_i$, and so on. But we don't know how to continue *from* r to R_{i+2}. We proceed as follows. Arrange the (finite) runs that we do have in a run tree. That is, a Q-labelled tree which represents the runs of $\mathcal{A}$ that are obtained by concatenating the runs we found above. Note that this is an *infinite tree* (since there are infinitely many R_i's), but it has a *finite branching factor*, since from each node we have at most $|Q|$ outgoing runs (since $|R_i| \le |Q|$).

And what do we say to infinite trees with a finite branching factor? It's KÖNIG TIME! From König's lemma (Lemma 4.18) we now get that the run tree above has an infinite run, but then this infinite run of $\mathcal{A}$ visits α infinitely often, so $w \in L(\mathcal{A})$. □

The size of $\mathcal{D}$:

We now turn to analyze the size of $\mathcal{D}$. To do so, we need to count the number of possible enriched history trees of $\mathcal{A}$. Let n denote the number of states in $\mathcal{A}$. Recall that in a history tree, each node is a proper superset of the union of its children, and that the nodes in the children are disjoint. It easily follows (by induction on n) that a history tree can have at most n nodes.

Now, by Cayley's tree formula, there are exactly n^{n-2} labelled trees over n nodes. In our case, however, we consider *ordered* labelled trees. So we can actually permute the nodes on each level. As an upper bound, we can just say that each of the above trees can induce at most $n!$ ordered trees. So our running total is $n^{n-2} \cdot n!$ trees.

Next, it is not only important to us that the labels are distinct, we are also interested in what the actual labels are. A-priori, each node can be labelled by a set of state $S \subseteq Q$, so a tree with n nodes induces $(2^n)^n = 2^{n^2}$ labellings, which would be too many. Instead, observe that by the structure of the labelling, each state in Q appears along (at most) a single path from the root, ending in some node. Thus, the labelling is uniquely determined by setting, for each state $q \in Q$, the node in which this path ends. So for every state we have $n+1$ options (one of the nodes in the tree, or none of them), giving us a total of $(n+1)^n$ possible labellings.

We thus have a running total of $n^{n-2} \cdot n! \cdot (n+1)^n$.

Finally, for each of the $n^{n-2} \cdot n! \cdot (n+1)^n$ trees, we need to mark the breakpoint and unstable nodes. This gives another 4^n options per tree (4^n because each node is marked with breakpoint, unstable, both or neither). So the number of states of $\mathcal{D}$ is at most $n^{n-2} \cdot n! \cdot (n+1)^n \cdot 4^n = 2^{O(n \log n)}$

Also, since each tree has at most n nodes, the index of $\mathcal{D}$ is at most n.

5 Kripke Structures and Model Checking

Recall that our goal is to model and reason about systems. So far we have provided the technical basis for this, which is automata. But we still haven't seen how they aid us to achieve this goal.

In this section, we introduce our formalisms for describing systems and semantics. We'll start with an example.

Example 5.1 (Mutex Protocol)**.** We have two processes that wish to gain access to a critical section in their code. Their behaviour is modeled by the following code:

```
Process 0:  Repeat                    Process 1:  Repeat
00:  <non-critical region 0>          00:  <non-critical region 1>
01:  wait unless turn = 0             01:  wait unless turn = 1
10:  <critical region 0>              10:  <critical region 1>
11:  turn := 1                        11:  turn := 0
```

How do we describe the current state of the system (where the "system" is the composition of the two processes)? We need to describe, for each process, in which line it currently is, as well as the value of the shared variable `turn`. Thus, a *state* of the system is of the form (a_1, a_2, b_1, b_2, t), where $a_1 a_2$ is the line number of Process 0, $b_1 b_2$ is that of Process 1, and t is the shared variable. For convenience we'll write the states as e.g., $(00, 10, 1)$.

We remark that we enumerate the lines in binary rather than using actual numbers, since we will later want to describe everything using Boolean propositions (see Definition 5.2 below).

Now, the states give rise to *transitions* between them. For example, from the (initial) state $(00, 00, 0)$, the next state can be $(01, 00, 0)$ or $(00, 01, 0)$. Then, from $(00, 01, 0)$, the next state can be either $(00, 01, 0)$ (if the "wait" was taken in Process 1) or $(01, 01, 0)$. This can proceed on and on.

Note that we assume each transition can be taken, so in a sense we have *nondeterminism* here. But the world is deterministic (to the best of our knowledge), so where did this nondeterminism come from? Well, the idea is that these processes actually act in within some larger system, in the presence of a scheduler. But since we (the designers) don't know the scheduler, we abstract it away by saying "everything is possible".

Now that we have our model for the system, what kind of *properties* to we want to make sure it satisfies?

- Safety: it is never the case that both critical sections are entered simultaneously. This can be formulated as "states of the form $10, 10, t$ are not reachable".
- Liveness: we want to see some progress in the program. This can be formulated as "infinitely often $10, b_1 b_2, t$ or $a_1 a_2, 10, t$ are reached"
- Fairness: each process reaches the critical section infinitely often: "infinitely often $10, b_1 b_2, t$ and $a_1 a_2, 10, t$ are reached"

Does the system satisfy these properties? it satisfies safety, but not (necessarily) liveness nor fairness. △

In the following we will investigate how to formally define systems and specifications, and how to check whether a system satisfies its specification.

5.1 Kripke Structures

We model our systems using Kripke structures. Intuitively, a Kripke structure is a directed graph, whose states are labeled with sets of truth values for Boolean atomic propositions $\{p_1, \ldots, p_m\}$. The infinite paths in this graph model the various behaviours of the system.

Kripke Structure

Definition 5.2. *A* Kripke Structure *over atomic propositions* $P = \{p_1, \ldots, p_m\}$ *is* $\mathcal{K} = (S, R, \mathscr{L}, S_0)$ *where:*

- S *is a finite set of states.*
- $R \subseteq S \times S$ *is a total transition relation (i.e., for every* $s \in S$ *there exists* $s' \in S$ *such that* $(s, s') \in R$*)*
- $\mathscr{L} : S \to 2^P$ *is a labeling function, assigning to each state a set of propositions.*
- $S_0 \subseteq S$ *are the initial states.*

path
labeling

A *path* in a Kripke structure $\mathcal{K} = (S, R, \mathscr{L}, S_0)$ is an infinite sequence of states $\rho = s_0, s_1, s_2, \ldots$ such that $s_0 \in S_0$ is an initial state, and for every $i \geq 0$ we have $(s_i, s_{i+1}) \in R$. The *labeling* of ρ is $\mathscr{L}(\rho) = \mathscr{L}(s_0)\mathscr{L}(s_1)\cdots$, and is regarded as a word in $(2^P)^\omega$. A *computation* of $\mathcal{K}$ is the labeling of a path of $\mathcal{K}$.

computation
language

The *language* of $\mathcal{K}$ (over the alphabet 2^P) is then $L(\mathcal{K}) = \{\pi : \pi \text{ is a computation of } \mathcal{K}\}$.

Since our technical tool is automata, we want to translate Kripke structures to automata.

Theorem 5.3. *Let* $\mathcal{K} = (S, R, \mathscr{L}, S_0)$ *be a Kripke structure over atomic propositions* P*. We can construct in polynomial time an* NBW $\mathcal{A}_{\mathcal{K}}$ *over alphabet* 2^P *such that* $L(\mathcal{A}_{\mathcal{K}}) = L(\mathcal{K})$*.*

Proof. The idea is quite simple: $\mathcal{A}_\mathcal{K}$ looks exactly like $\mathcal{K}$, but we "push" the labels to the transitions instead of the states.

Formally, $\mathcal{A}_\mathcal{K} = \langle S, 2^P, \delta, S_0, S\rangle$ where for every state $s \in S$ and letter $\sigma \in 2^P$ we have

$$\delta(s,\sigma) = \begin{cases} \{s' \in S : (s,s') \in R\} & \sigma = \mathscr{L}(s) \\ \emptyset & \sigma \neq \mathscr{L}(s) \end{cases}$$

That is, from every state s we are allowed to read only the label of s, in which case we progress to all neighbors. Note that all the states in $\mathcal{A}_\mathcal{K}$ are accepting.

Clearly a word is accepted by $\mathcal{A}_\mathcal{K}$ iff it is a computation of $\mathcal{K}$. In addition, words that are not accepted simply do not have runs in $\mathcal{A}_\mathcal{K}$. □

5.2 Model Checking

Now that we have a formalism for systems, comes the question of how to specify properties of such systems. We first need to define what properties are.

linear-time property

Definition 5.4. *Let $P = \{p_1, \dots, p_n\}$ be atomic propositions. A* linear-time property *over P is an ω-language $L \subseteq (2^P)^\omega$.*

Definition 5.4 doesn't tell us much, it only sets a baseline as to what we mean by properties. In particular, we do not restrict the class of ω-languages in any way (but we will very shortly).

Given a system and a property, the problem of deciding whether the system satisfies the property is called *model checking*. Without some restrictions on the properties, however, this problem is not entirely well defined.[7]

The first type of properties we can restrict to are ω-regular ones, given by NBWs.

NBW Model Checking:

Given:	A Kripke structure $\mathcal{K}$ and an NBW $\mathcal{A}$.
Question:	Is every computation of $\mathcal{K}$ accepted by $\mathcal{A}$?

Theorem 5.5. *The* NBW *model checking problem is* **PSPACE-Complete***.*

Proof. Observe that the NBW model checking problem actually asks whether $L(\mathcal{K}) \subseteq L(\mathcal{A})$. By Theorem 5.3 we can construct $\mathcal{A}_\mathcal{K}$ in polynomial time, and by Theorem 3.30, we can check whether $L(\mathcal{A}_\mathcal{K}) \subseteq L(\mathcal{A})$ in polynomial space.

For the lower bound, we easily reduce from the universality problem for NBW: given an NBW $\mathcal{A}$, assume w.l.o.g. its alphabet is of the form 2^P for $P = \{p_1, \dots, p_k\}$ (it is easy to adapt all our proofs to such settings). Then, construct a Kripke structure $\mathcal{K}_{\text{ALL}}$ such that $L(\mathcal{K}_{\text{ALL}}) = (2^P)^\omega$. This is easy to do: $\mathcal{K}_{\text{ALL}}$ is a complete graph on 2^P states, where each state has a unique label, and all the states are initial.

Then, $L(\mathcal{K}_{\text{ALL}}) \subseteq L(\mathcal{A})$ iff $L(\mathcal{A}) = (2^P)^\omega$.

Also, observe that this reduction is indeed in polynomial time, since computing $\mathcal{K}_{\text{ALL}}$ requires building $2^{|P|}$ states, but 2^P is the alphabet of $\mathcal{A}$, and so the description of $\mathcal{A}$ includes this, and it is thus linear in the size of the input. □

Remark 5.6. An important aspect of NBW model checking is that it relies on checking containment of NBWs. Recall from Theorem 3.30 that for containment, we only complement the NBW on the "right hand side", so we only have an exponential blow up there.

For model checking, this translates to the algorithm being exponential in the NBW (the specification), but not the system. This is very good news: typical systems are huge, but typical specifications are quite small. △

[7] the reason is that it is not clear how to specify the input. An algorithm cannot take as input a "language".

6 Linear Time Temporal Logic (LTL)

It is unnatural for humans to design automata. Indeed, systems are typically obtained by abstractions of code in various languages (e.g., VHDL for chip design, procedural or declarative languages for software, etc.). The same goes for specifying properties. Hence, there is a need for a more convenient specification formalism.

6.1 LTL – Syntax and Semantics

In 1977, Amir Pnueli proposed a logic for reasoning about (linear) time, called Linear Temporal Logic (LTL) [21]. This turned out to be a seminal contribution to the field of formal verification, for which Pnueli received the Turing award in 1996.

LTL allows us to specify properties of infinite computations over the alphabet 2^{AP} for a finite set AP of *atomic propositions*.

atomic propositions

Before diving into the formal syntax and semantics, let's start with an example.

Example 6.1. Recall our mutex example (Example 5.1). We'll write some LTL formulas that describe certain properties. The atomic propositions are $AP = \{p_1, p_2, p_3, p_4, t\}$.

- The property "infinitely often Process 0 reaches its critical section" is written as $\mathsf{GF}(p_1 \wedge \neg p_2)$, and reads as "Always Eventually p_1 and not p_2". This formula is satisfied in an infinite computation $\pi \in (2^{AP})^\omega$ iff in every suffix (i.e. always, G) there exists some point (i.e. eventually, F) where p_1 is true and p_2 is not true (i.e. we reached a code line starting with 10).
- The property "it is never the case that both processes are in their critical section" is written as $\mathsf{G}\neg(p_1 \wedge (\neg p_2) \wedge p_3 \wedge (\neg p_4))$. This is read as "Always it is not the case that..."
- The property "if Process 1 is waiting, and turn=1, then in the next step Process 1 is in its critical section" is written as
$$\mathsf{G}(((\neg p_3) \wedge p_4 \wedge t) \to \mathsf{X}(p_3 \wedge \neg p_4))$$
where X is read as "next" or "in the next step". Note that here, G was implicit in the English description of the property.
- The property "Always, if process 0 is in its critical section, then turn=0 until the process leaves the critical section" is written as
$$\mathsf{G}((p_1 \wedge (\neg p_2)) \to ((\neg t)\mathsf{U}(p_1 \wedge p_2)))$$
and read as "always, if p_1 and not p_2, then not t until $(p_1 \wedge p_2)$"
- The properties can of course also say things that are undesirable: $\mathsf{FG}((\neg p_1) \wedge p_2)$ says that from some point and on, Process 0 is waiting.

△

We are now ready for the formal definitions.

LTL *formula*

Definition 6.2 (LTL Syntax)**.** *Let AP be a finite set of atomic propositions. An* LTL formula φ *over AP is defined inductively as follows:*

- **true** *or p for $p \in AP$.*
- *$\neg\psi$, $\psi_1 \vee \psi_2$, $\mathsf{X}\psi_1$, or $\psi_1 \mathsf{U} \psi_2$, where ψ_1 and ψ_2 are* LTL *formulas.*

Equivalently, we can say that an LTL formula is given by the grammar

$$\varphi := \textbf{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \varphi\mathsf{U}\varphi$$

The semantics of LTL are defined with respect to an infinite computations over AP. Let $\pi = \sigma_1\sigma_2 \ldots \in (2^{AP})^\omega$, for $i \geq 1$ we define its suffix $\pi^i = \sigma_i\sigma_{i+1} \ldots \in (2^{AP})^\omega$.

Definition 6.3 (LTL Semantics)**.** *Let φ be an* LTL *formula over AP, and let $\pi = \sigma_1\sigma_2 \ldots \in (2^{AP})^\omega$.*
satisfies *We say that π* satisfies *φ, and write $\pi \models \varphi$, to indicate that ψ holds in the computation π.*

The relation $\models$ is defined inductively (on the structure of the formula) as follows.

- $\pi \models \textbf{true}$ *for all* π.
- $\pi \models p$ *for* $p \in AP$ *iff* $p \in \sigma_1$.
- $\pi \models \psi_1 \vee \psi_2$ *iff* $\pi \models \psi_1$ *or* $\pi \models \psi_2$.
- $\pi \models \mathsf{X}\psi_1$ *iff* $\pi^2 \models \psi_1$.
- $\pi \models \psi_1 \mathsf{U} \psi_2$ *iff there exists* $k \geq 1$ *such that* $\pi^k \models \psi_2$ *and for all* $1 \leq i < k$, $\pi^i \models \psi_1$.

The operators **true**, $\vee$, and $\neg$ are straightforward ("true", "or", and "not"). The atomic proposition p means "p is true now". Recall that the computation is over 2^{AP}, so each letter is in fact a subset of the atomic propositions. The *temporal* operator $\mathsf{X}\psi$, read as "next ψ", says that ψ will hold in the next step. The temporal operator $\psi_1\mathsf{U}\psi_2$, read as "ψ_1 until ψ_2", says that ψ_2 holds at some point in the future, and until then, ψ_1 holds.

In Example 6.1, we saw some additional operators, which we now define as abbreviations of existing operators. Specifically, we define F ("eventually"), G ("always"), and R ("release")[8].

Definition 6.4 (Additional Operators)**.**

- *The standard propositional operators $\wedge, \rightarrow, \leftrightarrow$, etc. are defined as usual (e.g., $\psi_1 \rightarrow \psi_2 \equiv \neg\psi_1 \vee \psi_2$).*
- $\mathsf{F}\psi \equiv \textbf{true}\mathsf{U}\psi$. *Then,* $\pi \models \mathsf{F}\psi$ *iff there exists* $k \geq 1$ *such that* $\pi^k \models \psi$.
- $\mathsf{G}\psi \equiv \neg\mathsf{F}\neg\psi$. *Then,* $\pi \models \mathsf{G}\psi$ *iff for every* $k \geq 1$ *we have that* $\pi^k \models \psi$.
- $\psi_1\mathsf{R}\psi_2 \equiv \neg((\neg\psi_1)\mathsf{U}(\neg\psi_2))$. *Then,* $\pi \models \psi_1\mathsf{R}\psi_2$ *iff for all* $k \geq 1$, *if* $\pi^k \not\models \psi_2$ *then there is* $1 \leq i < k$ *such that* $\pi^i \models \psi_1$.

Example 6.5. Let's see some simple LTL examples, over various sets of atomic propositions.

- $\mathsf{GF}p$ — p holds infinitely often.
- $\mathsf{FG}p$ — p holds from some point on.
- $\mathsf{G}(req \rightarrow \mathsf{X}ack \vee \mathsf{XX}ack)$ — every request is followed by an ack after one or two steps.
- $p \wedge \mathsf{G}(p \leftrightarrow \mathsf{XX}p)$ — p holds exactly in all odd indices.
- $\mathsf{G}(err \rightarrow (fix\mathsf{R}\neg operate))$ — once an error occurs, the system does not operate until it is fixed.
- $\neg hope \wedge \mathsf{F}bad$ — there is no hope, something bad will happen eventually.

△

language An LTL formula φ defines the *language* $L(\varphi) = \{\pi \in (2^{AP})^\omega : \pi \models \varphi\}$ of computations that satisfy it. We say that an LTL formula is *satisfiable* if $L(\varphi) \neq \emptyset$.

satisfiable
$\mathcal{K} \models \varphi$ We lift the definition of satisfaction from computations to Kripke structure. We say that a Kripke structure $\mathcal{K}$ satisfies an LTL formula φ, written $\mathcal{K} \models \varphi$, if $L(\mathcal{K}) \subseteq L(\varphi)$. That is, if every computation of $\mathcal{K}$ satisfies φ.

[8] $\psi_1\mathsf{R}\psi_2$ means, intuitively, that ψ_2 must hold either forever, or until ψ_1 holds.

6.2 Decision Problems for LTL

We defined LTL to help us formulate specifications. So the natural questions are whether we can use LTL for specification-checking procedures.

In particular, the following two problems are of interest:

LTL **Satisfiability:**

Given: LTL formula φ.
Question: Is φ satisfiable?

LTL **Model Checking:**

Given: LTL formula φ, Kripke Structure $\mathcal{K}$
Question: Does $\mathcal{K} \models \varphi$?

Remark 6.6. Notice that the model-checking problem has a "universal flavor" – *all* the computations of $\mathcal{K}$ must satisfy φ. This is somewhat arbitrary. We could have asked that *there exists* a computation that satisfies φ.

This, however, is completely dual: *there exists* a computation that satisfies φ iff $\mathcal{K} \not\models \neg\varphi$. Thus, for all intents and purposes, it's enough to consider the current definition. △

Recall that satisfiability amounts to deciding whether $L(\varphi) = \emptyset$, and model checking amounts to deciding whether $L(\mathcal{K}) \subseteq L(\varphi)$. Thus, in order to solve these questions, we need some tool to handle languages. Can you think of such a tool? if not, maybe a course on **automata**, logic, and games just isn't for you.

6.3 From LTL to NGBW

Naturally, we will show how to translate LTL to automata, and use the rich theory we've developed. Specifically, we will translate LTL to NGBW (nondeterministic generalized Büchi automata, that you saw in the exercise). Then, we can translate NGBW to NBW, if we so choose.

The construction is due to Vardi and Wolper [28], and is one of several known constructions. We will prove the following theorem.

Theorem 6.7. *Let φ be an* LTL *formula of size n, then there exists an* NGBW $\mathcal{A}_\varphi$ *with* $2^{O(n)}$ *states and index $O(n)$, such that $L(\varphi) = L(\mathcal{A}_\varphi)$*

Unlike some other constructions, the construction in the proof of Theorem 6.7 is not "magic". There is a very sensible intuition behind it, and understanding it gives good insight both into LTL and into automata. Naturally, this is said in hindsight, and coming up with the construction initially requires some brilliance.

We give some intuition before diving into the proof. The following doesn't have to really make sense at this point, but read through it nonetheless, as it might make the actual proof more understandable.

How does a path π satisfy an LTL formula? Assume you (the reader) are π.

- If the formula says that p should be seen now, then you better have p in your current letter.
- If it says that $p \vee (\neg q)$, then you should have p in your current letter, or don't have q in it.
- What if it says Xp? Well then, you're ok for now, no matter what you have, but in the next step you really should have p.
- And what about $p\mathsf{U}q$? Well, this is really interesting. If you have q, you're all good. If you don't have q and you don't have p, I'm afraid it's game over for you. But what if you have p? Well, you do get to live another day, but you're not in the clear yet. In the next step, you still need to satisfy $p\mathsf{U}q$.

`delayed U property`

This last point is closely related to the following useful identity, called (in these notes) the *delayed* U *property*:

$$\psi_1\mathsf{U}\psi_2 \;\equiv\; \psi_2 \vee (\psi_1 \wedge \mathsf{X}(\psi_1\mathsf{U}\psi_2))$$

which shows how U can be "delayed" to the next step.

This is all lovely, but you probably don't see yet where it's going. The point of this is that most of the checks the automaton needs to do are *local*: either make sure something is true now, or in the next step. This is right up the alley for automata. The only problem is how to handle these pesky U operators. We'll get to that later.

There is, however, another idea incorporated into this construction: consider the formula $\mathsf{X}p$. It is satisfied if the second letter has p. But what should we check for in the first letter? Intuitively – nothing. This, however, makes things slightly complicated, since we need to tell the automaton that at some points we "don't care" about the values of certain propositions. It gets messy. To overcome this, the Vardi-Wolper construction does something clever: the automaton "commits" to a certain value for each proposition, and moreover – for each *subformula.* It guesses, at each stage, exactly which subformulas hold. Then, it makes sure that the guesses are consistent with what is actually read, using the locality principles mentioned above.

A bit vague? Sure. Let's make it precise.

The Construction

For an LTL formula φ, the *closure* of φ, denoted $\mathrm{cl}(\varphi)$, the set of subformulas of φ. For example, $\mathrm{cl}(\mathsf{X}(p \vee q)\mathsf{U}\neg q) = \{\mathsf{X}(p \vee q)\mathsf{U}\neg q, (p \vee q)\mathsf{U}\neg q, p \vee q, \neg q, p, q\}$. Then, the *extended closure* of φ, denoted $\mathrm{xcl}(\varphi)$, is the set of all the subformulas and their negations. In our example, $\mathrm{xcl}(\mathsf{X}(p \vee q)\mathsf{U}\neg q) = \{\mathsf{X}(p \vee q)\mathsf{U}\neg q, \neg\mathsf{X}(p \vee q)\mathsf{U}\neg q, (p \vee q)\mathsf{U}\neg q, \neg(p \vee q)\mathsf{U}\neg q, p \vee q, \neg p \vee q, \neg q, \neg p, p, q\}$.

extended closure

Formally, we define it as follows.

Definition 6.8 (Extended Closure)**.** *Let φ be an LTL formula over AP. $\mathrm{xcl}(\varphi)$ is the smallest set of formulas that satisfies the following:*

- $\varphi \in \mathrm{xcl}(\varphi)$.
- *If* $\psi \in \mathrm{xcl}(\varphi)$ *then* $\neg\psi \in \mathrm{xcl}(\varphi)$ *(we identify* $\neg\neg\psi$ *with* ψ*).*
- *If* $\neg\psi \in \mathrm{xcl}(\varphi)$ *then* $\psi \in \mathrm{xcl}(\varphi)$.
- *If* $\psi_1 \vee \psi_2 \in \mathrm{xcl}(\varphi)$ *then* $\psi_1 \in \mathrm{xcl}(\varphi)$ *and* $\psi_2 \in \mathrm{xcl}(\varphi)$.
- *If* $\mathsf{X}\psi \in \mathrm{xcl}(\varphi)$ *then* $\psi \in \mathrm{xcl}(\varphi)$.
- *If* $\psi_1\mathsf{U}\psi_2 \in \mathrm{xcl}(\varphi)$ *then* $\psi_1 \in \mathrm{xcl}(\varphi)$ *and* $\psi_2 \in \mathrm{xcl}(\varphi)$.

Consider a computation $\pi = \sigma_1\sigma_2 \ldots \in (2^{AP})^\omega$. For every $i \geq 1$, we can think of the set of formulas $S \subseteq \mathrm{xcl}(\varphi)$ that are true in the suffix π^i. Observe that the set S cannot be completely arbitrary. For example, exactly one of $p \in S$ or $\neg p \in S$ holds, since either $p \in \sigma_1$ or $p \notin \sigma_1$. This motivates the following definition:

Definition 6.9 (Propositional Consistency)**.** *A set $S \subseteq \mathrm{xcl}(\varphi)$ is* propositionally consistent *if:*

propositionally consistent

- *for all* $\psi \in \mathrm{xcl}(\varphi)$ *we have* $\psi \in S$ *iff* $\neg\psi \notin S$.
- *for all* $\psi_1 \vee \psi_2 \in \mathrm{xcl}(\varphi)$ *we have* $\psi_1 \vee \psi_2 \in S$ *iff* $\psi_1 \in S$ *or* $\psi_2 \in S$.

We are now ready to describe the construction. We build the NGBW $\mathcal{A} = \langle Q, 2^{AP}, Q_0, \delta, \alpha\rangle$ as follows.

The set of states $Q \subseteq 2^{\mathrm{xcl}(\varphi)}$ is the set of all propositionally-consistent subsets of $\mathrm{xcl}(\varphi)$. The transitions are defined as follows. Let $S, S' \in Q$ and let $\sigma \in 2^{AP}$. Then $S' \in \delta(S, \sigma)$ iff all the following hold:

- $\sigma = S \cap AP$. That is, $p \in \sigma$ iff $p \in S$ for all $p \in AP$.
- For all $\mathsf{X}\psi \in \mathrm{xcl}(\varphi)$ we have $\mathsf{X}\psi \in S$ iff $\psi \in S'$.
- For all $\psi_1\mathsf{U}\psi_2 \in \mathrm{xcl}(\varphi)$, we have $\psi_1\mathsf{U}\psi_2 \in S$ iff either $\psi_2 \in S$ or $\psi_1 \in S$ and $\psi_1\mathsf{U}\psi_2 \in S'$.

The set of initial states Q_0 is the set of all $S \in Q$ such that $\varphi \in S$.

Finally, for each subformula $\psi_1 \mathsf{U} \psi_2 \in \mathrm{xcl}(\varphi)$ we add to the generalized Büchi condition α the set

$$\alpha_{\psi_1 \mathsf{U} \psi_2} = \{S \in Q \,:\, \psi_2 \in S \text{ or } \neg(\psi_1 \mathsf{U} \psi_2) \in S\}$$

Observe that we can equivalently write

$$\alpha_{\psi_1 \mathsf{U} \psi_2} = \{S \in Q \,:\, \psi_1 \mathsf{U} \psi_2 \in S \implies \psi_2 \in S\}$$

which is arguably the "correct" way of viewing $\alpha_{\psi_1 \mathsf{U} \psi_2}$. Intuitively, it means that "if $\psi_1 \mathsf{U} \psi_2$ should hold, then its eventuality is satisfied right now."

We now turn to prove correctness and analyze the size of the construction.

The size is easy: the number of states is $2^{O(|\varphi|)}$, since the states are subsets of $\mathrm{xcl}(\varphi)$, whose size is at most $2|\varphi|$. Recall, however, that in NGBW we also care about the *index*. Since each U subformula contributes a single accepting set, then the total index is at most $|\varphi|$.

We can now complete the proof.

Lemma 6.10. *$L(\mathcal{A}) = L(\varphi)$*

Proof. We now prove the correctness of the construction. First, consider a word $\pi = \sigma_1 \sigma_2 \ldots \in (2^{AP})^\omega$. For each suffix π^i of π, let $\lambda(\pi^i) = \{\psi \in \mathrm{xcl}(\varphi) \,:\, \pi^i \models \psi\}$ be the set of formulas in $\mathrm{xcl}(\varphi)$ that are satisfied in π^i. We then associate with π the infinite sequence $\Lambda(\pi) = \lambda(\pi^1)\lambda(\pi^2)\ldots$.

Note that clearly each $\lambda(\pi^i)$ is a propositionally consistent set, so we can think of it as a state of $\mathcal{A}$.

For the first direction, we claim that if $\pi \models \varphi$, then $\Lambda(\pi)$ is an accepting run of $\mathcal{A}$ on π. This is easy to show: since $\varphi \in \lambda(\pi^1)$, then $\lambda(\pi^1) \in Q_0$. Furthermore, we always have $\lambda(\pi^{i+1}) \in \delta(\lambda(\pi^i), \sigma_i)$, since:

- We trivially have that $\lambda(\pi^i) \cap AP = \sigma_i$,
- for all $\mathsf{X}\psi \in \mathrm{xcl}(\varphi)$, $\mathsf{X}\psi \in \lambda(\pi^i)$ iff $\psi \in \lambda(\pi^{i+1})$, and
- for every $\psi_1 \mathsf{U} \psi_2 \in \mathrm{xcl}(\varphi)$, by the "delayed until" property we have that $\psi_1 \mathsf{U} \psi_2 \in \lambda(\pi^i)$ iff either $\psi_2 \in \lambda(\pi^i)$, or $\psi_1 \in \lambda(\pi^i)$ and $\psi_1 \mathsf{U} \psi_2 \in \lambda(\pi^{i+1})$

so the conditions for a transition hold. Finally, $\Lambda(\pi)$ is an accepting run. Indeed, for every $\psi_1 \mathsf{U} \psi_2 \in \mathrm{xcl}(\varphi)$, whenever $\psi_1 \mathsf{U} \psi_2 \in \lambda(\pi^i)$ then by the definition of U, there exists some $k \geq i$ such that $\psi_2 \in \lambda(\pi^k)$ (and hence also $\psi_1 \mathsf{U} \psi_2 \in \lambda(\pi^k)$, otherwise there aren't any outgoing transitions from $\lambda(\pi^k)$) and $\psi_1 \in (\lambda(\pi^j))$ (and hence also $\psi_1 \mathsf{U} \psi_2 \in \lambda(\pi^j)$) for every $i \leq j < k$. Thus, if $\psi_1 \mathsf{U} \psi_2 \in \lambda(\pi^i)$ for infinitely many i's, then a state in $\alpha_{\psi_1 \mathsf{U} \psi_2}$ is visited infinitely often, and otherwise, $\alpha_{\psi_1 \mathsf{U} \psi_2}$ is visited always from some point on. We conclude that $\pi \in L(\mathcal{A})$.

For the converse direction, assume $\pi \in L(\mathcal{A})$, and let $\rho = \rho_1, \rho_2, \ldots$ be an accepting run of $\mathcal{A}$ on π. We claim that $\rho = \Lambda(\pi)$. To prove this, we prove that for every $i \geq 0$ and for every $\psi \in \mathrm{xcl}(\varphi)$ we have $\psi \in \rho_i$ iff $\psi \in \lambda(\pi^i)$ (that is, iff $\pi^i \models \psi$). We prove this by induction over the structure of ψ.

- If $\psi = p \in AP$, then clearly $p \in \rho_i$ iff $p \in \sigma_i$ iff $\pi^i \models p$ (otherwise there wouldn't be an outgoing transition from ρ_i with σ_i).
- If $\psi = \neg\psi'$ or $\psi = \psi_1 \vee \psi_2$, the claim follows immediately by the induction hypothesis (and by the consistency requirement of the states of $\mathcal{A}$).
- If $\psi = \mathsf{X}\psi'$, then $\mathsf{X}\psi' \in \rho_i$ iff $\psi' \in \rho_{i+1}$, which by the induction hypothesis happens iff $\pi^{i+1} \models \psi'$, so $\pi^i \models \mathsf{X}\psi'$.
- If $\psi = \psi_1 \mathsf{U} \psi_2$ (the interesting case), consider ρ_i. Since ρ is accepting, there exists $k \geq i$ such that $\rho_k \in \alpha_{\psi_1 \mathsf{U} \psi_2}$. Let k be minimal with this property. Then, by a simple inductive argument, we see that $\psi_1 \mathsf{U} \psi_2 \in \rho_i$ iff for every $i \leq j < k$ it holds that $\psi_1 \in \rho_j$ and $\psi_1 \mathsf{U} \psi_2 \in \rho_j$, and in addition, $\psi_1 \mathsf{U} \psi_2 \in \rho_k$. This happens iff $\psi_2 \in \rho_k$ (since $\rho_k \in \alpha_{\psi_1 \mathsf{U} \psi_2}$), and applying the induction hypothesis on ψ_1 and ψ_2 shows that this is equivalent to $\pi^i \models \psi_1 \mathsf{U} \psi_2$.

Since $\lambda(\pi^1) \in Q_0$, then $\varphi \in \lambda(\pi^1)$, so by the above we have $\pi \models \varphi$. □

Remark 6.11. As the proof of Lemma 6.10 shows, it is crucial that the run is accepting in order for it to reflect $\Lambda(\pi)$ correctly. Indeed, for example a run on the formula $p \vee \mathsf{F}q$ can "wrongly" guess that the formula is satisfied due to $p \wedge \mathsf{F}q$ being satisfied, rather than just p, and wait in a non accepting loop without visiting q, even though a satisfying computation, e.g. p^ω was given.

△

Since the translation from NGBW (with size n and index k) to NBW results in an NBW with $n \cdot k$ states, we have the following.

Theorem 6.12. *Let φ be an* LTL *formula of size n, then there exists an* NBW $\mathcal{A}_\varphi$ *with* $2^{O(n)}$ *states such that* $L(\varphi) = L(\mathcal{A}_\varphi)$

Remark 6.13. Observe that the construction above is easy to perform *on-the-fly*. As usual, this will become useful when using it in space-saving algorithms. △

Lower Bound on LTL → NBW

The construction in Theorem 6.12 involves an exponential blowup. The immediate question is: is this blowup necessary? The answer (of course) is yes, as we shall now show.

The proof is a bit indirect. We will actually show the following.

Theorem 6.14. *The translation from* LTL *to* DPW *(or* DRW *or* DSW*) is double-exponential.*

Recall that the translation from NBW to DPW (or DRW or DSW) has a blowup of $2^{O(n \log n)}$. Thus, if there is a translation from LTL to NBW with a sub-exponential blowup, we would get a contradiction to Theorem 6.14. Thus, the above would give us the following.

Theorem 6.15. *The translation from* LTL *to* NBW *is exponential.*

Proof of Theorem 6.14. We construct a family of languages L_n such that L_n can be defined by an LTL formula of size $O(n^2)$, and every DPW for L_n requires at least 2^{2^n} states.

Let $\Sigma = \{0, 1, \#, \$\}$. Observe that Σ is not of the form 2^{AP}, but it can easily be described as such using two atomic propositions. Define

$$L_n = \{\{0, 1, \#\}^* \cdot \# \cdot x \cdot \# \cdot \{0, 1, \#\}^* \cdot \$ \cdot x \cdot \#^\omega \ : \ x \in \{0, 1\}^n\}$$

That is, a word w is in L_n if after the single occurrence of \$, the word of length n over $\{0, 1\}$ has appeared before, between two #'s.

We start by proving that every DPW for L_n needs at least 2^{2^n} states (this is adapted from [4]). Assume by way of contradiction that $\mathcal{D}$ is a DPW for L_n with less than 2^{2^n} states. For every $S \in 2^{\{0,1\}^n}$, thought of as a set $S = \{y_1, y_2, \ldots, y_k\}$ of words of length n over $\{0, 1\}$), define $u_S = \#y_1\#y_2\# \cdots \#y_k\#\$$.

By our assumption, there are two distinct sets S, R such that the run of $\mathcal{D}$ on u_S and on u_R reaches the same state q. W.l.o.g. let $x \in S \setminus R$, then $u_S \cdot x\#^\omega \in L_n$, but $u_R \cdot x\#^\omega \notin L_n$, which is a contradiction.

We now show how to define L_n in LTL. Denote by X^i the nesting $\underbrace{\mathsf{XX}\ldots\mathsf{X}}_{i \text{ times}}$. We construct a formula $\varphi = \psi_1 \wedge \psi_2$ as follows:

- ψ_1 states that \$ occurs exactly once, and is followed by n letters from $\{0, 1\}$ and then $\#^\omega$. Formally,

$$\psi_1 = (\neg\$)\mathsf{U}(\$ \wedge \bigwedge_{i=1}^{n} \mathsf{X}^i(0 \vee 1) \wedge \mathsf{X}^{n+1}\mathsf{G}\#)$$

- ψ_2 states that some string of length n, delimited between #'s before \$ occurs also immediately after \$. Formally,

$$\psi_2 = \mathsf{F}(\# \wedge \mathsf{X}^{n+1}\# \wedge \bigwedge_{i=1}^{n} (\mathsf{X}^i 0 \longleftrightarrow \mathsf{G}(\$ \to \mathsf{X}^i 0) \wedge (\mathsf{X}^i 1 \longleftrightarrow \mathsf{G}(\$ \to \mathsf{X}^i 1))))$$

□

6.4 Solving the Decision Problems

In Section 6.2 we defined the two central decision problems for LTL: satisfiability and model-checking. We are now ready to tackle them. We start with upper bounds.

Lemma 6.16. *The* LTL*-satisfiability problem is in* **PSPACE**.

Proof. An LTL formula φ is satisfiable iff $L(\mathcal{A}_\varphi) \neq \emptyset$, where $\mathcal{A}_\varphi$ is the NBW obtained as per Theorem 6.12. By Remark 6.13, we can construct $\mathcal{A}_\varphi$ on the fly, and check emptiness in **NL**. Thus, the overall complexity is **coNPSPACE** = **PSPACE**. □

Lemma 6.17. *The* LTL *model-checking problem is in* **PSPACE**.

Proof. Given a Kripke structure $\mathcal{K}$ and an LTL formula φ, we have $\mathcal{K} \models \varphi$ iff $L(\mathcal{K}) \subseteq L(\mathcal{A}_\varphi)$, where $\mathcal{A}_\varphi$ is the NBW obtained as per Theorem 6.12.

A naive approach would be to construct $\mathcal{A}_\varphi$, then complement it and check whether $L(\mathcal{K}) \cap \overline{L(\mathcal{A}_\varphi)} = \emptyset$. Unfortunately, this would incur a double-exponential blowup: one for LTL → NBW, and one for complementing the NBW.

In order to circumvent this, we instead construct the NBW $\mathcal{A}_{\neg\varphi}$ for the negated formula. Note that $L(\mathcal{A}_{\neg\varphi}) = \overline{L(\mathcal{A}_\varphi)}$, but $|\neg\varphi| = |\varphi| + 1$, so the size of $\mathcal{A}_{\neg\varphi}$ is still $2^{O(n)}$.

Thus, we can proceed with checking whether $L(\mathcal{K}) \cap L(\mathcal{A}_{\neg\varphi}) = \emptyset$, which can be done in **PSPACE** by constructing $\mathcal{A}_{\neg\varphi}$ on the fly. □

Exercise 6.18. Show that in fact, $\mathcal{A}_{\neg\varphi}$ has the same number of states as $\mathcal{A}_\varphi$.

We now proceed to show a **PSPACE** lower bound. We start with model-checking.

Lemma 6.19 ([26])**.** *The* LTL *model-checking problem is* **PSPACE***-hard.*

Proof. We reduce from the canonical problem of space-bounded TM acceptance. Let

$$L = \{\langle M, w, 1^s\rangle \,:\, M \text{ accepts } w \text{ using at most s cells}\}$$

Given $\langle M, w, 1^s\rangle$ as above, we will construct a Kripke structure $\mathcal{K}$ and a formula φ such that M accepts w using at most s cells iff *there exists* a computation of $\mathcal{K}$ that satisfies φ. Thus, the reduction is in fact to the complement of model checking (or alternatively, we can just output $\neg\varphi$ and reduce from $\overline{L}$).

Let $M = \langle Q, \Sigma, \delta, q_0, q_{acc}, q_{rej}\rangle$ be the given TM. Recall that a *configuration* of M is a string in $(\Sigma \cup Q \times \Sigma)^*$, representing for each cell which letter it has, and whether it has the head (and what the state is). The Kripke structure $\mathcal{K}$ is defined over the atomic propositions $AP = \{p_\sigma \,:\, \sigma \in \Sigma \cup Q \times \Sigma\} \cup \{\vdash, \dashv\}$, and is depicted in Figure 13.

The idea is that $\mathcal{K}$ generates sequences of configurations of length s. We then construct φ to state the following properties:

1. The first sequence represents the initial configurations (i.e. $\vdash \wedge \mathsf{X}(q_0, w_1) \wedge \bigwedge_{i=2}^{s} \mathsf{X}^{2i-1} w_i$).
2. Each configuration follows correctly from the previous configuration. This is done using disjunctions over the possible transitions, and using $2s$ X operators to compare one entry of the configuration with the corresponding entry on the next configuration.
3. q_{acc} appears in some configuration, after which no more configurations appear (we should loop on $\dashv$).

It is not hard to write this formula explicitly (but too painful for these notes). □

Finally, in order to show hardness of satisfiability, we can take the same approach as in Lemma 6.19. Instead, we actually reduce MC to satisfiability (note that this gives an alternative proof of Lemma 6.17).

Lemma 6.20 ([26])**.** LTL *model checking reduces in polynomial time to* LTL *non-satisfiability.*

Proof. Given a Kripke structure $\mathcal{K} = \langle S, R, \mathscr{L}, S_0\rangle$ and a formula φ over atomic propositions AP, we construct an LTL formula Ψ over the atomic propositions $AP \cup S$ (i.e. the atomic propositions of φ, and the states of S) such that $\mathcal{K} \not\models \varphi$ iff Ψ is satisfiable.

The idea is that Ψ states the following: "there is a path in $\mathcal{K}$ whose computation does not satisfy φ". We construct $\Psi = \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \neg\varphi$ where

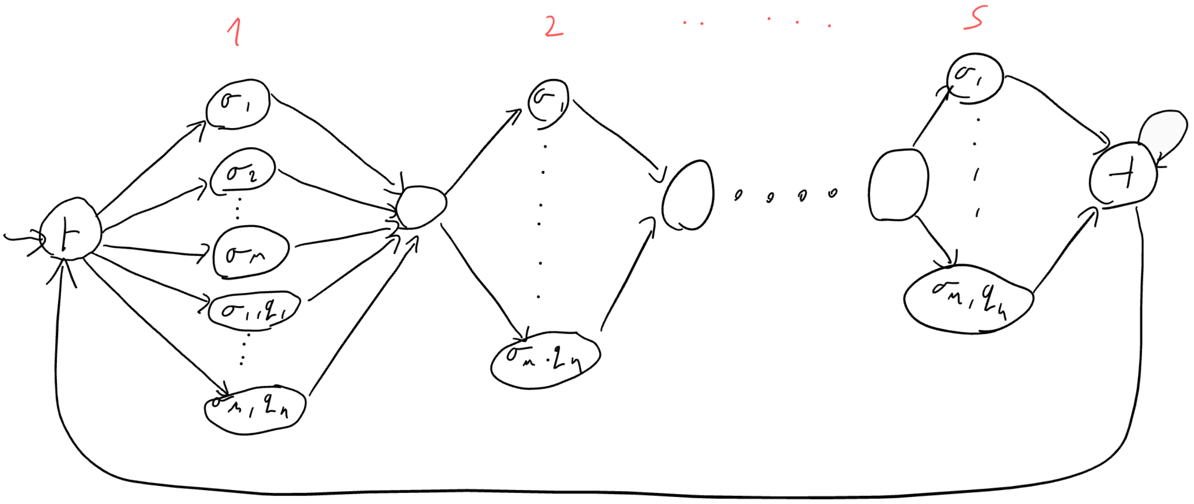


Figure 13: The Kripke structure $\mathcal{K}$.

1. $\psi_1 = \bigvee_{s \in S_0} s$. That is, the initial state is in S_0.
2. $\psi_2 = \mathsf{G} \bigwedge_{s \in S}(s \to \mathsf{X} \bigvee_{s' : R(s,s')} s')$. That is, the transitions follow those of $\mathcal{K}$.
3. $\psi_3 = \mathsf{G} \bigwedge_{s \in S}(s \to \bigwedge_{p \in \mathscr{L}(s)} p \wedge \bigwedge_{p \notin \mathscr{L}(s)} \neg p)$. That is, the atomic propositions in AP that hold in each state s, correspond exactly to $\mathscr{L}(s)$.

We now have that Ψ is satisfiable iff there exists a description of an infinite path through $\mathcal{K}$ whose corresponding computation does not satisfy φ, iff $\mathcal{K} \not\models \varphi$. □

We conclude with the following.

Theorem 6.21. *The* LTL *satisfiability and model-checking problems are* **PSPACE** *complete.*

6.5 Expressive power of LTL

In Theorem 6.12 we saw that every LTL formula has an equivalent NBW. In particular, $L(\varphi)$ is ω-regular for every φ. This begs the question – is the converse true? Is every ω-regular language LTL-definable?

Unfortunately, the answer is no. We prove this by describing a condition that is satisfied by every LTL definable language, and then demonstrating ω-regular languages that do not satisfy it.

non-counting

Definition 6.22 (non-counting)**.** *We say that a language $L \subseteq \Sigma^\omega$ is* non-counting *if there exists $n_0 \geq 0$ such that for every $n \geq n_0$ and for every $u, v \in \Sigma^*$ and $\beta \in \Sigma^\omega$ we have $uv^n\beta \in L \iff uv^{n+1}\beta \in L$.*

Simply put, a language is non-counting if, when you repeat some finite infix enough times, then it doesn't matter how many times you repeat it.

counting

A language is called *counting* if it is not non-counting.

Example 6.23. A couple of examples:

- The language $L_1 = \{w \in \{0,1\}^\omega : w \text{ has infinitely many 1's}\}$ is non-counting. Indeed, take $n_0 = 1$, then for every $u, v \in \Sigma^*$ and $\beta \in \Sigma^\omega$ we have that $uv^k\beta \in L_1$ iff β has infinitely many 1's, and in particular this is true for $k = n$ and $k = n + 1$ for every n.
- The language $L_2 = (00)^*1^\omega$ is not non-counting: for every $n_0 \in \mathbb{N}$ we have $0^{2n_0}1^\omega \in L_2$ but $0^{2n_0+1}1^\omega \notin L_2$.

△

Theorem 6.24. *Every* LTL *definable language is non counting.*

Proof. Let φ be an LTL formula over AP. We prove that $L(\varphi)$ is non-counting by induction over the structure of φ.

- If $\varphi = \textbf{true}$, set $n_0 = 0$, the condition holds trivially.
- If $\varphi = p$ for $p \in AP$, set $n_0 = 1$, then for every $u, v \in (2^{AP})^\star$ and $\beta \in (2^{AP})^\omega$, if $u \neq \epsilon$, then $uv^n\beta \in L(\varphi)$ iff u starts with p, regardless of n. If $u = \epsilon$ (and $v \neq \epsilon$, otherwise the claim is trivial), then for every $n \geq 1$, $v^n\beta \in L(\varphi)$ iff v starts with p, iff $v^{n+1}\beta \in L(\varphi)$.
- If $\varphi = \psi_1 \vee \psi_2$, let n_1, n_2 be the thresholds defined for ψ_1 and ψ_2 by the induction hypothesis, then for $n_0 = \max\{n_1, n_2\}$ the claim follows easily.
- If $\varphi = \neg\psi_1$, let n_1 be the thresholds defined for ψ_1 by the induction hypothesis, then n_1 clearly also works as a threshold for φ.
- If $\varphi = \mathsf{X}\psi$, let n_0 be the thresholds defined for ψ by the induction hypothesis. Let $n_1 = n_0 + 1$, and let $u, v \in (2^{AP})^\star$ and $\beta \in (2^{AP})^\omega$. Denote $x = (uv)^2$ the second suffix of uv, then for $n \geq n_1 = n_0 + 1$ we have
$$uv^n\beta \in L(\mathsf{X}\psi) \iff uvv^{n-1}\beta \in L(\mathsf{X}\psi) \iff xv^{n-1}\beta \in L(\psi) \iff$$
$$\iff xv^n\beta \in L(\psi) \iff uvv^n\beta \in L(\mathsf{X}\psi) \iff uv^{n+1}\beta \in L(\mathsf{X}\psi)$$
- $\varphi = \psi_1 \mathsf{U} \psi_2$, let n_1, n_2 be the thresholds defined for ψ_1 and ψ_2 by the induction hypothesis. take $n_0 = \max\{n_1, n_2\} + 1$, and let $u, v \in (2^{AP})^\star$ and $\beta \in (2^{AP})^\omega$.

  Claim 1: For all $n \geq n_0$, $uv^n\beta \models \psi_1 \mathsf{U} \psi_2 \implies uv^{n+1}\beta \models \psi_1 \mathsf{U} \psi_2$.

  If $uv^n\beta \models \psi_1 \mathsf{U} \psi_2$ then there exists $j \geq 0$ such that $(uv^n\beta)^j \models \psi_2$ and for every $0 \leq i < j$, $(uv^n\beta)^i \models \psi_1$. We now split to cases.

  – If $j \leq |u|$, then $(uv^n\beta)^j = u^jv^n\beta$, so by the induction hypothesis we have $u^jv^{n+1}\beta \models \psi_2$ and similarly, for every $i \leq j$ we have $u^iv^{n+1}\beta \models \psi_1$, so $uv^{n+1}\beta \models \psi_1 \mathsf{U} \psi_2$.

  – If $j > |u|$, then $(uv^{n+1}\beta)^{j+|v|} \models \psi_2$. Also, for $|u| + |v| \leq i < j + |v|$, we have $(uv^{n+1}\beta)^i \models \psi_1$. Finally, for $i < |u| + |v|$, $(uvv^n\beta)^i \models \psi_1$, since $(uvv^{n-1}\beta)^i \models \psi_1$ and by the induction hypothesis. We again conclude that $uv^{n+1}\beta \models \psi_1 \mathsf{U} \psi_2$.

  Claim 2: For all $n \geq n_0$, $uv^{n+1}\beta \models \psi_1 \mathsf{U} \psi_2 \implies uv^n\beta \models \psi_1 \mathsf{U} \psi_2$.

  If $uv^{n+1}\beta \models \psi_1 \mathsf{U} \psi_2$ then there exists $j \geq 0$ such that $(uv^{n+1}\beta)^j \models \psi_2$ and for every $0 \leq i < j$, $(uv^{n+1}\beta)^i \models \psi_1$. We now split to cases.

  – If $j \leq |u|+|v|$, then $(uv^{n+1}\beta)^j = (uv)^jv^n\beta$, so by the induction hypothesis we have $(uv)^jv^{n-1}\beta \models \psi_2$ and similarly, for every $i \leq j$ we have $(uv)^iv^{n-1}\beta \models \psi_1$, so $uvv^{n-1}\beta \models \psi_1 \mathsf{U} \psi_2$.

  – If $j > |u| + |v|$, then $(uv^n\beta)^{j-|v|} \models \psi_2$. Also, for $|u| \leq i < j - |v|$, we have $(uv^n\beta)^i = (uv^{n+1}\beta)^{i+|v|} \models \psi_1$.
  Finally, for $i < |u|$, $(uv^n\beta)^i \models \psi_1$, since $(uv^{n+1}\beta)^i = (u)^iv^{n+1}\beta \models \psi_1$ and by the induction hypothesis. We again conclude that $uv^n\beta \models \psi_1 \mathsf{U} \psi_2$.

□

We are now ready to show an ω-regular language that is not LTL definable.

Example 6.25 (LTL non definable, ω-regular language)**.** Let $AP = \{p\}$, and consider the language $L = \{w \in (2^{\{p\}})^\omega \, : \, p \text{ occurs in all even places}\}$. Note that p may or may not occur in odd places.

It is very easy to construct an NBW with two states for L. Thus, L is ω-regular. However, L is counting. Indeed, let $n_0 \in \mathbb{N}$, then $\{p\}^{2n_0} \cdot (\emptyset \cdot \{p\})^\omega \in L$, but $\{p\}^{2n_0+1} \cdot (\emptyset \cdot \{p\})^\omega \notin L$. △

7 S1S and Büchi's Theorem

S1S is the *monadic second order logic of one-successor*, and it is equivalent in expressive power to NBWs. But let's start at the beginning.

7.1 A Very Short Introduction to Logics

Recall that *propositional logic*, also called 0^{th}*-order logic*, allows us to connect atomic propositions using Boolean connectives $(\vee, \wedge, \neg)$ and has relation symbols (predicates) such as (Love(you,me)) and possibly other function symbols (apart from the Boolean connectives).

First-order logic extends propositional logic by allowing us to quantify elements in the domain. For example, if our domain is $\mathbb{N}$, we can consider the first-order logic of natural numbers with addition, given by the *signature* $\langle \mathbb{N}, 0, 1, + \rangle$ where the only relation is $=$, and the only function symbol is $+$, with the standard semantics. This logic is called *Presburger Arithmetic*, and plays an important role in many CS areas. In first order logic, we can write formulas such as $\varphi(x, y) : \quad \exists z \ . \ x + z = y$. The "intuitive" meaning of this formula is that $\varphi(x, y)$ iff $x < y$. In this example, we used the quantification $\exists z$, where z ranges over the domain $\mathbb{N}$. The *theory* of the logic is the set of (fully quantified) sentences in the logic that evaluate to **true**. For example, the sentence $\forall x \exists y \ . \ x + 1 = y$ is in the theory of Presburger arithmetic.

Second-order logic extends quantification even further: instead of only quantifying over elements in the domain, we can now quantify over *relations.* For example, we can say something like $\forall R \subseteq \mathbb{N} \times \mathbb{N} \forall x \in \mathbb{N} \ . \ R(x, x)$, which states that "every binary relation is reflexive", which is of course false.

Monadic Second-order logic (MSO) restricts the above a bit, by allowing us to only quantify over "unary" relations, i.e., sets. We can still say fairly interesting stuff, such as $\forall X \subseteq \mathbb{N}, \exists x \forall y (x \in X \wedge y \in X) \to x \leq y$, which states that every set has a minimal element.

This short introduction discusses logics in general. In our case, we want to discuss a logic called S1S that expresses properties of infinite words. We remark that there are several reasons for studying this logic:

- Historically, the connection between S1S and NBWs was the first time logic was connected to automata, and tools from automata theory proved to be of great importance.
- Many problems can be reformulated in S1S, and thus we get an immediate solution for them.
- MSO is a more "logicians' logic" than LTL, and it is worth knowing in order to read results formulated from a logic perspective.

7.2 The logic S1S

S1S
successor

The logic S1S (*monadic second order logic of one-successor*) is an MSO with the signature $\langle \mathbb{N}, \boldsymbol{s}, \in \rangle$, where $\boldsymbol{s}$ is the *successor* function[9] $\boldsymbol{s}x = x + 1$, and $\in$ is the membership relation of elements in sets.

Example 7.1 (S1S Example)**.** Before we formally define S1S, lets see an example. Consider the following formula

$$\varphi(P_a) := \exists X.(\exists z.z \in X) \wedge (\forall y.y \in X \to (y \in P_a \wedge \boldsymbol{s}y \in X))$$

What does φ "mean"? Well, think of $P_a \subseteq \mathbb{N}$ as describing an infinite word over $2^{\{a\}}$, where P_a are the indices of where a occurs, then $\varphi(P_a)$ is true iff there exists some set X of "indices", that contains some index, and from that index and on, contains all indices (the operator "$\boldsymbol{s}$" means "successor", that is, $\boldsymbol{s}y$ is "$y + 1$"), and X is contained in P_a.

Or, in simpler terms, $\varphi(P_a)$ is true iff from some point on a is true in the corresponding word. △

We are now ready to define S1S formally.

Definition 7.2 (S1S Syntax)**.** *The logical system* S1S *comprises the following elements.*

Variables

- Variables*: 1st-order variables (denoted by small letters x, y, z, etc.) range over* $\mathbb{N}$.
  2nd-order variables (denoted by capital letters X, Y, Z, etc.) range over subsets of $\mathbb{N}$.

Terms

- Terms*: 1st-order variables are terms, and if t is a term then so is* $\boldsymbol{s}t$.

Atomic formulas

- Atomic formulas*: expressions of the form $t \in X$ where t is a term and X is a 2nd-order variable.*

[9] observe that we write $\boldsymbol{s}x$ and not $\boldsymbol{s}(x)$, for convenience.

- Formulas*:* S1S *formulas are built up from atomic formulas using standard Boolean connectives and the quantifier* $\exists$ *over 1st and 2nd-order variables.*

Observe that the atomic formulas are all of the form $\boldsymbol{ss}\cdots\boldsymbol{s}x \in X$ for some 1st order variable x and 2nd order variable X.

We now want to define the semantics of S1S. We write formulas as $\varphi(x_1,\ldots,x_m,X_1,\ldots,X_n)$, where $\bar{x}=(x_1,\ldots,x_m)$ are free (i.e., not bound by a quantifier) 1st-order variables and $\bar{X}=(X_1,\ldots,X_n)$ are free 2nd-order variables.

Let $\bar{a}=(a_1,\ldots,a_m)\in\mathbb{N}^m$ and $\bar{P}=(P_1,\ldots,P_n)\in(2^{\mathbb{N}})^n$ (think of $\bar{a}$ and $\bar{P}$ as assignments to the free variables), then we write $(\bar{a},\bar{P})\models\varphi(\bar{x},\bar{X})$ to mean "the assignment $\bar{x}\mapsto\bar{a}$ and $\bar{P}\mapsto\bar{X}$ satisfies φ".

Definition 7.3 (S1S Semantics)**.** *The satisfaction relation* $(\bar{a},\bar{P})\models\varphi$ *is defined inductively on the structure of* φ *as follows.*

- $(\bar{a},\bar{P})\models\underbrace{\boldsymbol{ss}\cdots\boldsymbol{s}}_{k}x_i\in X_j$ *iff* $a_i+k\in P_j$*.*
- $(\bar{a},\bar{P})\models\neg\varphi(\bar{x},\bar{X})$ *iff* $(\bar{a},\bar{P})\not\models\varphi(\bar{x},\bar{X})$*.*
- $(\bar{a},\bar{P})\models\psi_1(\bar{x},\bar{X})\vee\psi_2(\bar{x},\bar{X})$ *iff* $(\bar{a},\bar{P})\models\psi_1(\bar{x},\bar{X})$ *or* $(\bar{a},\bar{P})\models\psi_2(\bar{x},\bar{X})$
- $(\bar{a},\bar{P})\models\exists y.\varphi(\bar{x},y,\bar{X})$ *iff there exists* $b\in\mathbb{N}$ *such that* $(\bar{a},b,\bar{P})\models\varphi(\bar{x},y,\bar{X})$*.*
- $(\bar{a},\bar{P})\models\exists Z.\varphi(\bar{x},\bar{X},Z)$ *iff there exists* $Q\subseteq\mathbb{N}$ *such that* $(\bar{a},\bar{P},Q)\models\varphi(\bar{x},\bar{X},Z)$*.*

Remark 7.4 (Universal Quantifier)**.** Note that the syntax and semantics do not contain $\forall$. However, we can express $\forall$ as $\forall\chi.\varphi(\chi):=\neg\exists\chi.\neg\varphi(\chi)$.

Similarly, we will allow all standard Boolean shorthands ($\wedge,\rightarrow$, etc.). △

Some useful S1S **constructs** As we saw in Example 7.1, S1S is missing some basic expressions, such as "<". This is because we can define them ourselves. Once these are defined, we can use them as shorthand.

- "$X\subseteq Y$" $:=\forall x.x\in X\rightarrow x\in Y$
- "$X=Y$" $:=X\subseteq Y\wedge Y\subseteq X$
- "$x=y$" $:=\forall X.x\in X\longleftrightarrow y\in X$ (cool, isn't it? "there is no separating set".)
- "$x=0$" $:=\forall y.\neg(x=\boldsymbol{s}y)$.
- "$x=1$" $:=x=\boldsymbol{s}0$ (more precisely, $\exists y.y=0\wedge x=sy$).
- "$x\leq y$" $:=\forall X.\,(x\in X\wedge(\forall z.z\in X\rightarrow\boldsymbol{s}z\in X))\rightarrow y\in X$.
- "X is finite" $:=\exists x.\forall y.(y\in X\rightarrow y\leq x)$.

7.3 S1S and ω-Languages

characteristic word

Let $AP=\{p_1,\ldots,p_k\}$ be a set of atomic propositions. For $q\in AP$, we associate a set $P_q\subseteq\mathbb{N}$ with the infinite *characteristic word* $\llbracket P_q\rrbracket\in(2^{\{q\}})^\omega$ such that $\llbracket P_q\rrbracket=\sigma_1\sigma_2\cdots$ with $\sigma_i=\begin{cases}\{q\} & i\in P_q\\ \emptyset & i\notin P_q\end{cases}$. Similarly, we associate with $P_1,\ldots,P_k$ a word $\llbracket P_1,\ldots,P_k\rrbracket\in(2^{AP})^\omega$ where each letter is determined by the corresponding sets. That is, $\llbracket P_1,\ldots,P_k\rrbracket=\tau_1\tau_2\cdots$ where $\tau_i=\{p_j\,:\,i\in P_j\}$.

For uniformity, we associate with a constant $a\in\mathbb{N}$ the word of the singleton set $\llbracket\{a\}\rrbracket$.

language

Let $\varphi(X_1,\ldots,X_n)$ be an S1S formula (with only 2nd order variables). We define the *language* of φ to be $L(\varphi(\bar{X}))=\{\llbracket\bar{P}\rrbracket\,:\,P\models\varphi(\bar{X})\}$.

Example 7.5. Consider the LTL formula $\mathsf{G}(p \to \mathsf{X}q)$. We construct an equivalent S1S formula.

$$\varphi_1(X_p, X_q) = \forall x, x \in X_p \to \boldsymbol{s}x \in X_q$$

Let's try something harder: $\mathsf{F}(p \wedge \mathsf{X}(\neg q \mathsf{U} p))$

$$\varphi_2(X_p, X_q) = \exists x. (x \in X_p \wedge \exists y. (\boldsymbol{s}x \leq y \wedge y \in X_p \wedge \forall z. (\boldsymbol{s}x \leq z \wedge z < y) \to \neg(z \in X_q)))$$

Note that this simple LTL formula yielded quite a long S1S formula. △

7.4 Decidability of S1S

Given an S1S formula $\varphi(\bar{X})$, can we determine whether it is satisfiable? Or whether it is valid? Or whether two formulas are equivalent? A-priori, these questions are quite difficult (indeed, try coming up with decision procedures!).

In 1962, Büchi [2] showed that S1S is decidable, by translating it to NBW. The proof, as it happens, is quite simple. The greatness in this work is the idea of translating a logic into a computational model. He also showed that the logic is exactly as expressive as NBW, by showing the inverse translation.

We say that an S1S formula $\varphi(\bar{X})$ is equivalent to an NBW $\mathcal{A}$ iff $L(\mathcal{A}) = L(\varphi)$.

We start with the simpler direction.

Theorem 7.6 (Büchi)**.** *Let $AP = \{p_1, \dots, p_k\}$. For every* NBW *$\mathcal{A}$ over alphabet 2^{AP}, there exists an equivalent* S1S *formula $\varphi(\bar{X})$. Moreover, we can effectively construct φ from $\mathcal{A}$.*

Proof. The proof is fairly straightforward: we construct a formula $\varphi(\bar{X})$ that, given $\bar{P} = (P_1, \dots, P_k)$, intuitively states that "there is an accepting run of $\mathcal{A}$ on $[\![\bar{P}]\!]$".

Let $\mathcal{A} = \langle Q, 2^{AP}, \delta, Q_0, \alpha \rangle$, where $Q = \{q_1, \dots, q_n\}$. We want to encode a run $\rho = \rho_0, \rho_1, \dots \in Q^\omega$ of $\mathcal{A}$ using n sets $Y_1, \dots, Y_n$ such that for all $i \in \mathbb{N}$ we have $i \in Y_j$ iff $\rho_i = q_j$.

In order to obtain such an encoding, we formulate certain properties of a run. First, each index has a unique state, that is the sets $Y_1, \dots, Y_n$ form a *partition* of $\mathbb{N}$. We thus define

$$\texttt{Partition}(Y_1, \dots, Y_n) := \forall x. \left(\bigvee_{i=1}^{n} x \in Y_i \right) \wedge \neg \left(\exists y. \bigvee_{i \neq j} y \in Y_i \wedge y \in Y_j \right)$$

Next, each state follows from the previous state according to the transition function. Given a letter $\sigma \in 2^{AP}$ and a (1st-order) variable x, we use the shorthand

$$x \in X_\sigma := \bigwedge_{p_j \in \sigma} x \in P_j \wedge \bigwedge_{p_j \notin \sigma} \neg(x \in P_j)$$

to denote that x represents exactly the atomic propositions that correspond to σ.

Then, we define

$$\texttt{Transitions}(Y_1, \dots, Y_n) := \forall x. \bigvee_{\substack{q_i \in Q, \sigma \in 2^{AP} \\ q_j \in \delta(q_i, \sigma)}} x \in Y_i \wedge x \in X_\sigma \wedge \boldsymbol{s}x \in Y_j$$

Finally, we define

$$\texttt{Accepting}(Y_1, \dots, Y_n) := \forall x. \exists y. (x \leq y \wedge \bigvee_{q_i \in \alpha} y \in Y_i)$$

We can now define the entire formula

$$\varphi_{\mathcal{A}}(X_1, \dots, X_n) := \exists Y_1, \dots, Y_n. \left(\begin{array}{ll} & \texttt{Partition}(Y_1, \dots, Y_n) \\ \wedge & \bigvee_{q_i \in Q_0} 0 \in Y_i \\ \wedge & \texttt{Transitions}(Y_1, \dots, Y_n) \\ \wedge & \texttt{Accepting}(Y_1, \dots, Y_n) \end{array} \right)$$

The correctness of this construction is easy to verify, by looking at each component of the formula. □

Theorem 7.7 (Also Büchi)**.** *Let $AP = \{p_1, \dots, p_k\}$. For every* S1S *formula $\varphi(\bar{x}, \bar{X})$ there exists an equivalent*[10] NBW *$\mathcal{A}$ over alphabet 2^{AP}. Moreover, we can effectively construct $\mathcal{A}$ from φ.*

Proof. The proof of this theorem is also fairly sensible, but it does involve some thought. We construct $\mathcal{A}$ by induction over the structure of φ.

- *Atomic formulas:* If $\varphi(\bar{x}, \bar{X}) = \underbrace{\mathbf{ss}\cdots\mathbf{s}}_{k} x_i \in X_j$, we build $\mathcal{A}$ as follows: once it sees a letter $\sigma_n \in 2^{AP}$ such that $p_i \in \sigma_n$, it checks that $p_j \in \sigma_{k+n}$, by waiting for k steps (see Figure 14).

  In addition, we check that p_i is seen exactly once.

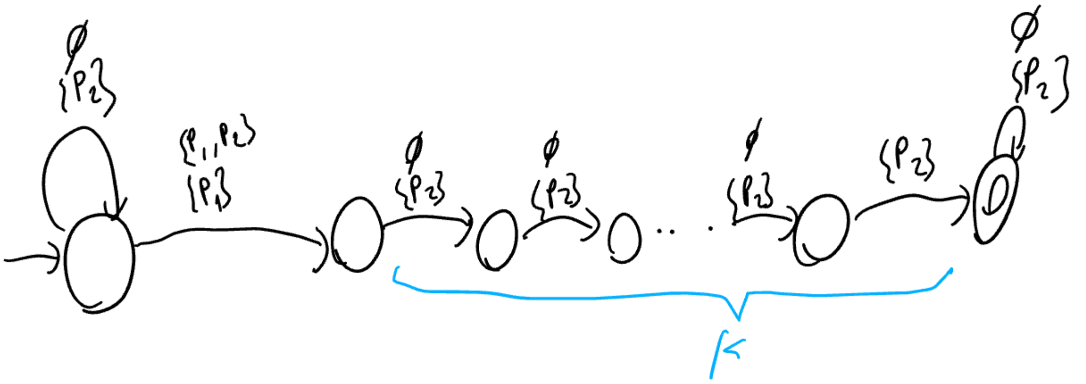


Figure 14: Translation of atomic formula. Here, x_i corresponds to p_1 and X_j to p_2.

- *Disjunction:* If $\varphi(\bar{x}, \bar{X}) = \psi_1(\bar{x}, \bar{X}) \vee \psi_2(\bar{x}, \bar{X})$, we simply take the union of the respective NBWs for ψ_1 and ψ_2, as per Theorem 3.6.

- *Negation:* If $\varphi(\bar{x}, \bar{X}) = \neg\psi(\bar{x}, \bar{X})$, we take the complement NBW of ψ, as per Theorem 3.23.

  Notice that while this step is now easy, Büchi had to work hard to prove closure under complement!

- Second-order existential quantification: If $\varphi(\bar{x}, \bar{X}) = \exists Z.\psi(\bar{x}, \bar{X}, Z)$, we proceed as follows. Let $\mathcal{B} = \langle Q, 2^{AP\cup\{p_{k+1}\}}, \delta, Q_0, \alpha\rangle$ be the NBW equivalent to $\psi(\bar{x}, \bar{X}, Z)$, as per the induction hypothesis. We are going to "project" the Z coordinate away (p_{k+1}), in a nondeterministic manner.

  Formally, we define $\mathcal{A} = \langle Q, 2^{AP}, \delta', Q_0, \alpha\rangle$ by setting, for every state $q \in Q$ and $\sigma \in 2^{AP}$, $\delta'(q, \sigma) = \delta(q, \sigma_0) \cup \delta(q, \sigma_1)$ where σ_0 and σ_1 are obtained by taking σ without and with p_{k+1}, respectively[11].

  Effectively, $\mathcal{A}$ "guesses" nondeterministically, which entries are in Z. Clearly $\mathcal{A}$ accepts a word iff there exists a set Z for which $\mathcal{B}$ accepts the corresponding word combined with the Z coordinate. See Figure 15.

- First-order existential quantification: This is fairly similar to the previous construction, only we also need to make sure we guess exactly one entry to have p_{k+1}. In order to achieve this, we take two copies of $\mathcal{B}$, as follows.

  Let $\mathcal{B} = \langle Q, 2^{AP\cup\{p_{k+1}\}}, \delta, Q_0, \alpha\rangle$ be the NBW equivalent to $\psi(\bar{x}, z, \bar{X})$, as per the induction hypothesis. We are going to "project" the z coordinate away (call it again p_{k+1}).

  We define $\mathcal{A} = \langle Q \times \{0, 1\}, 2^{AP}, \delta', Q_0 \times \{0\}, \alpha \times \{1\}\rangle$ by setting, for every state $q \in Q$ and $\sigma \in 2^{AP}$, $\delta'((q, 0), \sigma) = (\delta(q, \sigma_0), 0) \cup (\delta(q, \sigma_1), 1)$ where σ_0 and σ_1 are obtained by taking σ without and with p_{k+1}, respectively. In the second copy, we behave exactly like $\mathcal{B}$, without guessing additional p_{k+1} entries. That is, $\delta'((q, 1), \sigma) = (\delta(q, \sigma_0), 1)$.

  We now have that $\mathcal{A}$ accepts a word iff there exists a singleton z for which $\mathcal{B}$ accepts the corresponding word combined with the z coordinate.

[10] Notice that we allow φ to involve 1st-order variables. In this case, equivalence means that we also check that words only have a single index corresponding to those variables.

[11] We could have just taken σ and $\sigma \cup \{p_{k+1}\}$, but we want ot emphasize that we change alphabet

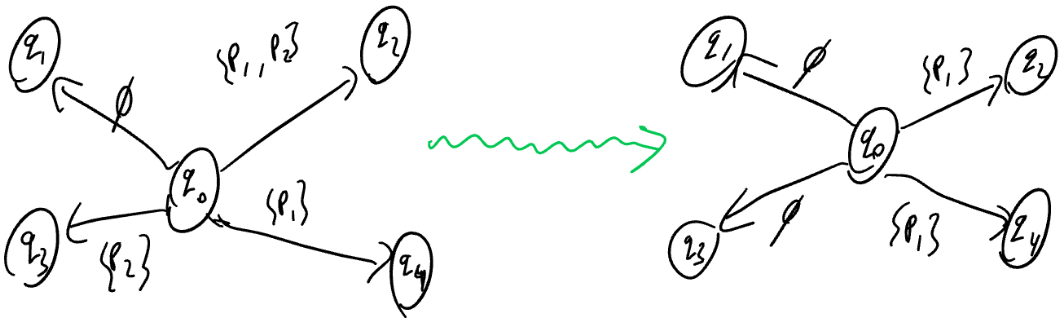


Figure 15: Translation of existential quantifier. Here, Z corresponds to p_2.

□

Remark 7.8 (S1S and NBW translation blowup)**.** Observe that the construction in Theorem 7.6, given an NBW of size n, yields a formula of size polynomial in n.

Conversely, the construction in Theorem 7.7, given a formula of size n, could potentially complement and project repeatedly (which is in fact quite common, when there are alternating quantifiers $\exists x \forall y \exists z \ldots$), in which case the blowup would be a tower of exponents $\underbrace{2^{2^{\cdot^{\cdot^{n}}}}}_{n}$ (a.k.a O(my god)).

Unfortunately, the latter is, in a sense, optimal. Indeed, Meyer [16] proved in 1975 that S1S satisfiability is NON-ELEMENTARY, meaning that the complexity cannot be bounded by a tower of exponents of a fixed height. △

8 Games Motivation – LTL Synthesis

8.1 Model Checking Reactive Programs

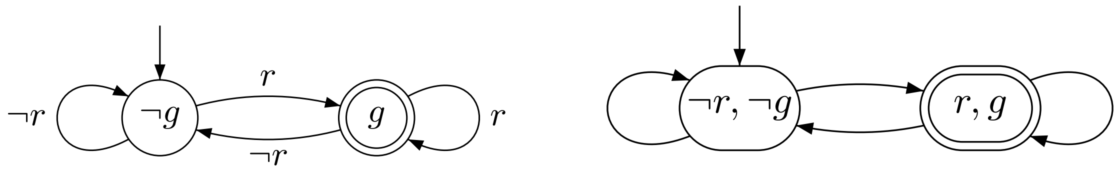
Recall that in LTL Model checking, we are given a formula φ and a structure $\mathcal{K}$, and we ask whether $\mathcal{K} \models \varphi$. This makes sense because our program is modeled as $\mathcal{K}$, and we want all its possible behaviours to satisfy φ. Why did we choose to model programs using a nondeterministic structure? Real programs are deterministic, or probabilistic. Well, there are several reasons. One reason is that sometimes we abstract away parts of the program, such as conditions, and obtain something nondeterministic. But why would a program even have conditions? It needs them only if there is something inherently unknown about the behaviour. This unknown element is simply the *input.*

Indeed, programs don't just hang around, doing mindless computations. They typically interact with users, sensors, data streams, etc. These can effect the execution of the program, and we need to model it somehow, Hence – nondeterminism.

A bit more formally, suppose our atomic propositions are partitioned into inputs I and outputs O. Then, a letter over $2^{I \cup O}$ describes the inputs for the system, and it's current outputs (in our previous framework, we essentially only had outputs).

Then, a *program* (which we will later call a *reactive system*) reads an input, outputs something, moves on to the next state, and repeats the process (see Figure 16(a)). How does this fit into our previous framework? Suppose we have such a program, and a specification φ over $2^{I \cup O}$, then we say that the program *realizes* the specification if for every sequence of inputs, the sequence of outputs induced by the program satisfies the specification. And indeed, this fits really well into Kripke structures: Given a program P, we obtain a Kripke structure $\mathcal{K}_P$ by nondeterministically choosing an input $i \subseteq I$, and for each input, marking the state we reach with the relevant output $o \subseteq O$. Then, $\mathcal{K}_P \models \varphi$ iff for every input sequence, if we combine it with the output sequence associated with it by P, we get a computation that satisfies φ (see Figure 16)

Checking whether a program we wrote satisfies the specification (i.e., is bug-free) is great. But what happens when we find a bug? We have to fix the bug, and recheck. And as you know, this check can be expensive. Wouldn't it be great, if we could just take our specifications, and from them *automatically*



(a) A transducer (program) satisfying $\mathsf{G}(r \longleftrightarrow g)$.

(b) The Kripke structure obtained from Figure 16(a).

Figure 16: Translating a program to a Kripke structure.

generate a program that satisfies the specifications? That is, completely get rid of human programming? It certainly would.

The first thing that comes to mind toward this effort is LTL satisfiability – given an LTL specification φ, we can check whether it's satisfiable, and if it is, output a satisfying computation. Is this enough? Not really. A satisfying computation can be thought of as a sequence of inputs and outputs that satisfies the formula. But it's not a program, since the inputs are controlled. What we want is a program that reacts to an uncontrollable input.

Solving this problem opens a wonderful field of study, relating to game theory, which we now turn to explore.

8.2 LTL Synthesis

We will start with an example for LTL synthesis.

Example 8.1. Let $I = \{r\}$ and $O = \{g\}$ (where we think of r as "request" and g as "grant"), and consider the specification $\mathsf{G}(r \longleftrightarrow g)$. Can we find a "good" program for this formula? (if you're re-reading this, then "good" simply means "realizing").

Sure (we've already seen that in Figure 16), the program works as follows: for every sequence of inputs $x \in (2^I)^\star$, if x ends with r, then the output will be g, and if it ends with $\neg r$ (or $\emptyset$, in set terminology) the output will be $\neg g$ (or $\emptyset$, in set terminology).

How about the specification $\mathsf{G}(r \to g)$? Well, the same program as above works, but we can also just have a program that always outputs g. △

The formal setting we consider is the following: the *system* controls the outputs O, whereas the *environment* controls the inputs I. Then, the system tries to make sure that every sequence of inputs is matched with a sequence of outputs such that the specification φ is satisfied. This setting is thus sort of a "game" between the system and the environment, where the system wins if the specification is satisfied. Then, a winning strategy for the system describes a program.

$x \otimes y$

Consider two finite sets I and O of input and output signals, respectively. For two words $x = i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$ and $y = o_0 \cdot o_1 \cdot o_2 \cdots \in (2^I)^\omega$, we define $x \otimes y$ as the word in $(2^{I \cup O})^\omega$ obtained by merging x and y. That is, $x \otimes y = (i_0 \cup o_0) \cdot (i_1 \cup o_1) \cdot (i_2 \cup o_2) \cdots$. The definition is similar for finite x and y of the same length. For a word $w \in (2^{I \cup O})^\omega$, we use $w_{|I}$ to denote the projection of w on I. In particular, $(x \otimes y)_{|I} = x$.

strategy

A *strategy* is a function $f : (2^I)^+ \to 2^O$. Intuitively, f models the interaction of the system that generates in each moment in time a letter in 2^O with an environment that generates letters in 2^I. For an input sequence $x = i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$, we use $f(x)$ to denote the output sequence $f(i_0) \cdot f(i_0 \cdot i_1) \cdot f(i_0 \cdot i_1 \cdot i_2) \cdots \in (2^O)^\omega$. Then, $x \otimes f(x) \in (2^{I \cup O})^\omega$ is the *computation* of f on x. Note that the environment "initiates" the interaction, by inputting i_0.

An arbitrary strategy can be thought of as an infinite tree, which specifies what to output after every finite input sequence. These, however, cannot generally be finitely described. Obviously we care about things we can describe, and so we restrict the discussion (for now) to finite strategies, represented by *transducers*, which are essentially deterministic automata with outputs.

I/O-transducer

An *I/O-transducer* is $\mathcal{T} = \langle I, O, S, s_0, M, \tau \rangle$, where S is a finite set of states, $s_0 \in S$ is an initial state, $M : S \times 2^I \to S$ is a transition function, and $\tau : S \to 2^O$ is a labelling function. For $x = i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^*$, let $M^*(x)$ be the state in S that $\mathcal{T}$ reaches after reading x. That is, $M^*(\epsilon) = s_0$ and for every

$j \geq 0$, we have that $M^*(i_0 \cdot i_1 \cdot i_2 \cdots i_j) = M(M^*(i_0 \cdot i_1 \cdot i_2 \cdots i_{j-1}), i_j)$. Then, $\mathcal{T}$ induces the strategy $f_{\mathcal{T}} : (2^I)^+ \to 2^O$, where for every $x \in (2^I)^+$, we have that $f_{\mathcal{T}}(x) = \tau(M^*(x))$. We use $\mathcal{T}(x)$ and $x \otimes \mathcal{T}(x)$ to denote the output sequence and the computation of $\mathcal{T}$ on x, respectively, and talk about $\mathcal{T}$ realizing a specification, referring to the strategy $f_{\mathcal{T}}$.

realizes Consider an LTL specification φ over $2^{I \cup O}$. We say that the transducer $\mathcal{T}$ *realizes* φ if for every $x \in (2^I)^\omega$, the induced computation $x \otimes \mathcal{T}(x)$ satisfies φ. If such a transducer exists, we say φ is *realizable*. The two central problems we study are the following:

LTL **Realizability:**

Given: an LTL formula φ over $2^{I \cup O}$

Question: Is φ realizable?

LTL **Synthesis:**

Given: an LTL formula φ over $2^{I \cup O}$

Question: If φ is realizable, output a realizing transducer.

Let's try to solve these problems (we can't just yet, but let's see where we get stuck). Our input is just an LTL formula φ over $2^{I \cup O}$. It's quite natural to convert is to an NBW $\mathcal{A}_\varphi$ as per Theorem 6.12. It's not quite clear how to proceed, though. We need to decide if there exists a strategy, such that given a word $x \in (2^I)^\omega$, outputs something "letter by letter", such that the induced word has an accepting run through $\mathcal{A}_\varphi$. One of the annoying aspects here is that $\mathcal{A}_\varphi$ is itself nondeterministic, which adds another complicating layer to everything. So let's circumvent this for now. By Theorem 4.19, we can determinize $\mathcal{A}_\varphi$ to obtain an equivalent DPW $\mathcal{D}_\varphi$. So now we want to decide if there exists a strategy, such that given a word $x \in (2^I)^\omega$, outputs something "letter by letter", such that the induced (unique) run through $\mathcal{D}_\varphi$ is accepting.

But the latter doesn't really fit any of the algorithms we have. Intuitively, we have the following setting: at each point, we *get* some input $i \in 2^I$, and we need to *choose* some output $o \in 2^O$, and so on, such that for *every* i_0 there *exists* o_0 such that for *every* i_1 there *exists* o_1 such that... the computation $(i_0 \cup o_0), (i_1 \cup o_1), \ldots$ is accepted by $\mathcal{D}_\varphi$.

This already starts to feel like a game: the environment (input player) chooses an input, and the system (the output player) needs to respond with an output, such that some winning condition is met.

We are now well motivated to start from the beginning.

9 Games – The Basics

The games we consider in this course are 2-player games played over a finite arena. The players will usually be called Player 0 and Player 1.

A lot of my notes and examples are taken from the excellent survey [14].

game **Definition 9.1** (Game)**.** *A* game *is* $\mathcal{G} = \langle V, V_0, V_1, E, \mathsf{Win} \rangle$ *where*

- *V is a finite set of vertices,*
- *V_0 and V_1 form a partition of V, denote the sets of Player 0 and Player 1, respectively,*
- *$E \subseteq V \times V$ is a total edge relation (i.e. every state has at least one outgoing edge),*
- *$\mathsf{Win} \subseteq V^\omega$ is a winning condition.*

arena We refer to $\langle V, V_0, V_1, E \rangle$ as the *arena* of the game.

strategy Given a game $\mathcal{G} = \langle V, V_0, V_1, E, \mathsf{Win} \rangle$, a *strategy* for Player $i \in \{0, 1\}$ is a function $f : V^* V_i \to V$ such that for every $u \in V^*$ and $v \in V_i$ it holds that $(v, f(uv)) \in E$.

memoryless A strategy f is called *memoryless* (a.k.a. "positional", and sometimes "sequential") if f only depends on the current state. That is, $f(uv) = f(v)$ for every $u \in V^*$ and $v \in V_i$ (for the corresponding Player i).

play A *play* in $\mathcal{G}$ is a sequence $\pi = v_0, v_1, \ldots \in V^\omega$ such that $(v_i, v_{i+1}) \in E$ for all $i \geq 0$. (We similarly

winning define a finite play). A play π is *winning* for Player 0 if $\pi \in \mathsf{Win}$. Otherwise, π is winning for Player 1 (we also say that a play is *losing* for Player i if it is winning for Player $1 - i$).

consistent

Fix strategies f, g for players 0 and 1, then for every initial vertex $v_0 \in V$, f and g uniquely determine a play $\mathsf{Play}_{v_0}(f, g)$ in $\mathcal{G}$, by starting from v_0 and proceeding according to the strategies. A play π is *consistent* with strategy f for Player i if there exists a strategy g for Player $(1 - i)$ such that $\pi = \mathsf{Play}_{v_0}(f, g)$.

wins

A strategy f is *winning* for Player i from vertex v_0 if for every strategy g of Player $1 - i$ it holds that $\mathsf{Play}_{v_0}(f, g)$ is winning for Player i. We say that Player 0 *wins* from v_0 if she has a winning strategy from v_0 (and similarly for Player 1).[12]

winning region

The *winning region* of Player $i \in \{0, 1\}$, denoted $\mathsf{Win}_i \subseteq V$, is the set of states from which Player i has a winning strategy.

Example 9.2. Consider the game in Figure 17. Suppose the goal for Player 0 is to reach either v_4 or v_5. That is, Win contains all plays that visit v_4 or v_5 at some point.

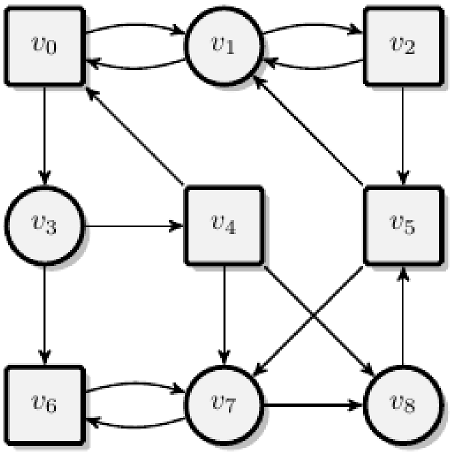


Figure 17: A game. Circle/Square vertices belong to Player 0/1 respectively.

Obviously Player 0 wins from v_4 and from v_5, and even with a memoryless strategy. Player 0 also wins from v_8, by forcing the play to reach v_5 in the next step (still, using a memoryless strategy).

From v_0, however, Player 1 wins, with the following memoryless strategy: from v_0 go to v_1, and from v_2 go to v_1 (and the rest of the vertices can have any arbitrary strategy, as they won't be reached). △

determined

Note that a-priori, it could be that for some vertex v, neither player has a winning strategy from v. We say that a game is *determined* if $\mathsf{Win}_0 \cup \mathsf{Win}_1 = V$. That is, if from each state, either Player 0 or Player 1 has a winning strategy. If a game is determined and from every state in Win_i, Player i has a memoryless winning strategy, we say that the game is *memoryless-determined*.

Example 9.3 (Determinacy)**.** It may seem that any reasonable game (or indeed, any game) is determined. To illustrate a non-determined game, consider rock-paper-scissors: from the "initial" state, there is no strategy that is guaranteed to win for either player.

However, this example is not in the spirit of our games, since this game does not have "perfect-information": it is not turn based, so e.g. Player 1 doesn't know what Player 0 is going to play in the first turn. △

It turns out that determinacy of the games we study is a complicated issue. Martin [13] proved that any game whose winning condition is a Borel set is determined. Coming up with a non-determined game in our framework is non-trivial (and depends on which axioms you allow – if you assume the Axiom of Determinacy, then all our games are determined. If you assume the Axiom of Choice, you can build a non-determined game). Gowers has an excellent entry on this in his Blog [6].

The fundamental problem about games is *solving* them:

Game Solving:

[12] Note that a-priori, it could be that neither player wins from a certain vertex.

Given: a game $\mathcal{G}$
Question: Compute Win_0 and Win_1.

In addition, we will usually want respective winning strategies from each state. Naturally, the actual algorithms would depend on the type of winning conditions we use.

10 Reachability and Safety Games

10.1 Reachability Games

The first, and simplest, type of game we consider is reachability games. Consider an arena $\langle V, V_0, V_1, E\rangle$, and let $R \subseteq V$. The *reachability* objective is

reachability

$$\textsc{Reach}(R) = \{\pi \in V^\omega \,:\, \exists i \geq 0,\ \pi_i \in R\}$$

that is, the set of plays that visit R at some point. The corresponding *reachability game* is $\mathcal{G} = \langle V, V_0, V_1, E, \textsc{Reach}(R)\rangle$.

Note that for reachability objective R, Player 1 is winning if it can keep the play in $V \setminus R$ forever.

Example 10.1. Consider the game described in Figure 17, with the reachability objective $\textsc{Reach}(R)$ for $R = \{v_4, v_5\}$. It's quite intuitive how to solve this game. First, $\{v_4, v_5\}$ are clearly in Win_0. Then, look at v_3, for example: $v_3 \in V_0$, so Player 0 can choose to take the edge (v_3, v_4), and thus win from v_3. Similarly, v_8 is winning by going to v_4. So $\{v_3, v_4, v_5, v_8\}$ are in Win_0. Now, v_7 can reach v_8, so it's also winning for Player 0.

Interestingly, v_6 is also winning for Player 0, even though $v_6 \in V_1$. This is because *all* the edges from v_6 get to Win_0. So far we have $\{v_3, v_4, v_5, v_6, v_8\} \subseteq \mathsf{Win}_0$. Is anything else winning for Player 0?

Doesn't seem so. In fact, as we reasoned in Example 9.2, Player 1 wins from $\{v_0, v_1, v_2\}$.

So we discovered that this game is determined, with $\mathsf{Win}_0 = \{v_3, v_4, v_5, v_6, v_8\}$ and $\mathsf{Win}_1 = \{v_0, v_1, v_2\}$. △

10.2 Attractors, Traps, and Subgames

In order to solve reachability games, we introduce some tools that will be very useful later. The first is the *attractor*.

Force$_i(S)$

Definition 10.2. *Consider an arena $\langle V, V_0, V_1, E\rangle$. Let $S \subseteq V$ be a set of vertices and $i \in \{0,1\}$ be a player. Define*

$$\mathsf{Force}_i(S) = \{v \in V_i \,:\, \exists v' \in S, (v, v') \in E\} \cup \{v \in V_{1-i} \,:\, \forall v' \in V \text{ such that } (v, v') \in E, v' \in S\}.$$

Equivalently,

$$\mathsf{Force}_i(S) = \{v \in V_i \,:\, v' \in S \text{ for some succesor of } v\} \cup \{v \in V_{1-i} \,:\, v' \in S \text{ for all succesors of } v\}$$

attractor

The attractor *for Player i to S is defined inductively as follows.*

- $\mathsf{Attr}_i^0(S) = S$,
- *For all $n \geq 0$, $\mathsf{Attr}_i^{n+1}(S) = \mathsf{Attr}_i^n(S) \cup \mathsf{Force}_i(\mathsf{Attr}_i^n(S))$, and*
- $\mathsf{Attr}_i(S) = \bigcup_{n\in\mathbb{N}} \mathsf{Attr}_i^n(S)$.

Intuitively, $\mathsf{Attr}_i(S)$ is the set of vertices from which Player i can force the play to reach S in finite time.

Remark 10.3. Definition 10.2 suggests an algorithm for computing $\mathsf{Attr}_i(S)$, by inductively constructing $\mathsf{Attr}_i^n(S)$. If the arena is finite, then this process must converge within at most $|V|$ iterations. That is, $\mathsf{Attr}_i^{|V|}(S) = \mathsf{Attr}(S)$.

In particular, $\mathsf{Attr}_i(S)$ can be computed by traversing each edge in E at most once. △

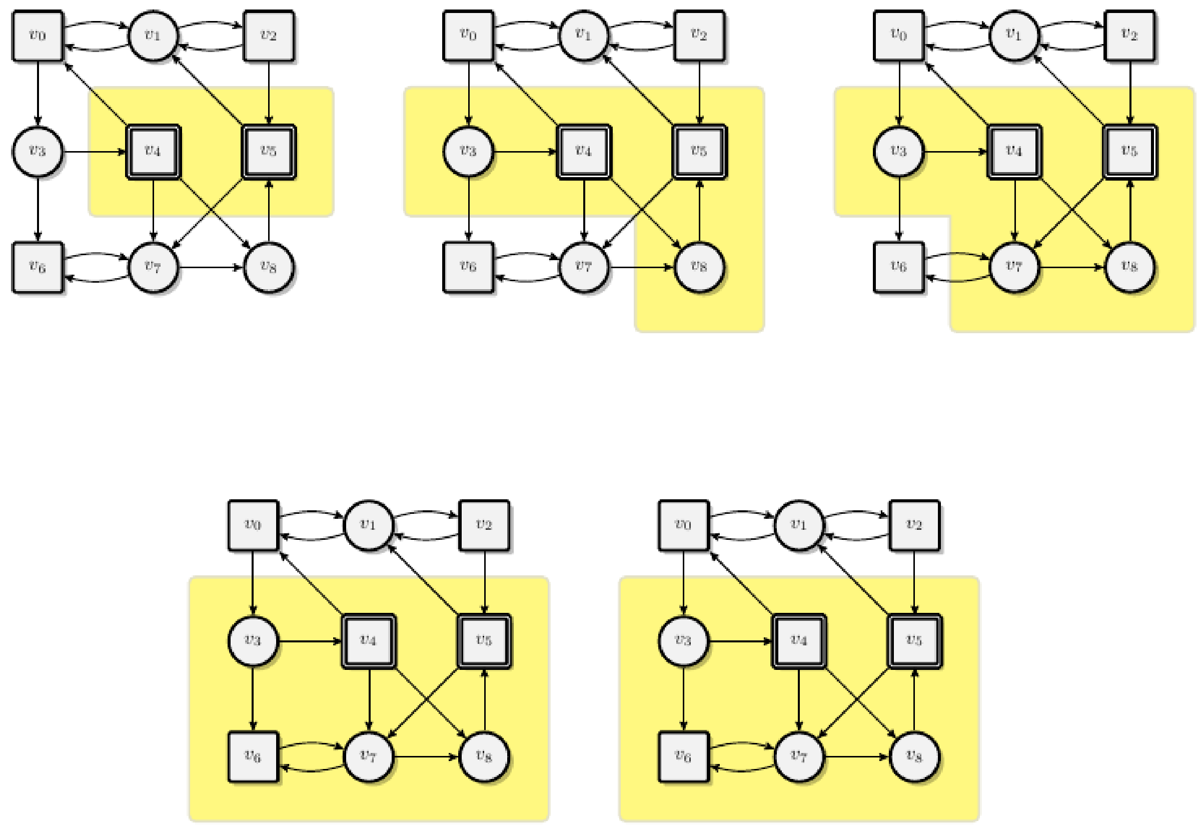


Figure 18: Attractor computation for $\mathsf{Attr}_0(\{v_4, v_5\})$.

Example 10.4. Let's see an example of computing attractors, on the reachability game from Example 10.1 (in Figure 17). The computation is depicted in Figure 18. The computation proceeds as follows:

$\mathsf{Attr}_0^0(\{v_4, v_5\}) = \{v_4, v_5\}$

$\mathsf{Attr}_0^1(\{v_4, v_5\}) = \{v_4, v_5\} \cup \{v_3, v_8\}$

$\mathsf{Attr}_0^2(\{v_4, v_5\}) = \{v_4, v_5, v_3, v_8\} \cup \{v_7\}$

$\mathsf{Attr}_0^3(\{v_4, v_5\}) = \{v_4, v_5, v_3, v_7, v_8\} \cup \{v_6\}$

$\mathsf{Attr}_0^4(\{v_4, v_5\}) = \{v_4, v_5, v_3, v_7, v_6, v_8\}$

After 4 iterations, no further vertices are added, so we have completed computing the attractor. Note that it just so happens that $\mathsf{Attr}_0(\{v_4, v_5\}) = \mathsf{Win}_0$. Coincidence? $\triangle$

In fact, we can even extract strategies from this computation, as we now show.

Lemma 10.5. *Consider an arena $\langle V, V_0, V_1, E\rangle$. Let $S \subseteq V$ be a set of vertices and $i \in \{0, 1\}$ be a player.*

- *There is a memoryless strategy σ for Player i such that every play from $\mathsf{Attr}_i(S)$ that is consistent with σ reaches S.*

  Moreover, if $v \in \mathsf{Attr}_i^n(S)$ then the above play reaches S within at most n steps.

- *There is a memoryless strategy τ for Player $1 - i$ such that every play from $V \setminus \mathsf{Attr}_i(S)$ that is consistent with τ remains within $V \setminus \mathsf{Attr}_i(S)$.*

Proof. For every $v \in \mathsf{Attr}_i(S)$, let $\delta(v) = \min\{n \in \mathbb{N} : v \in \mathsf{Attr}_i^n(S)\}$. Intuitively, $\delta(v)$ is the "distance" of v from S. In particular, $\delta(v) = 0$ iff $v \in S$. Define $\delta(v) = \infty$ for $v \notin \mathsf{Attr}_i(S)$.

Observe that for every $v \in V_i$ such that $\infty > \delta(v) > 0$ there exists v' such that $(v, v') \in E$ and $\delta(v') < \delta(v)$, and for every $v \in V_{1-i}$ such that $\infty > \delta(v) > 0$, it holds that $\delta(v') < \delta(v)$ for every v' such that $(v, v') \in E$ (this follows immediately from the definition of the attractor).

We define the memoryless strategy σ for Player i as follows. For every $v \in \mathsf{Attr}_i(S) \cap V_i$, if $\infty > \delta(v) > 0$ let $\sigma(v) = v'$ where v' is such that $\delta(v') < \delta(v)$ (which exists by the above). If $\delta(v) \in \{0, \infty\}$ then $\sigma(v)$ can be an arbitrary neighbor.

Consider a play $\pi = \pi_0, \pi_1, \ldots$ that is consistent with σ, where $\pi_0 \in \mathsf{Attr}_i^n(S)$ for some $n \in \mathbb{N}$. Thus, $\delta(\pi_0) \leq n$, and it can be easily shown by induction that $\delta(\pi_j) \leq n - j$ until $\delta(\pi_j) = 0$ for some $j \leq n$, at which point $\pi_j \in S$.

For the second part of the lemma, we define the memoryless strategy τ for Player $1 - i$ as follows. For every $v \in V_{1-i} \setminus \mathsf{Attr}_i(S)$, by definition we have that $\delta(v) = \infty$. Still by definition, there exists $v' \in V \setminus \mathsf{Attr}_i(S)$ with $(v, v') \in E$ such that $\delta(v') = \infty$. We define $\tau(v) = v'$.

Further note that for every $v \in V_i \setminus \mathsf{Attr}_i(S)$ we have that $\delta(v) = \infty$ and for every v' such that $(v, v') \in E$ we still have $\delta(v') = \infty$. Consequently, it is easy to show that for every play π that is consistent with τ and such that $\pi_0 \in V \setminus \mathsf{Attr}_i(S)$ it holds that $\delta(\pi_j) = \infty$ for all $j \geq 0$, which concludes the claim. □

Observe that for reachability games, we essentially want to compute the attractor. Thus, as an immediate corollary of Remark 10.3 and Lemma 10.5, we have the following.

Theorem 10.6. *Reachability games are memoryless determined, and can be solved in polynomial time.*

Before proceeding, observe that, as we have seen, Player $1 - i$ can force the play to remain in $V \setminus \mathsf{Attr}_i(S)$ indefinitely. This motivates the following definition, which is a dual of attractors.

trap

Definition 10.7 (Trap). *Let $\langle V, V_0, V_1, E \rangle$ be an arena and let $i \in \{0, 1\}$ be a player. A set $S \subseteq V$ is a* trap *for Player i if:*

- *For every $v \in S \cap V_i$ and for every $(v, v') \in E$ we have $v' \in S$ (i.e., all the successors of v are in S), and*
- *For every $v \in S \cap V_{1-i}$ there exists $v' \in S$ such that $(v, v') \in E$ (i.e., there exists a successor of v in S).*

The proof of Lemma 10.5 shows that Player $1 - i$ has a memoryless strategy that keeps the play in a Player i-trap, once it reaches the trap.

sub-game

The last definition we want is that of a *sub-game*, which is, as the name suggests, a game that is played on the subset of the arena.

Definition 10.8 (Sub-Game). *Let $\mathcal{G} = \langle V, V_0, V_1, E, \mathsf{Win} \rangle$ be a game. A set $S \subseteq V$ induces a valid sub-game $\mathcal{G}\restriction_S$ if for every $v \in S$ there exists $v' \in S$ such that $(v, v') \in E$. Then,*

$$\mathcal{G}\restriction_S = \langle S, V_0 \cap S, V_1 \cap S, E \cap (S \times S), \mathsf{Win} \cap S^\omega \rangle$$

Finally, we observe that every trap induces a valid subgame. This will be useful later on.

10.3 Safety Games

safety

The next winning condition we consider is safety. Intuitively, in a safety game Player 0 tries to keep the play within some given set. Consider an arena $\langle V, V_0, V_1, E \rangle$, and let $S \subseteq V$. The *safety* objective is

$$\textsc{Safety}(S) = \{\pi \in V^\omega \ : \forall i \geq 0,\ \pi_i \in S\}$$

The corresponding *safety game* is $\mathcal{G} = \langle V, V_0, V_1, E, \textsc{Safety}(S) \rangle$.

Safety and reachability are dual conditions. Indeed, it is easy to see that Player 0 wins a safety game $\mathcal{G} = \langle V, V_0, V_1, E, \textsc{Safety}(S) \rangle$ from vertex v_0 iff Player 0 loses in the reachability game $\widetilde{\mathcal{G}} = \langle V, V_1, V_0, E, \textsc{Reach}(V \setminus S) \rangle$. Indeed, Player 0 in $\widetilde{\mathcal{G}}$ controls the vertices V_1, so it essentially mimics Player 1 in $\mathcal{G}$, so we essentially require that Player 1 in $\mathcal{G}$ cannot force the game to reach $V \setminus S$. Thus, Theorem 10.6 immediately gives us the following:

Theorem 10.9. *Safety games are memoryless determined, and can be solved in polynomial time.*

11 Büchi Games

Let's try to tackle Büchi games. There, the goal of Player 0 is to reach a certain set infinitely often.

The main conceptual difference from reachability games lies in that for reachability games, if Player 0 wins, then she wins after some finite prefix, after which the rest doesn't matter. In Büchi games, in order to win she must force the play to revisit α over and over, and no finite prefix will determine the outcome.

Büchi Formally, consider an arena $\langle V, V_0, V_1, E\rangle$, and let $\alpha \subseteq V$. The *Büchi* objective is

$$\text{BÜCHI}(\alpha) = \{\pi \in V^\omega \;:\; \inf(\pi) \cap \alpha \neq \emptyset\}$$

The corresponding *Büchi game* is $\mathcal{G} = \langle V, V_0, V_1, E, \text{BÜCHI}(\alpha)\rangle$.

Example 11.1. Consider the game in Figure 19, where $\alpha = \{v_4, v_6\}$.

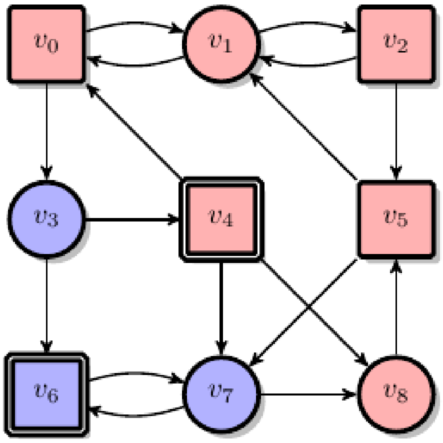


Figure 19: A Büchi game

Let's try to find the winning sets of the players (yes, they're already marked with pink and blue). Consider $\mathsf{Attr}_0(\alpha) = \{v_3, v_4, v_6, v_7\}$, and recall that $V \setminus \mathsf{Attr}_0(\alpha) = \{v_0, v_1, v_2, v_5, v_8\}$ is a trap for Player 0, meaning Player 0 cannot force the play to escape it. In particular, she cannot reach α, not even once. Thus, we immediately have that $V \setminus \mathsf{Attr}_0(\alpha) \subseteq \mathsf{Win}_1$.

Now let's look at v_4. It's in α, so it may seem like a good place to try and reach. But since it's in V_1, then Player 1 can choose to move it to v_8, from which Player 0 is losing. Thus, $v_4 \in \mathsf{Win}_1$. More precisely, we have $v_4 \in \mathsf{Force}_1(V \setminus \mathsf{Attr}_0(\alpha))$.

So we have $\mathsf{Win}_0 = \{v_3, v_6, v_7\}$, but we didn't quite prove it. Let's try to argue why this is the case. From $\{v_3, v_6, v_7\}$ Player 0 can force the game to reach v_6, but then from v_6 Player 0 can again force $\{v_3, v_6, v_7\}$, and so on forever. △

The solution of Büchi games roughly follows the lines of Example 11.1, but it's slightly more elaborate. The basic construct we use for Büchi games is the *recurrence* construction.

Definition 11.2 (Recurrence, Avoidance)**.** *Let $\langle V, V_0, V_1, E\rangle$ be an arena, $i \in \{0,1\}$ be a player, and* recurrence *$S \subseteq V$ be a set of vertices. The* recurrence *set for Player i to S is defined inductively as follows:*

- $\mathsf{Recur}_i^0(S) = S$*,*
- *For every $n \geq 0$, $\mathsf{Avoid}_{1-i}^n(S) = V \setminus \mathsf{Attr}_i(\mathsf{Recur}_i^n(S))$, and*
- *For every $n \geq 0$, $\mathsf{Recur}_i^{n+1}(S) = \mathsf{Recur}_i^n(S) \setminus \mathsf{Force}_{1-i}(\mathsf{Avoid}_{1-i}^n(S))$.*

We then define $\mathsf{Recur}_i(S) = \bigcap_{n\in\mathbb{N}} \mathsf{Recur}_i^n(S)$ and $\mathsf{Avoid}_{1-i}(S) = \bigcup_{n\in\mathbb{N}} \mathsf{Avoid}_{1-i}^n(S)$.

Since the sets $\mathsf{Recur}_i^n(S)$ form a decreasing chain, there exists $m \leq |V|$ such that $\mathsf{Recur}_i^m(S) = \mathsf{Recur}_i(S)$, and thus we can compute these sets in polynomial time (note that the operations in the inductive definition can be applied in polynomial time). Similarly, the sets $\mathsf{Avoid}_{1-i}^n(S)$ form an increasing chain, and the sets converge together.

Intuitively, $\mathsf{Recur}_i^n(S)$ is the set of vertices in S from which Player i can force the play to reach S again at least n times. Thus, $\mathsf{Recur}_i(S)$ is the set of vertices from which Player i can reach S infinitely often. It follows that the winning vertices for Player 0 in a Büchi game is $\mathsf{Attr}_0(\mathsf{Recur}_0(\alpha))$. This is formalized in the following.

Lemma 11.3. *Consider a Büchi game $\mathcal{G} = \langle V, V_0, V_1, E, \text{BÜCHI}(\alpha)\rangle$, then $\mathsf{Win}_0 = \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha))$, and $\mathsf{Win}_1 = V \setminus \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha))$.*

In addition, there are memoryless winning strategies for both players (from their respective winning regions).

Proof. Define $X = \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha))$, we will show that $X \subseteq \mathsf{Win}_0$ and $V \setminus X \subseteq \mathsf{Win}_1$, and since $\mathsf{Win}_0 \cap \mathsf{Win}_1 = \emptyset$, this will conclude the proof. Recall that $\mathsf{Recur}_0(\alpha) = \mathsf{Recur}_0^m(\alpha)$ for some m. Similarly, $\mathsf{Avoid}_1(\alpha) = \mathsf{Avoid}_1^m(\alpha)$ (since the sets converge together).

We start with Player 0, and construct a winning memoryless strategy from X. Consider a vertex $v \in \mathsf{Recur}_0(\alpha)$, then by the inductive construction we have $v \in \mathsf{Recur}_0(\alpha) \setminus \mathsf{Force}_1(\mathsf{Avoid}_1(\alpha))$. Accordingly:

1. If $v \in V_0$, then there exists a successor v' of v such that $v' \notin \mathsf{Avoid}_1(\alpha)$. But $\mathsf{Avoid}_1(\alpha) = V \setminus \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha))$, and hence $v' \in \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha)) = X$.

2. If $v \in V_1$, then by a similar reasoning, we have $v' \in X$ for every successor v' of v.

We can now construct a memoryless strategy σ for Player 0 in X. Let σ' be the (memoryless) attractor strategy for Player 0 that ensures reaching $\mathsf{Recur}_0(\alpha)$ from $\mathsf{Attr}_0(\mathsf{Recur}_0(\alpha))$, as per Lemma 10.5. We then define for every $v \in V_0$:

$$\sigma(v) = \begin{cases} \sigma'(v) & v \in \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha)) \setminus \mathsf{Recur}_0(\alpha), \\ v' & v \in \mathsf{Recur}_0(\alpha), \text{ where } v' \in X \text{ as per Item 1 above}, \\ v'' & \text{otherwise, where } v'' \text{ is some arbitrary successor.} \end{cases}$$

We claim that σ is winning for Player 0 from every $v \in X$. Indeed, let $\pi = \pi_0, \pi_1, \ldots$ be a play consistent with σ with $\pi_0 \in X$. First, it is easy to show by induction that $\pi_n \in \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha))$ for every $n \geq 0$ (using the construction of σ, and Property 2 above). Next, since σ' guarantees that a play visits $\mathsf{Recur}_0(\alpha)$ within at most $|V|$ steps from any vertex $v \in \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha)) \setminus \mathsf{Recur}_0(\alpha)$. Since $\mathsf{Recur}_0(\alpha) \subseteq \alpha$, this guarantees the the play is winning for Player 0.

We now turn to Player 1. Observe that $V \setminus X = \mathsf{Avoid}_1(\alpha)$. Observe that $\mathsf{Avoid}_1(\alpha)$ is a trap for Player 0, hence, Player 1 can force the play within it. However, it is not quite enough, since $\mathsf{Avoid}_1(\alpha)$ may contain vertices in α. Thus, we also need to make sure that Player 1 can force the play to eventually stay out of α. This is done as follows.

Recall that $\mathsf{Avoid}_1(\alpha) = \mathsf{Avoid}_1^m(\alpha)$ for some $m \leq |V|$. For every $v \in \mathsf{Avoid}_1(\alpha)$, define $\delta(v) = \min\{k : v \in \mathsf{Avoid}_1^k(\alpha)\}$, then we have $\delta(v) \leq |V|$ for every $v \in \mathsf{Avoid}_1(\alpha)$.

Furthermore, observe that $\mathsf{Avoid}_1^0(\alpha) = V \setminus \mathsf{Attr}_0(\alpha)$. In particular, $\mathsf{Avoid}_1^0(\alpha)$ is a trap for Player 0 that is disjoint[13] from α, so Player 1 has a memoryless strategy τ_0 that can keep the play within $\mathsf{Avoid}_1^0(\alpha)$.

Next, consider a vertex $v \in \mathsf{Avoid}_1(\alpha)$ with $\delta(v) = k > 0$, then $v \notin \mathsf{Attr}_0(\mathsf{Recur}_0^k(\alpha))$. We now show some properties of v.

1. If $v \in V_1$ then there exists a successor v' of v such that $\delta(v') \leq \delta(v)$, and if $v \in \alpha$ then the inequality is strict.

   Indeed, assume by way of contradiction that every successor v' of v satisfies $\delta(v') > \delta(v)$, and hence $v' \notin \mathsf{Avoid}_1^{\delta(v)}(\alpha)$ (since v' enters "later" than v). Thus, every successor v' satisfies $v' \in \mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)})(\alpha)$. But then $v \in \mathsf{Force}_0(\mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)}(\alpha))) \subseteq \mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)}(\alpha))$ (since the force

[13] observe that $\mathsf{Avoid}_1^0(\alpha) \neq \emptyset$, otherwise $\mathsf{Win}_0 = V$ and we're done.

of the attractor is contained in the attractor). But this is a contradiction, since $v \in \mathsf{Avoid}_1^{\delta(v)}(\alpha) = V \setminus \mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)}(\alpha))$.

Intuitively, the argument above simply says that if v entered $\mathsf{Avoid}_1(\alpha)$ at iteration $\delta(v)$, then at least one of its neighbors was already previously in $\mathsf{Avoid}_1(\alpha)$ before. Otherwise, v could not have gotten in.

Now, assume that in addition $v \in \alpha$. Since $v \in \mathsf{Avoid}_1^{\delta(v)}(\alpha) = V \setminus \mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)}(\alpha))$, then in particular $v \notin \mathsf{Recur}_0^{\delta(v)}(\alpha)$, i.e. by the inductive definition $v \notin \mathsf{Recur}_0^{\delta(v)-1}(\alpha) \setminus \mathsf{Force}_1(\mathsf{Avoid}_1^{\delta(v)-1}(\alpha))$. From this we get that either $v \in \mathsf{Force}_1(\mathsf{Avoid}_1^{\delta(v)-1}(\alpha))$ or $v \notin \mathsf{Recur}_0^{\delta(v)-1}(\alpha)$

In the former case, we get that there exists a successor v' of v with $\delta(v') \leq \delta(v) - 1 < \delta(v)$, and we are done.

In the latter case, we repeat the same argument, until we get the the former case at some point, since otherwise we will eventually get $v \notin \mathsf{Recur}_0^0(\alpha) = \alpha$, which is a contradiction since $v \in \alpha$.

2. If $v \in V_0$ then every successor v' of v has $\delta(v') \leq \delta(v)$, and if $v \in \alpha$ then the inequality is strict.

   This follows similar lines to the former part. Assume by way of contradiction that there exists a successor v' of v with $\delta(v') > \delta(v)$, and hence $v' \notin \mathsf{Avoid}_1^{\delta(v)}(\alpha)$ (since v' enters "later" than v). Thus, $v' \in \mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)})(\alpha)$. But then $v \in \mathsf{Force}_0(\mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)}(\alpha))) = \mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)}(\alpha))$ (since the force of the attractor is contained in the attractor). But this is a contradiction, since $v \in \mathsf{Avoid}_1^{\delta(v)}(\alpha) = V \setminus \mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)}(\alpha))$.

   Now, assume that in addition $v \in \alpha$. Since $v \in \mathsf{Avoid}_1^{\delta(v)}(\alpha) = V \setminus \mathsf{Attr}_0(\mathsf{Recur}_0^{\delta(v)}(\alpha))$, then in particular $v \notin \mathsf{Recur}_0^{\delta(v)}(\alpha)$, i.e. by the inductive definition $v \notin \mathsf{Recur}_0^{\delta(v)-1}(\alpha) \setminus \mathsf{Force}_1(\mathsf{Avoid}_1^{\delta(v)-1}(\alpha))$. From this we get that either $v \in \mathsf{Force}_1(\mathsf{Avoid}_1^{\delta(v)-1}(\alpha))$ or $v \notin \mathsf{Recur}_0^{\delta(v)-1}(\alpha)$

   In the former case, we get that every successor v' of v satisfies $\delta(v') \leq \delta(v) - 1 < \delta(v)$, and we are done.

   In the latter case, we repeat the same argument, until we get the the former case at some point, since otherwise we will eventually get $v \notin \mathsf{Recur}_0^0(\alpha) = \alpha$, which is a contradiction since $v \in \alpha$.

We can now define the winning strategy τ for Player 1 from $V \setminus X$. Recall that $X = \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha))$, and hence $V \setminus X = \mathsf{Avoid}_1(\alpha)$. Let τ' be a memoryless strategy for Player 1 that keeps a play within $\mathsf{Avoid}_1^0(\alpha) = V \setminus \mathsf{Attr}_0(\alpha)$ (which is a trap, and hence admits such a strategy).

For every $v \in V_1$, we define:

$$\tau(v) = \begin{cases} \tau'(v) & v \in \mathsf{Avoid}_1^0(\alpha) \\ v' & v \in \mathsf{Avoid}_1(\alpha) \setminus \alpha, \text{ with } \delta(v') \leq \delta(v) \\ v'' & v \in \mathsf{Avoid}_1(\alpha) \cap \alpha, \text{ with } \delta(v') < \delta(v) \\ v''' & \text{otherwise} \end{cases}$$

Note that the existence of v', v'' is guaranteed by the arguments above.

It remains to show that τ is winning for Player 1 from $V \setminus X$. Let $\pi = \pi_0, \pi_1, \ldots$ be a play consistent with τ, with $\pi_0 \in V \setminus X$. By the construction of τ, we have that $\pi_i \in V \setminus X$ for all $i \geq 0$. Indeed, for vertices in V_1, τ always chooses a successor that remains in $V \setminus X$, and for vertices in V_0, Property 2 above guarantees that we remain in $V \setminus X$. Moreover, Properties 1 and 2 guarantee that $\delta(\pi_i)$ is a decreasing function, and strictly decreasing whenever α is reached. Since $\delta(v) = 0$ implies $v \notin \alpha$, it follows that π visits α finitely often, and is hence winning for Player 1. □

Note that in particular, the proof of Lemma 11.3 shows that $\mathsf{Win}_1 = V \setminus \mathsf{Attr}_0(\mathsf{Recur}_0(\alpha)) = \mathsf{Avoid}_1(\alpha)$.
11.3 readily gives us the following.

Theorem 11.4. *Büchi games are memoryless determined, and can be solved in polynomial time.*

Intermission – Non Memoryless Games

To complement Ex4 Q3, we show that Player 1 does not necessarily have a memoryless winning strategy in a Rabin game. Or, viewed dually, we show that Player 0 does not have a memoryless strategy in a Streett game.

Consider the 3-vertex Street game $\mathcal{G} = \langle V, V_0, V_1, E, \alpha\rangle$ with $V = \{q, p, r\}.$, $V_0 = V$, $V_1 = \emptyset$, and $E = \{(p, q), (q, p), (p, r), (r, p)\}$ with $\alpha = \{(V, \{q\}), (V, \{r\})\}$. Thus, a play is winning for Player 0 iff it visits both q and r infinitely often.

Observe that a strategy for Player 0 amounts to deciding what to play from p, whenever it is reached. Clearly, if the strategy is memoryless, then the play from p would be either $(pq)^\omega$ or $(pr)^\omega$, both of which are not winning.

However, a strategy that alternates between q and r is winning for Player 0.

12 Parity Games

We now turn our attention to parity games. Recall that by Theorem 4.20, deterministic parity automata can capture all ω-regular languages. Thus, in a way, parity games are as general as we need.

Parity games have proved to be useful in many aspects of formal verification, logic and automata. Despite that, the complexity of solving them has remained one of the most exciting open questions of the past few decades:

The parity condition was first considered as a condition on games in 1984 by Mostowski [19] and by Emerson and Jutla [5] in 1991. There, it was shown that parity games are memoryless determined. This already gave the first fundamental result, which says that solving parity games can be solved in **NP** $\cap$ **coNP**.

In 1993, McNaughton [15] developed a general algorithm for solving Muller games, which was adapted by Zielonka [29] to parity games, and whose runtime was exponential ($O(2^n)$). Soon after, the importance of parity games was noticed (due to connections to μ-calculus and to mean-payoff games), and many papers started dealing with the complexity of the problem.

In 1998, Jurdziński [8] shows that in fact, parity games can be solved in **UP** $\cap$ **coUP**. He also improved Zielonka's deterministic algorithm based on "progress measures" [9].

Since then, we have seen a significant body of work tackling parity games, with the "holy grail" being a polynomial-time algorithm. Until 2017, the state of the art algorithms achieved complexity of roughly $n^{\sqrt{d}}$, where n is the number of vertices, and d is the index. However, an unexpected breakthrough in 2017 by Calude, Jain, Khoussainov, Li and Stephan [3] described the first quasi-polynomial algorithm, working in $n^{\mathrm{poly}(\log d)}$. This work sprouted many other works showing quasi-polynomial algorithms, with the most recent (at the time of writing these notes) being that of Parys [20], which, interestingly, shows that quasi-polynomial time can be obtained by a small (but clever) modification of Zielonka's 1993 algorithm.

Let's start with the definition of parity games. Consider an arena $\langle V, V_0, V_1, E\rangle$ and let $\Omega : V \to \{0, \dots, d\}$ be a priority function. The *parity* objective is

parity

$$\textsc{Parity}(\Omega) = \{\pi \in V^\omega \ : \ \min\{\Omega(q) \ : \ q \in \inf(\pi)\} \text{ is even}\}$$

That is, all the paths such that the minimal priority that appears infinitely often is even.

index

The corresponding *parity* game is $\mathcal{G} = \langle V, V_0, V_1, E, \textsc{Parity}(\Omega)\rangle$. We refer to d as the *index* of the game (or of Ω).

We're now going to jump right in and give McNaughton/Zielonka's algorithm. After understanding it and proving it's correctness, we will see what results it gives us.

Lemma 12.1. *Given a parity game $\mathcal{G}$, Algorithm 1 computes the winning regions of the players in $\mathcal{G}$. Moreover, we can extract memoryless winning strategies for both players.*

Proof. Let $\mathcal{G} = \langle V, V_0, V_1, E, \textsc{Parity}(\Omega)\rangle$ be a parity game. We prove the claim by induction over $n = |V|$.

The base case is trivial: if $n = 0$, there are no vertices, and thus the winning regions are $\emptyset$ for both players (c.f. Line 3), and the winning strategies are vacuously memoryless.

Algorithm 1 Parity Games

```
 1: procedure SOLVE($\mathcal{G} = \langle V, V_0, V_1, E, \textsc{Parity}(\Omega)\rangle$)
 2:     if $V = \emptyset$ then
 3:         return $\mathsf{Win}_0 = \mathsf{Win}_1 = \emptyset$
 4:     else
 5:         $l \leftarrow \min\{\Omega(v) : v \in V\}$                  ▷ the minimal priority
 6:         $i \leftarrow l \mod 2$                                   ▷ parity of minimal priority
 7:         $L \leftarrow \Omega^{-1}(l)$                              ▷ the set of minimal priority vertices
 8:         $V' \leftarrow V \setminus \mathsf{Attr}_i(L)$             ▷ Remove Player $i$'s attractor to $L$
 9:         $(W'_0, W'_1) \leftarrow \textsc{Solve}(\mathcal{G}\restriction_{V'})$   ▷ Recursively solve the subgame over $V'$
10:         if $W'_{1-i} = \emptyset$ then
11:             return $\mathsf{Win}_i = V$, $\mathsf{Win}_{1-i} = \emptyset$.
12:         else
13:             $B \leftarrow \mathsf{Attr}_{1-i}(W'_{1-i})$           ▷ Find the Player $1-i$ attractor to $W'_{1-i}$
14:             $V'' = V \setminus B$                                ▷ Remove the attractor
15:             $(W''_0, W''_1) \leftarrow \textsc{Solve}(\mathcal{G}\restriction_{V''})$   ▷ Recursively solve the game over $V''$
16:             return $\mathsf{Win}_i = W''_i$ and $\mathsf{Win}_{1-i} = B \cup W''_{1-i}$
```

Assume correctness for every $k < n$, we prove for n. Let l, i, L be as in Lines 5,6,7. We define $V' = V \setminus \mathsf{Attr}_i(L)$. Since $L \neq \emptyset$ by definition, then $|V'| < |V|$. In addition, V' is a trap for Player i, and in particular $\mathcal{G}\restriction_{V'}$ is a subgame (see Definition 10.8), so we can apply the induction hypothesis, corresponding to the recursive call in Line 8 (see Figure 20).

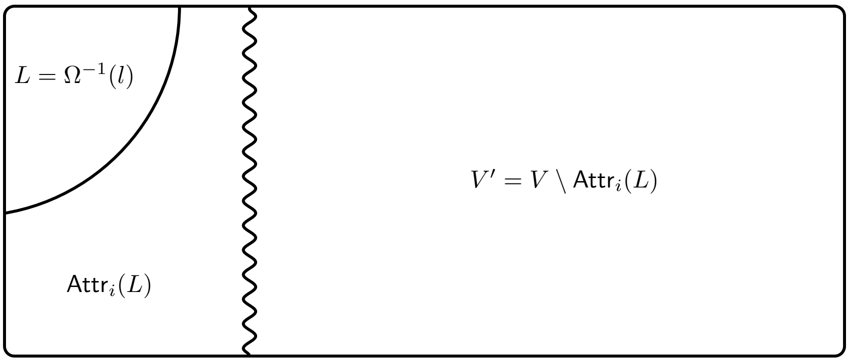


Figure 20: Line 8.

Thus, let $(W'_0, W'_1) = \textsc{Solve}(\mathcal{G}\restriction_{V'})$, and by the induction hypothesis, W'_0 and W'_1 are the correct winning regions in the subgame $\mathcal{G}\restriction_{V'}$. Moreover, there are memoryless strategies σ', τ' for Players i and $1-i$ in this subgame, respectively (see Figure 21).

We now separate the analysis to cases.

If $W'_{1-i} = \emptyset$: this is the easy case. We claim that $\mathsf{Win}_i = V$ and $\mathsf{Win}_{1-i} = \emptyset$. Indeed, let $\widehat{\sigma}$ be the memoryless attractor strategy for Player i from $\mathsf{Attr}_i(L)$ to L, as per Lemma 10.5, and consider the Player i strategy $\sigma : V_i \to V$ defined as follows.

$$\sigma(v) = \begin{cases} \sigma'(v) & v \in V' \\ \widehat{\sigma}(v) & v \notin V' \end{cases}$$

We claim that σ is winning for Player i from every vertex $v_0 \in V$. Indeed, consider a play π consistent with σ from v_0. By the definition of $\widehat{\sigma}$, whenever a vertex $\pi_i \notin V'$ occurs, then $\pi_i \in \mathsf{Attr}_i(L)$ and thus π visits L within at most n steps from π_i. Thus, if π goes outside V' infinitely often, then $\min\{\Omega(q) : q \in \inf(\pi)\} = l$, and $i = l \mod 2$, so Player i wins.

Otherwise, π goes outside V' only finitely often, so π eventually remains in V'. Then, however, σ behaves identically to σ', which is winning from every vertex in V', so Player i again wins.

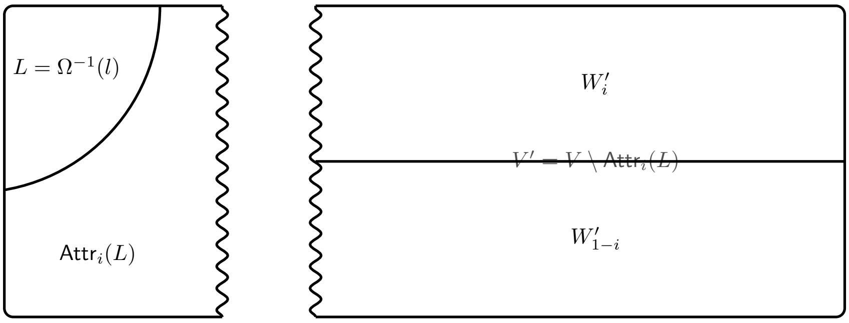


Figure 21: Line 9.

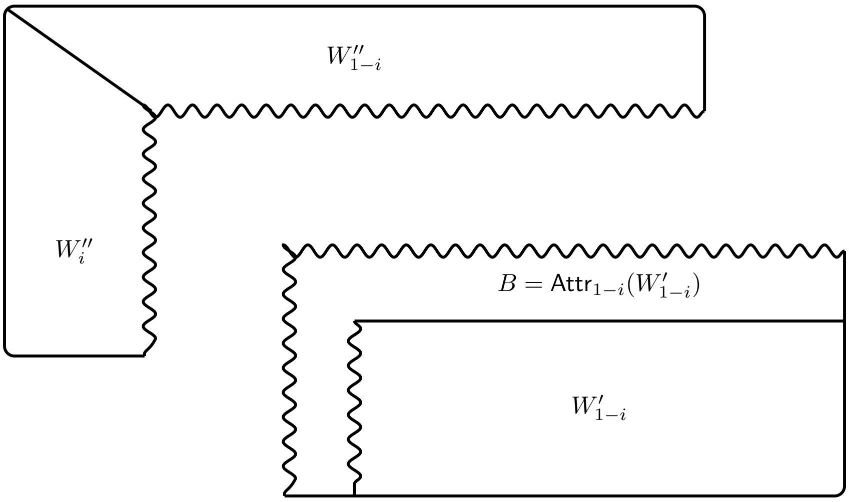


Figure 22: Lines 13–14.

If $W'_{1-i} \neq \emptyset$: this is the interesting case. Set $B = \mathsf{Attr}_{1-i}(W'_{1-i})$ and let $V'' = V \setminus B$ as per lines 13–14. (see Figure 22).

Since $W'_{1-i} \neq \emptyset$, it follows that $|V''| < |V|$, and V'' is a trap and hence a subgame. Therefore, we can again apply the induction hypothesis, corresponding to the recursive call in Line 15.

Thus, let $(W''_0, W''_1) = \textsc{Solve}(\mathcal{G}\upharpoonright_{V''})$, and by the induction hypothesis, W''_0 and W''_1 are the correct winning regions in the subgame $\mathcal{G}\upharpoonright_{V''}$. Moreover, there are memoryless strategies σ'', τ'' for Players i and $1-i$ in this subgame, respectively.

We now claim $\mathsf{Win}_i = W''_i$ and $\mathsf{Win}_{1-i} = B \cup W''_{1-i}$ (note that these indeed form a partition of V). In order to prove this, it's enough to prove that $W''_i \subseteq \mathsf{Win}_i$ and that $B \cup W''_{1-i} \subseteq \mathsf{Win}_{1-i}$.

We start with the former. Consider the Player i strategy $\sigma : V_i \to V$ defined as follows.

$$\sigma(v) = \begin{cases} \sigma''(v) & v \in V'' \\ w & v \notin V'', \text{ where } w \text{ is an arbitrary successor} \end{cases}$$

We claim σ is winning for Player i from V''. Indeed, since V'' is a trap for Player $1-i$, then $\sigma''(v)$ ensures that every play consistent with it remains within V''. Thus, σ is identical to σ'' when starting from V'', and is therefore winning for Player i.

Next, let $\widehat{\tau}$ be the attractor strategy for Player $1-i$ from B to W'_{1-i}, and consider the Player $1-i$ strategy $\tau : V_{1-i} \to V$ defined as follows.

$$\tau(v) = \begin{cases} \widehat{\tau}(v) & v \in B \setminus W'_{1-i} \\ \tau'(v) & v \in W'_{1-i} \\ \tau''(v) & v \in W''_{1-i} \\ w & \text{otherwise, where } w \text{ is an arbitrary successor} \end{cases}$$

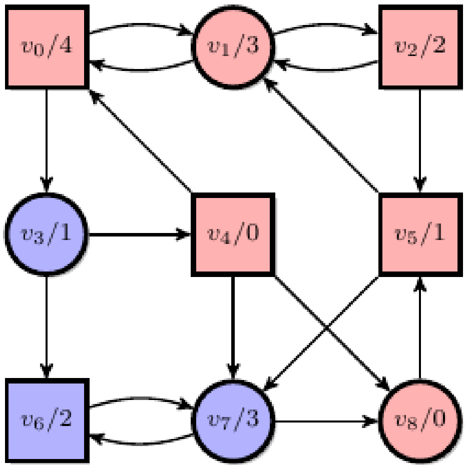


Figure 23: A parity game.

We show that τ is winning for Player $1-i$ from $B \cup W''_{1-i}$. Indeed, consider a play π consistent with τ. Observe that if, at any point, π reaches B (in particular, if π starts from B), then within at most n steps, π reaches W'_{1-i}, which is a trap for Player i, and hence τ is identical to τ' on W'_{1-i}, and in particular τ is winning for Player $1-i$.

Otherwise, if π never reaches B, and starts from W''_{1-i}, then π is also a play in $\mathcal{G}\restriction_{V''}$, where τ is identical to τ'', and in particular is winning from W''_{1-i}, and we conclude the claim. □

Now that we understand the algorithm, let's demonstrate it on an example.

Example 12.2. Consider the game depicted in Figure 23. The run of Algorithm 1 on this game proceeds as follows.

$L = \Omega^{-1}(0) = \{v_4, v_8\}$.
$\mathsf{Attr}_0(L) = \{v_3, v_4, v_6, v_7, v_8\}$.
$V' = \{v_0, v_1, v_2, v_5\}$.
Solving $\mathcal{G}\restriction_{V'}$ recursively yields: $W'_0 = \emptyset$ and $W'_1 = \{v_0, v_1, v_2, v_5\}$.
$B = \mathsf{Attr}_1(W'_1) = \{v_0, v_1, v_2, v_4, v_5, v_8\}$.
$V'' = \{v_3, v_6, v_7\}$.
Solving $\mathcal{G}\restriction_{V''}$ recursively yields: $W''_0 = \{v_3, v_6, v_7\}$ and $W''_1 = \emptyset$.
We have $\mathsf{Win}_0 = \{v_3, v_6, v_7\}$ and $\mathsf{Win}_1 = \{v_0, v_1, v_2, v_4, v_5, v_8\}$. △

Finally, lets analyze the complexity of Algorithm 1: assume $|V| = n$ and that there are d priorities. In the first recursive call, we reduce the size of V to at most $n-1$, and reduce the number of priorities to at most $d-1$. In the second recursive call, we only reduce the number of vertices. Thus, the number of recursive calls $T(n,d)$ satisfied $T(n,d) \le T(n-1,d-1) + T(n-1,d)$, with $T(0,d) = T(n,1) = 0$ (if there is only one priority, the winner is determined immediately). It is easy to prove by induction that $T(n,d) \le n^d$. Indeed,

$$\begin{aligned} T(n,d) \le& (n-1)^{d-1} + (n-1)^d = (n-1)^{d-1} + (n-1)^{d-1}(n-1) = \\ =& (n-1)^{d-1}(1+n-1) = n(n-1)^{d-1} \le n \cdot n^{d-1} = n^d \end{aligned}$$

This means that the overall complexity is at most $n^d \mathrm{poly}(n)$, as in each iteration we only compute attractors.

13 LTL Synthesis Revisited

Recall the LTL realizability and synthesis problems from 8.2: we are given an LTL formula φ over $I \cup O$, and we want to decide whether there exists an I/O transducer $\mathcal{T}$ that realizes φ, and if so – to find one.

We are now ready to solve this problem.

Theorem 13.1. LTL *realizability is* **2EXPTIME**-*Complete. When a formula is realizable, we can compute a transducer that realizes it.*

Proof. We only prove the upper bound. For the lower bound (obtained by reduction from the word problem for Alternating Exponential-Space TMs) see [22].

Let φ be an LTL formula of length n over $I \cup O$. Consider a DPW $\mathcal{D}_\varphi = \{Q, 2^{I \cup O}, \delta, q_0, \Omega\}$ that is equivalent to φ. We obtain from $\mathcal{D}_\varphi$ a parity game $\mathcal{G}$ as follows: the vertices of the game are $V = V_1 \cup V_0$ with $V_0 = Q \times 2^I$ and $V_1 = Q$, and we have the edges $E = \{(q,(q,i)) : q \in V_1, i \in 2^I\} \cup \{((q,i), \delta(q, i \cup o)) : (q,i) \in V_0, o \in 2^O\}$. The parity ranks in $\mathcal{G}$ are determined according to the state component. That is, the ranking functions is $\alpha : V \to \{1, \ldots, d\}\}$ where $\alpha(q) = \Omega(q)$ and $\alpha((q,i)) = \Omega(q)$.

Intuitively, in every vertex $q \in V_1$, Player 1 (the environment) chooses an input $i \in 2^I$. Then, the game moves to the Player 0 vertex (q,i), from which Player 0 chooses an output $o \in 2^O$, at which point the game moves to $\delta(q, i \cup o)$.

Observe that a memoryless winning strategy for Player 0 from q_0 exactly corresponds to an I/O transducer, which outputs whatever Player 0 decides to play in each state (q,i).

We thus have that Player 0 wins from q_0 iff φ is realizable, if so, a memoryless strategy corresponds to a transducer.

It remains to figure out the complexity of this procedure (that is, constructing $\mathcal{D}_\varphi$, and solving the corresponding game).

The size of $\mathcal{D}_\varphi$ is $2^{2^{O(n)}}$, by Theorems 4.20 and 6.12. However, the *index* of $\mathcal{D}_\varphi$ is $2^{O(n)}$ (again by Theorems 4.20 and 6.12). The complexity of solving the game (using Algorithm 1) is then $(2^{2^{O(n)}})^{2^{O(n)}} = 2^{2^{O(n)}}$, which is double exponential. □

The horrible complexity of synthesis could have maybe been forgiven, if the resulting transducers would have been small. This, of course, is not the case.

Theorem 13.2. *There exists a realizable* LTL *formula φ of length $O(n^2)$ over $I \cup O$ whose minimal realizing transducer is of size $\Omega(2^{2^n})$.*

Proof. Recall that in Theorem 6.14, we showed there exists an LTL formula φ of size n^2 such that the minimal equivalent DPW has at least 2^{2^n} states. Specifically, φ was over the alphabet $\Sigma = \{0, 1, \#, \$\}$ (where Σ is not of the form 2^{AP}, but can easily be described as such using two atomic propositions), and the language of φ is

$$L_n = \{\{0,1,\#\}^* \cdot \# \cdot x \cdot \# \cdot \{0,1,\#\}^* \cdot \$ \cdot x \cdot \#^\omega \ : \ x \in \{0,1\}^n\}$$

We modify this language slightly, and define:

$$K_n = \{\{0,1,\#\}^* \cdot \# \cdot x \cdot \# \cdot \{0,1,\#\}^* \cdot \$ \cdot x \cdot \# \cdot \{0,1,\#,\$\}^\omega \ : \ x \in \{0,1\}^n\}$$

That is, once $\$x\#$ is seen, the rest of the word is arbitrary (instead of $\#^\omega$). Clearly K_n is still definable by an LTL formula φ of length $O(n^2)$.

Denote by I the propositions of φ, and let $O = \{o\}$ be a new proposition. Define $\psi := \varphi \longleftrightarrow \mathsf{F}o$.

Thus, ψ is satisfied by a computation $\pi \in (2^{I \cup O})^\omega$ iff either $\pi|_I \models \varphi$ and o occurs in π, or $\pi|_I \not\models \varphi$ and o does not occur in π.

Observe that φ is realizable. Indeed, a realizing transducer can "remember" each word (of length exactly n) that appears between two # symbols, and once \$ is seen, compare the next n letters to one of the remembered words. Then, if a match is found, output o when # is seen. Otherwise, the transducer always outputs $\emptyset$.

We now claim that the minimal transducer is of size at least 2^{2^n}. The proof is practically identical to that of Theorem 6.14: if (by way of contradiction) $\mathcal{T}$ is a realizing transducer with less than 2^{2^n} states, then we can find two sequences $x_0\#x_1\#\ldots\#x_k$ and $y_0\#y_1\#\ldots\#y_m$ where the x_i and y_i are strings over $\{0,1\}^n$, such that w.l.o.g. there exists a string z that appears within the x_i, but not within the y_i, and such that the run of $\mathcal{T}$ on these strings leads to the same state of $\mathcal{T}$. Then, an environment input of z will either cause $\mathcal{T}$ to output o, or not. Either answer is wrong. □

14 ω-Regular Games

While parity games are enough for our analysis of LTL synthesis, other acceptance conditions can also be useful for games, in certain scenarios.

We now show that any ω-regular game, can be reduced, in a way, to parity games.

14.1 The Composition Game

Consider a game $\mathcal{G} = \langle V, V_0, V_1, E, \mathsf{Win}\rangle$, and suppose Win can be described by a parity automaton. That is, we are given a DPW $\mathcal{D} = \langle Q, V, \delta, q_0, \Omega\rangle$ (notice that the alphabet of $\mathcal{D}$ is V – the vertices of the game) such that $L(\mathcal{D}) = \mathsf{Win}$.

We define the *composition game* $\mathcal{D} \circ \mathcal{G} = \langle Q \times V, Q \times V_0, Q \times V_1, E', \textsc{Parity}(\Omega')\rangle$ such that:

composition game

- $((q, v), (q', v')) \in E'$ iff $(v, v') \in E$ and $q' = \delta(q, v)$.
- $\Omega'((q, v)) = \Omega(q)$

Intuitively, $\mathcal{D} \circ \mathcal{G}$ proceeds similarly to $\mathcal{G}$, but in addition the DPW $\mathcal{D}$ keeps track of the state of $\mathcal{D}$ while reading the play of the game.

The main property of the composition game is the following.

Lemma 14.1. *In the notations above, Player i wins from a vertex (q, v) in $\mathcal{D} \circ \mathcal{G}$ iff Player i wins from vertex v in $\mathcal{G}$ for $i \in \{0, 1\}$.*

Proof. We prove the claim for $i = 0$. The claim for $i = 1$ is completely dual. Consider a vertex $(q, v) \in \mathsf{Win}_i$ and a memoryless winning strategy σ for Player 0 from (q, v).

We construct a strategy σ' in $\mathcal{G}$ which uses $\mathcal{D}$ as "memory": σ' keeps track of the state of $\mathcal{A}$ by updating it with the current state, and outputting whichever state σ dictates in $\mathcal{A} \circ \mathcal{G}$ (this is not completely formal, but it's easy to formulate similarly to the transducers discussed in Section 8.2).

Since σ is winning for Player i, then any sequence of states (of $\mathcal{A}$) induced by it is accepted by $\mathcal{A}$ (if $i = 1$, this changes to "rejected"), and is hence winning (by definition) in $\mathcal{G}$. It follows that σ is winning in $\mathcal{G}$.

Modifying the proof also for Player 1, we get that for every $i \in \{0, 1\}$, if Player i wins from (q, v) in $\mathcal{D} \circ \mathcal{G}$, then Player i wins from v in $\mathcal{G}$. But this already implies the "only if" direction, and we are done. □

The type of strategy obtained in the proof of Lemma 14.1 is not memoryless, but does have a *finite* auxiliary memory structure. This type of strategy is called finite-memory, and is the "next best thing" after memoryless strategies.

As we showed in 4, every ω-regular language can be described by a DPW. Hence, we conclude with the following.

Theorem 14.2. *Let $\mathcal{G} = \langle V, V_0, V_1, E, \mathsf{Win}\rangle$ be a game where Win is an ω-regular objective, then Win_0 and Win_1 are computable, and the players have finite-memory winning strategies.*

14.2 Nondeterministic Automata and Games

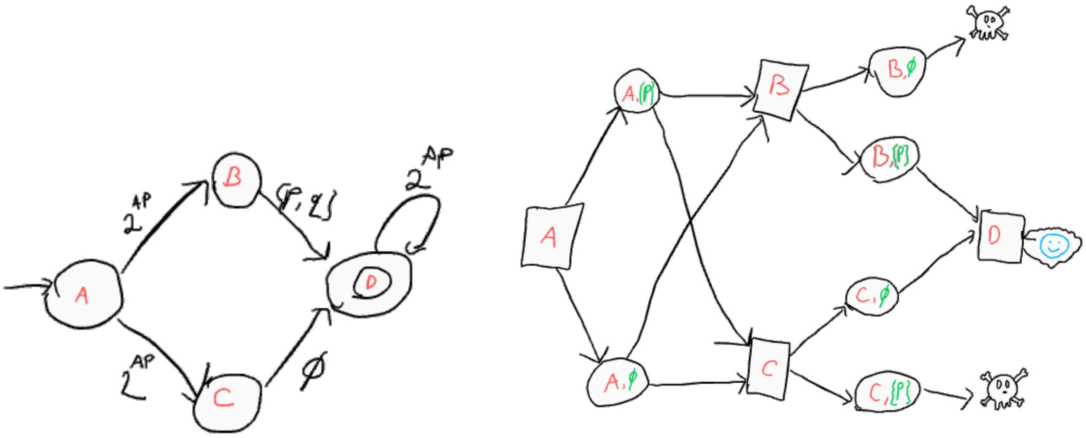
Observe that throughout our study of games, whenever we used automata to obtain games (in particular, in Theorem 13.1 and Section 8.2) we always used deterministic automata. One may wonder why not use nondeterministic automata?

We start by demonstrating where this approach fails.

Example 14.3. Consider the NBW in Figure 24(a), which models (stupidly) the formula $\mathsf{X}(p \longleftrightarrow q)$.

Suppose $I = \{p\}$ and $O = \{q\}$. If we attempt to convert this NBW to a game by state-splitting, as we did in Theorem 13.1, we obtain the game in Figure 24(b).

Observe that in the obtained game, Player 0 does not have a winning strategy, despite the formula being realizable. Indeed, the reason is that Player 0 needs to resolve the nondeterminism of state A (in states $(A, \{p\})$ and $(A, \emptyset)$) in addition to responding with an output. However, Player 0 does not know how to resolve the nondeterminism, since the nondeterministic automaton requires that she "commits" to the output before seeing the input. △



(a) An NBW for $\mathsf{X}(p \longleftrightarrow q)$.

(b) The game obtained from Figure 24(a).

Figure 24: A game obtained from a nonodeterministic automaton.

The fundamental reason for the problem in Example 14.3 is that nondeterminism is much “stronger” than strategies, in that sometimes one needs to “know the future” in order to select an accepting run. Indeed, this is the source of strength of nondeterministic models. Thus, requiring the player to resolve nondeterminism in addition to selecting an output is just too much.

This problem has a partial workaround by means of *good-for-games* automata [7], which allow only nondeterminism that can be resolved in a “strategic” way.

1. Alternating automata:
   - Definition as extension of nondet.
   - Example: UCW for infinitely many a.
   - Alternation removal from ABW (Miyano Hayashi 1984)
   - Corollary: NCW=DCW: NCW to UBW to DBW to DCW.
   - NBW complementation: NBW to AWW with n^2 blowup, and then AWW can be seen as ACW, so comeplement ABW obtained in $O(1)$, and then alternation removal Kupferman-Vardi 2001
2. Tree automata:
   - Definitions. The run on a tree is a Q-labelled tree.
   - Example: NBT Language of trees where there exists a branch with infinitely many a’s.
   - Example: DMT trees where each branch has finitely many a’s
   - No NBT for latter, since guesses need to be verified on entire subtree.
   - NPT and parity games: given a tree t we can check whether it’s accepted by building a game where Player 0 tries to resolve nondeterminism, and Player 1 chooses a branch. Player 0 wins if there is a resolution of the nonodeterminism at each state, such that every branch is accepting.
   - Similarly, we can allow Player 0 to also choose the letter, thus creating an emptiness-game.
   - Memoryless strategy for parity games implies a regular witness.
   - Rabin’s theorem: translation between S2S and NPT.
   - There are also alternating tree automata, of course. And they are useful.
3. Weighted automata:
   - model.

- example. No determinization.

4. Dynamical systems:
   - Linear loops as programs.
   - Reachability analysis.
   - LRS. Eigenvalues. Skolem problem.
   - Connections to number theory, Baker. Scary stuff.
   - Semialgebraic sets and reachability. Quantifier elimination. Math.

5. Planning as synthesis.

References

[1] Udi Boker. Why these automata types? In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57 of *EPiC Series in Computing*, pages 143–163. EasyChair, 2018. URL: https://easychair.org/publications/paper/G5dD, doi:10.29007/c3bj.

[2] J Richard Büchi. On a decision method in restricted second order arithmetic. In *The Collected Works of J. Richard Büchi*, pages 425–435. Springer, 1990.

[3] Cristian S Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasi-polynomial time. *SIAM Journal on Computing*, (0):STOC17–152, 2020.

[4] Ashok K Chandra and Larry J Stockmeyer. Alternation. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 98–108. IEEE, 1976.

[5] E Allen Emerson and Charanjit S Jutla. Tree automata, mu-calculus and determinacy. In *FoCS*, volume 91, pages 368–377. Citeseer, 1991.

[6] Timothy Gowers. Determinacy of borel games. URL: https://gowers.wordpress.com/2013/08/23/determinacy-of-borel-games-i/.

[7] Thomas A Henzinger and Nir Piterman. Solving games without determinization. In *International Workshop on Computer Science Logic*, pages 395–410. Springer, 2006.

[8] Marcin Jurdziński. Deciding the winner in parity games is in UP $\cap$ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.

[9] Marcin Jurdziński. Small progress measures for solving parity games. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer, 2000.

[10] Orna Kupferman. Automata theory and model checking. In *Handbook of Model Checking*, pages 107–151. Springer, 2018.

[11] Orna Kupferman and Moshe Y Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic (TOCL)*, 2(3):408–429, 2001.

[12] LH Landweber. Decision problems for ω-automata. *Mathematical systems theory*, 3(4):376–384, 1969.

[13] Donald A Martin. Borel determinacy. *Annals of Mathematics*, pages 363–371, 1975.

[14] Felix Klein Martin Zimmermann and Alexander Weinert. Infinite games – lecture notes, 2016. URL: https://www.react.uni-saarland.de/teaching/infinite-games-16/lecture-notes.pdf.

[15] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[16] Albert R Meyer. Weak monadic second order theory of succesor is not elementary-recursive. In *Logic colloquium*, pages 132–154. Springer, 1975.

[17] Max Michel. Complementation is more difficult with automata on infinite words. *CNET, Paris*, 15, 1988.

[18] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32(3):321–330, 1984.

[19] Andrzej Włodzimierz Mostowski. *Games with forbidden positions.* 1991.

[20] Pawel Parys. Parity games: Zielonka's algorithm in quasi-polynomial time. In *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[21] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[22] Roni Rosner. *Modular synthesis of reactive systems.* PhD thesis, PhD thesis, Weizmann Institute of Science, 1992.

[23] Shmuel Safra. On the complexity of!-automata. In *Proc. 29th IEEE Symp. Found. of Comp. Sci*, pages 319–327, 1988.

[24] Sven Schewe. Büchi complementation made tight. In *26th International Symposium on Theoretical Aspects of Computer Science*, page 661.

[25] Sven Schewe. Tighter bounds for the determinisation of büchi automata. In *International Conference on Foundations of Software Science and Computational Structures*, pages 167–181. Springer, 2009.

[26] A Prasad Sistla and Edmund M Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.

[27] Wolfgang Thomas. Handbook of theoretical computer science, volume b, chapter automata on infinite objects, 1990.

[28] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.

[29] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.