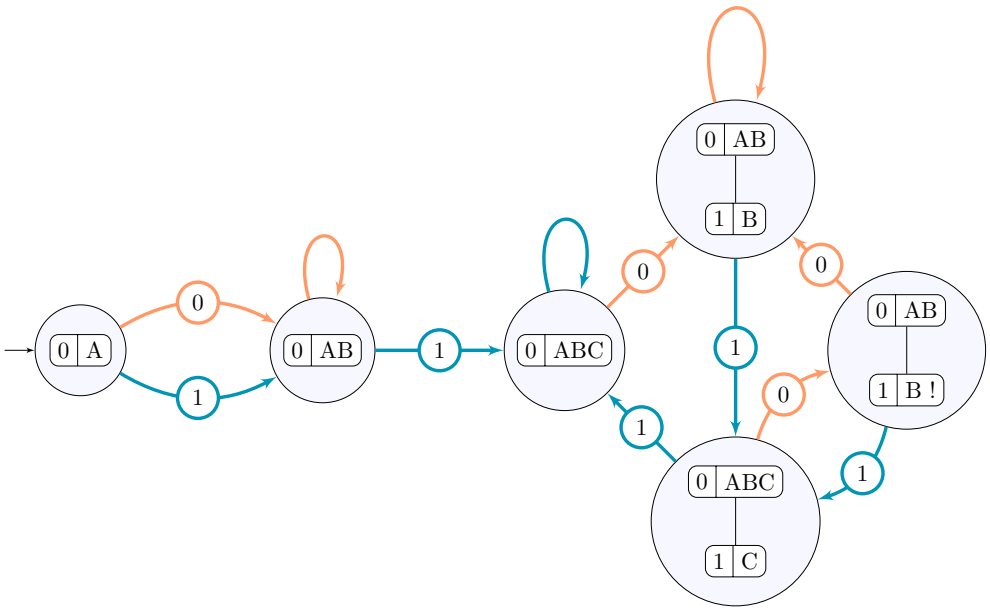
Semester Project – Spring 2022

Between decideable logics: ω-automata and infinite games

With 31 Illustrations



Author
Diego Dorn

EPFL

Supervisor
Jacques Duparc

Introduction

In the world of logics, it is quite a surprising fact when one of them is decidable. It is even more remarkable when such decidable logic is expressive enough to be used in real world applications, and not only as a toy for logicians. We present here two decidable logics, *monadic second order logic* over infinite strings and *μ-calculus*. Those two logics are tightly related to each other and can be studied by means of infinite games of perfect information between two player and also automata that read infinite words.

The results developped here show the potential of those logic for tools of formal verification: they can express a lot of properties that we would like to formally prove on our computer programs, and since they are decidable, we can write complete automated tools to verify that those properties hold. This model checking of μ-calculus can be done in quasipolynomial time, but it is still an open question if it is possible to do it in polynomial time.

Contents

Conventions

Ordinals We use $\mathbb{N}$ to denote the set of integers, which contains 0. It is the same set as ω, the first infinite ordinal. We use them interchangebly, but ω makes focus on the order whereas $\mathbb{N}$ is used more as a set.

Sequences A sequence, or ω-word, on a set X is a function from ω to X. We denote the set of sequences on X as X^ω and given $x \in X$ and an integer $n \in \mathbb{N}$, we write x_n instead of $x(n)$ to have less parentheses.

A finite sequence, or word, on a set X is a function $x : \{0, ..., n\} \to X$, where n is the length of the finite sequence, also denoted $|x|$. The set of finite sequences on X is $X^{<\omega}$. Given two finite sequences s and t, we write $s^\frown t \in X^{<\omega}$ for the concatenation of the two finite sequences.

Given a sequence x, finite or infinite, and an integer $n \in \mathbb{N}$, we write $x{\restriction_n}$ for the sequence x restricted to the first n elements. If $n < m$, we write $x_{n..m}$ for the subsequence of x that starts at position n and ends at position m, both included.

Sets An alphabet is any finite set, whose elements are called letters or symbols. Given a set X, $\mathcal{P}(X)$ is the power set of X, which contains every subset of X. We use the symbol $\subseteq$ for the non-strict inclusion of one set to the other and $\subsetneq$ for the strict inclusion.

Formulas Formulas are trees, where each node contains a symbol, variable, logical connector, quantifer or term. Given a formula φ and two terms x and y, we write $\varphi[x := y]$ for the formula where every occurence of x is replaced by y.

1. Automata on infinite words

1.1. Three kinds of automata

In the following sections we will make good use of automata that read infinite words. Those automata are very similar to the more common finite state machines but differ in how they accept or reject their inputs.

Definition 1.1. Let Σ be a finite alphabet. An **ω-automaton** $\mathcal{A} = (Q, \Sigma, \delta, q_0, Acc)$ consists of:

- a finite set of states, Q;
- a finite alphapet, Σ;
- a transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$;
- an initial state $q_0 \in Q$;
- a set $Acc \subseteq Q^\omega$ of accepting runs.

A run of the ω-automaton on an input $w \in \Sigma^\omega$ is any sequence of states $r \in Q^\omega$ such that $r_0 = q_0$ and for all $n > 0$, $r_n \in \delta(r_{n-1}, w_n)$. Such run is **accepting** if it belongs to Acc. Otherwise, if $r \notin Acc$, it is **rejecting**.

We say that $\mathcal{A}$ accepts the infinite word $w \in \Sigma^\omega$ if there exists an accepting run on w.

An ω-automaton is **deterministic** if for all letters $s \in \Sigma$ and each state $q \in Q$, there is exactly one transition from q to another state when s is read. That is, $|\delta(q, s)| = 1$. Otherwise, it is **non-deterministic**.

This provides a general context in which automata can read infinite words. However, it is too general because the accepting condition can be arbitrarily complex and even non-computable. We will mostly look at three types of accepting conditions:

- Buchi conditions
- Muller conditions
- Parity conditions

Each of these conditions defines the Acc set from simpler data and rely on the set of states that a run passes through infinitely many times. To make the notation more convenient, given a sequence w, we write $\mathrm{Inf}(w)$ to denote the elements of w that appear at infinitely many indices in w.

Definition 1.2. Let $w \in X^\omega$ be sequence on any set X. We define $\mathrm{Inf}(w) \subseteq X$ as

$$\mathrm{Inf}(w) := \{x \in X \mid \forall N \in \mathbb{N}\ \exists n > N\ w_n = x\}$$

Definition 1.3 (Acceptance conditions)**.** An ω-automaton has a **Büchi acceptance condition** if for some set of states $F \subseteq Q$, a run passes through F infinitely many times:

$$Acc := \{r \in Q^\omega \mid \mathrm{Inf}(r) \cap F \neq \varnothing\}.$$

An ω-automaton has a **Muller acceptance condition** if for some collection of subsets of the states $F \subseteq \mathcal{P}(Q)$, a run is accepted if and only if the set of states that are visited infinitely many times belongs to F.

$$Acc := \{r \in Q^\omega \mid \mathrm{Inf}(r) \in F\}.$$

An ω-automaton has a **parity acceptance condition** if for some function $p : Q \to \mathbb{N}$ called the priority function, a run is accepted if and only if the state with the lowest priority that is visited infinitely many times belongs has an even priority:

$$Acc := \left\{ r \in Q^\omega \;\middle|\; \min_{q \in \mathrm{Inf}(r)} p(q) \text{ is even} \right\}.$$

Remark. As a shorthand, we will say "a Büchi automaton" instead of "an ω-automaton with a Büchi acceptance condition", and similarly for each other acceptance conditions.

Note that by default we consider non-deterministic automata, so we will mention every time we consider deterministic automata.

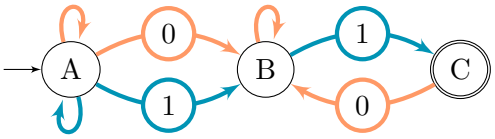
Definition 1.4. Let $\mathcal{A}$ be an ω-automaton on an alphabet Σ. The **language** of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all infinite words accepted by $\mathcal{A}$.

$$\mathcal{L}(\mathcal{A}) := \{w \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } w\}$$

We say that a ω-language $L \subseteq \Sigma^\omega$ is **Büchi-definable** if there is a Büchi automaton $\mathcal{A}$ whose language is L, that is, $\mathcal{L}(\mathcal{A}) = L$ and similarly for the other acceptance conditions.

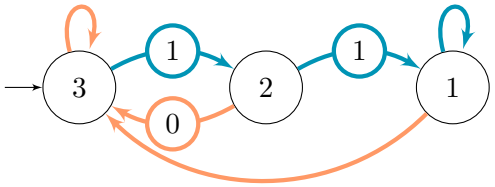
Examples. The following automata all recognise the same language, which consist of words made of `0`s and `1`s that contain infinitely many times the letter `1`, but only finitely many time the sequence `11`.

To facilitate the reading of the automata, we always use orange arrows for transitions when reading the letter `0`, and blue arrows for the letter `1`, and we may sometimes omit the labels. The initial state is denoted by an incoming black arrow and for Büchi automata, accepting states are double circled.



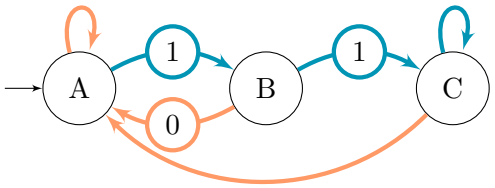
The above automaton is a Büchi automaton, so it accepts a word if and only if there is a way to follow the arrows that passes infinitly many times through the state C, which is the only accepting state. One can notice that there is no path from C to A, so any infinite run that passez through C just alternates between B and C. Since there is only one arrow reaching C, with a `1` on it, any word accepted must have infinitely many `1`s. We also notice that when cycling between states B and C, it is impossible to have two `1`s in a row, as one would make the transition from B to C, but there is not arrow with a `1` leaving C, and the run would just end. What is the purpose of the state A then ? It allows to skip any finite prefix of the input word, where anything could happen (for instance some `11`s). Once all the occurences of `11` have been read, and the tail of the word consists only of zeroes and isolated ones, the automaton can transition to B and then C.

With a deterministic parity automaton, the same language can be recognised (in fact, the next section is about showing that the two notions are equivalent).



This parity automaton, with the priorities written on the nodes also recognise the language with infinitely many `1`s, but finitely many `11`s. Indeed, in order for a run to be accepted, it the smallest priority visited infinitely many times must be even. Here, there is only one even priority, so it must visit the middle node infinitely many times, but not the rightmost node. However, every time a `0` is read, the automton goes beck to node 3, and every time a `1` is read, it moves towards node 1. So in order to avoid visiting node 1 infinitely many times, there must be a finite number of `11` in the word. We can also see that if the tail of the word contain only zeros, the automaton will stay in state 3, and reject the word, so an accepted word must have infinitely many ones.

A Muller automata that recognise the same language is very similar to the parirty automaton, but only the definition of the acceptance condition changes.



Here we define the family of accepting states as $\mathcal{F} = \big\{ \{A, B\} \big\}$. This is mostly an other way of expressing the condition of the parity automaton: we want an accepting run to visit infinitely many times the two states on the left, and only finitely many times the states on the right. In general, parity automata are a special case of Muller automata, when the family of accepting subsetsets of state can be defined by a priority function.

Lemma 1.5. Any parity automaton is also a Müller automaton.

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, p)$ be parity automaton. Its acceptance condition is also a Muller acceptance condition as we can take

$$\mathcal{F} = \left\{ F \subseteq Q \,\middle|\, \min_{q \in F} \; p(q) \text{ even} \right\}$$

which is a Muller condition. □

One might then wonder why we would use parity automata in the first place, but they have the advantage of being easier to understand for a human brain because of the more explicit structure of the set of accepting runs. They are also easier to represent and sometimes easier to reason about, for instance, they are a bit simpler to convert to Büchi automata than Muller automata are.

1.2. Equivalences between automata

The goal of the following section is to show the equivalences between different notions of ω-automata. Two kinds of ω-automata, for instance, Muller and parity automata are equivalent if they can recognise the exact same languages, that is, a language is Muller-definable if and only if it parity-definable. We will show the following:

Theorem 1.6. Let $L \subseteq \Sigma^\omega$ be a language.

$$\begin{gathered} L \text{ is recognised by a Büchi automaton} \\ \Longleftrightarrow \\ L \text{ is recognised by a deterministic Muller automaton} \\ \Longleftrightarrow \\ L \text{ is recognised by a deterministic parity automaton} \end{gathered}$$

However, some languages are recognised by the above automata that cannot be recognised by any deterministic Büchi automaton.

This section will focus on showing all those equivalences, and provide explicit constructions to convert one kind of automata into the other. The contruction to transform a Büchi automaton into a deterministic Muller automaton, however, is much more complex and will be carried in it own Section 1.3.

Lemma 1.7. Let $\mathcal{M}$ be a deterministic Muller automaton, then there exists a deterministic parity automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A})$.

Proof. The proof is carried out in two steps. First we assume that $\mathcal{M} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ with $|F| = 1$, that is, there is only one set of states that can occur infinitely many times. We then show that parity automata are closed under disjunctions and conclude that if $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ we can construct a parity automaton from the automata $\mathcal{A}_i$ made from $\mathcal{M}$ and replacing $\mathcal{F}$*by* $\{F_i\}$.

Case of one accepting subset Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be a Muller automaton with $\mathcal{F} = \{F\}$. Let also $F = \{f_1, f_2, \ldots, f_n\}$. We define $\mathcal{A} = (Q', \Sigma, \delta', q_0', p)$ as follows:

- The set of states is $Q' = Q \times F$.
- The initial state is $q_0' = (q_0, f_0)$.
- The priority of a state $(q, f) \in Q'$ is

$$p\big((q,f)\big) = \begin{cases} 1 & \text{if } q \notin F \\ 2 & \text{if } (q,f) = (f_0, f_0) \\ 3 & \text{otherwise.} \end{cases}$$

- Since the automaton is deterministic, we give the transition function as $\delta' : Q' \times \Sigma \to Q'$. Let $q \in Q$, $f_i \in F$ and $a \in \Sigma$. We set

$$\delta'\big((q, f_i), a\big) = \begin{cases} \big(\delta(q,a), f_{i+1}\big) & \text{if } q = f_i \text{ and } i < k \\ \big(\delta(q,a), f_0\big) & \text{if } q = f_i \text{ and } i = k \\ \big(\delta(q,a), f_i\big) & \text{otherwise.} \end{cases}$$

Correctness of the construction Let $w \in \Sigma^\omega$ be an ω-word accepted by $\mathcal{A}$ and let $r' \in {Q'}^\omega$ be the corresponding run. By the definition of δ', the run $r \in Q^\omega$ of w on $\mathcal{M}$ is exactly the projection of r' on its first coordinate, that is, $r = \pi_1(r')$. Since r' is accepted by $\mathcal{A}$, we know that r' visits finitely many times states of priority 1, which are the states in $(Q \setminus F) \times F = \pi_1^{-1}(Q \setminus F)$. Therefore $\mathrm{Inf}(\pi_1(r')) \subseteq F$ and equivalently $\mathrm{Inf}(r) \subseteq F$. For the other inclusion, we have that r' visits infinitely many times the only state of even priority, (f_0, f_0). However, every time it does so, the run proceeds into $Q \times \{f_1\}$, which can be escaped only when reaching (f_1, f_1), and then proceeds to the subset $Q \times \{f_2\}$, and so on. Therefore, it has visited every state of the form (f_i, f_i) before reaching (f_0, f_0) again, and thus $\mathrm{Inf}(r) = \pi_1(r') = F$

For the reverse implication, assume that $w \in \Sigma^\omega$ is accepted by $\mathcal{M}$ on the run $r \in Q^\omega$ and let $r' \in {Q'}^\omega$ be the run in $\mathcal{A}$. Since $\mathrm{Inf}(r) = F$ and $\mathrm{Inf}(r') \subseteq \mathrm{Inf}(\pi_1^{-1}(r)) = F \times F$, we know that r' visits nodes of priority 1 finitely many times. It remains to show that r' visits (f_0, f_0) infinitely many times. Let n be an integer. We show that, if $r'_n = (q, f_k)$, for some k, then there is some $m > n$ such that $r'_m = (f_k, f_k)$. Indeed, appart from the transions from (f_k, f_k), the transitions inside the subset $Q \times \{f_k\}$ of Q' match exactly those of the Muller automaton $\mathcal{M}$. Since in $\mathcal{M}$ the run r visits f_k, it must also be the case in $Q \times \{f_k\}$. Finally, this fact shows that every state of the form (f_k, f_k) is visited infinitely many times, and therefore r' visits (f_0, f_0) infinitely many times and accepts w.

Parity automata are closed under disjunctions Let $\mathcal{A}_0 = (Q_0, \Sigma, \delta_0, q_0, p_0)$ and $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_1, p_1)$ be two parity automata. We build an automata $\mathcal{B} = (Q, \Sigma, \delta, q, p)$ such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{A}')$. The construction follows the cartesian product of the automata, with a specific priority function:

- The set of states is $Q = Q_0 \times Q_1$.
- The initial state is $q = (q_0, q_1)$.
- The transition function is $\delta = (\delta_0, \delta_1)$, that is, for any state $(s, s') \in Q$, and any symbol $a \in \Sigma$,
$$\delta((s, s'), a) = \big(\delta_0(s, a), \delta_1(s', a)\big)$$
- The priority function $p : (Q_0 \times Q_1) \to \mathbb{N}$ is any function that verify those three conditions:
  1. it assigns to each pair $(s, s') \in Q$ an even number if and only if either $p(s)$ or $p(s')$ is even.
  2. for each $t \in Q_0$, $p(s, s') < p(t, s')$ if and only if $p_0(s) < p_0(t)$ so that the order is preserved.
  3. symetrically, for each $t \in Q_1$, $p(s, s') < p(s, t)$ if and only if $p_1(s') < p_1(t)$.

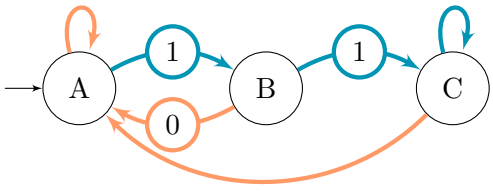
  To give an explit function, let $N := 2 \cdot \max_{q \in Q_1} p_1(q) + 2$ and $(s, s') \in Q$. We set
$$p(s, s') = \begin{cases} N \cdot p_0(s) + p_1(s') & \text{if } p_0(s) \text{ odd} \\ N \cdot p_0(s) + 2 \cdot p_1(s') & \text{if } p_0(s) \text{ even} \end{cases}$$

Correctness of the construction Let $w \in \Sigma^\omega$ be an ω-word and run r, r_0 and r_1 be the runs on $\mathcal{B}, \mathcal{A}_0$ and $\mathcal{A}_1$ respectively. By construction, the two coordinates of the run r are the runs r_0 and r_1.

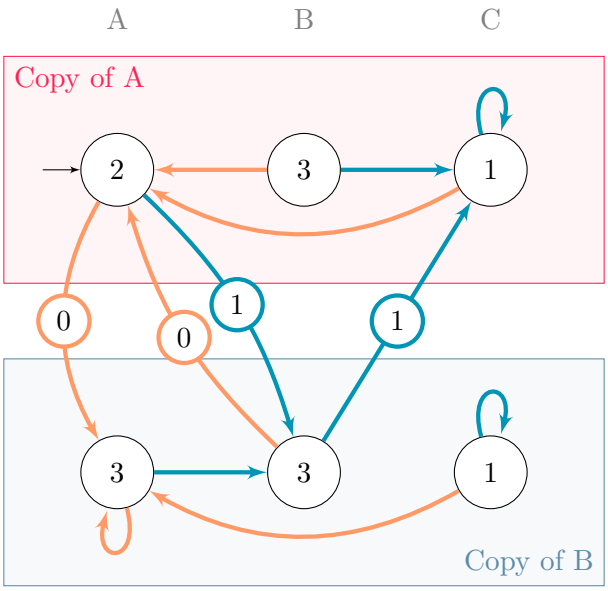
The minimal priority of $\mathrm{Inf}(r)$ is obtained on some state $(s, s') \in Q$. By the first condition of the priority function, $p(s, s')$ is even if and only if either $p_0(s)$ or $p_1(s')$ is even and by condition 2 and 3, s (resp. s') is also a state of $\mathrm{Inf}(r_0)$ (resp. $\mathrm{Inf}(r_1)$) with minimal priority. Therefore, r is accepted if and only if r_0 or r_1 is accepted, and $w \in \mathcal{L}(\mathcal{A}_0) \cup \mathcal{L}(\mathcal{A}_1)$.

□

Example. We illustrate the construction to convert a Müller automaton with only one accepting subset into a parity automaton with the now familiar automaton to recognise words with infinitely many `1`s but finitely many `11`s. We recall that this Muller automaton is



With acceptance condition $\mathcal{F} = \{\{A, B\}\}$. The resulting parity automaton must therefore have six states, and two copies of the Muller automaton. One of this copy will have the transitions going out of the A node modified to point the the second copy, and the B node of the second copy will have its transitions modified to point to the first copy. Copies of C have priority 1, as C does not belong to the accepting condition, and we set the only node with priority 2 to be the A node in the first copy. Every other node has a priority of three.



We can notice in this example that even if it uses the same set of priorities $\{1, 2, 3\}$, as the parity automaton we have already constructed, the resulting automaton is more complex. A few states, however, are not reachable at all from the start (BA and CB), but this is just what happens to be in this specific example.

In the next lemma, we prove the second implication, that is, we can convert a deterministic parity automaton into Büchi automaton.

Lemma 1.8. Let $\mathcal{A}$ be a deterministic parity automaton, then there exists a Büchi automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.

Proof. Given $\mathcal{A} = (Q, \Sigma, \delta, q_0, p)$ a parity automaton, we build a Büchi automaton $\mathcal{B} = (Q', \Sigma, \delta', q_0', F)$ that recognises the same language. Let Q_i be the states of $\mathcal{A}$ of priority i and $Q_{\geq i} := \{q \in Q \mid p(q) \geq i\}$ the states of $\mathcal{A}$ of priority greater than i. We set

- The set of states of $\mathcal{B}$ is the disjoint union of the $Q_{\geq i}$, for all even i, plus one copy of Q:
$$Q' = Q \sqcup \bigsqcup_{i \text{ even}} Q_{\geq i} = Q \sqcup \left\{(q, i) \in Q \times \mathbb{N} \,\middle|\, p(q) \geq i \wedge i \text{ even}\right\}$$
- The initial state is $q_0' = q_0 \in Q'$.

- The accepting states of $\mathcal{B}$ are the states of priority i in the copy of $Q_{\geq i}$:

$$F = \bigsqcup_{i \text{ even}} Q_i = \{(q, i) \in Q' \mid p(q) = i\}$$

- In each of the $Q_{\geq i}$, the transition function is the same as in $\mathcal{A}$, and the transitions from the copy of Q are the same as in $\mathcal{A}$ but the automaton can also decide to move to one of the $Q_{\geq i}$. So for all all $q \in Q, i \in \mathbb{N}$ and $s \in \Sigma$,

$$\delta'(q, s) = \delta(q, s) \cup \bigcup_{i \text{ even}} (\delta(q, s) \cap Q_{\geq i}) \times \{i\}$$

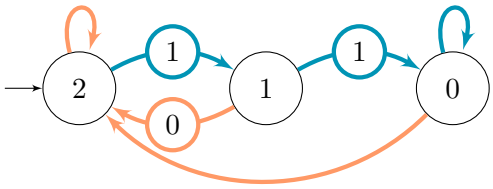
and

$$\delta'((q, i), s) = \big(\delta(q, s) \cap Q_{\geq i}\big) \times \{i\}$$

Correctness of the construction Let $w \in \mathcal{L}(\mathcal{A})$ be and $r \in Q^\omega$ be the corresponding accepting run. Let p_{min} be the minimum priority that appears infinitely many times in r and let $N = \min\{n \in \mathbb{N} \mid \forall k \geq n, p(r_k) \geq p_{min}\}$ be the time from which the run never visit a state with a priority lower than p_{min}. Then the run $q := r \restriction_{N-1} \frown (r_N, p_{min}) \frown (r_{N+1}, p_{min}) \frown \dots$ is an accepting run in $\mathcal{B}$.

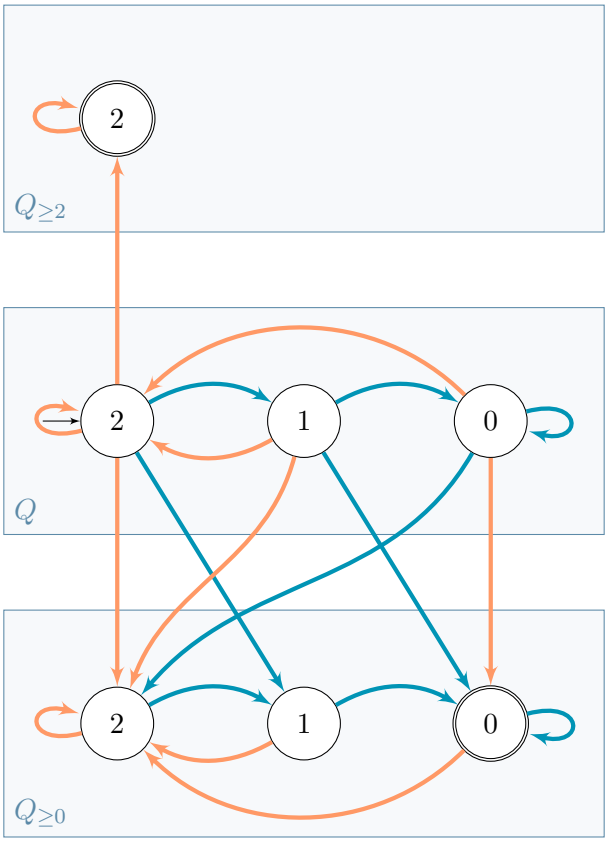
□

Example. In order to illustrate this construction, we will convert the complement of our favourite parity automaton into a Büchi automaton.



This automaton accepts words which if they have infinitely many `1`s, then they have infinitely many `11`s. The automaton can accept words in two different ways, either the minimal priority occurring infinitely many times is 2 or 0. If it is 2, there are finitely many ones, and the tail of the word is only zeros. If it is 0, then there are infinitely many `11`s.

With our construction, we name the nodes with their priority, as no two nodes have the same priority. We have $Q_{\leq 0} = Q = \{0, 1, 2\}$ and $Q_{\geq 2} = \{2\}$, so the Büchi automaton should have 7 states.



Note that the number in the states do not have any specific meaning, they are only reminder of which node from the original automaton they come from.

1.3. The Safra construction

There is one equivalence between automata that we did not prove in the last section: every language that is recognised by a non-deterministic Büchi automaton can be recognised by a deterministic Muller automaton. This result concludes the proof of Theorem 1.6 and is the most important result that we will use in the sequel. Indeed, it allows us to complement non-deterministic Büchi automata which are very hard to complement otherwise (as is usually the case with non-deterministic automata). The proof is also quite interesting by itself and makes use of Safra trees.

Definition 1.9. Let Q be any finite set. A **Safra tree** on Q is a finite tree whose children are ordered and each node has:

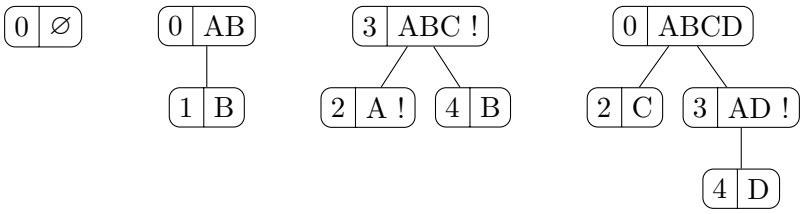
- a name, $n \in \{0, \dots, 2\,|Q|\}$
- a label, $l \subseteq Q$

- an optional marker !

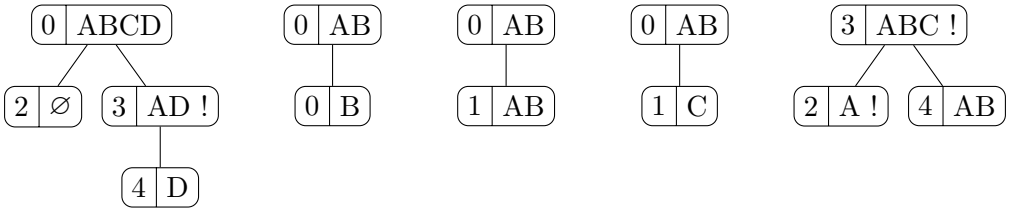
and satisfies the following constraints:

- Every node has a distinct name.
- No node can have a label of $\varnothing$, except the root.
- The label of a node is a subset of the label of its parent.
- Labels of siblings node are disjoint.
- The union of labels of siblings node is a proper subset of the label of their parent.

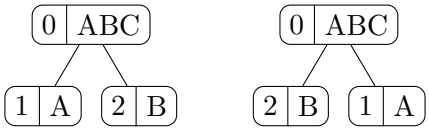
The following trees are Safra trees on $Q = \{A, B, C, D\}$, where the name is written on the left and the label on the right:



We can also have trees that are **not** Safra trees. For instance, each of the following trees violates exactly one of the conditions of a Safra tree:



Note, that, because we consider the order of children, those two Safra trees are different:



This fact will be important for the construction, as the order of the children will coincide with the order they are added.

Lemma 1.10. Given a non-deterministic Büchi automaton $\mathcal{B}$, there is an deterministic Muller automaton $\mathcal{M}$ that recognises the same languages as $\mathcal{B}$, that is, $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{M})$.

Proof. Let $\mathcal{B} = (Q, \Sigma, \delta, q_0, F)$ be a non-deterministic Büchi automaton. We build a deterministic automaton $\mathcal{M} = (Q', \Sigma, \delta', q_0', \mathcal{F}')$ with

- The states Q' is the set of Safra trees on Q.
- The alphabet Σ stays the same.
- The initial state is the Safra tree with only a root labeled with $\mathcal{B}$'s initial state q_0:

- The transition function is deterministic and associates to each Safra tree T and and input letter s a new safra tree in 5 steps:
  1. **Branch off accepting states**: for each node labeled l of T, create a child node with label $F \cap l$ if it is not empty, and take any name in $[1, \dots, 2\,|Q|]$ that is not already taken.
  2. **Power set**: replace each label $l \subseteq Q$ of T by $\bigcup_{q \in l} \delta(q, s)$, the set of state that are reachable from some state in l while reading the letter s.
  3. **Remove states**: remove from each label (and its children) the states that appear also in older siblings. Older siblings are nodes on the left of a given node, or equivalently nodes with a smaller index in the sibling list. At that point, labels of siblings are disjoints.
  4. **Remove empty**: remove all nodes that have an empty label, except the root.
  5. **Mark nodes**: erase all ! marks, then mark each node with a ! if the union of its children labels is the same as the parent label. In that case, remove the whole tree below the marked node.
- The accepting condition is given by the family $\mathcal{F}' \subseteq \mathcal{P}(Q')$. A set $S \subseteq Q'$ of Safra trees belongs to $\mathcal{F}'$ if some node name appears in each tree $s \in S$ and in some tree $s \in S$, the node with this name carries a ! marker.

For detailed examples of this construction, see Section 1.4. We now prove that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{M})$ by double inclusion.

Let $w \in \mathcal{L}(\mathcal{B})$ be an infinite word. Since $w \in \mathcal{L}(\mathcal{B})$, there is a run of the automaton $\mathcal{B}$ that encounters infinitely many times an accepting state $q \in F$. We look at the corresponding run $r \in Q'^{\omega}$ in $\mathcal{M}$ and show that it is accepting. Each of r_0, r_1, ... is a Safra tree. If the root node is marked infinitely many times with a ! in r, then $\mathcal{M}$ accepts w. Otherwise, at some point, q is put on a son of the tree, and the Büchi run stays in a fixed son of the root. Indeed, every time the Büchi run encounters q, the root has a son with label q, as a result from either the *Power set* or *Branch off accepting states* rules. This node can be removed only by the *Remove empty* or *Remove states* rules, but since there is a Büchi run, the *Power set* rule doesn't produce empty states after some point, and therefore the *Remove empty* can only be applied as many times as the *Remove states* rule. Finally, the *Remove states* rule can be applied only finitely

many times because it corresponds to push the Büchi run in an older sibling. We can then proceed recursively: either k_1 is marked infinitely many times and then $\mathcal{M}$ accepts, or it has a son k_2 in which the run ultimately stays. Since Safra trees have a depth of at most $|Q|$, there is a step at which a node is marked infinitely many times, and thus $\mathcal{M}$ accepts w.

For the other inclusion, let $w \in \mathcal{L}(\mathcal{M})$ be an infinite word and $r \in Q'^{\omega}$ the corresponding accepting run in $\mathcal{M}$. We need to construct a path $q \in Q^{\omega}$ in $\mathcal{B}$ that visits an accepting state infinitely many times. To that extent, the following claim will provide a way to find paths in $\mathcal{B}$ from paths in $\mathcal{M}$.

Claim. Let $n < m$ be integers and N be the name of a node that occurs in all the Safra trees $r_n, r_{n+1}, \ldots, r_m$. Let also $P \subseteq Q$ be the label of node N in t_n and $R \subseteq Q$ the label of node N in t_m.

Then for all states $r \in R$ there exists a path from $p \in P$ to r in the automaton $\mathcal{B}$ that can be taken on input $w_n, \ldots, w_{m-1}$. We write this as

$$\mathcal{B} : P \overset{w_{n..m}}{\rightsquigarrow} R \quad \text{for} \quad \forall r \in R \; \exists p \in P \left(\mathcal{B} : p \overset{w_{m..m}}{\rightsquigarrow} r \right)$$

Furthermore, if the node N is marked in both r_n and r_m, then any such path visits at least one accepting state of $\mathcal{B}$.

Proof. We this by induction on $m - n$. For the base case, if $m = n + 1$, each element of R was produced by the *Power set* rule, which means precisely that, $r \in R$ if and only if there is some $p \in P$ such that $r \in \delta(p, w_n)$. For the recursive case, let P' be the label of node N in t_{m-1}. By hypothesis, $\mathcal{B} : P \overset{w_{n..m-1}}{\rightsquigarrow} P'$ and $\mathcal{B} : P' \overset{w_{m-1..m}}{\rightsquigarrow} R$. Since the relation is clearly transitive, it follows that $\mathcal{B} : P \overset{w_{n..m}}{\rightsquigarrow} R$.

For the second part, let $S \subseteq Q$ be the label of a son of node N. Since this son has been created by the *Branch accepting* rule, it shows that every state $s \in S$ is reachable from P through an accepting state. If R is marked, it means that the union of N children is exactly R, and therefore every state in R is reachable from P via an accepting state. □

Since $\mathcal{M}$ accepts w, there is a node name N that appears in every tree of $\mathrm{Inf}(r)$, and is marked with a ! in at least one tree of $\mathrm{Inf}(r)$. Let $P_0, P_1, \ldots$ be the sequence of labels of N every time N is marked. We have

$$\{q_0\} \overset{u_0}{\rightsquigarrow} P_1 \overset{u_1}{\rightsquigarrow} P_2 \overset{u_2}{\rightsquigarrow} P_3 \rightsquigarrow \ldots$$

where the u_k are segments of w. Let $p_i \in Q^{|u_i|}$ be the set of paths from P_i to P_{i+1} that visit an accepting state on input u_i. Those are the path realise the fact that $\mathcal{B} : P_i \overset{u_i}{\rightsquigarrow} P_{i+1}$. Now, if we take two paths $v \in P_i$ and $v' \in P_{i+1}$, those might be stitched together to corresponds to a path on input $u_i{}^\frown u_{i+1}$, if and only if $v_{|u_i|} = v'_0$. In that case we write $v \sim v' := v^\frown v'_1 v'_2 \ldots v'_{|v'|-1}$.

Let T be the set of paths that can that start at q_0 and can be continued by paths from the p_i:

$$T = \bigcup_{k=0}^{\infty} \left\{ v_0 \sim v_1 \sim \cdots \sim v_k \;\middle|\; \begin{array}{c} \forall i \leq k \; v_i \in p_i \\ \wedge \\ v_0, \ldots v_k \text{ can be stitched} \end{array} \right\}$$

We can consider T as a tree, by making it closed under prefix. In this case, each node of T corresponds to a state in $\mathcal{B}$, This tree is infinite, since it has branches of unbounded length, and is finitely branching because the automaton $\mathcal{B}$ has finitely many states. By König's lemma there exists an infinite path q in T. This path q visits an accepting state infinitely many times on input w because it is composed of paths from the sets p_i which each contains at least one visit to an accepting state. Therefore q is an accepting run, and w is accepted by $\mathcal{B}$. □

This concludes the proof of Theorem 1.6:

Theorem (1.6)**.** Let $L \subseteq \Sigma^\omega$ be a language.

$$\begin{array}{c} L \text{ is recognised by a Büchi automaton} \\ \Longleftrightarrow \\ L \text{ is recognised by a deterministic Muller automaton} \\ \Longleftrightarrow \\ L \text{ is recognised by a deterministic parity automaton} \end{array}$$

However, some languages are recognised by the above automata that cannot be recognised by any deterministic Büchi automaton.

The following theorem is one of the main reasons for which we have proven the above theorem. Indeed, we will later want to combine automata to show that they have the same expressive power as some formulas. To that extent, we need to be able to take unions, intersections and complements of automata, which correspond respectively to the disjunction, conjunction and negation of formulas.

Theorem 1.11. (Closure properties of Büchi automata) The class of languages recognised by Büchi automata is closed under the operation of

- finite union;
- finite intersection;
- complementation.

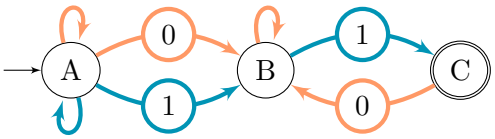
Proof. Since we have proven that the three classes of Büchi, deterministic Muller and deterministic parity automata are equivalent, we can show the properties on any of the classes.

We have already proven that parity automata are closed under union in Lemma 1.7 and the closure under intersection is a direct consequence of the closure under union and complements. It suffices to show that Muller automata are closed under complements.

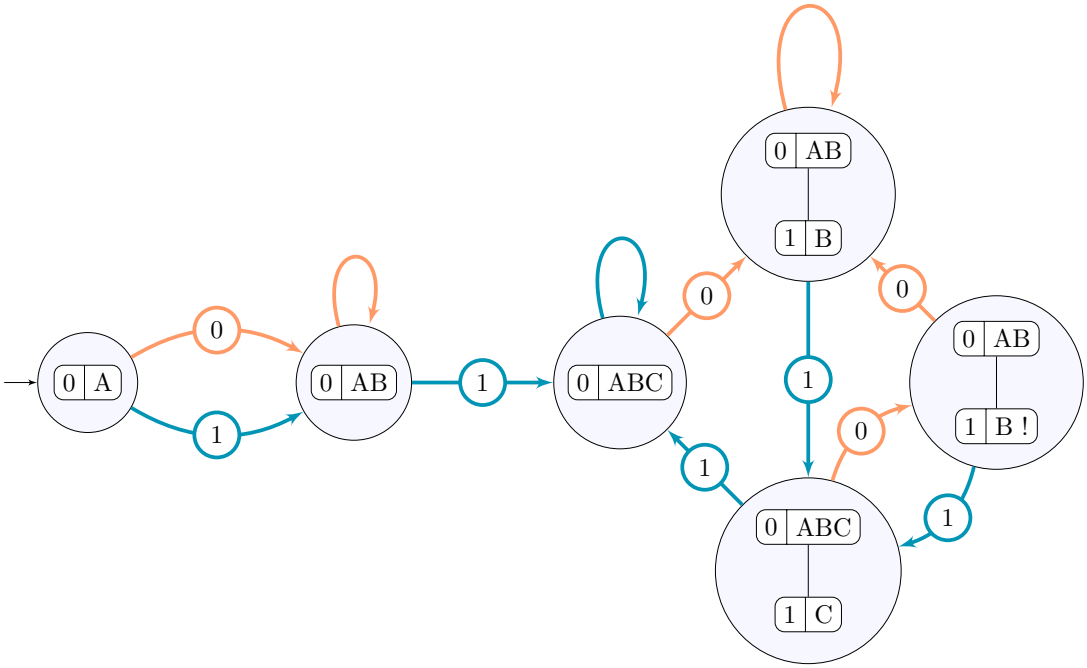
This is clear however, as given a Muller automaton $\mathcal{M} = (Q, \delta, \Sigma, q_0, \mathcal{F})$, the automata $\mathcal{M}^{\mathsf{c}} = (Q, \delta, \Sigma, q_0, \mathcal{P}(Q) \setminus \mathcal{F})$ accepts a run $r \in Q^\omega$ if and only if $\mathrm{Inf}(r) \in \mathcal{P}(Q) \setminus \mathcal{F}$, that is, $\mathrm{Inf}(r) \notin \mathcal{F}$, which corresponds to the fact that $\mathcal{M}$ does not accept r. □

1.4. Example of the Safra construction

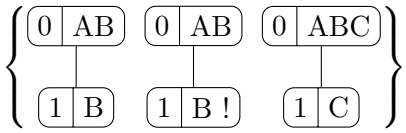
In this section, we we look at a few instances of the Safra construction. The following Büchi automaton on the language $\{0, 1\}$ recognises all the infinite words that have infinitely many `1` but only finitely many `11`.



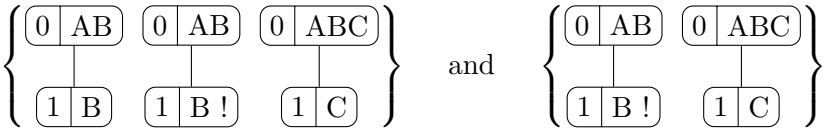
Converting this automaton to a deterministic Müller automaton, we obtain the following:



Which accepts a run if the set of states that occur infinitely many times is any subset of



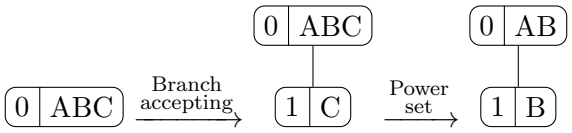
However, only two of those subsets are relevant here, because they are the only ones corresponding to loops containing a node with `!` in the graph of the Müller automaton. Those two sets are



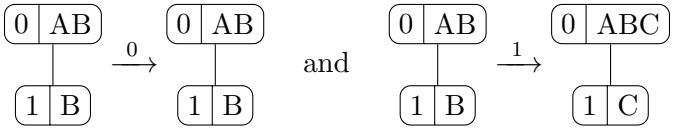
We now look at how to compute the transitions table of the Müller automaton. The initial state is $\boxed{0 \mid A}$ since the initial state of the Büchi automaton is A. Then, whether a `0` or a `1` is read, it is possible to transition to states A and B. All rules except the *Power set* rule do nothing here:

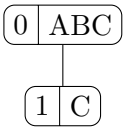
We now need to compute the transitions from $\boxed{0 \mid AB}$. When the autmaton reads `0` in the state A or B, we can reach the state A (from A) or B (from both A and B). The other rules don't apply here, so the transition is $\boxed{0 \mid AB} \xrightarrow{0} \boxed{0 \mid AB}$. However, if `1` is read, the set of reachable states is $\{A, B, C\}$. The transition is thus $\boxed{0 \mid AB} \xrightarrow{1} \boxed{0 \mid ABC}$.

The transition from $\boxed{0 \mid ABC}$ on input `0` makes also use of the *Branch accepting* rule, since C is an accepting state. According to this rule, we add a new node with a name that is not yet present in the tree, here 1, and create a son with label the set of accepting states of the node 0, which is just $\{C\}$. We then apply the *Power set* rule to each node. The new node that we have added, $\boxed{1 \mid B}$, tells us that if we are in B, then we have seen an accepting node (here C) before.

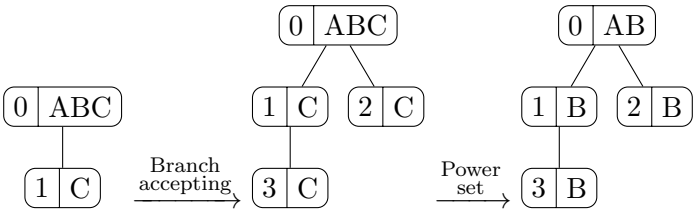


If we continue from this Safra tree and read `0` or `1` we obtain the two following transitions by using only the *Power set*:

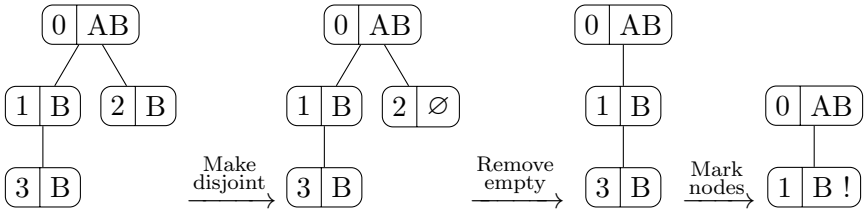


Finally we look at what happens if we read 1 from 

, because every rule is used. Since both node 0 and 1 contain C, we need to create a new node below them that contain C. Since the Büchi automaton has three nodes, we pick non-used labels in $\{0, 1, \ldots, 6\}$ and create two a new child for each. We then apply the *Power set* rule to each of the four nodes.

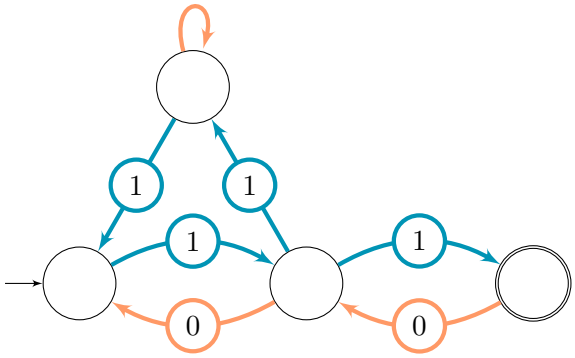


Now, both nodes 1 and 2 contain B, so we remove B from the newest node, which is 2 in this case. The idea behind this is that we do not need to keep track multiple times of what happens after we see state B, and all the information about runs That visit B will already be contained in node 1 or its children. We then remove node 2 entirely with the *Remove empty* rule. Finally, the *Mark nodes* rule is used on node 1, since its only child has the same label. We therefore remove node 3 and add a ! marker to node 1.



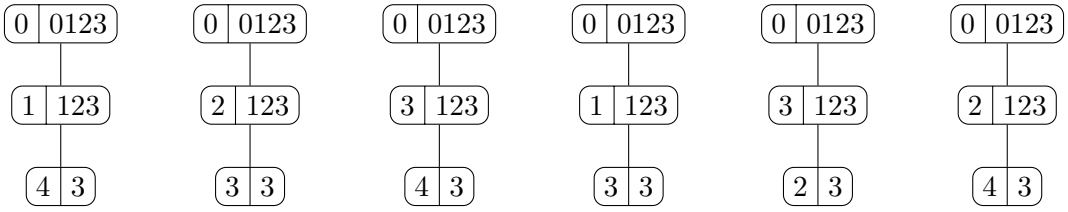
If we look at all the paths the Büchi automaton might have taken since the creation of this node, the ! mark indicates that there is a way to reach any state on the label of the node and visit an accepting state along the way. It doesn't mean that all paths since the creation of the node did visit an accepting state, but there's some path that did. This is also true if we consider the possible paths since the node was previously marked, as all children have been removed when marking the node, and have since been re-created by visiting an accepting state (via the *Branch accepting* rule).

Notice that this construction can produce very complex automata, even if there exists a simpler Muller automaton for the same language. For instance, the following Büchi automaton consisting of only four states

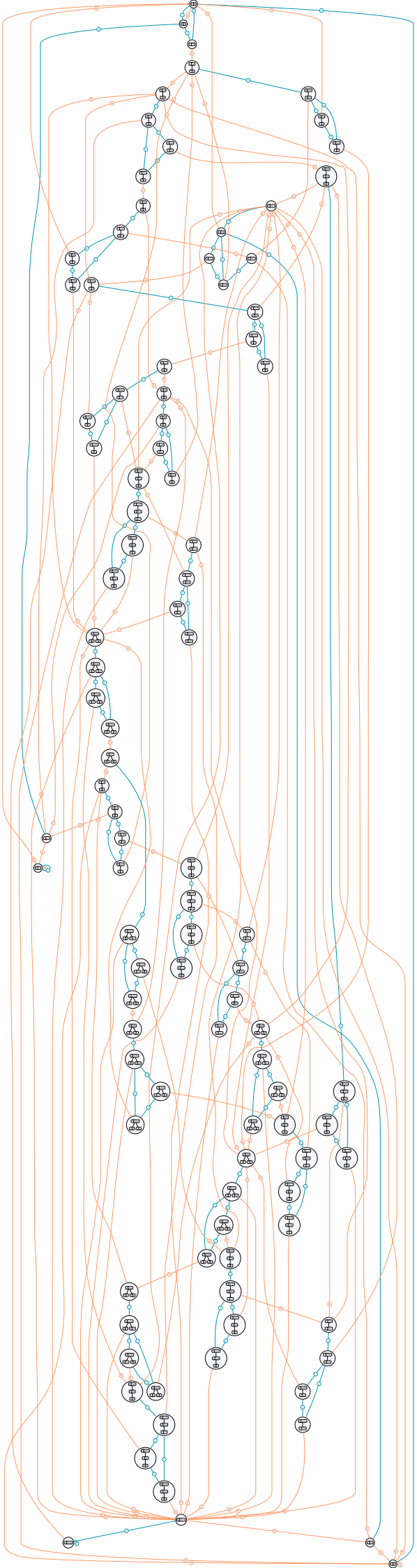


produces a Muller automaton with 105 states depicted on the next page. This is not because this automaton recognises a very complex language but rather that that each nod of Safra trees has a label and by following path of different length there can be multiple tree that have the same labels but different names.

For instance the tree below has six nodes labeled with the with $\{0, 1, 2, 3\}$, $\{1, 2, 3\}$ and $\{3\}$, but only the labels changes.



There is however, no obvious way to remove the names of each node as they are required in some cases to have the correct output.



2. Monadic second order logic

2.1. Definition

The goal of this section is to introduce the reader to some extensions of first order logic (FO), and most importantly to develop monadic second order logic.

In first order logic, quantifications happen only over the domain elements. Second order logic (SO) is a generalization of FO, where quantification over relations is allowed. It adds second order variables, usually denoted by capital letters, that are interpreted by relations, that is, if the relation is of arity n, subsets of $\mathcal{M}^n$.

For instance, the following formulas are syntactically valid:

- If the language contains the symbol $\leq$ and A is a second order variable of arity 1, then:
$$\forall A\ \exists x\ (A(x)\implies \forall y\ (A(y)\implies x\leq y)$$
is a formula expressing that every subset of the model a has a minimal element.

- If the language contains a binary relation E and M is a second order variable of arity 2, consider
$$\exists M\,\forall x\,\forall y\,\big(M(x,y)\implies E(x,y)\big)\wedge\forall x\,\exists!y\,M(x,y).$$
If E is symmetrical, we can view any model as an undirected graph with edges given by E, then the formula above expresses the fact that there exists a perfect matching, namely M.

Definition 2.1. Given a vocabulary σ consisting of relations and constants symbols, the **terms** of SO are constants symbols and first order variables.

The **atomic formulas** are of the form:

- $t=t'$ when t,t' are terms.
- $R(t_1,\dots,t_n)$ when $t_1,\dots,t_n$ are terms and $R\in\sigma$ is a relation of arity n.
- $X(t_1,\dots,t_n)$ when $t_1,\dots,t_n$ are terms and X is a second-order variable of arity n.

The set of SO **formulas** is the smallest set that contains all atomic formulas and is closed under:

- boolean operators: $\neg$, $\wedge$, $\vee$ and first order quantification
- second order quantification: if $\varphi(\vec{x},Y,\vec{X})$ is a SO formula, then $\forall Y\varphi(\vec{x},Y,\vec{X})$ and $\exists Y\varphi(\vec{x},Y,\vec{X})$ are SO formulas.

The **semantic** of SO logic is defined similarly to FO logic so we only need to define the semantics for new constructs. Let $\mathcal{M}$ be a σ-structure and $\varphi(x_1,\dots,x_k,X_1,\dots,X_n)$

a SO formula. We define $\mathcal{M} \models \varphi(\vec{b}, \vec{B})$ where $b \in \mathcal{M}^k$ is a tuple of elements of $\mathcal{M}$ and if X_i is of arity n_i, then B_i is a subset of $\mathcal{M}^{n_i}$.

- If $\varphi(x_1, \ldots, x_k, X)$ is $X(t_1, \ldots, t_n)$ with X a second-order variable of arity n and $t_1, \ldots, t_n$ terms with free variables among $x_1, \ldots, x_i$, then $\mathcal{M} \models \varphi(\vec{b}, B)$ if $(t_1^{\mathcal{M}}\vec{b}), \ldots, t_k^{\mathcal{M}}(\vec{b}))$ is in B.
- If $\varphi(\vec{x}, Y, \vec{X})$ is $\forall Y \psi(\vec{x}, Y, \vec{X})$ with Y a second-order variable of arity n then $\mathcal{M} \models \varphi(\vec{b}, B)$ if for all $C \subseteq \mathcal{M}^n$, $\mathcal{M} \models \psi(\vec{x}, C, \vec{X})$
- If $\varphi(\vec{x}, Y, \vec{X})$ is $\exists Y \psi(\vec{x}, Y, \vec{X})$ with Y a second-order variable of arity n then $\mathcal{M} \models \varphi(\vec{b}, B)$ if for some $C \subseteq \mathcal{M}^n$, $\mathcal{M} \models \psi(\vec{x}, C, \vec{X})$

We will not study much of second order logic, but instead look at one of its fragments, monadic second order logic.

Definition 2.2. Monadic second order logic or MSO is an extension of fisrt order logic and a restriction of second order logic. In MSO, valid formulas are formulas of second order logic where second order quantification happens only on unary relations. This corresponds to being able to quantify over elements of the domain (first-order quantification) or over subsets of the domain (second-order quantification).

The semantics of MSO is the same as SO semantics.

Examples. We give some formulas of MSO in the context of directed graphs, that is, we consider models of MSO with only one symbol of relation, E of arity 2. We read $E(x, y)$ as there is an edge between x and y.

The following formula states that a graph is disconected:

$$\underbrace{\exists X}_{\text{there exists a set}} \quad \underbrace{(\forall x\ \forall y\ X(x) \wedge E(x,y) \implies X(y))}_{\text{closed under neighbours}} \wedge \underbrace{(\exists x\ X(x)) \wedge (\exists x\ \neg X(x))}_{\text{and neither empty or full}}$$

And this formula states that it is 3-colorable:

$$\underbrace{\exists X_1\, \exists X_2\, \exists X_3}_{\text{there exists three sets}} \quad \underbrace{\forall x\ X_1(x) \vee X_2(x) \vee X_3(x)}_{\text{covering the graph}}$$
$$\wedge \underbrace{\forall x \bigwedge_{i \neq j} \big(X_i(x) \implies \neg X_j(x)\big)}_{\text{disjoint}}$$
$$\wedge \underbrace{\forall x\ \forall y\ E(x,y) \implies \bigwedge_{i=1..3} \big(X_i(x) \implies \neg X_i(y)\big)}_{\text{and no edge has twice the same color}}$$

Note that here the symbols $\bigwedge$ are not part of the language of MSO, they are meta shorthands to indicate the longer formula it expands to.

The examples above show that MSO has more expressive power than first order logic, as those formulas cannot be expressed in FOL. However, checking whether some MSO formula hold on a given graph is computationally hard, as this problem belongs to PSPACE. It even belongs to PSPACE when the graph (but not the formula) is fixed, as QBF can be encoded straightforwardly in MSO on the graph with only one vertex.

2.2. Decidability

The decidability problem for monadic second order logic, in general, is impossible, because it contains first order logic. However, in monadic second order logic, the theory of $\mathbb{N}$ with the successor (S1S) has been shown to be decidable. This theory is also called the theory of infinite strings for reasons that will be clear in the next section.

Definition 2.3. Let $n \in \mathbb{N}$ be a positive integer. A **language** of S1S is $\mathcal{L} = \{\underline{0}, \underline{S}, \underline{\leq}, \underline{P_1}, ..., \underline{P_n}\}$, where 0 is a constant symbol, S is a unary function, $<$ is a binary relation, and the P_i are unary relations.

Let $P_1, \ldots, P_n \subseteq \mathbb{N}$. A model of S1S is $\mathcal{M} = \{\mathbb{N}, 0, +1, <, P_1, ..., P_n\}$, where the domain is always $\mathbb{N}$, $S^{\mathcal{M}} = +1$ is the successor function which maps x to $x+1$, $<^{\mathcal{M}}$ is interpreted as the usual order on $\mathbb{N}$, and $P_i^{\mathcal{M}} = P_i$.

Remark. To get a model of S1S, we only need to give the sets $P_1, \ldots, P_n$. Therefore, a model can be coded by an ω-word $\mathcal{P} \in (\mathbb{B}^n)^\omega$ where $i \in P_k \iff (\mathcal{P}_i)_k = 1$.

In other words, each letter of the ω-word $\mathcal{P}$ is a n-tuple of 0's and 1's, and if the k-th element of i-th letter is a 1, then $P_i(k)$ holds.

The reason why S1S is called the theory of infinite strings is because its models can be considered as words on the alphabet $\mathcal{P}(\{P_1, \ldots, P_n\})$.

An other reason is that if for each $i \in \mathbb{N}$, there is exactly one of the $P_1(i), \ldots, P_n(i)$ that hold, then we can represent the model as an infinite string on the alphabet $\Sigma = \{P_1, \ldots, P_n\}$.

Those two reasons will be used to change our setting from models of S1S to and from ω-languages recognised by automata.

The semantics of S1S are the same as MSO. However, we only use $\mathbb{N}$ as the universe for first order variables and $2^{\mathbb{N}}$ as the universe for second order variables. If $w \in (\mathbb{B}^n)^\omega$ is an infinite word inducing the model $\mathcal{M} = \{\mathbb{N}, 0, +1, <, P_1, ..., P_n\}$, and $\varphi(X_1, \ldots, X_n)$ is a S1S formula with n free second order variables, we write

$$w \models \varphi(X_1, \ldots, X_n) \iff \mathcal{M} \models \varphi_{[P_1/X_1, ..., P_n/X_n]}$$

Definition 2.4. An ω-language $L \subseteq (\mathbb{B}^n)^\omega$ is **S1S-definable** if there is some S1S

formula $\varphi(X_1, \ldots, X_n)$ such that

$$L = \{w \in (\mathbb{B}^n)^\omega \mid w \models \varphi(X_1, \ldots, X_n)\}$$

Example. $L = \{w \in \mathbb{B}^\omega \mid w \text{ has infinitly many 1's}\}$ is first order definable by $\varphi(X_1) = \forall s \, \exists t \, (s < t \wedge X_1(t))$

Lemma 2.5. A Büchi-definable ω-language is S1S-definable.

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, q_1, F)$ be a Büchi automaton. We need to construct a formula $\varphi(X_1, \ldots, X_n)$ such that for all ω-word $w \in \Sigma^\omega$ we have $w \models \varphi(X_1, \ldots, X_n)$ if and only if $\mathcal{A}$ accepts w. For this, we set $n := |\Sigma|$ and

$$\varphi(X_1, \ldots, X_n) = \exists Q_1, \ldots, Q_n \; \varphi_{part} \wedge \varphi_{start} \wedge \varphi_{trans} \wedge \varphi_{accept}$$

Informally, for each integer t there is exactly one of the Q_i such that $t \in Q_i$, which corresponds to the state of the automaton on an accepting run. Formally,

- φ_{part} asserts that the sets $Q_1, \ldots, Q_n$ form a partition of $\mathbb{N}$

$$\varphi_{part}(Q_1, \ldots, Q_n) := \forall t \bigvee_{i=1..n} Q_i(t) \wedge \forall t \bigwedge_{i \neq j} \neg \left(Q_i(t) \wedge Q_j(t)\right)$$

- φ_{start} encodes the facts that a run must start at q_1, so

$$\varphi_{start}(Q_1, \ldots, Q_n) := Q_1(0)$$

- φ_{trans} encodes the transition function.

$$\varphi_{trans}(Q_1, \ldots, Q_n) = \forall t \bigwedge_{q' \notin \delta(q,l)} Q_q(t) \wedge X_l(t) \implies \neg Q_{q'}(t+1)$$

  where the conjunction is on every triple $(q, l, q') \in Q \times \Sigma \times Q$ such that $q' \notin \delta(q, l)$. This corresponds to forbidding the automaton to move from the state q to q' by reading the letter l. Note that we do not need to explicitly require that the automaton goes to some state in $\delta(q, l)$ as the fact that the Q_i form a partition already requires that the automaton goes to *some* state.

- φ_{accept} encodes the fact that the automaton visits infinitely many times an accepting state:

$$\varphi_{accept}(Q_1, \ldots, Q_n) = \forall t \, \exists s \left(t < s \wedge \bigwedge_{i \in F} Q_i(t)\right)$$

By construction, if there is an accepting run $r \in Q^\omega$ then we have $w \models \varphi(X_1, \ldots, X_n)$ (where, remember, the X_i are interpreted as the set of integer where w has the letter i). On the other side, if $w \models \varphi$, an accepting run is given by the sets $Q_1, \ldots, Q_n$ by $r_i = j$ if and only if $i \in Q_j$. This is well defined because the Q_i form a partition of $\mathbb{N}$, and corresponds to an accepting run since it starts at q_1 (by φ_{start}), moves according to the transition function (by φ_{trans}) and visits infinitely many time an accepting state(by φ_{accept}). □

We will later show that the converse is also true, that is, given some S1S-decidable language, we can build a Büchi automaton that recognises the same language. The two notions are therefore equivalent.

To that extent, we will introduce deterministic Muller automata and show that they are equivalent to (non-deterministic) Büchi automata. This will be used to show that it is possible to complement a Büchi automaton. We will then show that S1S is equivalent to a simpler version with fewer constructs, $\mathrm{S1S}_0$. Finally, we will show that Büchi automata are as expressive as $\mathrm{S1S}_0$ formulas.

Definition 2.6. A formula is a $\mathrm{S1S}_0$ formula if

- its atomic formulas are one of $X \subseteq Y$, $\mathrm{Succ}(X, Y)$ or $\mathrm{Sing}(X)$, for X and Y some second order variable. We interpret that $X \subseteq Y$ holds if X is a subset of Y, $\mathrm{Succ}(X, Y)$ holds if $(X, Y) = (\{a\}, \{a+1\})$ for some $a \in \mathbb{N}$ and $\mathrm{Sing}(X)$ holds if $X = \{a\}$ for some a.
- The only connectors are $\vee$ and $\neg$
- The only quantifiers are second order existential quantifers.

Lemma 2.7. Every S1S formula $\varphi(X_1, \ldots, X_n)$ has an equivalent $\mathrm{S1S}_0$ formula $\varphi_0(X_1, \ldots, X_n)$.

Proof. First, we know that we can eliminate all conjunction with De Morgan's laws and replace all implications with their definition. Similarly, we can eliminate universal quantifiers since $\forall x\, \psi(x) \iff \neg\exists x\, \neg\psi(x)$.

We can eliminate the constant 0, by using a fresh variable (say z) that does not appear in φ. Then, φ is equivalent to

$$\exists z\ \left(\neg\exists x\, (x < z) \wedge \varphi_{[z:=0]}\right)$$

where $\varphi_{[z:=0]}$ is the formula where every 0 of φ is replaced by z.

The formula $x < y$ is equivalent to

$$\forall X\ \Big(X(x) \wedge \forall t\big(X(t) \implies X(S(t))\big)\Big) \implies X(y)$$

That is, every set that contains x and is closed under the successor function also contains y.

Then we can ensure that the successor function occurs only in formulas of the form $S(x) = y$. For instance, we replaces instances of $X(S(S(x)))$ by

$$\exists s \exists t \big(S(x) = s \wedge S(s) = t \wedge X(t)\big)$$

We can similarly remove all instances of the form $S(...(S(x)...) = S(...(S(y)...)$ by adding intermediate variables.

We have shown so far that we can consider formulas that consist only of $S(x) = y$, $X(x)$, connectors $\neg$ and $\wedge$ and first and second order existential quantifiers. In order to remove completely first order variables and quantifers, we need to modify every occurence of first order variables:

- If φ is of the form $\exists x\ \psi(x)$, we replace it by $\exists X\ \mathrm{Sing}(X) \wedge \psi_{[X/x]}$ where X is a variable not apearing in ψ.
- If φ is $X(y)$, we replace it by $\mathrm{Sing}(Y) \wedge Y \subseteq X$
- If φ is $x' = y$ we replace it by $\mathrm{Succ}(x, y)$.

□

Theorem 2.8. An ω-language is Büchi-definable if and only if it is S1S definable.

Proof. Let L be an ω-language. By Lemma 2.5, we know that if it is Büchi-definable, it is S1S-definable.

For the other direction, by Lemma 2.7, formulas of S1S are as expressive as formulas of $\mathrm{S1S}_0$, so it suffices construct a Büchi automaton that corresponds to each $\mathrm{S1S}_0$ formula. We consider formulas of the form $\varphi(P_1, \ldots, P_n, X_1, \ldots, X_m)$ where

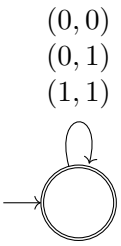
- the P_i are the second order variables that correspond to the ω-word.
- the X_i are second order free variables.

We will abbreviate this as $\varphi(\vec{P}, \vec{X})$, as it does not matter for the construction which of the second order variable is which. It is only relevant to keep in mind that some of those variables correspond to the unary relations of the model.

For each such formula, we construct an automaton on the language $\mathbb{B}^n \times \mathbb{B}^m = \mathbb{B}^{n+m}$ such that given any $\vec{P}$ and $\vec{X}$, φ holds if and only if the automaton accepts the word $(\vec{P}, \vec{X})$.

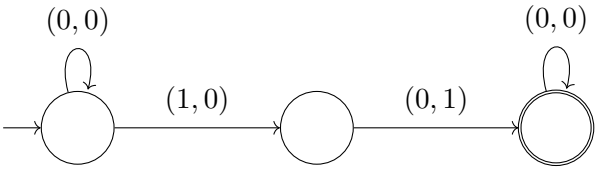
We proceed by induction on the height of the formula φ.

- If $\varphi(X, Y)$ is $X \subseteq Y$, a corresponding automaton is:



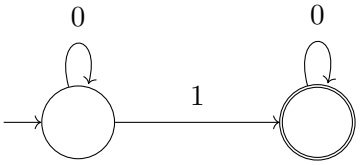
  In this automaton, an infinite run is possible if and only if each digit of X is smaller that the corresponding digit of Y, so the only impossible case is $(1, 0)$, which would correspond to having some $x \in X$ that does not belong to Y.

- If $\varphi(X, Y)$ is $\mathrm{Succ}(X, Y)$, a corresponding automaton is:



There, a one must be read in X just before it is read in Y, and nowhere else.

- If $\varphi(X)$ is $\mathrm{Sing}(X)$, a corresponding automaton is



- If $\varphi(\vec{X})$ is $\psi(\vec{X}) \vee \chi(\vec{X})$, or $\neg\psi(\vec{X})$ where ψ and χ have equivalent automaton, we know by Theorem 1.11 that the class of Büchi automata is closed under union and complementation, so there is a automaton corresponding to φ.

- If $\varphi(\vec{X})$ is $\exists Y\ \psi(\vec{X}, Y)$ and A is the automaton corresponding to $\psi(\vec{X}, Y)$, an automaton for φ is a copy of A in which the transition constraint for the variable Y are erased. That is, if A works on the alphabet $\mathbb{B}^{k+1}$ and has a transition function $\delta : \mathbb{B}^{k+1} \times S \to \mathcal{P}(S)$, then the automaton A' for φ works on the alphabet $\mathbb{B}^k$ and has transition function

$$\begin{array}{rccl} \delta' : & \mathbb{B}^k \times S & \longrightarrow & \mathcal{P}(S) \\ & (b, s) & \longmapsto & \bigcup_{y \in \mathbb{B}} \delta\big((b_1, \ldots, b_k, y), s\big) \end{array}$$

  It is the projection on the coordinate Y of the automaton A.

□

Given a S1S formula, we want to know whether it admits a model, that is, if some infinite word satifies the formula. The following theorem tells us that this problem is decidable.

Theorem 2.9. The satisfiability problem for S1S is decidable.

Proof. Let *phi* be a S1S formula. By Theorem 2.8 we have an automaton $\mathcal{A}$ that recognise the same words as φ accepts. It suffices to know whether $\mathcal{A}$ recognises at least one word.

It is easy to find wether a Büchi automaton accepts at least one word, and if so, to give one such word. Indeed, it suffices to find a cycle in the graph of the automaton which is reachable from the initial state and which contains an accepting state. this can be done by a depth first search. □

2.3. Bisimilarity

In this section, we will look at another remarkable fact about MSO, which brings it closer to theoretical computer science. An important class of objects in theoretical computer science is the class of labelled transition systems, which are directed graphs with labels on both the edges and vertices. Those labels correspond to different kinds of transitions that can happen from one state to the other. It is a concept similar to automata but without a notion of initial and accepting states. It also differs from automata as the set of states and transitions are not necessarily finite and not even countable.

Definition 2.10. Given to sets of symbols P and M, a **labeled transition system** is a quadruple $\mathbb{S} = (S, \Lambda, \Gamma, R, V)$, where:

- S is a set of **states**.
- Λ is a set of **edge labels**.
- Γ is a set of **state labels**.
- $R : \Lambda \to \mathcal{P}(S \times S)$ is the transition rule, so that for $\alpha \in \Lambda$, $R(\alpha)$ is a binary relation that discribes a directed graph on S.
- $V : \Gamma \to \mathcal{P}(S)$ describes the state labels, so that for $\alpha \in \Gamma$, $V(\alpha)$ is the set of state labelled with α.

Note that this notion encompasses that of Kripke's structure.

We will look at the notion of bisimilarity, or bisimulation, which is an equivalence relation on labelled transition systems. Intuitively, two labelled transition systems are bisimilar if they can simulate each other and an observer cannot distinguish between them. We formalise the idea of being undistinguishable by an observer with Ehrenfreucht-Fraïssé games.

Definition 2.11. Let $\mathbb{S} = (S, \Lambda, \Gamma, R, V)$ and $\mathbb{T} = (T, \Lambda, \Gamma, R', V')$ be two labeled transition system that share the same set of labels.

The **Ehrenfreucht-Fraïssé game** of $\mathbb{S}$ and $\mathbb{T}$ is an infinite game, between two players that we call Attacker and Defender. The game plays as follows:

1. The Attacker picks a model $M \in \{\mathbb{S}, \mathbb{T}\}$, and a state $q_A \in M$
2. Let $M' \in \{\mathbb{S}, \mathbb{T}\}$ be the model not chosen by the Attacker. The Defender picks a state $q_D \in M'$ such that q_A and q_D have the same labels. That is, for every state label $\alpha \in \Gamma$, $q_A \in V(\alpha)$ if and only if $q_D \in V(\alpha)$.
3. At each turn, let $(p, p') \in M \times M'$ be the two states q_A and q_D picked by the

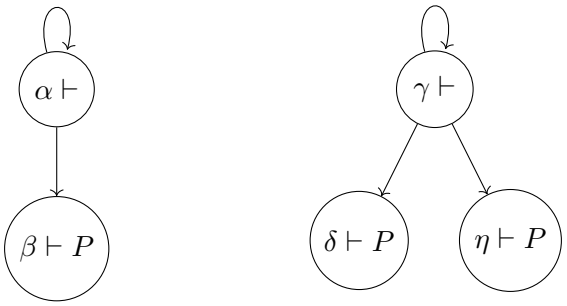
two players in the last turn. the Attacker picks $p_A \in \{p, p'\}$ and a transition $p_A \xrightarrow{\alpha} q_A$ for some $\alpha \in \Lambda$ and q_A in the same model as p_A.

4. To a move $p_A \xrightarrow{\alpha} q_A$ the Defender picks a transition $p_D \xrightarrow{\alpha} q_D$. where p_D is the state of $\{p, p'\}$ not picked by the Attacker, and q_D is in the same model as p_D and such that q_A and q_A have the same labels.

5. If a player cannot pick a transition, he loses the game. Otherwise, if the game is infinitely long, the Defender wins.

Definition 2.12. Two labeled transition systems $\mathbb{S}$ and $\mathbb{T}$ are **bisimilar** if the Defender has a winning strategy in the Ehrenfreucht-Fraïssé game with $\mathbb{S}$ and $\mathbb{T}$.

Example. The two transition systems below, with one edge label that we represent by a simple arrow, and vertice labels $\{P\}$ that we write on the node, next to the name of the node (which is only used for clarity, states in transition systems are not named, and thus cannot be distinguished by the names.), are bissimilar:



Indeed, we can exhibit a winning strategy for the Defender:

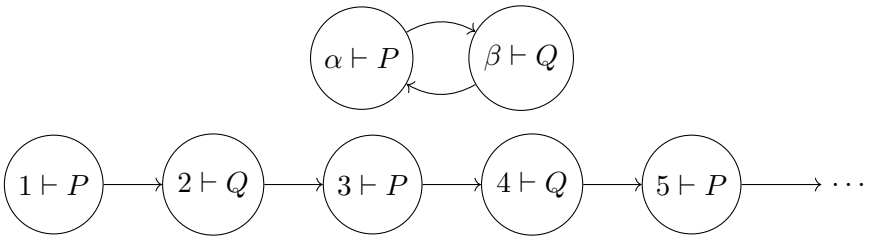
- If the Attacker starts with β, the Defender can pick δ, which has the same labels. On the second turn, the Attacker needs to choose one of the two previous nodes, β or δ and a transition out of it. Since no transition leaves either node, she cannot play and loses.
- Similarly, if the Attacker starts with δ or η, the Defender can pick β, and the Attacker is not able to play in the next round.
- If the Attacker picks α or γ, the Defender can answer respectively with γ or α, since it also has the same labels. On the next turn,
  - if the Attacker picks the first model and β, the Defender can pick δ in the second model, and wins for the same reason as in the first case.
  - if she picks the second model and either δ or η, the Defender can pick β in the first model, the Attacker will not be able to play in the next turn.

– if she picks either α or γ, the Defender picks the other one, and the game is back to the same state as the first turn. Since all other moves of the Attacker are immediately loosing, she might play either α or γ infinitely many times, which gives the victory to the Defender, because the Attacker fails to reveal any difference between the two models.

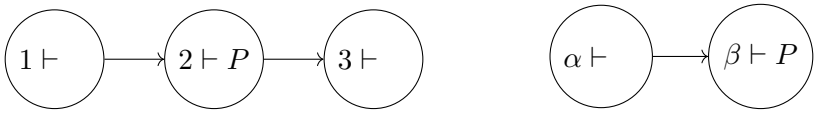
In this example, we can feel why the two models are bisimilar: however the Attacker moves in either graph, we can always move in a symmetrical fashion in the other graph.

We can also notice that at any point in a game, the winner does not depend on the history of the game, but only on the two positions picked at the last turn. This kind of game is called memoryless because there is always a winning strategy that depends only on the last move.

Example. The two transition systems below are bissimilar:



Example. Those two transition systems are not bissimilar:



Indeed, the Attacker has a winning strategy:

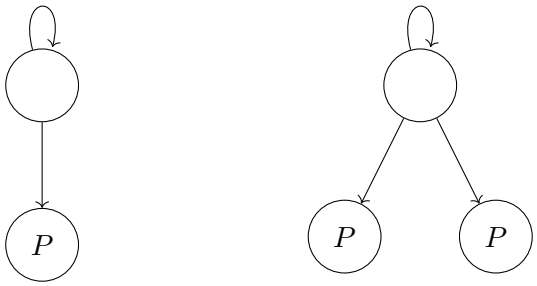
- On the first turn, the Attacker picks state 3 in the left model.
- The Defender is forced to pick α since it is the only state with an empty label in the right model.
- On the second turn, the Attacker can pick either 3 or α but since 3 has no outgoing transition, she picks α and the Defender cannot pick any transition in the left model and looses.

3. Mu calculus

3.1. A strange logic

We have seen the notion of bisimilarity, which is one way of considering different models the same. However, in MSO, there are formulas that do not have the same truth value in two bisimilar models One example is the formula φ saying that there are two states that with label P, which is true in the second model but not the first:

$$\varphi := \exists x \ \exists y \ P(x) \wedge P(y) \wedge \neg(x = y)$$



On the other hand, some formulas always have the same truth value in bisimilar models. If one is to study which formulas of MSO are invariant under bisimilarity, they find a class of formulas that corresponds to the μ-calculus.

The μ-calculus is an extension of propositional logic, extended with modal and fixpoint operators.

Definition 3.1. The **signature** of a μ-calculus language is $\sigma = P \cup M$, where

- P contains propositional symbols of arity 0.
- M contains modal operators symbols.

Both of those sets can be finite or infinite.

Definition 3.2. The **formulas** of the μ-calculus with signature $\sigma = P \cup M$ is the smallest set such that:

- each proposition $p \in P$ and each variable is a formula.
- if φ and ψ are formulas, then $\varphi \wedge \psi$ is a formula.
- if φ is a formula, then $\neg\varphi$ is a formula.
- for each modal operator $m \in M$, and formula φ, $[m]\,\varphi$ is a formula read "m box φ".
- if Z is a variable and φ a formula, and every occurence of Z occurs positively in φ, that is, is under an even number of negations, then $\nu Z\,\varphi$ is a formula.

Furthermore, if φ, ψ are formulas, m is a modal operator and Z is a variable, we have the following shorthands:

- $\varphi \vee \psi$ stands for $\neg(\neg\varphi \wedge \neg\psi)$
- $\langle m\rangle\,\varphi$ stands for $\neg[m]\,\neg\varphi$. This is read "a diamond φ".

- $\mu Z\, \varphi$ stands for $\neg \nu Z\, \neg \varphi[Z := \neg Z]$. where $\varphi[Z := \neg Z]$ is the formula obtained by replacing every occurence of Z by $\neg Z$ in φ.

Definition 3.3. A **model** of μ-calculus with signature $\sigma = P \cup M$ is a **labeled transition system** $\mathbb{S} = (S, \Lambda, \Gamma, R, V)$, where:

- S is the set of states.
- $\Lambda = M$ is the set of edge labels, or modal operators.
- $\Gamma = P$ is the set of state labels, or propositional symbols.
- $R : M \to \mathcal{P}(S \times S)$ maps each modal operators to a binary relation on states. This can be seen as $|M|$ directed graphs.
- $V : P \to \mathcal{P}(S)$ maps each propositional symbol to the set of states on which it is true.

Since $\Lambda = P$ and $\Gamma = M$ are part of the signature of the language, we will not write them explicitly and introduce labeled transition systems as $\mathbb{S} = (S, R, V)$.

Finally, we need to describe how μ-calculus formulas are interpreted in a given labelled transition system. Four facts differ from classical propositional logic:

- The meaning of a formula is the set of states on which it is true. In particular, a formula does not have a truth value in a given model, but only at a given state in the model.
- The modal operators $[m]$ and $\langle m \rangle$ correspond to movements in the directed graph corresponding to m. The box operator $[m]\, \varphi$ means that φ is satisfied in every m-neighbour of the current state. On the other hand, the diamond operator $\langle m \rangle\, \varphi$ means that there exists a m-neighbour in which φ is satisfied.
- Variables are not interpreted as states, but as sets of states on which they are true.
- The fixpoint operator μ is the least fixpoint whereas ν is the greatest fixpoint in the sense that $\nu Z\, \varphi$ is true in the largest set of states Z such that φ is satisfied on Z.

Definition 3.4. Let $\mathbb{S} = (S, R, V)$ be a labeled transition system with signature $\sigma = P \cup M$. An interpretation i is a map that associates each variable Z to the set of states $i(Z) \subseteq S$ on which it is true.

The **meaning** of a formula φ is the set of states $[\![\varphi]\!]_i^{\mathbb{S}} \subseteq S$ defined as follows.

- $[\![p]\!]_i^{\mathbb{S}} = V(p)$ if $p \in P$.

- $[\![Z]\!]^{\mathbb{S}}_i = i(Z)$ if Z is a variable.
- $[\![\varphi \wedge \psi]\!]^{\mathbb{S}}_i = [\![\varphi]\!]^{\mathbb{S}}_i \cap [\![\psi]\!]^{\mathbb{S}}_i$ if φ, ψ are formulas.
- $[\![\neg\varphi]\!]^{\mathbb{S}}_i = S \setminus [\![\varphi]\!]^{\mathbb{S}}_i$ if φ is a formula.
- $[\![[m]\,\varphi]\!]^{\mathbb{S}}_i = \left\{ s \in S \,\middle|\, \forall t \in S\ (s,t) \in R(m) \implies t \in [\![\varphi]\!]^{\mathbb{S}}_i \right\}$ if φ is a formula and m a modal operator.
- $[\![\nu Z\,\varphi]\!]^{\mathbb{S}}_i = \bigcup \left\{ T \subseteq S \,\middle|\, T \subseteq [\![\varphi]\!]^{\mathbb{S}}_{i[Z:=T]} \right\}$ if φ is a formula. Where $i[Z := T]$ maps Z to T and the rest of the map i is not modifed.

If follows that the interpretations of $\vee$, $\langle m \rangle$ and μ are, φ, ψ are formulas:

- $[\![\vee\varphi \wedge \psi]\!]^{\mathbb{S}}_i = [\![\varphi]\!]^{\mathbb{S}}_i \cup [\![\psi]\!]^{\mathbb{S}}_i$.
- $[\![\langle m \rangle\,\varphi]\!]^{\mathbb{S}}_i = \left\{ s \in S \,\middle|\, \exists t \in S\ (s,t) \in R(m) \implies t \in [\![\varphi]\!]^{\mathbb{S}}_i \right\}$
- $[\![\mu Z\,\varphi]\!]^{\mathbb{S}}_i = \bigcap \left\{ T \subseteq S \,\middle|\, [\![\varphi]\!]^{\mathbb{S}}_{i[Z:=T]} \subseteq T \right\}$

Informally, we have that

- p holds for all states in $V(p)$
- Z holds for all states in $i(Z)$
- $\varphi \wedge \psi$ holds for states where both φ and ψ hold.
- $\varphi \vee \psi$ holds for states where either φ or ψ hold.
- $\neg\varphi$ holds for states where φ doesn't hold.
- $[m]\,\varphi$ holds in s if all m-transitions from s lead to states where φ holds.
- $\langle m \rangle\,\varphi$ holds in s if there exists a m-transition from s to a state where φ holds.
- $\nu Z\,\varphi$ holds in a state s if there exist a subset $T \subseteq S$ of states that contains s and when Z is interpreted as T in φ, the set on which φ holds contains T.

  Note that the application $[\![\varphi]\!]^{\mathbb{S}}_{i[Z:=T]}$ is increasing in T, because every syntactic construction but the negation are clearly increasing functions of T, and the negation is a decreasing function. Nevertheless, the variable Z is always under an even number of negation, therefore by composition of decreasing function, $[\![\varphi]\!]^{\mathbb{S}}_{i[Z:=T]}$ is increasing in T.

  By Knaster-Tarski's theorem, $[\![\nu Z\,\varphi]\!]^{\mathbb{S}}_i$ is then the largerst fixed point of $[\![\varphi]\!]^{\mathbb{S}}_{i[Z:=T]}$.
- On the other hand, $\mu Z\,\varphi$ is the smallest fixed point of $[\![\varphi]\!]^{\mathbb{S}}_{i[Z:=T]}$.

We give a few examples of formulas and their meanings but without detailed explanations, as the semantics defined above are concise, but hard to interpret. In fact, we give an other caraterisation of the meaning of formulas using games, which makes reasonning with μ-calculus easier.

Example. We adopt the convention that when a language has only one modal operator, for instance $[m]$ and $\langle m \rangle$, we write them simply as $\Box$ and $\Diamond$ respectively. Let φ any formula of the μ-calculus, (for instance, a proposition):

- $\nu Z\,(\varphi \wedge \Box Z)$ is interpreted as "φ is true along every path". Indeed, Z is the largest set of states in which φ is true, and which remains true after moving along any edge.
- $\mu Z\,(\varphi \vee \Diamond Z)$ is interpreted as the existence of a path leading to a node where φ holds.
- $\nu Y\ \mu X\ (\varphi \wedge \Diamond Y) \vee \Diamond X$ is the largest set of nodes which is closed under the property that there is a path leading to a node where φ holds. Said otherwise, it is the set of paths on which φ holds inifinitely many times.

3.2. Model checking

As is often the case in logic, the semantics of μ-calculus can be defined alternatively defined via a two player game between the Verifier and Falsifier.

To that extent, we need to define parity games.

Parity games are an important subject in complexity theory, as the problem of deciding whether the first or second player has a winning strategy is one of the few problems we know is in $\mathsf{NP} \cap \mathsf{coNP}$ be don't know if it is in P. Finding the best bounds for the complexity of this problem is an active field of reasearch, with the first quasipolynomial time algorithm, due to [Calude et al., 2017]. There have been many variants of this algorithm since and it is still an open question to understand them well and know whether a better algorithm can be devised. Lastly, this problem is one of the few problems in $\mathsf{UP} \cap \mathsf{coUP}$ that are not found to be in P yet [Jurdziński, 1998]. The most famous problem that fit in this category was primality checking.

As a reminder, the class UP is the class of decision problems that can be verified in polynomial time and have unique short certificates, that is, there is a polynomial Turing machine in which at most one computation path accepts.

This makes μ-calculus a good logic with both a good expressivity and good algorithmic properties, as its model checking is based on parity games.

Definition 3.5. The **parity game** is an inifinite two player game played on a directed graph $G = (V, E)$. Let V_0, V_1 be a partition of the vertices of G, and $p : V \to \mathbb{N}$ any function, called the **priority function**. Let also $v_0 \in V$ be the starting point of the game.

The game goes as follows:

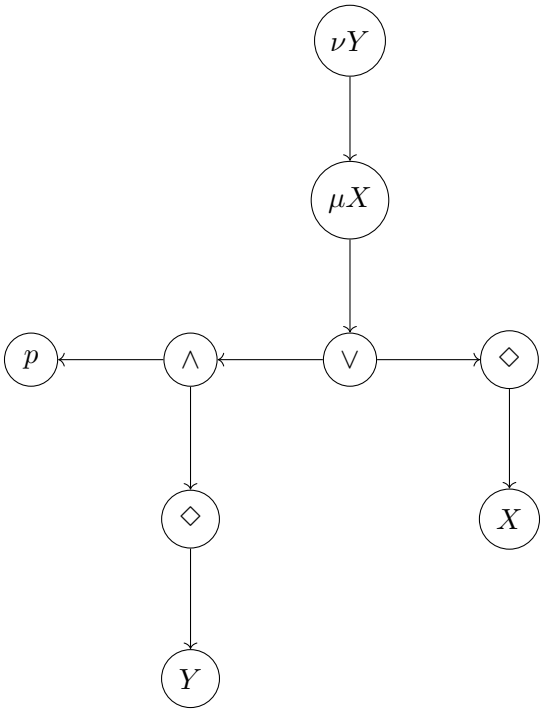
- It starts at v_0
- At a given turn i, if $v_i \in V_0$, player 0 decides to move to an adjacent node v_{i+1}. Otherwise if $v_i \in V_1$, it is player 1 that picks a neighbour of v_i.
- If a player cannot pick a neighbour,
- After ω turns, they have produced an infinite sequence of vertices $(v_n)_{n\in\mathbb{N}} \in V$. The least priority that was seen infinitely many times determines the winner, playermax $\{p(v) \mid v \in \mathrm{Inf}(v)\}$ is even, player 0 wins. If it is odd, player 1 wins.

Note that any parity game is determined, as the winning set is

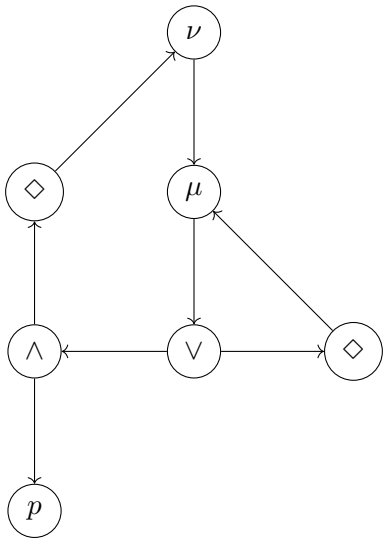
$$W = \left\{ b \in T \;\middle|\; \exists z \in V \; \forall z' \in V \left(\begin{array}{c} p(z) \text{ even} \\ \wedge \\ \exists^\infty i,\ v_i = z \\ \wedge \\ p(z') > p(z) \implies \exists N \forall n > N v_n \neq z' \end{array} \right) \right\}$$

which is a Borel set (in fact, it is $\mathbf{\Sigma}^0_3$). Since all Borel games are determinated [Martin, 1975] there is always a player that has a winning strategy.

We give an example of the model-checking game of μ-calculus before defining it. Let $\varphi = \nu Y \ \mu X \ (p \wedge \Diamond Y) \vee \Diamond X$ be a formula and we want to know whether some model satifies it. We first draw the formula as a tree:

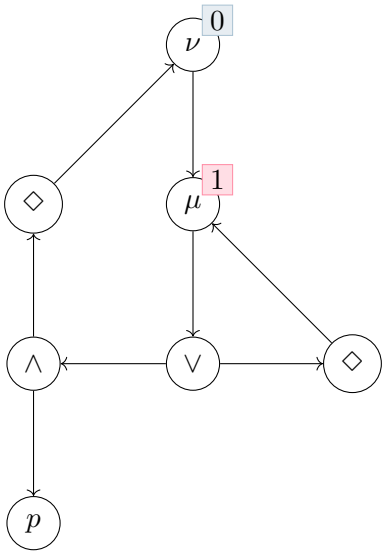


And then we connect each variable to the node where it is bound. With some rearanging of the nodes for clarity and removing the variable names as they will not be used anymore, we get:

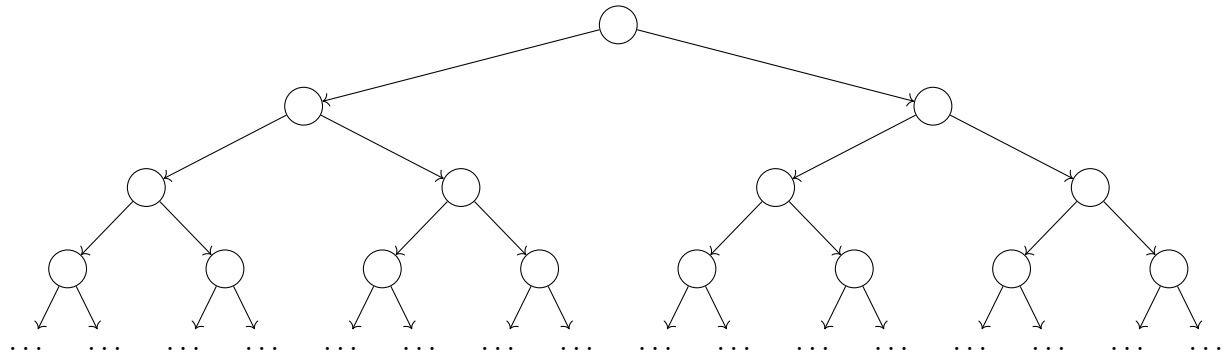


We then assign a priority to each node depending on their position in the formula φ:

- Every greatest fixpoint operator ν is assigned an even number, greater than the priority of every least fixpoint operator μ above it in the tree of the formula. Here the ν has no μ above it, so we can give it a priority of 0.
- Conversely, every least fixpoint operator μ is assigned an odd number, larger than the priority of every greatest fixpoint operator ν above it in the tree of the formula. Here μ has a ν of priority 0 in the formula above it so we can give it priority 1.
- Every non-fixpoint node gets an odd priority, higher than every other priority. In this example, we can assign 3 to each other node, but for clarity we will not draw them on the graph.



The formula consists of half of the game, indeed, it needs to be checked against a model. Given a model $\mathbb{S} = (S, R, V)$ and a state $s \in S$, we want to know whether φ holds at s. For the first example, we take as model an infinite binary tree with no node satisfying the property p, that is $V(p) = \varnothing$.



The game is played both in the formula and the model at the same time. Each turn, one of the two players decides where to move in both structures. The game starts at the root of the formula and the state s_0 in the model. The person that plays and their allowed moves are given by the current node of the formula. At each turn, the current player must move in the tree of the formula to a neighbouring node, and potentially also make an extra action.

Formula node	Player	Action
p		End of the game
$\vee$	Player 0	
$\wedge$	Player 1	
$\Diamond$	Player 0	Pick $s_{n+1} \in S$ s.t. $s_n \to s_{n+1}$
$\Box$	Player 1	Pick $s_{n+1} \in S$ s.t. $s_n \to s_{n+1}$
$\neg$	Any, there is only one choice	Player 0 and 1 switch roles
μ	Any, there is only one choice	
ν	Any, there is only one choice	

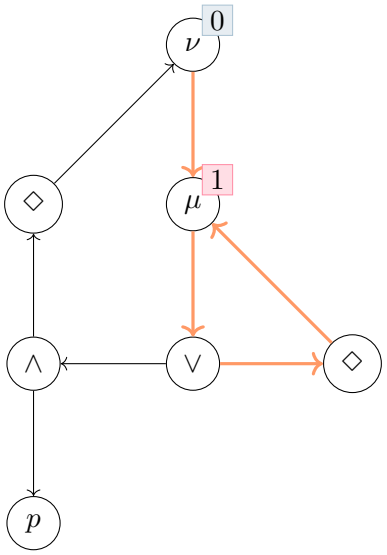
If the game ends on a proposition (after a finite number of turns), the winner is Player 0 if the last visited state in the model satisfies the proposition, otherwise, Player 1 wins. If the game runs for ω turns, the winner is determined by the smallest priority that was visited infinitely many times.

In the current example, the position in the model does not matter much, because every state looks like any other (formally, there is a bisimilarity relation between any two states). We can therefore focus on what happens in the graph of the formula.

The game starts on ν, and since there is only one choice, proceeds to μ. On the next turn, there is still only one choice, so the game proceeds to the $\vee$ node. On the $\vee$ node, it is Player 0, also known as the Verifier that has to play. She can choose to move either to the $\Diamond$ node on the right or the $\wedge$ node. We examine both cases:

- If she picks $\wedge$ it is then Player 1's turn to play and we can chose between the node p and the node $\Diamond$ above. However, moving to the node p is a winning move for him! Indeed, because they have reached a proposition symbol, the game ends and Player 1 wins if the current state in the model does not satisfy the proposition p. But since no state in the model satisfies p, he surely wins.
- On the other hand, if she picks $\Diamond$ it is again Player 0's turn. For this turn, she needs takes two actions, she first moves to the only neighbour of $\Diamond$ which is μ and picks $s_1 \in S$, a neighbour of s_0. From μ the game proceeds to the node $\vee$ and the same choice is offered to Player 0.

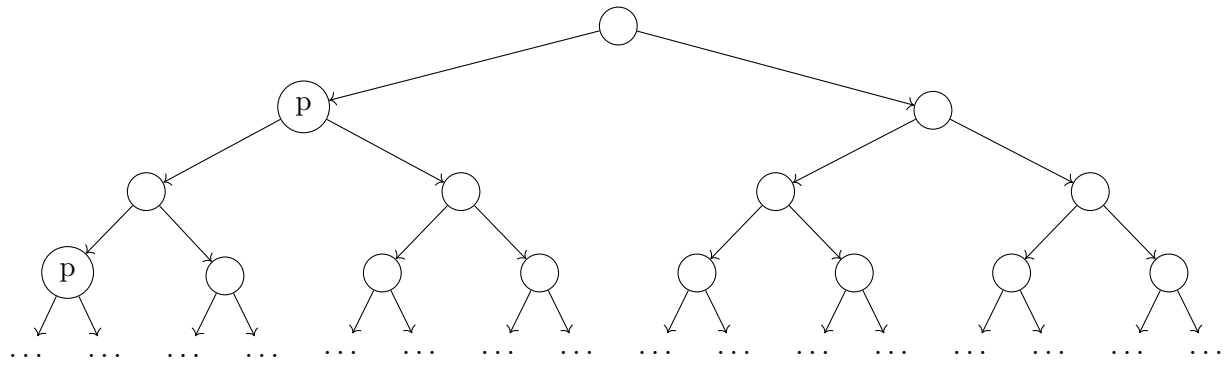
So if Player 0 decides to move to $\wedge$, she looses. On the other hand, if she always decides to move to $\Diamond$, the infinite path built by the two player is $\nu, \mu, \vee, \Diamond, \mu, \ldots$, as shown in orange below.



Here, the set of states visited infinitely many times is then $\left\{\mu, \vee, \diamond\right\}$ and the minimum priority is $\min\{1,3,3\} = 1$ and Player 0 loses. Indeed, the run passes infinitely many times by the μ of priority 1, but only finitely many times through the ν of priority 0 (in fact, only once).

We have shown here that there is a winning strategy for Player 1, regardless of the starting node $s \in S$, the formula φ holds nowhere in this model.

To understand what φ does, we look at the game with a model in which it is verified. This model is the infinite binary tree on which every second node on the leftmost branch satisfy the property p:



The difference with the previous example is that now, Player 0 can sometimes choose to move to the node $\wedge$ without directly losing in the next turn. Indeed, if she chooses to move to $\wedge$ from $\vee$ when the current position in the model satisfies p, Player 1

will not move to the node (p) as he would directly lose. Instead, a wise choice is to move up to $(\Diamond)$. From there, it is up to Player 0 to choose where to move inside the model before reaching the initial state (ν) in the formula.

If Player 0 somehow manages to do this loop infinitely many, they will have visited (ν) of priority 0 infinitely many times and she would win. She can do this if and only if the run inside the model visits infinitely many times a node with p, so that she can decide to go to the loop on the left of the formula (according to the chosen layout of the graph).

If the run starts on the leftmost branch, her strategy is then deceptively simple, as it is always her that chooses where to move in the model (there are no box operators in the formula), which just stays in the leftmost branch. Every time she reaches $(\vee)$, if the current node has label p, she decides to take the left loop in the formula, otherwise she takes the left loop. In the model given in the example, she would alternate between the left and right loop, as every second state has the label p. However, in a more general setting, she could wait as long as she wants on the right loop, waiting for a state with p and then take the left loop.

Even if this might not be obvious at first sight, this means that φ is valid in every node from which there exists a path that visits infinitely many times a node with label p.

Definition 3.6. Let φ be a formula of μ-calculus, $\mathbb{S} = (S, R, V)$ be a model and $s_0 \in S$ a starting state. The **model checking game** of φ on $(\mathbb{S}, s_0)$ is a parity game as follows.

Vertices The game starts at (φ, s_0). At each turn, the two players play both a subformula of φ and a node $s_i \in S$. If we denote the set of subformulas of φ by Φ, the vertices of the graph is thus $\mathcal{V} = \Phi \times S$.

Edges A node $(\psi, s) \in \mathcal{V}$ is connected to other nodes depending on the type of ψ. and if Z is a variable in φ, we write $\varphi_Z \in \Phi$ for the subformula of φ that binds Z. The edges of the graph are given as follows:

ψ	Neighbours of (ψ, s)	Partition
Atomic propositon p	$\varnothing$	$\begin{cases} 1-N & \text{if } s \in V(p) \\ N & \text{otherwise} \end{cases}$
Variable Z	$\{(\varphi_Z, s)\}$	0
$\neg\psi'$	$\{(\psi', s)\}$	0
$\psi_0 \vee \psi_1$	$\{(\psi_0, s), (\psi_1, s)\}$	N
$\psi_0 \wedge \psi_1$	$\{(\psi_0, s), (\psi_1, s)\}$	1 - N
$\langle m \rangle \psi'$	$\{(\psi', t) \mid t \in \mathrm{Neigh}^m_{\mathbb{S}}(s)\}$	N
$[m] \psi'$	$\{(\psi', t) \mid t \in \mathrm{Neigh}^m_{\mathbb{S}}(s)\}$	1 - N
$\mu Z\ \psi'$	$\{(\psi', s)\}$	0
$\nu Z\ \psi'$	$\{(\psi', s)\}$	0

Partition The last column of the table discribes the partition, V_0 or V_1 in which the node (ψ, s) belongs. This partition depends on whether there is an even number of negations above ψ in φ or an odd number. We denoted by $N \in \{0, 1\}$ the parity of the number of negations, so that if a node in the partition N is under an even number of negations, it is in V_0 and otherwise in V_1, and conversly for $1 - N$. When there is only one neighbour in the graph, it doesn't matter who plays, so we arbitrarily put all those nodes in V_0.

This formalises the fact that the two players switch roles when encountering a negation. Note that there is no problem with loops in the graph, as the only loops are made by variables, which are always under an even number of negations.

For atomic propositions, the game ends and we want the winner to be Player 0 if p holds at s, except if p is negated (or under an odd number of negations), in which case the winner is Player 1.

Priorities To define the priority function $p : \mathcal{V} \to \mathbb{N}$ we need an auxilary function $h : \Phi \to \mathbb{N}$ defined recursively from the root of φ and downward to each subformula ψ. Given $\psi \in \Phi$, we write $\psi^\wedge$ for the set of all formulas that contain strictly ψ. We define h as follows:

- If ψ is a μ-formula or ν-formula, that is, a formula with the operator μ (resp. ν) at the root and $\varphi^\wedge$ contains no fixpoint operator. Then

$$h(\psi) = \begin{cases} 0 & \text{if } \psi \text{ is a } \nu \text{ formula} \\ 1 & \text{if } \psi \text{ is a } \mu \text{ formula} \end{cases}$$

- If ψ is a μ-formula (ie. a formula with the operator μ at the root), then

$$h(\psi) := 1 + \max\left\{p(\chi) \,\middle|\, \chi \in \psi^{\wedge} \wedge \chi \text{ is a } \nu \text{ formula}\right\}$$

- If ψ is a ν-formula, then

$$h(\psi) := 1 + \max\left\{p(\chi) \,\middle|\, \chi \in \psi^{\wedge} \wedge \chi \text{ is a } \mu \text{ formula}\right\}$$

- Otherwise $h(\varphi)$ is any number larger that every value given to μ and ν formulas, for instance, the size of the whole formula $h(\psi) := |\varphi|$

We then simply define $p(\psi, s) := h(\psi)$.

A. Computing the Safra construction

This appendix contains the Python code that was developped to convert any Büchi automaton into a Muller automaton. The code runs in any linux environement where its three dependencies are installed:

- The Graphviz software, used to layout and draw graphs: https://www.graphviz.org/
- Python 3.9 (or higher): https://www.python.org/
- The python packages `click` (for the command line interface) and `dot2tex` (conversion of Graphviz output to LaTeX) which are available on PyPI and can be installed with `pip install click dot2tex`.

This code can be run with `python safra.py --help` to get a list of the available options and usage examples. Because scripts and software are living beings, a more up-to-date version can be found at https://gitlab.com/ddorn/safra.

```python
"""
Conversion from non-deterministic Büchi automata to deterministic Muller
automata via the Safra construction.

This script includes visualisation of the automata with the help of Graphviz and Tikz/Latex.
Both must be installed to use this script.

Author: Diego Dorn, 2022
License: WTFPL
"""

from __future__ import annotations

import dataclasses
import subprocess
from collections import deque
from dataclasses import dataclass
from pprint import pprint
from textwrap import indent
from typing import Generator, Iterator

import click
import dot2tex

fset = frozenset
Label = str | int
Transition = dict[tuple[Label, int], set[Label]]
DetTransition = dict[tuple[Label, int], Label]


@dataclass(frozen=True)
class SafraNode:
    """
    A node in the Safra tree.
    """
    name: int
    label: fset[Label]
    children: tuple[int, ...]
    marked: bool = False


class SafraTree(dict[int, SafraNode]):
    """
    A Safra tree, where each node is labeled with an integer.
    """

    @classmethod
    def new(cls, initial_state: Label) -> SafraTree:
        """Create a Safra tree with only one node containing one state."""
        return cls({0: SafraNode(0, fset([initial_state]), ())})

    def power(self, buchi: BuchiAutomaton, input: int) -> SafraTree:
```

```
        """Apply the power set rule."""
        result = SafraTree()
        for name, node in self.items():
            result[name] = SafraNode(name,
                                     label=fset().union(*(buchi.next(s, input)
                                                          for s in node.label)),
                                     children=node.children,
                                     marked=False)

        return result

    def branch_accepting(self, accepting: set[Label]) -> SafraTree:
        """Apply the branch accepting rule."""

        available_names = [i for i in reversed(range(2 * len(self))) if i not in self]

        def get_nodes(name: int) -> Iterator[SafraNode]:
            node = self[name]

            acc = node.label.intersection(accepting)
            if acc:
                new_name = available_names.pop()
                yield dataclasses.replace(node, children=node.children + (new_name, ))
                yield SafraNode(new_name, acc, ())
            else:
                yield node
            for c in node.children:
                yield from get_nodes(c)

        return SafraTree({n.name: n for n in get_nodes(0)})

    def make_disjoint(self) -> SafraTree:
        """Apply the make disjoint rule."""

        def get_nodes(name: int, parent: SafraNode | None) -> Iterator[SafraNode]:
            node = self[name]

            if parent is None:
                new = node
            else:
                siblings = parent.children
                older_siblings = siblings[:siblings.index(name)]
                older_labels = fset().union(*(self[c].label for c in older_siblings))
                # We remove those that have been removed from the parent also
                new_label = node.label.difference(older_labels).intersection(parent.label)
                new = dataclasses.replace(node, label=new_label)

            yield new
            for c in node.children:
                yield from get_nodes(c, new)

        return SafraTree({n.name: n for n in get_nodes(0, None)})

    def mark_nodes(self) -> SafraTree:
        """Apply the mark nodes rule."""

        def get_nodes(name: int) -> Iterator[SafraNode]:
            node = self[name]

            if set().union(*(self[c].label
                             for c in node.children)) == node.label and node.label != fset():
                yield SafraNode(node.name, node.label, (), True)
            else:
                yield node
                for c in node.children:
                    yield from get_nodes(c)

        return SafraTree({n.name: n for n in get_nodes(0)})

    def remove_empty(self) -> SafraTree:
        """Remove empty nodes."""
        result = SafraTree()
        for name, node in self.items():
            if node.label or node.name == 0:                                    # We allways keep the root
                result[name] = SafraNode(name, node.label,
                                         tuple(c for c in node.children if self[c].label),
                                         node.marked)
        return result

    def next(self, buchi: BuchiAutomaton, input: int) -> SafraTree:
        """Apply the 5 rules of the safra construction."""
        return (self.branch_accepting(buchi.accepting_states) # 2: create new children
                .power(buchi, input)                          # 1: First step
                .make_disjoint()                              # 3: Clean up
```

```
                .remove_empty()                                 #    and remove empty nodes
                .mark_nodes())                                  # 4: Mark children

    def to_latex_forest(self) -> str:
        """Get the tree in LaTeX format."""

        def node_str(name: int) -> str:
            node = self[name]
            if node.label:
                label = ''.join(sorted(map(str, node.label)))
            else:
                label = '$\\emptyset$'
            full = "\\nodepart{one}" + str(name) + "\\nodepart{two}" + label
            if node.marked:
                full += ' !'
            full = "{" + full + "}"

            children = '\n'.join(map(node_str, node.children))
            if children:
                return f"[{full}\n{indent(children, '  ')}]"
            else:
                return f"[{full}]"

        r = "\\begin{forest}safra,\n"
        r += node_str(0)
        r += '\n\\end{forest}'
        return r


@dataclass
class Automaton:
    """Base class for automaton"""

    initial_state: Label

    def transitions(self) -> Generator[tuple[Label, int, Label], None, None]:
        raise NotImplementedError

    def to_graphviz(self, labels: dict[Label, str] | None = None, edge_len: float = 1.0) -> str:
        """
        Convert the automaton to the graphviz format.

        Arguments:
            labels: An optional dictionary of state labels to show.
            edge_len: The length of the edges for use with the neato algorithm.
        """

        r = f"digraph {self.__class__.__name__} {{\n" #rankdir=LR;\n"

        # Nodes / states
        if labels:
            r += "  {"
            for state, txt in labels.items():
                escaped = txt.replace('"', r'\"')
                initial = ',initial' if state == self.initial_state else ''
                r += f'    {state} [label="{escaped}",style="safrastate{initial}"];\n'
            r += "  }\n"

        # Edges / transitions
        r += f'edge [len={edge_len}];\n'
        for l in (0, 1):
            if l == 0:
                r += 'edge [style="edge0"];\n'
            else:
                r += 'edge [style="edge1"];\n'

            for start, label, end in self.transitions():
                if label == l:
                    r += f"{start} -> {end} [label=\"{label}\"];\n"

        r += "}\n"

        return r


@dataclass
class BuchiAutomaton(Automaton):
    """A non-deterministic Büchi automaton."""

    transition_function: Transition
    accepting_states: set[Label]

    def transitions(self) -> Generator[tuple[Label, int, Label], None, None]:
        for (s, i), next_states in self.transition_function.items():
```

```
            for next_state in next_states:
                yield s, i, next_state

    def next(self, state: Label, input: int) -> set[Label]:
        return self.transition_function.get((state, input), set())

    def to_muller(self) -> tuple[MullerAutomaton, dict[int, SafraTree]]:
        """Convert to a deterministic Muller automaton using the Safra construction"""

        start = SafraTree.new(self.initial_state)
        to_compute = deque([(start, 0), (start, 1)])
        transition: dict[tuple[Label, int], SafraTree] = {}
        ids: list[SafraTree] = []
        while to_compute:
            # print("left:", len(to_compute), "computed:", len(transition))
            tree, input = to_compute.popleft()
            try:
                id = ids.index(tree)
            except ValueError:
                id = len(ids)
                ids.append(tree)

            if (id, input) in transition:
                continue # already computed
            new = tree.next(self, input)
            transition[id, input] = new
            to_compute.append((new, 0))
            to_compute.append((new, 1))

        transition_with_ids: DetTransition = {
            key: ids.index(value)
            for key, value in transition.items()
        }

        return MullerAutomaton(0, transition_with_ids, []), dict(enumerate(ids))


@dataclass
class MullerAutomaton(Automaton):
    """A deterministic Muller automaton."""

    transition_function: DetTransition
    accepting_states: list[set[Label]]

    def transitions(self) -> Generator[tuple[Label, int, Label], None, None]:
        for (s, i), next_state in self.transition_function.items():
            yield s, i, next_state


A, B, C, D, E, F, G, Z = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'Z']
AUTOMATA: list[BuchiAutomaton] = [
    BuchiAutomaton(  # Infinitely many 1 but finitely many 11
        initial_state=A,
        transition_function={
            (A, 0): {A, B}, (A, 1): {A, B}, (B, 0): {B}, (B, 1): {C},
            (C, 0): {B}, (C, 1): set(),
        },
        accepting_states={C},
    ),
    BuchiAutomaton(  # Infinitely many 1 and iff infinitely many 11 then infinitely many 111
        transition_function={
            (A, 0): {A, B, D}, (A, 1): {A, B, D}, (B, 0): {B}, (B, 1): {C},
            (C, 0): {B}, (C, 1): set(), (D, 0): {D}, (D, 1): {E}, (E, 0): {D},
            (E, 1): {F}, (F, 0): {D}, (F, 1): {G}, (G, 0): {D}, (G, 1): {D, G},
        },
        accepting_states={C, G},
        initial_state=A,
    ),
    BuchiAutomaton(  # zeros always followed by the same parity of ones
        transition_function={
            (A, 0): {B}, (A, 1): {C}, (B, 0): {A}, (B, 1): {D}, (C, 0): set(),
            (C, 1): {D, Z}, (D, 0): set(), (D, 1): {C}, (Z, 0): {B}, (Z, 1):
            set(),
        },
        accepting_states={Z},
        initial_state=A
    ),
    BuchiAutomaton(  # 4 states but produces 270 in the Muller automtaton
        initial_state=0,
        transition_function={
            (0, 0): {0}, (0, 1): {3}, (1, 0): {0, 3}, (1, 1): {1}, (2, 0): {1},
            (2, 1): {0, 2}, (3, 0): {2}, (3, 1): set()},
        accepting_states={0}
    )
```

```
]


@click.command()
@click.option('-n', '--no-trees', is_flag=True, default=False)
@click.option('--prog',
              default='dot',
              type=str,
              help='The program to use to render the graph. "dot" and "neato" work best.')
@click.option('--edge-len',
              default=1.0,
              type=float,
              help="Target length for edges for the neato layout.")
@click.option('-o', '--output', default='graph.tex', help="Latex file to store the graph to.")
@click.option('-b',
              '--draw-buchi',
              is_flag=True,
              help="Draw the Buchi automaton instead of the Muller automton.")
@click.argument('automaton', default=-1)
def main(edge_len: float,
         no_trees: bool,
         prog: str,
         output: str,
         automaton: int,
         draw_buchi: bool = False) -> None:
    """Draw an convert Buchi automata into Muller automata.
    AUTOMATA is the index of the automaton to work with inside the AUTOMATA list."""

    buchi = AUTOMATA[automaton]

    if draw_buchi:
        dot = buchi.to_graphviz()
    else:
        muller, safra_trees = buchi.to_muller()
        if no_trees:
            labels = None
        else:
            labels = {k: v.to_latex_forest() for k, v in safra_trees.items()}
        dot = muller.to_graphviz(labels, edge_len)

    template = open('template.tex', 'r').read() + '\n'
    tex = dot2tex.dot2tex(
        dot,
        prog=prog,
        format='tikz',
        texmode='raw',
        crop=True,
        autosize=True,
        tikzedgelabels=True,
        template=template,
    )
    with open(output, 'w') as f:
        f.write(tex)
    subprocess.check_call(['pdflatex', '--output-directory', 'out/', output])

    pprint(buchi)


if __name__ == "__main__":
    main()
```

References

[Calude et al., 2017] Calude, C. S., Jain, S., Khoussainov, B., Li, W., and Stephan, F. (2017). Deciding parity games in quasipolynomial time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 252–263.

[Janin and Walukiewicz, 1996] Janin, D. and Walukiewicz, I. (1996). On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *International Conference on Concurrency Theory*, pages 263–277. Springer.

[Jurdziński, 1998] Jurdziński, M. (1998). Deciding the winner in parity games is in up∩ co-up. *Information Processing Letters*, 68(3):119–124.

[Libkin, 2004] Libkin, L. (2004). *Elements of finite model theory*, volume 41. Springer.

[Martin, 1975] Martin, D. A. (1975). Borel determinacy. *Annals of Mathematics*, 102(2):363–371.

[Walukiewicz, 2001] Walukiewicz, I. (2001). Automata and logic.