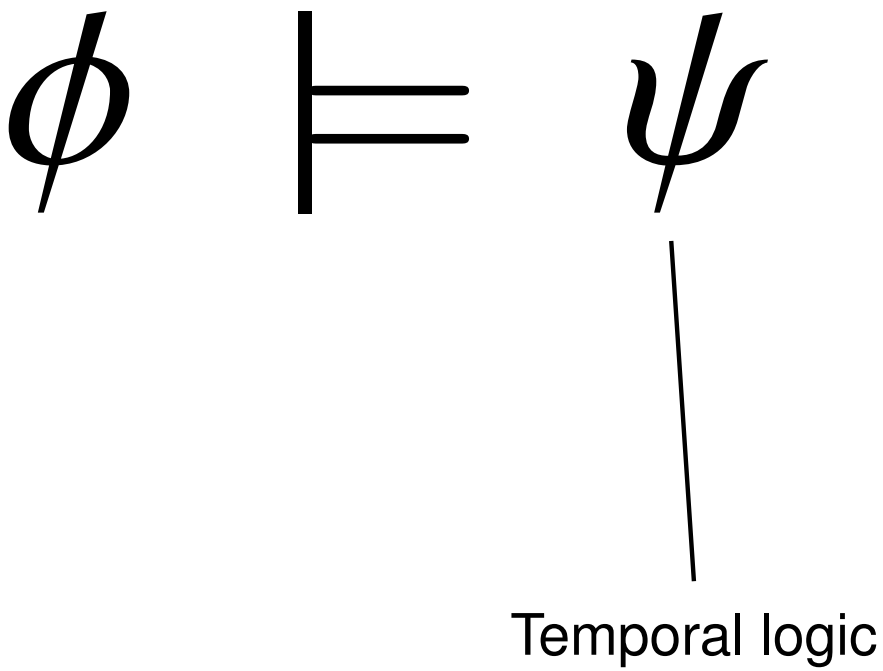
Model Checking and Games

Part IV - Model checking LTL

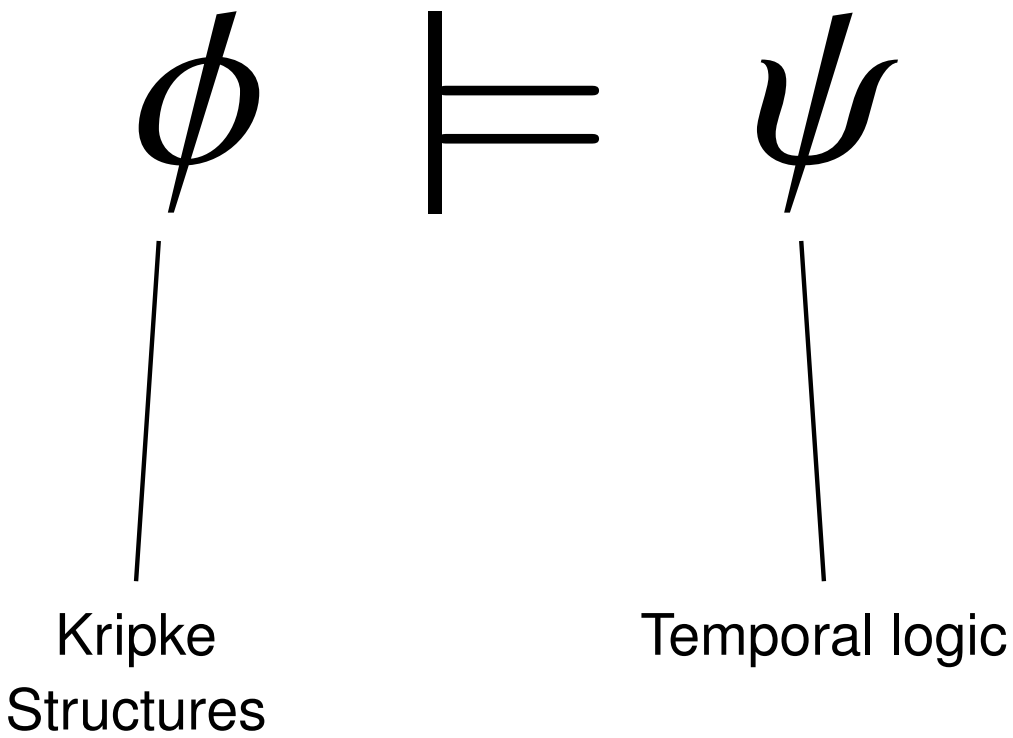
Rüdiger Ehlers, Clausthal University of Technology

September 2019

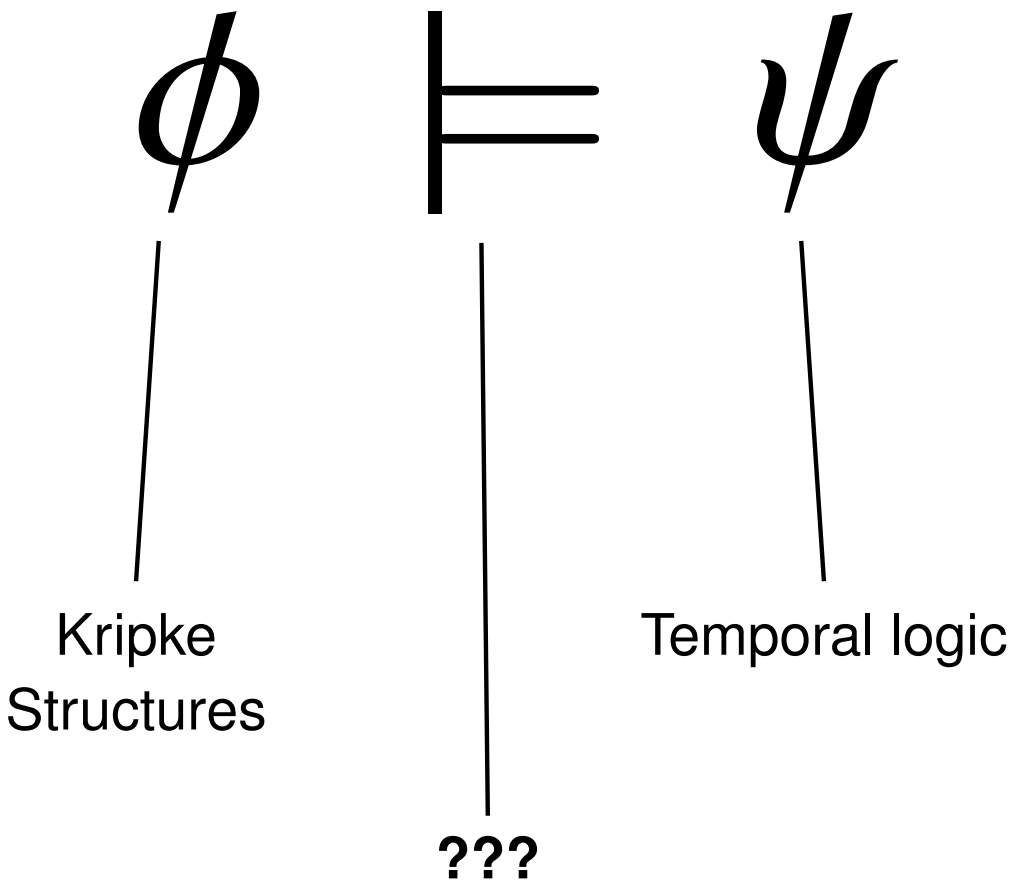
Introduction



Introduction



Introduction

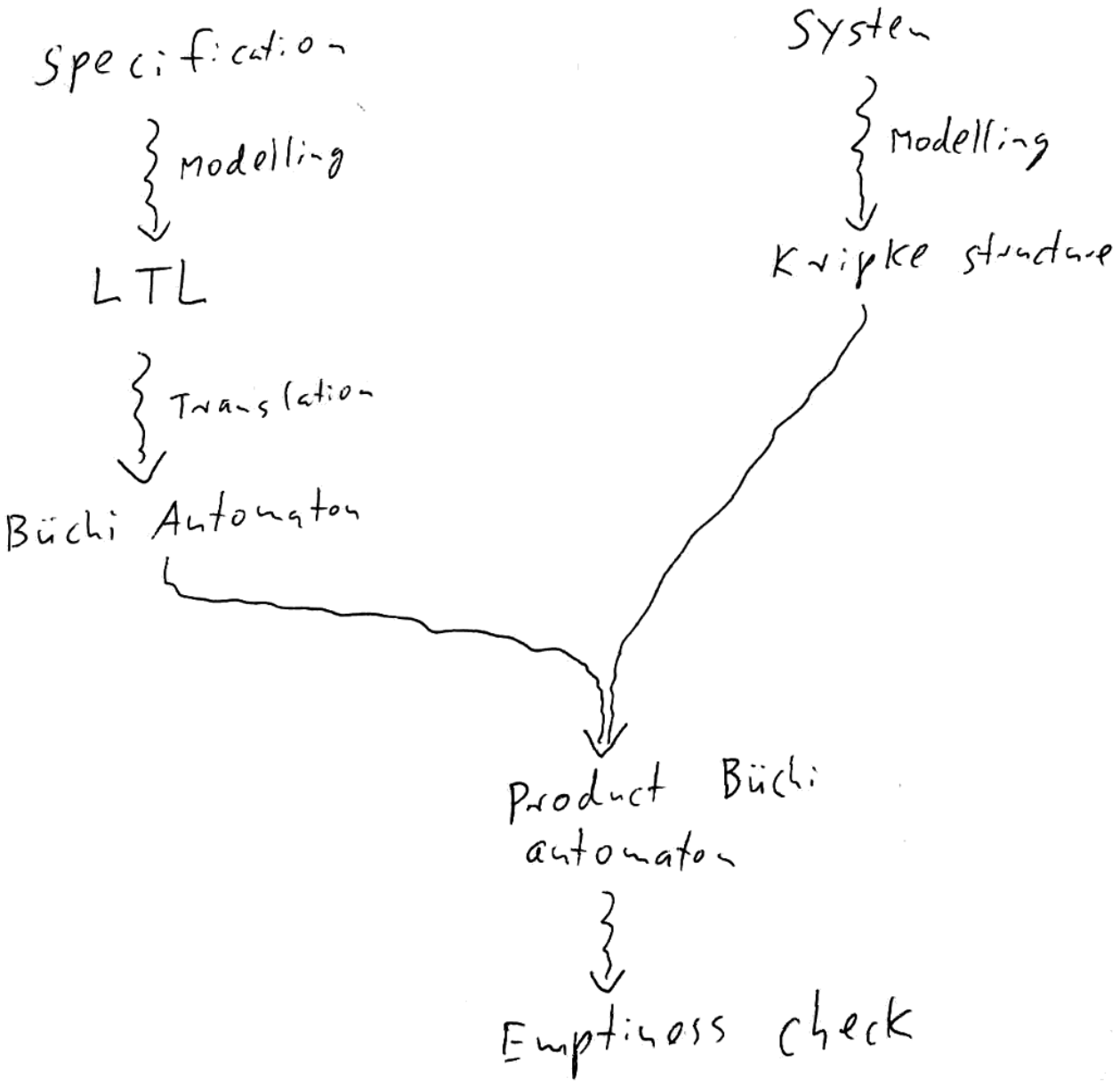


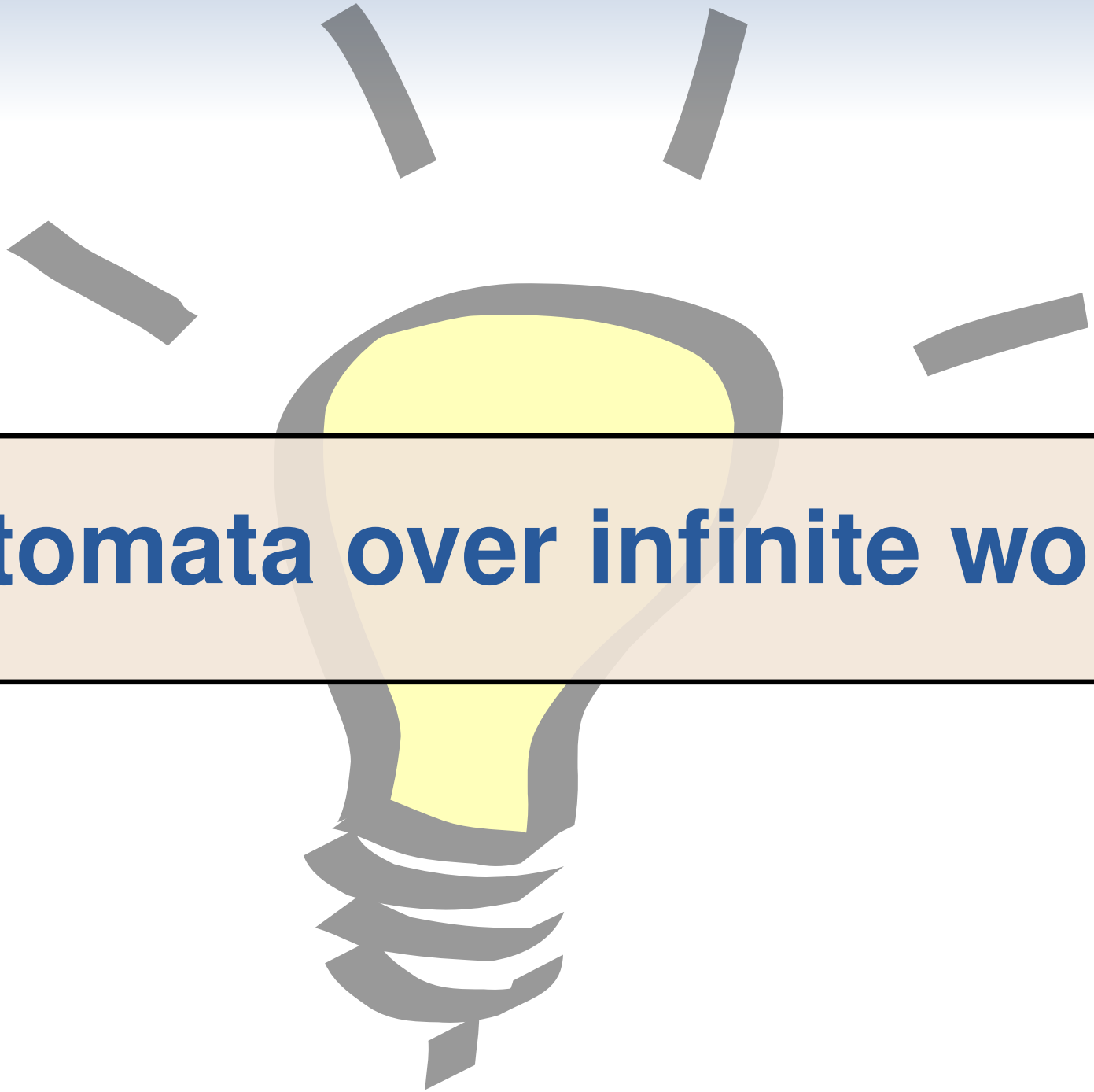
Approach pursued in the following

Steps

1. Translating the specification to an *automaton over infinite words*.
2. Building the product of the automaton and the Kripke structure
3. Checking the product for *language emptiness*

Overview


Specification
modelling
LTL
Translation
Büchi Automaton
System
Modelling
Kripke structure
Product Büchi automaton
Emptiness check



Automata over infinite words

Motivation

Automata over infinite words

- They capture specifications as *languages*
- They are a more “technical” representation for a specification, as needed for model checking.
- Well-researched topic since the 1960’s.

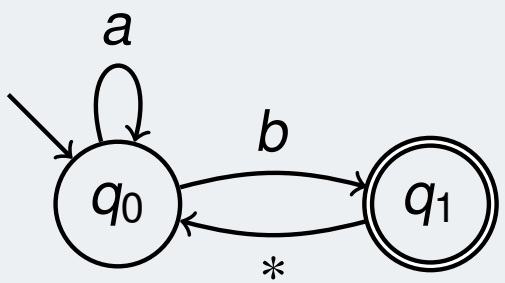
A first automaton type: (Non-deterministic) Büchi automata

Büchi automata

- Introduced by Büchi (1962)
- Have the same syntactic structure as automata over finite words
- *Accept* or *reject* infinite words over some alphabet Σ

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

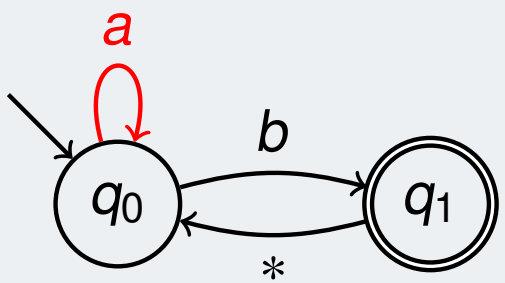


An example *run* of the automaton for a word

$\rho =$												
$\pi =$	$q0$											

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

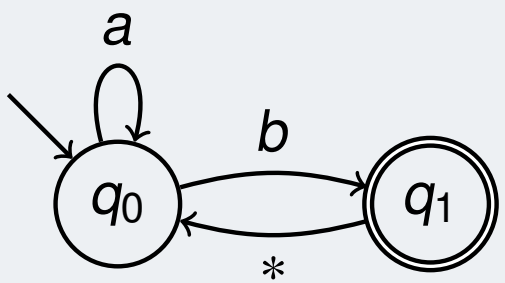


An example *run* of the automaton for a word

$\rho =$	a										
$\pi =$	$q0$										

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

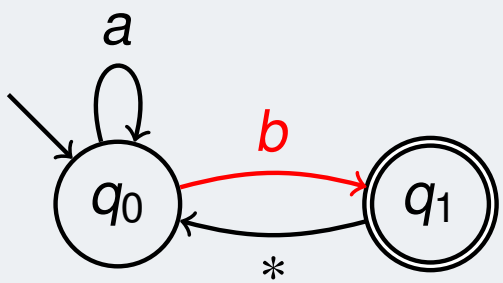


An example *run* of the automaton for a word

$\rho =$	a											
$\pi =$	$q0$	$q0$										

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

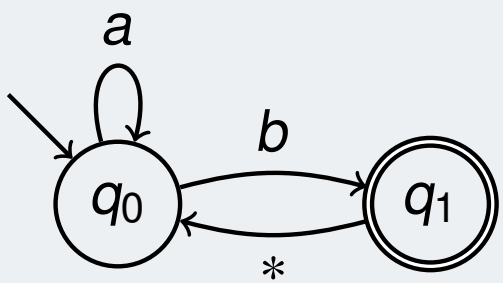


An example *run* of the automaton for a word

$\rho =$	a	b										
$\pi =$	$q0$	$q0$										

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

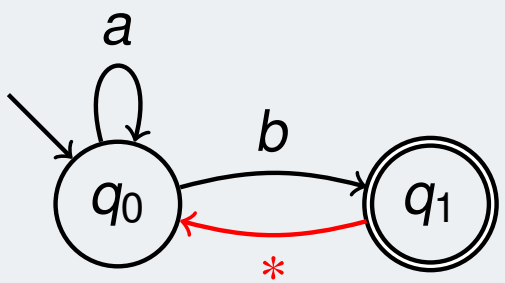


An example *run* of the automaton for a word

$\rho =$	a	b										
$\pi =$	$q0$	$q0$	$q1$									

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

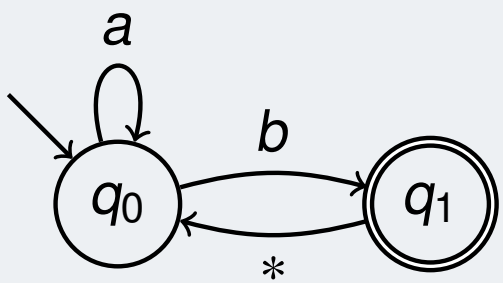


An example *run* of the automaton for a word

$\rho =$	a	b	a									
$\pi =$	$q0$	$q0$	$q1$									

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

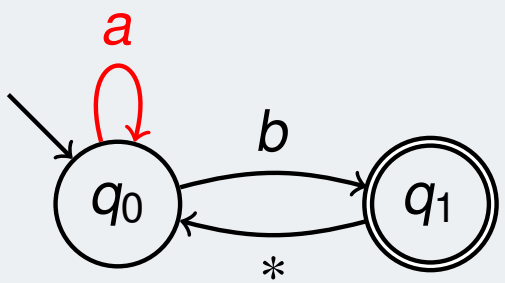


An example *run* of the automaton for a word

$\rho =$	a	b	a									
$\pi =$	$q0$	$q0$	$q1$	$q0$								

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

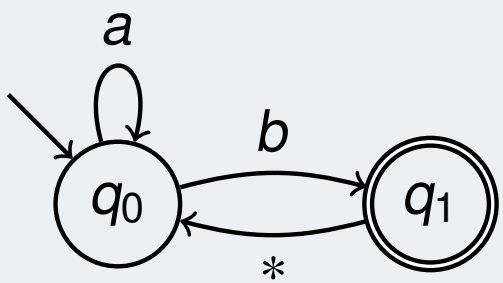


An example *run* of the automaton for a word

$\rho =$	a	b	a	a								
$\pi =$	$q0$	$q0$	$q1$	$q0$								

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

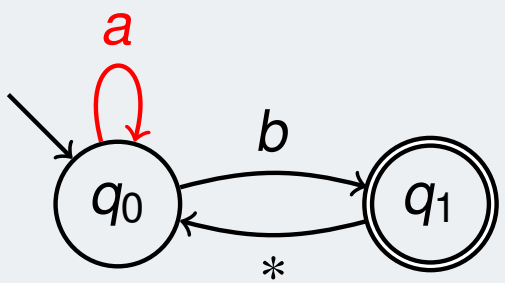


An example *run* of the automaton for a word

$\rho =$	a	b	a	a								
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$							

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

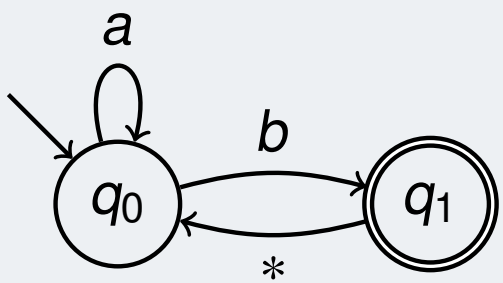


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a							
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$							

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

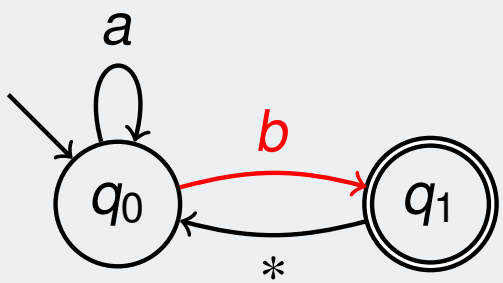


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a							
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$						

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

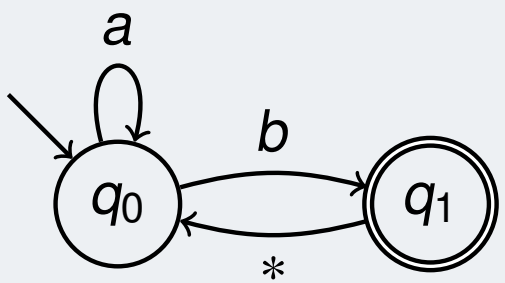


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b						
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$						

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

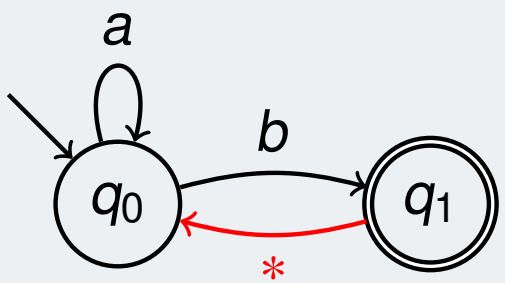


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b						
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$					

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

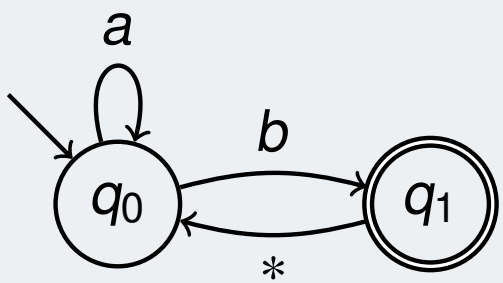


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a					
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$					

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

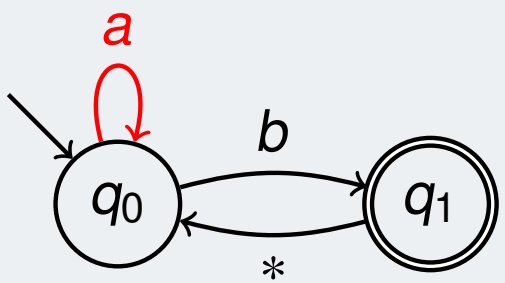


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a					
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$				

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

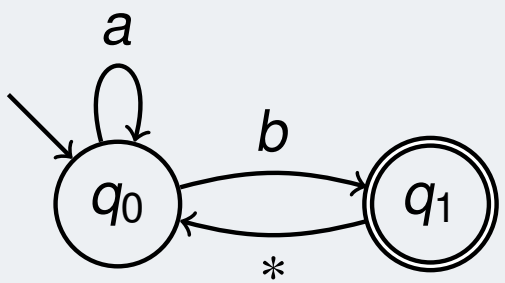


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a				
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$				

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

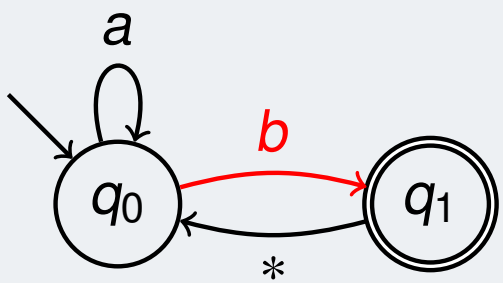


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a				
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$	$q0$			

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

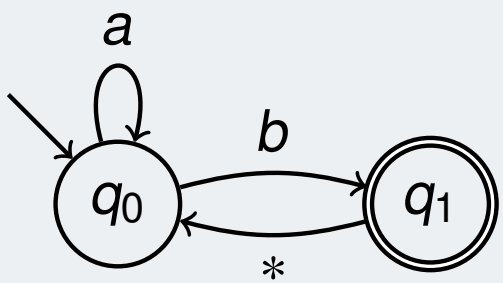


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a	b			
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$	$q0$			

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

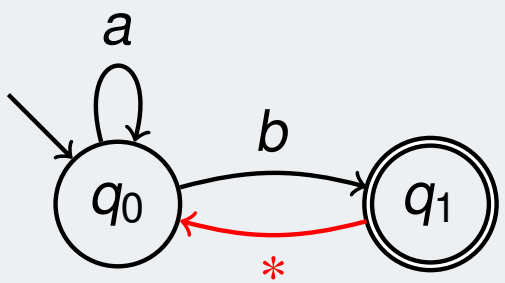


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a	b			
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$	$q0$	$q1$		

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

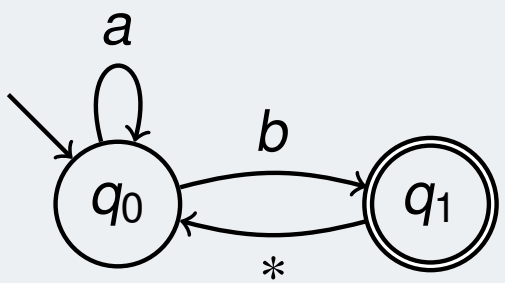


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a	b	b		
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$	$q0$	$q1$		

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

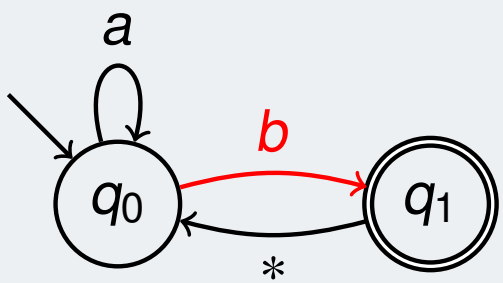


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a	b	b		
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$	$q0$	$q1$	$q0$	

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

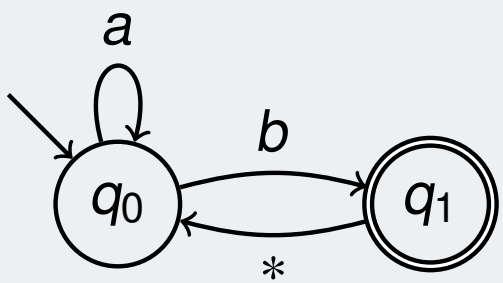


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a	b	b	b	
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$	$q0$	$q1$	$q0$	

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$

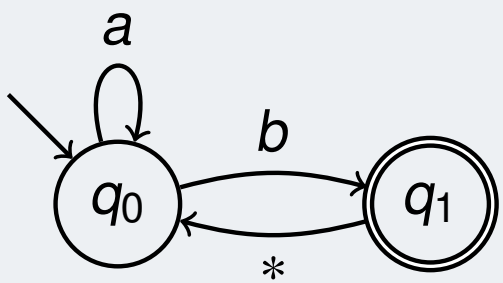


An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a	b	b	b	
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$	$q0$	$q1$	$q0$	$q1$

Büchi automata by example

Example automaton 1 for $\Sigma = \{a, b\}$



An example *run* of the automaton for a word

$\rho =$	a	b	a	a	a	b	a	a	b	b	b	$\ldots$
$\pi =$	$q0$	$q0$	$q1$	$q0$	$q0$	$q0$	$q1$	$q0$	$q0$	$q1$	$q0$	$q1$ $\ldots$

Formal definition of Büchi automata (1)

Definition

Given a finite alphabet Σ, a (non-deterministic) Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with the following components:

- A *finite* set of states Q,
- the alphabet Σ,
- the transition relation $\delta \subseteq Q \times \Sigma \times Q$,
- the initial states $Q_0 \subseteq Q$, and
- the set of *accepting states* $F \subseteq Q$.

Note that these are exactly the same components as for a (non-deterministic) automaton over finite words (NFAs). The similarities between these concepts do not end here, however.

Formal definition of Büchi automata (2)

Definitions

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be a Büchi automaton and $w = w_0 w_1 \ldots \in \Sigma^\omega$ be an infinite word.

Formal definition of Büchi automata (2)

Definitions

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be a Büchi automaton and $w = w_0 w_1 \ldots \in \Sigma^\omega$ be an infinite word. We say that some infinite sequence of states $\pi = \pi_0 \pi_1 \ldots \in Q^\omega$ is an infinite run of $\mathcal{A}$ on w if $\pi_0 \in Q_0$, and for every $j \in \mathbb{N}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta$.

Formal definition of Büchi automata (2)

Definitions

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be a Büchi automaton and $w = w_0 w_1 \ldots \in \Sigma^\omega$ be an infinite word. We say that some infinite sequence of states $\pi = \pi_0 \pi_1 \ldots \in Q^\omega$ is an infinite run of $\mathcal{A}$ on w if $\pi_0 \in Q_0$, and for every $j \in \mathbb{N}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta$.

Likewise, some finite sequence of states $\pi = \pi_0 \pi_1 \ldots \pi_n \in Q^\omega$ is a finite run of $\mathcal{A}$ on w if $\pi_0 \in Q_0$, and for every $0 \leq i < n \in \mathbb{N}$, we have $(\pi_i, w_i, \pi_{i+1}) \in \delta$.

Formal definition of Büchi automata (3)

As for NFAs, runs can be accepting or rejecting, and the automaton accepts all words that have accepting runs. Since we are dealing with words of infinite length here, however, we cannot define the acceptance of such a word over which state is reached last.

Formal definition of Büchi automata (3)

As for NFAs, runs can be accepting or rejecting, and the automaton accepts all words that have accepting runs. Since we are dealing with words of infinite length here, however, we cannot define the acceptance of such a word over which state is reached last.

Definition

A run $\pi = \pi_0\pi_1 \ldots \in Q^\omega$ of a Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is accepting if and only if the set $\{i \in \mathbb{N} \mid \pi_i \in F\}$ is infinitely large.

Formal definition of Büchi automata (3)

As for NFAs, runs can be accepting or rejecting, and the automaton accepts all words that have accepting runs. Since we are dealing with words of infinite length here, however, we cannot define the acceptance of such a word over which state is reached last.

Definition

A run $\pi = \pi_0\pi_1 \ldots \in Q^\omega$ of a Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is accepting if and only if the set $\{i \in \mathbb{N} \mid \pi_i \in F\}$ is infinitely large.

Note that since Q is a finite set of states, this is equivalent to saying that there is at least one state that occurs infinitely often along the run.

Formal definition of Büchi automata (3)

As for NFAs, runs can be accepting or rejecting, and the automaton accepts all words that have accepting runs. Since we are dealing with words of infinite length here, however, we cannot define the acceptance of such a word over which state is reached last.

Definition

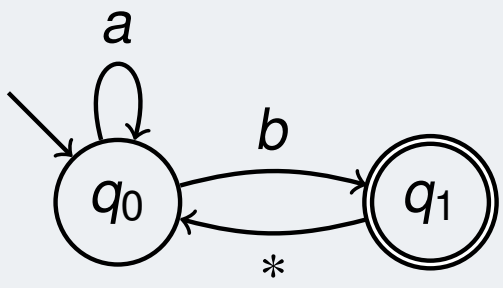
A run $\pi = \pi_0\pi_1 \ldots \in Q^\omega$ of a Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is accepting if and only if the set $\{i \in \mathbb{N} \mid \pi_i \in F\}$ is infinitely large.

Note that since Q is a finite set of states, this is equivalent to saying that there is at least one state that occurs infinitely often along the run.

The set of words accepted by the automaton is called its *language*.

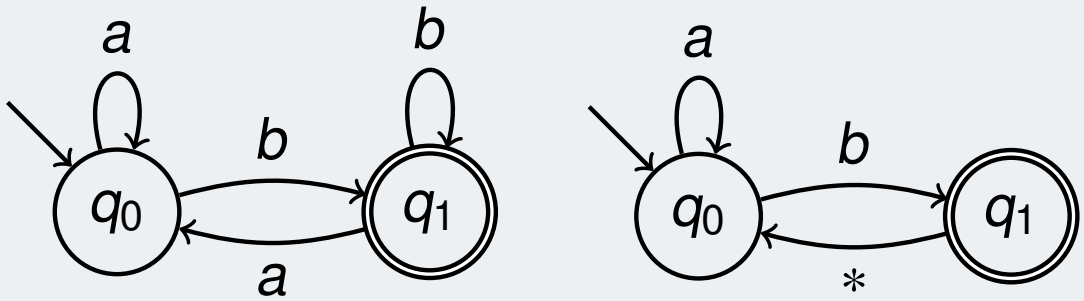
Revisiting the example

Example automata for $\Sigma = \{a, b\}$



Continued example

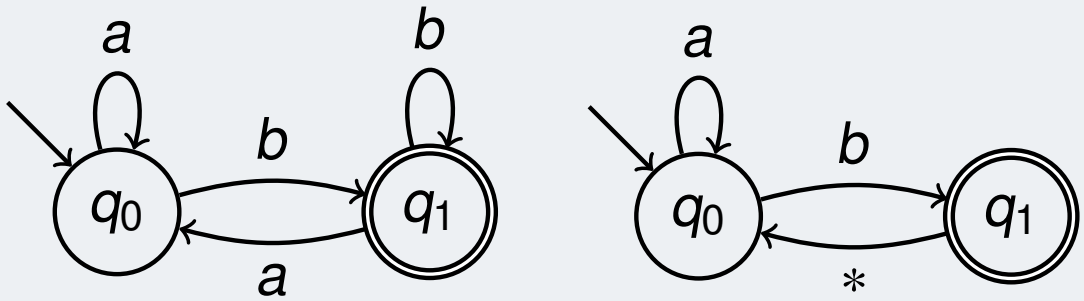
Example automata for $\Sigma = \{a, b\}$



Do they represent different languages?

Continued example

Example automata for $\Sigma = \{a, b\}$

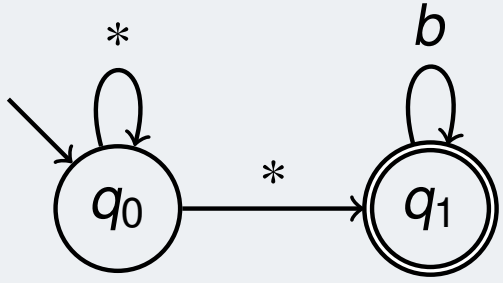


Do they represent different languages?

No, as the automaton still accepts all words in which there are infinitely many *b*s.

Some examples (2)

Example automaton 2 for $\Sigma = \{a, b\}$



Let's build some Büchi automata!

Example 1 for $\Sigma = \{a, b, c\}$

The automaton should accept all words for which a is infinitely often immediately followed by c.

Let's build some Büchi automata!

Example 1 for $\Sigma = \{a, b, c\}$

The automaton should accept all words for which a is infinitely often immediately followed by c.

Example 2 for $\Sigma = \{a, b, c\}$

The letter a should never be immediately be followed by b, otherwise every word should be accepted.

Let's build some Büchi automata!

Example 1 for $\Sigma = \{a, b, c\}$

The automaton should accept all words for which a is infinitely often immediately followed by c.

Example 2 for $\Sigma = \{a, b, c\}$

The letter a should never be immediately be followed by b, otherwise every word should be accepted.

Example 3 for $\Sigma = \{a, b, c\}$

At some point in the future, every sequence of as is immediately followed by a b.

Deterministic vs. non-deterministic Büchi automata (1)

Determinism

We say some that automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is *deterministic* if for every $q \in Q$ and $x \in \Sigma$, there is at most one state $q' \in Q$ with $(q, x, q') \in \delta$.

Comparison to automata over finite words

While for automata over finite words, deterministic and non-deterministic automata are equally expressive, *deterministic* Büchi automata are less expressive than non-deterministic Büchi automata.

Deterministic vs. non-deterministic Büchi automata (2)

Claim

The language $L = \{wa^\omega \mid w \in \{a, b\}^*\}$ for $\Sigma = \{a, b\}$ is not representable by a deterministic Büchi automaton.

Proof by contradiction (part 1)

Assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is a deterministic Büchi automaton for L.

Deterministic vs. non-deterministic Büchi automata (2)

Claim

The language $L = \{wa^\omega \mid w \in \{a, b\}^*\}$ for $\Sigma = \{a, b\}$ is not representable by a deterministic Büchi automaton.

Proof by contradiction (part 1)

Assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is a deterministic Büchi automaton for L.

We build a word $w = w_0 w_1 \ldots \notin L$ that $\mathcal{A}$ accepts if it accepts all words in L.

Deterministic vs. non-deterministic Büchi automata (2)

Claim

The language $L = \{wa^\omega \mid w \in \{a, b\}^*\}$ for $\Sigma = \{a, b\}$ is not representable by a deterministic Büchi automaton.

Proof by contradiction (part 1)

Assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is a deterministic Büchi automaton for L.

We build a word $w = w_0 w_1 \ldots \notin L$ that $\mathcal{A}$ accepts if it accepts all words in L.

We start w with some fragment $w^1 \in \{a\}^*$ such that $\mathcal{A}$ visits some state in F from Q_0 when reading w^1.

Deterministic vs. non-deterministic Büchi automata (2)

Claim

The language $L = \{wa^\omega \mid w \in \{a, b\}^*\}$ for $\Sigma = \{a, b\}$ is not representable by a deterministic Büchi automaton.

Proof by contradiction (part 1)

Assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is a deterministic Büchi automaton for L.

We build a word $w = w_0 w_1 \ldots \notin L$ that $\mathcal{A}$ accepts if it accepts all words in L.

We start w with some fragment $w^1 \in \{a\}^*$ such that $\mathcal{A}$ visits some state in F from Q_0 when reading w^1.

Note that such a fragment exists as otherwise $\mathcal{A}$ would not accept a^ω.

Deterministic vs. non-deterministic Büchi automata (2)

Claim

The language $L = \{wa^\omega \mid w \in \{a, b\}^*\}$ for $\Sigma = \{a, b\}$ is not representable by a deterministic Büchi automaton.

Proof by contradiction (part 1)

Assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is a deterministic Büchi automaton for L.

We build a word $w = w_0 w_1 \ldots \notin L$ that $\mathcal{A}$ accepts if it accepts all words in L.

We start w with some fragment $w^1 \in \{a\}^*$ such that $\mathcal{A}$ visits some state in F from Q_0 when reading w^1.

Note that such a fragment exists as otherwise $\mathcal{A}$ would not accept a^ω.

Now consider a word starting with $w^1 b w^2$ for $w^2 \in a^*$.

Deterministic vs. non-deterministic Büchi automata (2)

Claim

The language $L = \{wa^\omega \mid w \in \{a, b\}^*\}$ for $\Sigma = \{a, b\}$ is not representable by a deterministic Büchi automaton.

Proof by contradiction (part 1)

Assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is a deterministic Büchi automaton for L.

We build a word $w = w_0 w_1 \ldots \notin L$ that $\mathcal{A}$ accepts if it accepts all words in L.

We start w with some fragment $w^1 \in \{a\}^*$ such that $\mathcal{A}$ visits some state in F from Q_0 when reading w^1.

Note that such a fragment exists as otherwise $\mathcal{A}$ would not accept a^ω.

Now consider a word starting with $w^1 b w^2$ for $w^2 \in a^*$.

If w^2 is long enough, we then know that accepting states are visited at least two times, once before having head $w^1 b$, and once during reading w^2.

Deterministic vs. non-deterministic Büchi automata (2)

Claim

The language $L = \{wa^\omega \mid w \in \{a, b\}^*\}$ for $\Sigma = \{a, b\}$ is not representable by a deterministic Büchi automaton.

Proof by contradiction (part 1)

Assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is a deterministic Büchi automaton for L.

We build a word $w = w_0 w_1 \ldots \notin L$ that $\mathcal{A}$ accepts if it accepts all words in L.

We start w with some fragment $w^1 \in \{a\}^*$ such that $\mathcal{A}$ visits some state in F from Q_0 when reading w^1.

Note that such a fragment exists as otherwise $\mathcal{A}$ would not accept a^ω.

Now consider a word starting with $w^1 b w^2$ for $w^2 \in a^*$.

If w^2 is long enough, we then know that accepting states are visited at least two times, once before having head $w^1 b$, and once during reading w^2.

This is because otherwise the automaton would not accept $w^1 b a^\omega$.

Deterministic vs. non-deterministic Büchi automata (2)

Proof by contradiction (part 2)

By continuing this lines of reasoning, we can find an infinite sequence of finite words $w^1 w^2 w^3 \ldots \in a^*$ such that for every $n \in \mathbb{N}$, accepting states are visited at least n times when reading $w^1 b w^2 b \ldots b w^n$.

Deterministic vs. non-deterministic Büchi automata (2)

Proof by contradiction (part 2)

By continuing this lines of reasoning, we can find an infinite sequence of finite words $w^1 w^2 w^3 \ldots \in a^*$ such that for every $n \in \mathbb{N}$, accepting states are visited at least n times when reading $w^1 b w^2 b \ldots b w^n$.

Hence, for the word $w^1 b w^2 b \ldots \in \Sigma^*$, the unique run of $\mathcal{A}$ visits accepting states infinitely often.

Deterministic vs. non-deterministic Büchi automata (2)

Proof by contradiction (part 2)

By continuing this lines of reasoning, we can find an infinite sequence of finite words $w^1 w^2 w^3 \ldots \in a^*$ such that for every $n \in \mathbb{N}$, accepting states are visited at least n times when reading $w^1 b w^2 b \ldots b w^n$.

Hence, for the word $w^1 b w^2 b \ldots \in \Sigma^*$, the unique run of $\mathcal{A}$ visits accepting states infinitely often.

However, this word is not in L.

Deterministic vs. non-deterministic Büchi automata (2)

Proof by contradiction (part 2)

By continuing this lines of reasoning, we can find an infinite sequence of finite words $w^1 w^2 w^3 \ldots \in a^*$ such that for every $n \in \mathbb{N}$, accepting states are visited at least n times when reading $w^1 b w^2 b \ldots b w^n$.

Hence, for the word $w^1 b w^2 b \ldots \in \Sigma^*$, the unique run of $\mathcal{A}$ visits accepting states infinitely often.

However, this word is not in L.

Hence, $\mathcal{A}$ cannot represent L.

Deterministic vs. non-deterministic Büchi automata (2)

Proof by contradiction (part 2)

By continuing this lines of reasoning, we can find an infinite sequence of finite words $w^1 w^2 w^3 \ldots \in a^*$ such that for every $n \in \mathbb{N}$, accepting states are visited at least n times when reading $w^1 b w^2 b \ldots b w^n$.

Hence, for the word $w^1 b w^2 b \ldots \in \Sigma^*$, the unique run of $\mathcal{A}$ visits accepting states infinitely often.

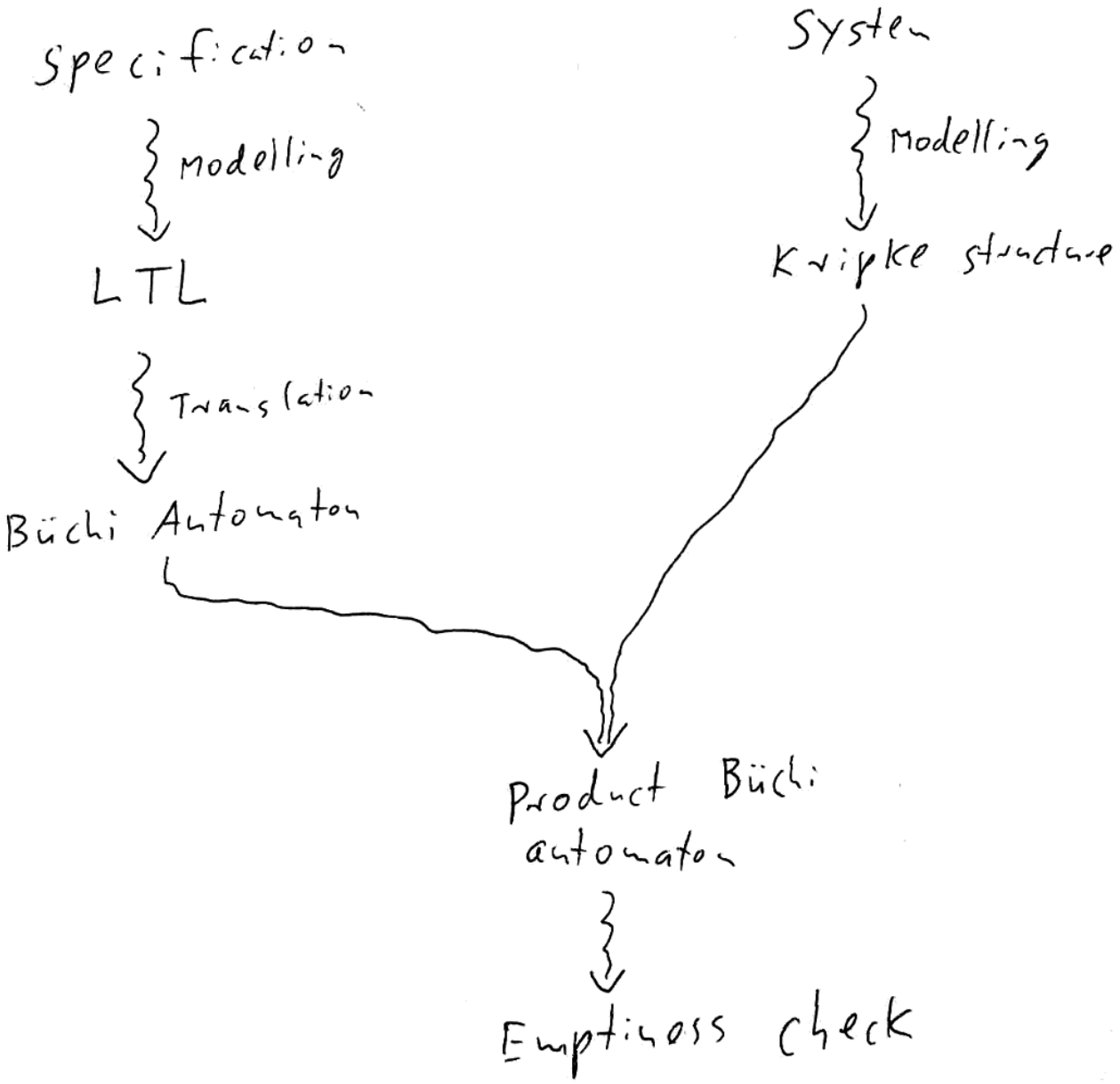
However, this word is not in L.

Hence, $\mathcal{A}$ cannot represent L.

Side note

This also shows that deterministic Büchi automata cannot capture all properties expressible in LTL.

Overview


Specification
modelling
LTL
Translation
Büchi Automaton
System
Modelling
Kripke structure
Product Büchi automaton
Emptiness check

Product construction between a Büchi automaton and a Kripke structure

Central Idea

We want to check if all *traces* of a Kripke structure satisfy a specification.

Product construction between a Büchi automaton and a Kripke structure

Central Idea

We want to check if all *traces* of a Kripke structure satisfy a specification.

This is the same as saying as saying that there should not exist a trace that satisfies the *complement of a specification*.

Product construction between a Büchi automaton and a Kripke structure

Central Idea

We want to check if all *traces* of a Kripke structure satisfy a specification.

This is the same as saying as saying that there should not exist a trace that satisfies the *complement of a specification*.

Let $\mathcal{A}$ be a Büchi automaton for the complement of a specification.

Product construction between a Büchi automaton and a Kripke structure

Central Idea

We want to check if all *traces* of a Kripke structure satisfy a specification.

This is the same as saying as saying that there should not exist a trace that satisfies the *complement of a specification*.

Let $\mathcal{A}$ be a Büchi automaton for the complement of a specification.

We want to find out of $\text{Traces}(\mathcal{K}) \cap \mathcal{L}(\mathcal{A}) = \emptyset$.

Product construction between a Büchi automaton and a Kripke structure

Central Idea

We want to check if all *traces* of a Kripke structure satisfy a specification.

This is the same as saying as saying that there should not exist a trace that satisfies the *complement of a specification*.

Let $\mathcal{A}$ be a Büchi automaton for the complement of a specification.

We want to find out of $\text{Traces}(\mathcal{K}) \cap \mathcal{L}(\mathcal{A}) = \emptyset$.

We will do so in two steps:

- Build a *product* Büchi automaton $\mathcal{B}$ accepting $\text{Traces}(\mathcal{K}) \cap \mathcal{L}(\mathcal{A})$
- Check the product automaton for emptiness.

Product construction – formal definition

Definition

Let $\mathcal{K} = (S, \mathrm{AP}, S_0, T, L)$ be a Kripke structure and $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ be a Büchi automaton with $\Sigma = 2^{\mathrm{AP}}$.

Product construction – formal definition

Definition

Let $\mathcal{K} = (S, \mathrm{AP}, S_0, T, L)$ be a Kripke structure and $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ be a Büchi automaton with $\Sigma = 2^{\mathrm{AP}}$. We define the product Büchi automaton $\mathcal{B} = (Q', \Sigma, Q'_0, \delta', F')$ with:

- $Q' = S \times Q$

Product construction – formal definition

Definition

Let $\mathcal{K} = (S, \mathrm{AP}, S_0, T, L)$ be a Kripke structure and $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ be a Büchi automaton with $\Sigma = 2^{\mathrm{AP}}$. We define the product Büchi automaton $\mathcal{B} = (Q', \Sigma, Q'_0, \delta', F')$ with:

- $Q' = S \times Q$
- $Q'_0 = S_0 \times Q_0$

Product construction – formal definition

Definition

Let $\mathcal{K} = (S, \mathrm{AP}, S_0, T, L)$ be a Kripke structure and $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ be a Büchi automaton with $\Sigma = 2^{\mathrm{AP}}$. We define the product Büchi automaton $\mathcal{B} = (Q', \Sigma, Q_0', \delta', F')$ with:

- $Q' = S \times Q$
- $Q_0' = S_0 \times Q_0$
- $\delta' = \{((q, s), x, (q', s')) \in Q' \times \Sigma \times Q' \mid (q, x, q') \in \mathcal{A}, (s, s') \in T, L(s) = x\}$

Product construction – formal definition

Definition

Let $\mathcal{K} = (S, \text{AP}, S_0, T, L)$ be a Kripke structure and $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ be a Büchi automaton with $\Sigma = 2^{\text{AP}}$. We define the product Büchi automaton $\mathcal{B} = (Q', \Sigma, Q'_0, \delta', F')$ with:

- $Q' = S \times Q$
- $Q'_0 = S_0 \times Q_0$
- $\delta' = \{((q, s), x, (q', s')) \in Q' \times \Sigma \times Q' \mid (q, x, q') \in \mathcal{A}, (s, s') \in T, L(s) = x\}$
- $F = S \times F$

Product construction – formal definition

Definition

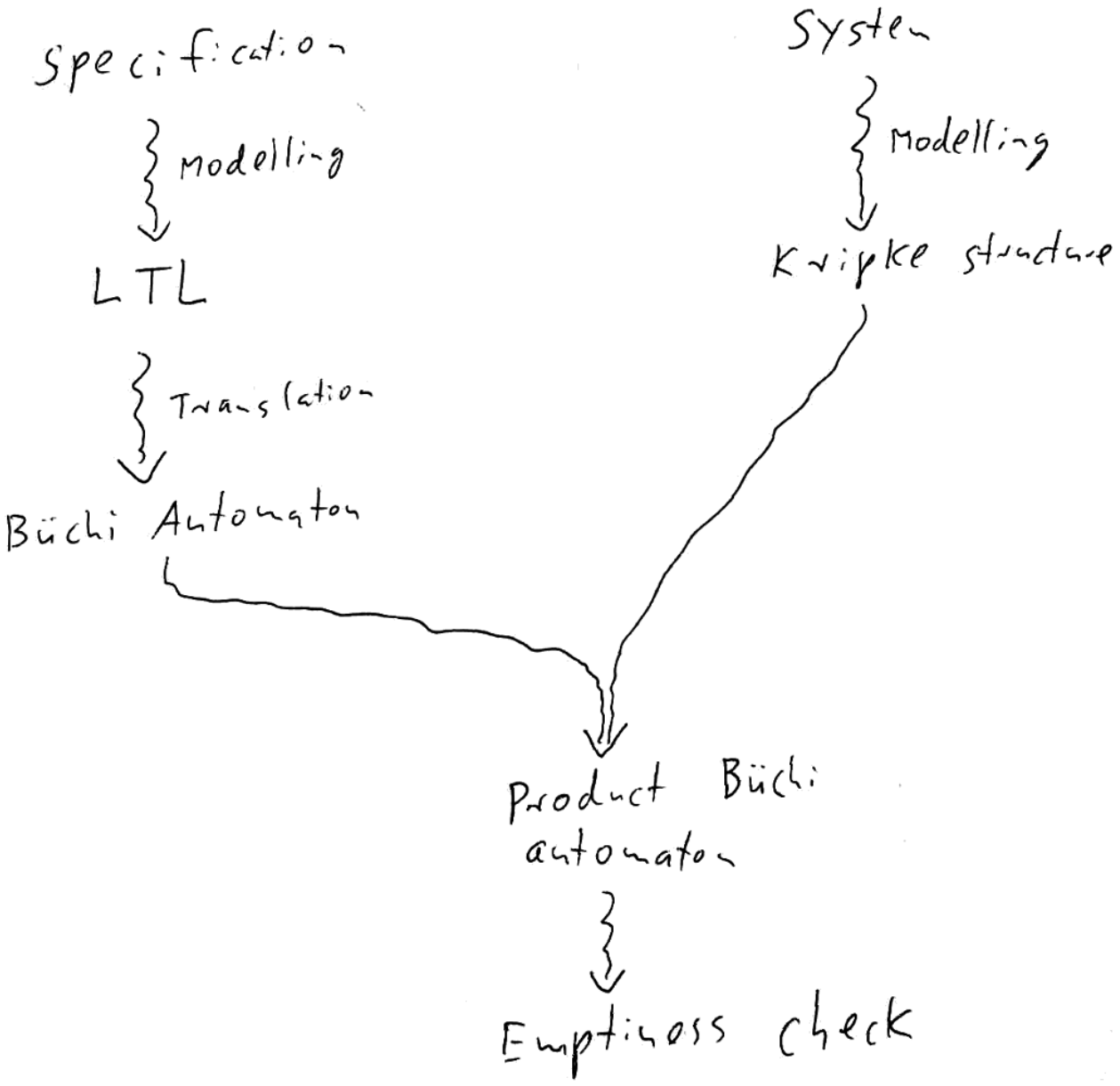
Let $\mathcal{K} = (S, \text{AP}, S_0, T, L)$ be a Kripke structure and $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ be a Büchi automaton with $\Sigma = 2^{\text{AP}}$. We define the product Büchi automaton $\mathcal{B} = (Q', \Sigma, Q_0', \delta', F')$ with:

- $Q' = S \times Q$
- $Q_0' = S_0 \times Q_0$
- $\delta' = \{((q, s), x, (q', s')) \in Q' \times \Sigma \times Q' \mid (q, x, q') \in \mathcal{A}, (s, s') \in T, L(s) = x\}$
- $F = S \times F$

Proposition (without proof here)

$\mathcal{B}$ accepts exactly those words that accepted by $\mathcal{A}$ **and** are traces of $\mathcal{K}$ at the same time.

Overview


Specification
modelling
LTL
Translation
Büchi Automaton
System
Modelling
Kripke structure
Product Büchi automaton
Emptiness check

Checking Büchi automata language emptiness

A classical question in automata theory

Can we check an automaton for *language emptiness*?

Checking Büchi automata language emptiness

A classical question in automata theory

Can we check an automaton for *language emptiness*?

Automata over finite words

We can test this by checking if there exists a reachable accepting state.

Checking Büchi automata language emptiness

A classical question in automata theory

Can we check an automaton for *language emptiness*?

Automata over finite words

We can test this by checking if there exists a reachable accepting state.

Automata over infinite words

We can test this by checking if there exists an accepting *lasso*.

A lasso consists of a *lasso handle* and a *lasso cycle*, where the lasso cycle contains at least one accepting state.

Checking automaton emptiness in practice

A simple algorithm

1. Use an algorithm such as Floyd–Warshall to find the distances between all pairs of states (q, q') in the graph.

Checking automaton emptiness in practice

A simple algorithm

1. Use an algorithm such as Floyd–Warshall to find the distances between all pairs of states (q, q') in the graph.
2. For all states $q_0 \in Q_0$ and $q \in F$:

Checking automaton emptiness in practice

A simple algorithm

1. Use an algorithm such as Floyd–Warshall to find the distances between all pairs of states (q, q') in the graph.
2. For all states $q_0 \in Q_0$ and $q \in F$:
   - Check if q_0 is reachable from q (i.e., their distance is not ∞)
   - Check if q is reachable from q (i.e., their distance is not ∞).

Checking automaton emptiness in practice

A simple algorithm

1. Use an algorithm such as Floyd–Warshall to find the distances between all pairs of states (q, q') in the graph.
2. For all states $q_0 \in Q_0$ and $q \in F$:
   - Check if q_0 is reachable from q (i.e., their distance is not ∞)
   - Check if q is reachable from q (i.e., their distance is not ∞).
   - Return **Non-empty** whenever both checks succeed.

Complexity

The Floyd-Warshall takes times **cubic** in $|Q|$.

Checking automaton emptiness in practice

A simple algorithm

1. Use an algorithm such as Floyd–Warshall to find the distances between all pairs of states (q, q') in the graph.
2. For all states $q_0 \in Q_0$ and $q \in F$:
   - Check if q_0 is reachable from q (i.e., their distance is not ∞)
   - Check if q is reachable from q (i.e., their distance is not ∞).
   - Return **Non-empty** whenever both checks succeed.

Complexity

The Floyd-Warshall takes times **cubic** in $|Q|$.

The Second step takes time **quadratic** in $|Q|$.

Checking automaton emptiness in practice

A simple algorithm

1. Use an algorithm such as Floyd–Warshall to find the distances between all pairs of states (q, q') in the graph.
2. For all states $q_0 \in Q_0$ and $q \in F$:
   - Check if q_0 is reachable from q (i.e., their distance is not ∞)
   - Check if q is reachable from q (i.e., their distance is not ∞).
   - Return **Non-empty** whenever both checks succeed.

Complexity

The Floyd-Warshall takes times **cubic** in $|Q|$.

The Second step takes time **quadratic** in $|Q|$.

$\rightarrow$ Cubic time overall (which is not good!)

The double-DFS algorithm

Aim

Since Kripke structures can be huge in practice, need an algorithm that is faster.

The double-DFS algorithm

Aim

Since Kripke structures can be huge in practice, need an algorithm that is faster.

The **Double-Depth-first-search** algorithm only takes time linear in $|\delta|$.

The double-DFS algorithm

Aim

Since Kripke structures can be huge in practice, need an algorithm that is faster.

The **Double-Depth-first-search** algorithm only takes time linear in $|\delta|$.

Basic idea

We combine two depth-first search procedures to find reachable cycles:

1. The first one for finding a path *to* an accepting state
2. The second one for finding a path *back* to the state

The double-DFS algorithm

Shared variables

```
stack = []
stackB = []
visited = set([])
marked = set([])
```

First DFS

```
def dfsA(node):
if node in visited:
  return
stack.push(node)
visited.add(node)
for all (v, v') ∈ E with v==node:
  dfsA(v')
if  v ∈ F:
  dfsB(v')
stack.pop()
```

Second DFS

```
def dfsB(node):
if node in marked:
  return
stackB.push(node)
marked.add(node)
for all (v,v') in E with v==node:
  if v' in stack:
    lasso found
  dfsB(v):
stackB.pop()
```

Calling dfsa

```
for all v ∈ Q0:
  dfsA(v)
```

Why is the double-DFS algorithm correct? (1/2)

Reasoning (1)

The case that if the algorithm returns `lasso found`, a lasso has been found is relatively clear:

Why is the double-DFS algorithm correct? (1/2)

Reasoning (1)

The case that if the algorithm returns `lasso found`, a lasso has been found is relatively clear:

`dfsB` is only called from an accepting state s, and if it finds a path back to a state s' in `stack`, then the lasso cycle consists of $s \to s' \to s$.

Why is the double-DFS algorithm correct? (1/2)

Reasoning (1)

The case that if the algorithm returns `lasso found`, a lasso has been found is relatively clear:

`dfsB` is only called from an accepting state s, and if it finds a path back to a state s' in `stack`, then the lasso cycle consists of $s \rightarrow s' \rightarrow s$.

The first DFS Procedure makes sure that only reachable lasso cycles are found.

Why is the double-DFS algorithm correct? (2/2)

Reasoning (2)

Now let us assume that the algorithm did not find a lasso.

Why is the double-DFS algorithm correct? (2/2)

Reasoning (2)

Now let us assume that the algorithm did not find a lasso.

We want to show that there is indeed no lasso then.

Why is the double-DFS algorithm correct? (2/2)

Reasoning (2)

Now let us assume that the algorithm did not find a lasso.

We want to show that there is indeed no lasso then.

The only way in which a lasso could be missed is by an accepting state v getting added to the `marked` states without a lasso containing it and reachable from v being found.

Why is the double-DFS algorithm correct? (2/2)

Reasoning (2)

Now let us assume that the algorithm did not find a lasso.

We want to show that there is indeed no lasso then.

The only way in which a lasso could be missed is by an accepting state v getting added to the `marked` states without a lasso containing it and reachable from v being found.

Since the `dfsA` function only calls `dfsB` after exploring the successors, the only way for this to happen is if for some other accepting state v', `dfsB` was called first, and there is a path from v' to v but not from v back to v' (as otherwise that could consistitute an accepting cycle).

Why is the double-DFS algorithm correct? (2/2)

Reasoning (2)

Now let us assume that the algorithm did not find a lasso.

We want to show that there is indeed no lasso then.

The only way in which a lasso could be missed is by an accepting state v getting added to the `marked` states without a lasso containing it and reachable from v being found.

Since the `dfsA` function only calls `dfsB` after exploring the successors, the only way for this to happen is if for some other accepting state v', `dfsB` was called first, and there is a path from v' to v but not from v back to v' (as otherwise that could consistitute an accepting cycle).

But now this contradicts the assumption above that v is reachable from v'.

Why is the double-DFS algorithm correct? (2/2)

Reasoning (2)

Now let us assume that the algorithm did not find a lasso.

We want to show that there is indeed no lasso then.

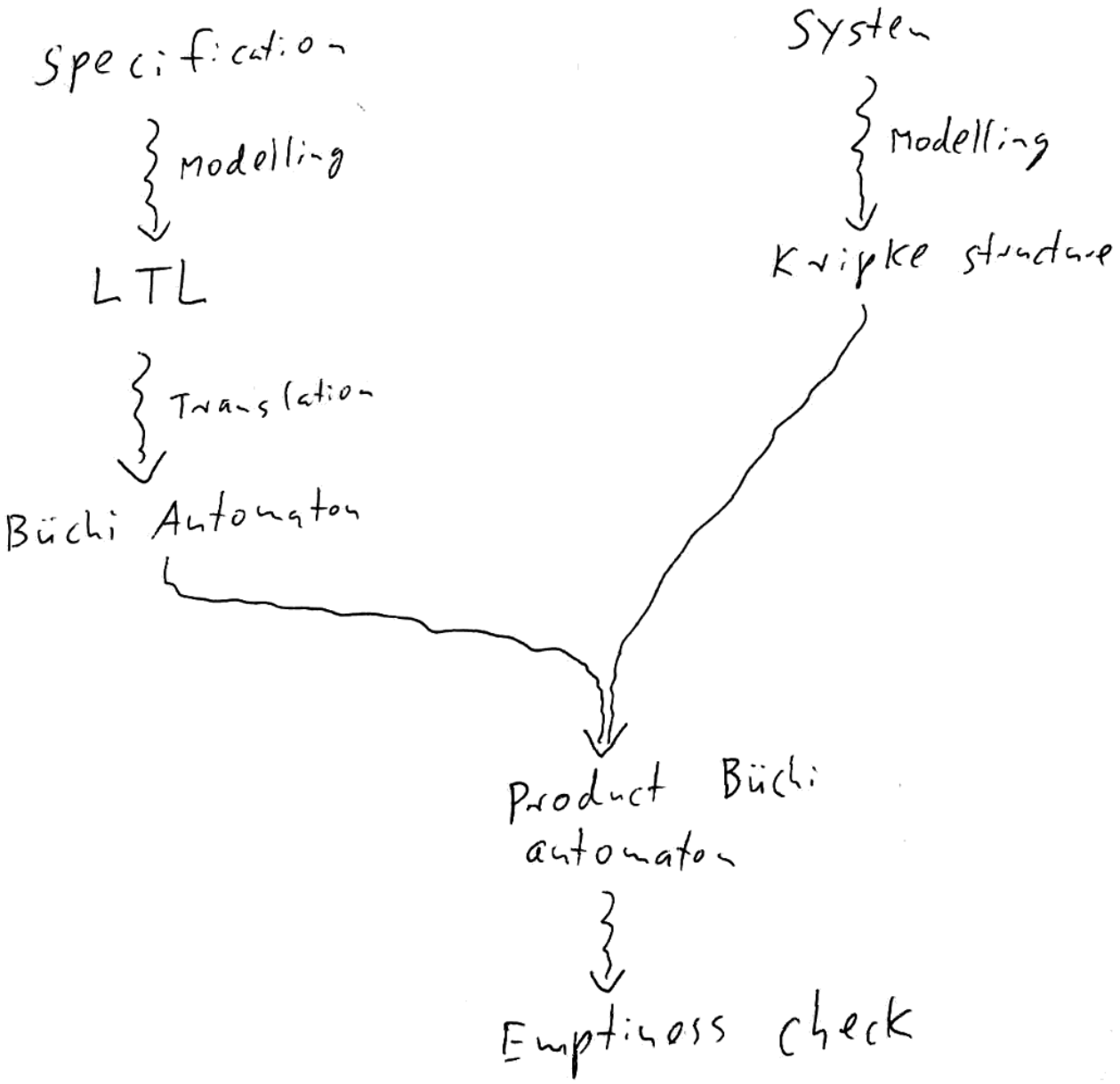
The only way in which a lasso could be missed is by an accepting state v getting added to the `marked` states without a lasso containing it and reachable from v being found.

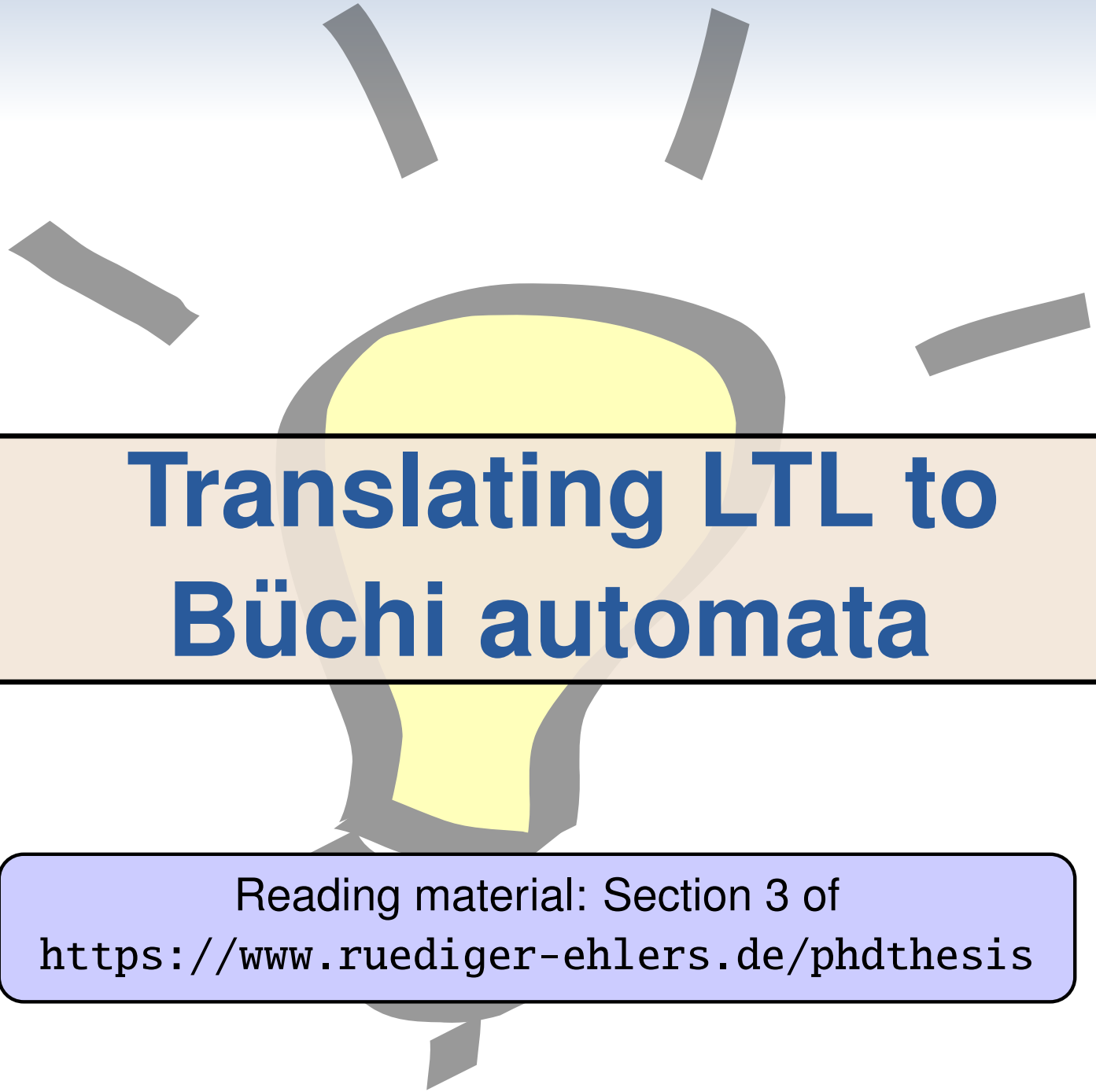
Since the `dfsA` function only calls `dfsB` after exploring the successors, the only way for this to happen is if for some other accepting state v', `dfsB` was called first, and there is a path from v' to v but not from v back to v' (as otherwise that could consistitute an accepting cycle).

But now this contradicts the assumption above that v is reachable from v'.

Hence, the assumption that the algorithm did not find a lasso cannot hold.

Overview


Specification
modelling
LTL
Translation
Büchi Automaton
System
Modelling
Kripke structure
Product Büchi automaton
Emptiness check



Translating LTL to Büchi automata

Reading material: Section 3 of
https://www.ruediger-ehlers.de/phdthesis

Translating from LTL to Büchi automata

Aim of the translation

Given an LTL formula, we want to translate an LTL formula to a Büchi automaton such that the language of the Büchi automaton is exactly the set of words satisfying the LTL formula.

Translating from LTL to Büchi automata

Aim of the translation

Given an LTL formula, we want to translate an LTL formula to a Büchi automaton such that the language of the Büchi automaton is exactly the set of words satisfying the LTL formula.

Note that this is the last missing piece for a full model checking workflow!

Translating from LTL to Büchi automata

Aim of the translation

Given an LTL formula, we want to translate an LTL formula to a Büchi automaton such that the language of the Büchi automaton is exactly the set of words satisfying the LTL formula.

Note that this is the last missing piece for a full model checking workflow!

Approach presented in the following

We do a multi-step translation:

LTL

$\Downarrow$

LTL in negation normal form

$\Downarrow$

Alternating Büchi automaton

$\Downarrow$

(Non-deterministic) Büchi automaton

LTL: Negation normal form

Definition

An LTL formula is in negation normal form (NNF) if all occurrences of the negation operator are in front of atomic propositions.

In NNF or not?

- $\mathbf{GF}(p)$

LTL: Negation normal form

Definition

An LTL formula is in negation normal form (NNF) if all occurrences of the negation operator are in front of atomic propositions.

In NNF or not?

- $\mathbf{GF}(p)$ - ✓
- $(p \,\mathcal{U}\, \neg q)$

LTL: Negation normal form

Definition

An LTL formula is in negation normal form (NNF) if all occurrences of the negation operator are in front of atomic propositions.

In NNF or not?

- $\mathbf{GF}(p)$ - ✓
- $(p \,\mathcal{U}\, \neg q)$ - ✓
- $\mathbf{F}\neg\mathbf{F}p$

LTL: Negation normal form

Definition

An LTL formula is in negation normal form (NNF) if all occurrences of the negation operator are in front of atomic propositions.

In NNF or not?

- $\mathbf{GF}(p)$ - ✓
- $(p \,\mathcal{U}\, \neg q)$ - ✓
- $\mathbf{F}\neg\mathbf{F}p$ - ×

LTL: Translation to NNF

Idea

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$

LTL: Translation to NNF

Idea

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$

LTL: Translation to NNF

Idea

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F}\psi \equiv \mathbf{G}\neg\psi$
- $\neg \mathbf{G}\psi \equiv \mathbf{F}\neg\psi$
- $\neg \mathbf{X}\psi \equiv \mathbf{X}\neg\psi$

LTL: Translation to NNF

Idea

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F}\psi \equiv \mathbf{G}\neg\psi$
- $\neg \mathbf{G}\psi \equiv \mathbf{F}\neg\psi$
- $\neg \mathbf{X}\psi \equiv \mathbf{X}\neg\psi$
- $\neg(\psi \mathcal{U} \psi') \equiv ?$

LTL: Translation to NNF

Idea

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$
- $\neg \mathbf{X} \psi \equiv \mathbf{X} \neg \psi$
- $\neg (\psi \mathcal{U} \psi') \equiv ?$

New LTL operator

We define the *release operator* as the complement of the *until operator*.

We say that for some word $w = w_0 w_1 \ldots \in (2^{\mathrm{AP}})^\omega$ and some subformulas ψ and ψ', we have $w \models \psi \, \mathbf{R} \, \psi'$ if and only if

LTL: Translation to NNF

Idea

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F}\psi \equiv \mathbf{G}\neg\psi$
- $\neg \mathbf{G}\psi \equiv \mathbf{F}\neg\psi$
- $\neg \mathbf{X}\psi \equiv \mathbf{X}\neg\psi$
- $\neg(\psi \mathcal{U} \psi') \equiv ?$

New LTL operator

We define the *release operator* as the complement of the *until operator*.

We say that for some word $w = w_0 w_1 \ldots \in (2^{\mathrm{AP}})^\omega$ and some subformulas ψ and ψ', we have $w \models \psi \,\mathbf{R}\, \psi'$ if and only if there exists some $j \in \mathbb{N}$ such that $w^j \models \psi$ and for all $0 \leq k \leq j$, we have $w^k \models \psi'$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$
- $\neg \mathbf{X} \psi \equiv \mathbf{X} \neg \psi$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$
- $\neg \mathbf{X} \psi \equiv \mathbf{X} \neg \psi$
- $\neg (\psi \mathcal{U} \psi') \equiv \neg \psi \mathbf{R} \neg \psi'$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$
- $\neg \mathbf{X} \psi \equiv \mathbf{X} \neg \psi$
- $\neg (\psi \mathcal{U} \psi') \equiv \neg \psi \mathbf{R} \neg \psi'$
- $\neg (\psi \mathbf{R} \psi') \equiv \neg \psi \mathcal{U} \neg \psi'$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F}\psi \equiv \mathbf{G}\neg\psi$
- $\neg \mathbf{G}\psi \equiv \mathbf{F}\neg\psi$
- $\neg \mathbf{X}\psi \equiv \mathbf{X}\neg\psi$
- $\neg(\psi \mathcal{U} \psi') \equiv \neg\psi \mathbf{R} \neg\psi'$
- $\neg(\psi \mathbf{R} \psi') \equiv \neg\psi \mathcal{U} \neg\psi'$
- $(\neg\neg\psi \equiv \psi)$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F}\psi \equiv \mathbf{G}\neg\psi$
- $\neg \mathbf{G}\psi \equiv \mathbf{F}\neg\psi$
- $\neg \mathbf{X}\psi \equiv \mathbf{X}\neg\psi$
- $\neg(\psi \mathcal{U} \psi') \equiv \neg\psi \mathbf{R} \neg\psi'$
- $\neg(\psi \mathbf{R} \psi') \equiv \neg\psi \mathcal{U} \neg\psi'$
- $(\neg\neg\psi \equiv \psi)$

Pushing the negation inwards

We can push every negation in front of the atomic propositions with these rules.

Example:

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F}\psi \equiv \mathbf{G}\neg\psi$
- $\neg \mathbf{G}\psi \equiv \mathbf{F}\neg\psi$
- $\neg \mathbf{X}\psi \equiv \mathbf{X}\neg\psi$
- $\neg(\psi \mathcal{U} \psi') \equiv \neg\psi \mathbf{R} \neg\psi'$
- $\neg(\psi \mathbf{R} \psi') \equiv \neg\psi \mathcal{U} \neg\psi'$
- $(\neg\neg\psi \equiv \psi)$

Pushing the negation inwards

We can push every negation in front of the atomic propositions with these rules.

Example:

$\neg \mathbf{G}(x \wedge (y \,\mathcal{U}\, z) \vee \neg z)$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F}\psi \equiv \mathbf{G}\neg\psi$
- $\neg \mathbf{G}\psi \equiv \mathbf{F}\neg\psi$
- $\neg \mathbf{X}\psi \equiv \mathbf{X}\neg\psi$
- $\neg(\psi \mathcal{U} \psi') \equiv \neg\psi \mathbf{R} \neg\psi'$
- $\neg(\psi \mathbf{R} \psi') \equiv \neg\psi \mathcal{U} \neg\psi'$
- $(\neg\neg\psi \equiv \psi)$

Pushing the negation inwards

We can push every negation in front of the atomic propositions with these rules.

Example:

$\neg \mathbf{G}(x \wedge (y \,\mathcal{U}\, z) \vee \neg z)$
$\equiv \mathbf{F}(\neg(x \wedge (y \,\mathcal{U}\, z) \vee \neg z))$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F}\psi \equiv \mathbf{G}\neg\psi$
- $\neg \mathbf{G}\psi \equiv \mathbf{F}\neg\psi$
- $\neg \mathbf{X}\psi \equiv \mathbf{X}\neg\psi$
- $\neg(\psi \mathcal{U} \psi') \equiv \neg\psi \mathbf{R} \neg\psi'$
- $\neg(\psi \mathbf{R} \psi') \equiv \neg\psi \mathcal{U} \neg\psi'$
- $(\neg\neg\psi \equiv \psi)$

Pushing the negation inwards

We can push every negation in front of the atomic propositions with these rules.

Example:

$\neg \mathbf{G}(x \wedge (y \,\mathcal{U}\, z) \vee \neg z)$
$\equiv \mathbf{F}(\neg(x \wedge (y \,\mathcal{U}\, z) \vee \neg z))$
$\equiv \mathbf{F}(\neg(x \wedge (y \,\mathcal{U}\, z)) \wedge \neg\neg z)$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$
- $\neg \mathbf{X} \psi \equiv \mathbf{X} \neg \psi$
- $\neg (\psi \mathcal{U} \psi') \equiv \neg \psi \mathbf{R} \neg \psi'$
- $\neg (\psi \mathbf{R} \psi') \equiv \neg \psi \mathcal{U} \neg \psi'$
- $(\neg \neg \psi \equiv \psi)$

Pushing the negation inwards

We can push every negation in front of the atomic propositions with these rules.

Example:

$\neg \mathbf{G}(x \wedge (y \, \mathcal{U} \, z) \vee \neg z)$
$\equiv \mathbf{F}(\neg(x \wedge (y \, \mathcal{U} \, z) \vee \neg z))$
$\equiv \mathbf{F}(\neg(x \wedge (y \, \mathcal{U} \, z)) \wedge \neg \neg z)$
$\equiv \mathbf{F}((\neg x \vee \neg(y \, \mathcal{U} \, z)) \wedge z)$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$
- $\neg \mathbf{X} \psi \equiv \mathbf{X} \neg \psi$
- $\neg (\psi \mathcal{U} \psi') \equiv \neg \psi \mathbf{R} \neg \psi'$
- $\neg (\psi \mathbf{R} \psi') \equiv \neg \psi \mathcal{U} \neg \psi'$
- $(\neg \neg \psi \equiv \psi)$

Pushing the negation inwards

We can push every negation in front of the atomic propositions with these rules.

Example:

$\neg \mathbf{G}(x \wedge (y \,\mathcal{U}\, z) \vee \neg z)$
$\equiv \mathbf{F}(\neg(x \wedge (y \,\mathcal{U}\, z) \vee \neg z))$
$\equiv \mathbf{F}(\neg(x \wedge (y \,\mathcal{U}\, z)) \wedge \neg\neg z)$
$\equiv \mathbf{F}((\neg x \vee \neg(y \,\mathcal{U}\, z)) \wedge z)$
$\equiv \mathbf{F}((\neg x \vee \neg y \,\mathbf{R}\, \neg z) \wedge z)$

LTL: Translation to NNF

Equivalences

We utilize some equivalences (for all LTL subformulas ψ):

- $\neg \mathbf{F} \psi \equiv \mathbf{G} \neg \psi$
- $\neg \mathbf{G} \psi \equiv \mathbf{F} \neg \psi$
- $\neg \mathbf{X} \psi \equiv \mathbf{X} \neg \psi$
- $\neg (\psi \mathcal{U} \psi') \equiv \neg \psi \mathbf{R} \neg \psi'$
- $\neg (\psi \mathbf{R} \psi') \equiv \neg \psi \mathcal{U} \neg \psi'$
- $(\neg \neg \psi \equiv \psi)$

Pushing the negation inwards

We can push every negation in front of the atomic propositions with these rules.

Example:

$\neg \mathbf{G}(x \wedge (y \,\mathcal{U}\, z) \vee \neg z)$
$\equiv \mathbf{F}(\neg(x \wedge (y \,\mathcal{U}\, z) \vee \neg z))$
$\equiv \mathbf{F}(\neg(x \wedge (y \,\mathcal{U}\, z)) \wedge \neg\neg z)$
$\equiv \mathbf{F}((\neg x \vee \neg(y \,\mathcal{U}\, z)) \wedge z)$
$\equiv \mathbf{F}((\neg x \vee \neg y \,\mathbf{R}\, \neg z) \wedge z)$

Translation procedure to NNF

$$
\mathit{toNNF}(\psi) = \begin{cases}
\psi & \text{if } \psi \in \{\mathbf{true}, \mathbf{false}\} \\
\psi & \text{if } \psi \in \{\neg p \mid p \in \mathrm{AP}\} \cup \mathrm{AP} \\
\mathit{toNNF}(\psi') \vee \mathit{toNNF}(\psi'') & \text{if } \psi = \psi' \vee \psi'' \\
\mathit{toNNF}(\psi') \wedge \mathit{toNNF}(\psi'') & \text{if } \psi = \psi' \wedge \psi'' \\
\mathbf{G}(\mathit{toNNF}(\psi')) & \text{if } \psi = \mathbf{G}\psi' \\
\mathbf{F}(\mathit{toNNF}(\psi')) & \text{if } \psi = \mathbf{F}\psi' \\
\mathbf{X}(\mathit{toNNF}(\psi')) & \text{if } \psi = \mathbf{X}\psi' \\
\mathit{toNNF}(\psi') \,\mathcal{U}\, \mathit{toNNF}(\psi'') & \text{if } \psi = \psi' \,\mathcal{U}\, \psi'' \\
\mathit{toNNF}(\psi') \,\mathbf{R}\, \mathit{toNNF}(\psi'') & \text{if } \psi = \psi' \,\mathbf{R}\, \psi'' \\
\mathit{toNNF}(\neg\psi') \wedge \mathit{toNNF}(\neg\psi'') & \text{if } \psi = \neg(\psi' \vee \psi'') \\
\mathit{toNNF}(\neg\psi') \vee \mathit{toNNF}(\neg\psi'') & \text{if } \psi = \neg(\psi' \wedge \psi'') \\
\mathbf{F}(\mathit{toNNF}(\neg\psi')) & \text{if } \psi = \neg\mathbf{G}\psi' \\
\mathbf{G}(\mathit{toNNF}(\neg\psi')) & \text{if } \psi = \neg\mathbf{F}\psi' \\
\mathbf{X}(\mathit{toNNF}(\neg\psi')) & \text{if } \psi = \neg\mathbf{X}\psi' \\
\mathit{toNNF}(\neg\psi') \,\mathbf{R}\, \mathit{toNNF}(\neg\psi'') & \text{if } \psi = \psi' \,\mathcal{U}\, \psi'' \\
\mathit{toNNF}(\neg\psi') \,\mathcal{U}\, \mathit{toNNF}(\neg\psi'') & \text{if } \psi = \psi' \,\mathbf{R}\, \psi'' \\
\mathit{toNNF}(\psi) & \text{if } \psi = \neg\neg\psi'
\end{cases}
$$

Checkpoint!

Steps of the translation

LTL

$\Downarrow$ ✓

LTL in Negation normal form

$\Downarrow$

Alternating Büchi automaton

$\Downarrow$

(Non-deterministic) Büchi automaton

Alternating Büchi automata

Idea

We want to translate from an LTL formula into a type of automaton that is powerful enough for a very simple translation.

Alternating Büchi automata

Idea

We want to translate from an LTL formula into a type of automaton that is powerful enough for a very simple translation. Only in the second part of the construction (translating and Alternating Büchi automaton to a Non-deterministic Büchi automaton) we will get rid of this additional power.

Alternating Büchi automata

Idea

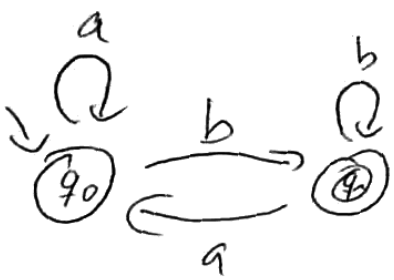
We want to translate from an LTL formula into a type of automaton that is powerful enough for a very simple translation. Only in the second part of the construction (translating and Alternating Büchi automaton to a Non-deterministic Büchi automaton) we will get rid of this additional power.

Implementing this idea

We will extend the *branching capabilities* of the automaton.

Branching modes

Det. Aut



One word

a b a b b a ...

One run

$q_0\ q_1 \ldots$

$w \in \mathcal{L}(A)$ if the run accepts

Non-det. Aut.

One word

a b a b b a ...

Many runs

$q_0 q_0 q_1 q_0 \ldots$

$q_0 q_0 q_0 q_0 \ldots$

$w \in \mathcal{L}(A)$ if <u>one</u> run accepts

Universal aut.

One word

a b a b b a ...

Many runs

$q_0\ q_0 q_0 q_0 \ldots$

$q_0\ q_1$

$q_0\ q_0 q_0 q_1 \ldots$

$w \in \mathcal{L}(A)$ if <u>all</u> infinite runs accept.

Representing branching modes in transition functions

Idea

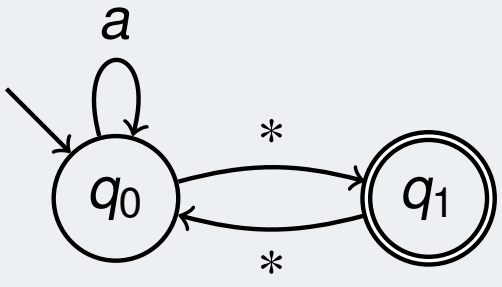
An alternative representation of the transition relation of a non-deterministic Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with $\delta \subseteq Q \times \Sigma \times Q$ is that of a Boolean transition function.

Representing branching modes in transition functions

Idea

An alternative representation of the transition relation of a non-deterministic Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with $\delta \subseteq Q \times \Sigma \times Q$ is that of a Boolean transition function.

For instance, consider the following automaton:



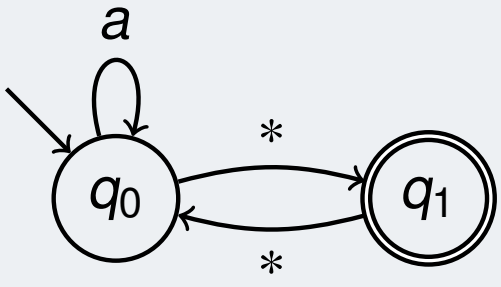
We can either write $\delta = \{(q_0, a, q_0), (q_0, a, q_1), (q_0, b, q_1), (q_1, a, q_0), (q_1, b, q_0)\}$

Representing branching modes in transition functions

Idea

An alternative representation of the transition relation of a non-deterministic Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with $\delta \subseteq Q \times \Sigma \times Q$ is that of a Boolean transition function.

For instance, consider the following automaton:



We can either write $\delta = \{(q_0, a, q_0), (q_0, a, q_1), (q_0, b, q_1), (q_1, a, q_0), (q_1, b, q_0)\}$

or

we define $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$, where $\mathcal{B}^+(Q)$ is the set of positive Boolean formulas over Q and define $\delta(q_0, a) = q_0 \vee q_1$, $\delta(q_0, b) = q_1$, $\delta(q_1, a) = q_0$, $\delta(q_1, b) = q_1$.

Why positive Boolean functions?

Idea

Using positive Boolean formulas, we can represent both *non-deterministic and universal branching*.

Why positive Boolean functions?

Idea

Using positive Boolean formulas, we can represent both *non-deterministic and universal branching*.
So a transition function element of the form

$$\delta(q, a) = q_1 \vee q_2$$

would represent non-deterministic branching, as the run needs to accept for *some* possible successor to witness an accepted word,

Why positive Boolean functions?

Idea

Using positive Boolean formulas, we can represent both *non-deterministic and universal branching*.
So a transition function element of the form

$$\delta(q, a) = q_1 \vee q_2$$

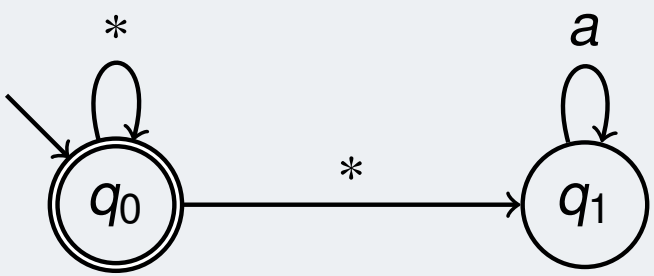
would represent non-deterministic branching, as the run needs to accept for *some* possible successor to witness an accepted word, and

$$\delta(q, a) = q_1 \wedge q_2$$

represents that the run needs for accept *all* possible successors to witness an accepted word.

Example

(Universal) automaton for $\Sigma = \{a, b\}$

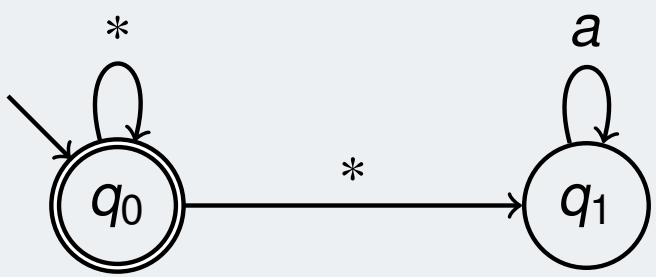


Alternative representation of the translation function/relation

$\delta(q_0, a) = q_0 \wedge q_1$
$\delta(q_0, b) = q_0 \wedge q_1$
$\delta(q_1, a) = q_1$ $\delta(q_1, b) =$ **true**

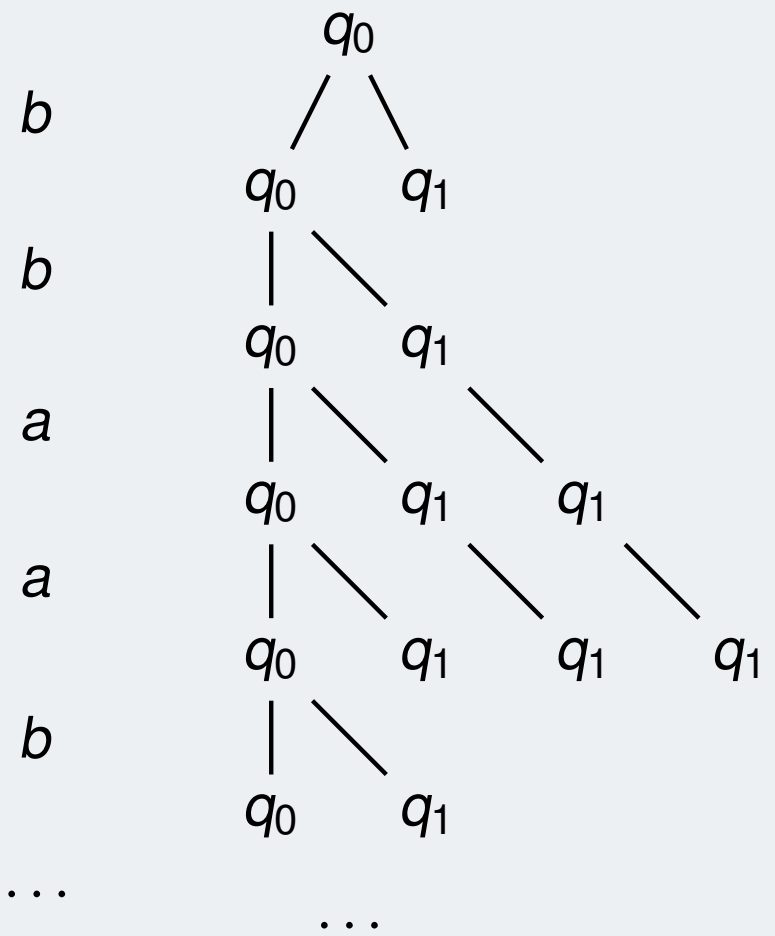
A closer look at universal branching

(Universal) automaton for $\Sigma = \{a, b\}$



A word and a **run tree**

$w =$



Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

- The finite set of states Q,

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

- The finite set of states Q,
- the alphabet Σ,

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

- The finite set of states Q,
- the alphabet Σ,
- the transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$,

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

- The finite set of states Q,
- the alphabet Σ,
- the transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$,
- the set of initial states Q_0, and

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

- The finite set of states Q,
- the alphabet Σ,
- the transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$,
- the set of initial states Q_0, and
- the set of accepting states F.

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

- The finite set of states Q,
- the alphabet Σ,
- the transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$,
- the set of initial states Q_0, and
- the set of accepting states F.

An alternating Büchi automaton accepts or rejects words $w = w_0 w_1 \ldots \in \Sigma^\omega$.

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

- The finite set of states Q,
- the alphabet Σ,
- the transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$,
- the set of initial states Q_0, and
- the set of accepting states F.

An alternating Büchi automaton accepts or rejects words $w = w_0 w_1 \ldots \in \Sigma^\omega$.

The set of accepted words is called its language.

Alternating automata

Main idea

Alternating automata combine universal and non-deterministic branching in one automaton type. Hence they are quite powerful!

Definition

An alternating Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with:

- The finite set of states Q,
- the alphabet Σ,
- the transition function $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$,
- the set of initial states Q_0, and
- the set of accepting states F.

An alternating Büchi automaton accepts or rejects words $w = w_0 w_1 \ldots \in \Sigma^\omega$.

The set of accepted words is called its language.

An alternating Büchi tree automaton accepts all words that have accepting run trees.

A run tree is accepting if all of its infinite branches visit accepting states infinitely often.

Run trees

Given an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, a tree $\langle T, \tau \rangle$ is a **run tree** if it satisfies the following conditions:

Run trees

Given an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, a tree $\langle T, \tau \rangle$ is a **run tree** if it satisfies the following conditions:

- $T \subseteq \mathbb{N}^*$, i.e., $\mathbb{N}$ is the set of directions of $\langle T, \tau \rangle$

Run trees

Given an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, a tree $\langle T, \tau \rangle$ is a **run tree** if it satisfies the following conditions:

- $T \subseteq \mathbb{N}^*$, i.e., $\mathbb{N}$ is the set of directions of $\langle T, \tau \rangle$
- $\tau : T \to Q$

Run trees

Given an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, a tree $\langle T, \tau \rangle$ is a **run tree** if it satisfies the following conditions:

- $T \subseteq \mathbb{N}^*$, i.e., $\mathbb{N}$ is the set of directions of $\langle T, \tau \rangle$
- $\tau : T \to Q$
- $\tau(\epsilon) \in Q_0$

Run trees

Given an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, a tree $\langle T, \tau \rangle$ is a **run tree** if it satisfies the following conditions:

- $T \subseteq \mathbb{N}^*$, i.e., $\mathbb{N}$ is the set of directions of $\langle T, \tau \rangle$
- $\tau : T \to Q$
- $\tau(\epsilon) \in Q_0$
- For every $t \in T$ with $j = |t|$:

Run trees

Given an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, a tree $\langle T, \tau \rangle$ is a **run tree** if it satisfies the following conditions:

- $T \subseteq \mathbb{N}^*$, i.e., $\mathbb{N}$ is the set of directions of $\langle T, \tau \rangle$
- $\tau : T \to Q$
- $\tau(\epsilon) \in Q_0$
- For every $t \in T$ with $j = |t|$:
  - Let $succ(t) = \{q \in Q \mid \exists d \in \mathbb{N}. td \in T, \tau(td) = q\}$.

Run trees

Given an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, a tree $\langle T, \tau \rangle$ is a **run tree** if it satisfies the following conditions:

- $T \subseteq \mathbb{N}^*$, i.e., $\mathbb{N}$ is the set of directions of $\langle T, \tau \rangle$
- $\tau : T \to Q$
- $\tau(\epsilon) \in Q_0$
- For every $t \in T$ with $j = |t|$:
  - Let $succ(t) = \{q \in Q \mid \exists d \in \mathbb{N}. td \in T, \tau(td) = q\}$.
  - We have that $succ(t) \models \delta(\tau(t), w_j)$

Use of alternating automata for LTL

Idea

With the branching capabilities of alternating automata, we can easily translate an LTL formula in *negation normal form* to LTL.

Use of alternating automata for LTL

Idea

With the branching capabilities of alternating automata, we can easily translate an LTL formula in *negation normal form* to LTL.

We will use one state per subformula in the LTL formula.

Use of alternating automata for LTL

Idea

With the branching capabilities of alternating automata, we can easily translate an LTL formula in *negation normal form* to LTL.

We will use one state per subformula in the LTL formula.

Example

When translating the LTL formula $a \vee \mathbf{X}(a \,\mathcal{U}\, b)$ we will have states for the subformulas:

- a
- b
- $a \,\mathcal{U}\, b$
- $\mathbf{X}(a \,\mathcal{U}\, b)$
- $a \vee \mathbf{X}(a \,\mathcal{U}\, b)$

How to build the automaton

Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{AP}$ as follows.

How to build the automaton

Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$

How to build the automaton

Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$
- $\delta(\neg p, x) = \mathbf{ff}$ if $p \in x$, $\delta(\neg p, x) = \mathbf{tt}$

How to build the automaton

Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$
- $\delta(\neg p, x) = \mathbf{ff}$ if $p \in x$, $\delta(\neg p, x) = \mathbf{tt}$
- $\delta(\mathbf{X}\psi', x) = \psi'$

How to build the automaton

Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$
- $\delta(\neg p, x) = \mathbf{ff}$ if $p \in x$, $\delta(\neg p, x) = \mathbf{tt}$
- $\delta(\mathbf{X}\psi', x) = \psi'$
- $\delta(\mathbf{G}\psi', x) = \delta(\psi', x) \wedge \mathbf{G}\,\psi'$

How to build the automaton

Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$
- $\delta(\neg p, x) = \mathbf{ff}$ if $p \in x$, $\delta(\neg p, x) = \mathbf{tt}$
- $\delta(\mathbf{X}\psi', x) = \psi'$
- $\delta(\mathbf{G}\psi', x) = \delta(\psi', x) \wedge \mathbf{G}\,\psi'$
- $\delta(\mathbf{F}\psi', x) = \delta(\psi', x) \vee \mathbf{F}\,\psi'$

How to build the automaton

Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$
- $\delta(\neg p, x) = \mathbf{ff}$ if $p \in x$, $\delta(\neg p, x) = \mathbf{tt}$
- $\delta(\mathbf{X}\psi', x) = \psi'$
- $\delta(\mathbf{G}\psi', x) = \delta(\psi', x) \wedge \mathbf{G}\,\psi'$
- $\delta(\mathbf{F}\psi', x) = \delta(\psi', x) \vee \mathbf{F}\,\psi'$
- $\delta(\psi' \,\mathcal{U}\, \psi'', x) = \delta(\psi', x) \wedge (\psi' \,\mathcal{U}\, \psi'') \vee \delta(\psi', x)$

How to build the automaton

Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$
- $\delta(\neg p, x) = \mathbf{ff}$ if $p \in x$, $\delta(\neg p, x) = \mathbf{tt}$
- $\delta(\mathbf{X}\psi', x) = \psi'$
- $\delta(\mathbf{G}\psi', x) = \delta(\psi', x) \wedge \mathbf{G}\,\psi'$
- $\delta(\mathbf{F}\psi', x) = \delta(\psi', x) \vee \mathbf{F}\,\psi'$
- $\delta(\psi' \,\mathcal{U}\, \psi'', x) = \delta(\psi', x) \wedge (\psi' \,\mathcal{U}\, \psi'') \vee \delta(\psi', x)$
- $\delta(\psi' \,\mathbf{R}\, \psi'', x) = (\delta(\psi', x) \vee (\psi' \,\mathbf{R}\, \psi'')) \wedge \delta(\psi'', x)$

How to build the automaton

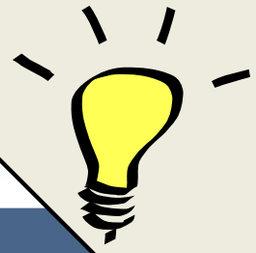
Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$
- $\delta(\neg p, x) = \mathbf{ff}$ if $p \in x$, $\delta(\neg p, x) = \mathbf{tt}$
- $\delta(\mathbf{X}\psi', x) = \psi'$
- $\delta(\mathbf{G}\psi', x) = \delta(\psi', x) \wedge \mathbf{G}\,\psi'$
- $\delta(\mathbf{F}\psi', x) = \delta(\psi', x) \vee \mathbf{F}\,\psi'$
- $\delta(\psi' \,\mathcal{U}\, \psi'', x) = \delta(\psi', x) \wedge (\psi' \,\mathcal{U}\, \psi'') \vee \delta(\psi', x)$
- $\delta(\psi' \,\mathbf{R}\, \psi'', x) = (\delta(\psi', x) \vee (\psi' \,\mathbf{R}\, \psi'')) \wedge \delta(\psi'', x)$
- $\delta(\psi' \vee \psi'', x) = \delta(\psi', x) \vee \delta(\psi'', x)$

How to build the automaton



Automaton construction

Given an LTL formula ψ over some set of atomic propositions AP, we build an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with Q being the subformulas of ψ and $\Sigma = 2^{\mathrm{AP}}$ as follows.

Let $p \in \mathrm{AP}$ be an arbitrary atomic proposition, $x \in \Sigma$, and ψ' and ψ'' be subformulas. We have:

- $\delta(p, x) = \mathbf{tt}$ if $p \in x$, $\delta(p, x) = \mathbf{ff}$
- $\delta(\neg p, x) = \mathbf{ff}$ if $p \in x$, $\delta(\neg p, x) = \mathbf{tt}$
- $\delta(\mathbf{X}\psi', x) = \psi'$
- $\delta(\mathbf{G}\psi', x) = \delta(\psi', x) \wedge \mathbf{G}\,\psi'$
- $\delta(\mathbf{F}\psi', x) = \delta(\psi', x) \vee \mathbf{F}\,\psi'$
- $\delta(\psi' \,\mathcal{U}\, \psi'', x) = \delta(\psi', x) \wedge (\psi' \,\mathcal{U}\, \psi'') \vee \delta(\psi', x)$
- $\delta(\psi' \,\mathbf{R}\, \psi'', x) = (\delta(\psi', x) \vee (\psi' \,\mathbf{R}\, \psi'')) \wedge \delta(\psi'', x)$
- $\delta(\psi' \vee \psi'', x) = \delta(\psi', x) \vee \delta(\psi'', x)$
- $\delta(\psi' \wedge \psi'', x) = \delta(\psi', x) \wedge \delta(\psi'', x)$

We define the states for all **G** and **R** subformulas as accepting, the rest is non-accepting. The state for the formula we started with is the only initial state.

From alternating automata to non-deterministic automata

So far

- We know how to build an alternating automaton from an LTL specification
- We know how to model check using a non-deterministic Büchi automaton

From alternating automata to non-deterministic automata

So far

- We know how to build an alternating automaton from an LTL specification
- We know how to model check using a non-deterministic Büchi automaton
- But how we obtain the non-deterministic automaton from the alternating one?

From alternating automata to non-deterministic automata

So far

- We know how to build an alternating automaton from an LTL specification
- We know how to model check using a non-deterministic Büchi automaton
- But how we obtain the non-deterministic automaton from the alternating one?
  $\rightarrow$ In a sense, we need to get rid of the universal branching.

Starting point

Observation: Run trees contain multiple runs of the automaton.

From alternating automata to non-deterministic automata

So far

- We know how to build an alternating automaton from an LTL specification
- We know how to model check using a non-deterministic Büchi automaton
- But how we obtain the non-deterministic automaton from the alternating one?
  $\rightarrow$ In a sense, we need to get rid of the universal branching.

Starting point

Observation: Run trees contain multiple runs of the automaton.

However, a non-deterministic automaton accepts if there is a *single* run.

From alternating automata to non-deterministic automata

So far

- We know how to build an alternating automaton from an LTL specification
- We know how to model check using a non-deterministic Büchi automaton
- But how we obtain the non-deterministic automaton from the alternating one?
  $\rightarrow$ In a sense, we need to get rid of the universal branching.

Starting point

Observation: Run trees contain multiple runs of the automaton.

However, a non-deterministic automaton accepts if there is a *single* run.

Hence, we need to compress complete run trees to individual runs.

From alternating automata to non-deterministic automata

So far

- We know how to build an alternating automaton from an LTL specification
- We know how to model check using a non-deterministic Büchi automaton
- But how we obtain the non-deterministic automaton from the alternating one?
  $\rightarrow$ In a sense, we need to get rid of the universal branching.

Starting point

Observation: Run trees contain multiple runs of the automaton.

However, a non-deterministic automaton accepts if there is a *single* run.

Hence, we need to compress complete run trees to individual runs.

The individual runs of a non-deterministic automaton then need to *simulate* a complete run tree.

From alternating automata to non-deterministic automata

So far

- We know how to build an alternating automaton from an LTL specification
- We know how to model check using a non-deterministic Büchi automaton
- But how we obtain the non-deterministic automaton from the alternating one?
  $\rightarrow$ In a sense, we need to get rid of the universal branching.

Starting point

Observation: Run trees contain multiple runs of the automaton.

However, a non-deterministic automaton accepts if there is a *single* run.

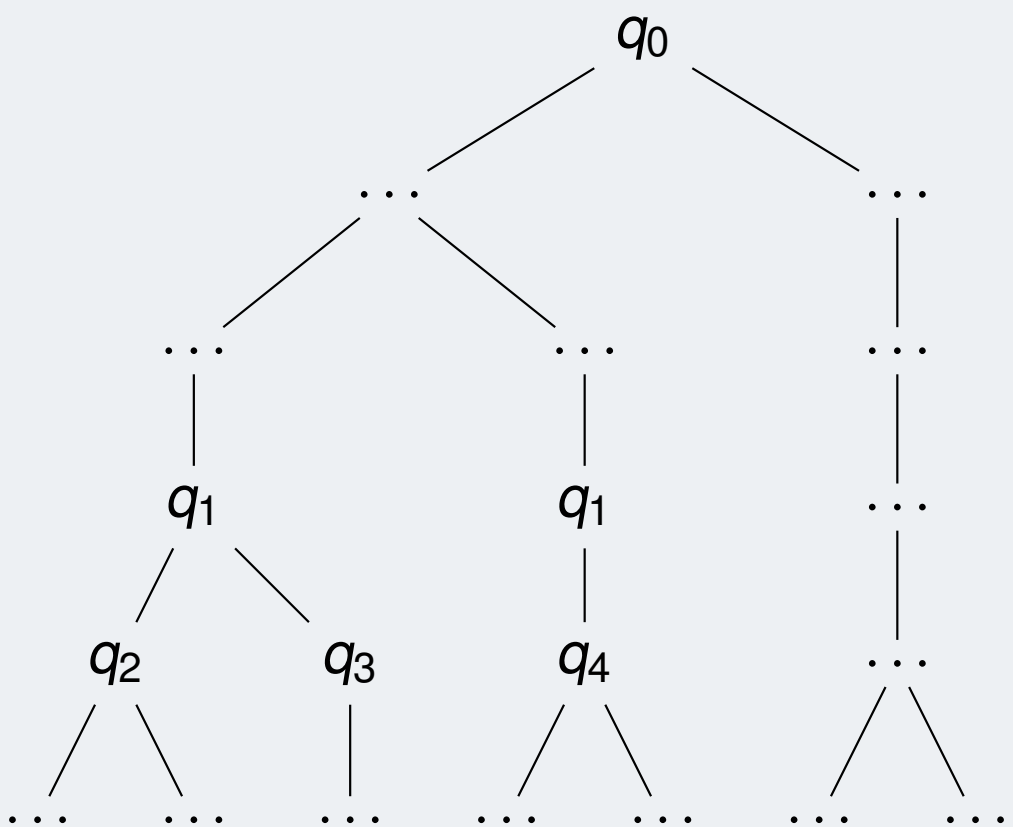
Hence, we need to compress complete run trees to individual runs.

The individual runs of a non-deterministic automaton then need to *simulate* a complete run tree.

For this to work, the run tree need to have a special shape.

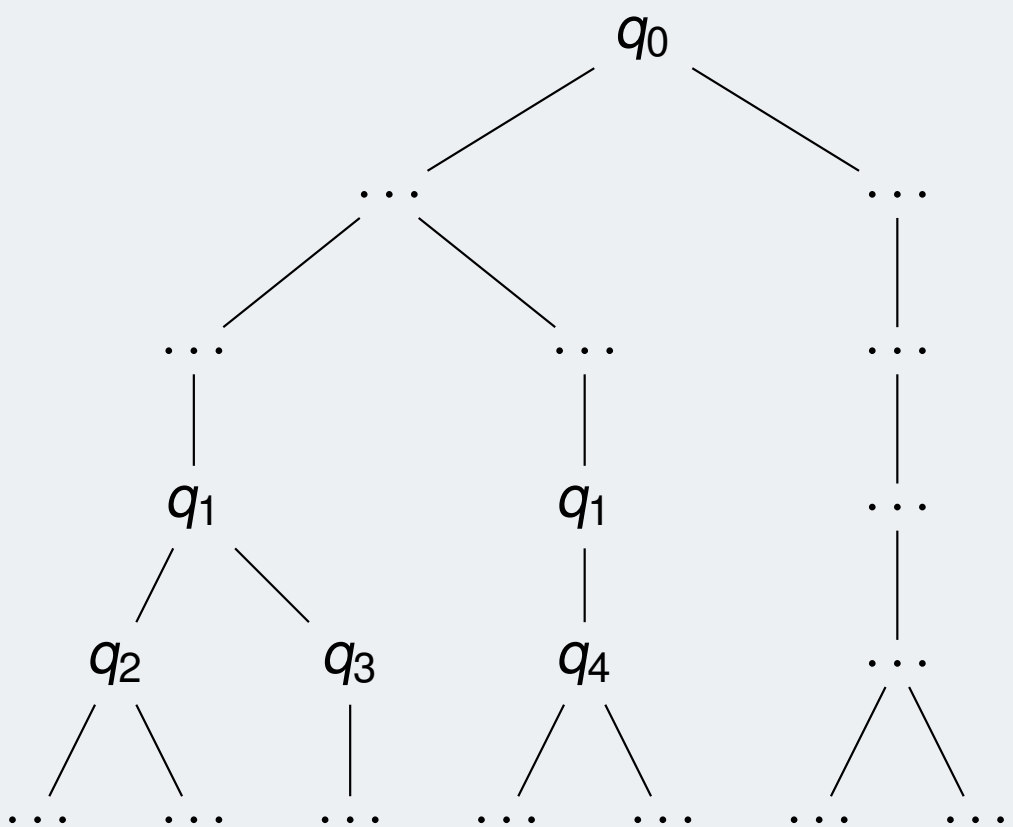
(Non-)Uniform run trees

A non-uniform run tree



(Non-)Uniform run trees

A non-uniform run tree



Definition of a uniform run tree

Formally, we call a run tree $\langle T_r, \tau_r \rangle$ *uniform* if for every two nodes t_1 and t_2, we have that if $|t_1| = |t_2|$ and $\tau(t_1) = \tau(t_2)$, then the sub-trees induced by t_1 and t_2 are the same.

Run trees

Observation

Every accepting run tree can be translated to a *uniform such tree*.

Run trees

Observation

Every accepting run tree can be translated to a *uniform such tree*.

Hence, we only need to search for *uniform* run trees henceforth.

Run trees

Observation

Every accepting run tree can be translated to a *uniform such tree*.

Hence, we only need to search for *uniform* run trees henceforth.

We can translate an accepting run tree by iterating through the levels and replacing subtrees by other subtrees. Since all subtrees are accepting, the tree stays accepting.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton.

For every transition, we use non-determinism in the target automaton to guess which transitions are taken along the branches in the run tree of the alternating automaton.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton.

For every transition, we use non-determinism in the target automaton to guess which transitions are taken along the branches in the run tree of the alternating automaton.

For a given word, we can thus read a valid run tree of the alternating automaton off the states of the non-deterministic Büchi automaton that occur along an accepting run of the latter.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton.

For every transition, we use non-determinism in the target automaton to guess which transitions are taken along the branches in the run tree of the alternating automaton.

For a given word, we can thus read a valid run tree of the alternating automaton off the states of the non-deterministic Büchi automaton that occur along an accepting run of the latter.

Additionally, to have our non-deterministic Büchi automaton accept only words that have an accepting run tree in the alternating automaton, we check that there is no infinite branch in the run tree on which accepting states occur only finitely often.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton.

For every transition, we use non-determinism in the target automaton to guess which transitions are taken along the branches in the run tree of the alternating automaton.

For a given word, we can thus read a valid run tree of the alternating automaton off the states of the non-deterministic Büchi automaton that occur along an accepting run of the latter.

Additionally, to have our non-deterministic Büchi automaton accept only words that have an accepting run tree in the alternating automaton, we check that there is no infinite branch in the run tree on which accepting states occur only finitely often.

We do this using a *break-point construction* that keeps track of along which branches in the run tree we have not “recently” seen accepting states.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton.

For every transition, we use non-determinism in the target automaton to guess which transitions are taken along the branches in the run tree of the alternating automaton.

For a given word, we can thus read a valid run tree of the alternating automaton off the states of the non-deterministic Büchi automaton that occur along an accepting run of the latter.

Additionally, to have our non-deterministic Büchi automaton accept only words that have an accepting run tree in the alternating automaton, we check that there is no infinite branch in the run tree on which accepting states occur only finitely often.

We do this using a *break-point construction* that keeps track of along which branches in the run tree we have not “recently” seen accepting states.

Once there is no such branch left, we reset our “recently” set, at which point we have reached a so-called *break-point*.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton.

For every transition, we use non-determinism in the target automaton to guess which transitions are taken along the branches in the run tree of the alternating automaton.

For a given word, we can thus read a valid run tree of the alternating automaton off the states of the non-deterministic Büchi automaton that occur along an accepting run of the latter.

Additionally, to have our non-deterministic Büchi automaton accept only words that have an accepting run tree in the alternating automaton, we check that there is no infinite branch in the run tree on which accepting states occur only finitely often.

We do this using a *break-point construction* that keeps track of along which branches in the run tree we have not “recently” seen accepting states.

Once there is no such branch left, we reset our “recently” set, at which point we have reached a so-called *break-point*.

We construct our non-deterministic automaton such that the states that witness break-points are the accepting ones.

The Miyano-Hayashi construction

Explanation

The Miyano-Hayashi construction makes use of the fact that we only have to consider uniform trees.

The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton.

The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton.

For every transition, we use non-determinism in the target automaton to guess which transitions are taken along the branches in the run tree of the alternating automaton.

For a given word, we can thus read a valid run tree of the alternating automaton off the states of the non-deterministic Büchi automaton that occur along an accepting run of the latter.

Additionally, to have our non-deterministic Büchi automaton accept only words that have an accepting run tree in the alternating automaton, we check that there is no infinite branch in the run tree on which accepting states occur only finitely often.

We do this using a *break-point construction* that keeps track of along which branches in the run tree we have not “recently” seen accepting states.

Once there is no such branch left, we reset our “recently” set, at which point we have reached a so-called *break-point*.

We construct our non-deterministic automaton such that the states that witness break-points are the accepting ones.

If and only if we visit break-points infinitely often, the run of the non-deterministic automaton is accepting and all branches in the corresponding run tree of the alternating automaton are accepting.

Miyano-Hayashi - the actual construction

Description

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be an alternating Büchi automaton. We define its translation to a non-deterministic Büchi automaton by $\mathcal{A}' = (Q', \Sigma, \delta', Q'_0, \mathcal{F}')$ with:

- $Q' = 2^Q \times 2^Q$

Miyano-Hayashi - the actual construction

Description

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be an alternating Büchi automaton. We define its translation to a non-deterministic Büchi automaton by $\mathcal{A}' = (Q', \Sigma, \delta', Q'_0, \mathcal{F}')$ with:

- $Q' = 2^Q \times 2^Q$
- $Q'_0 = \{(\{q_0\}, \emptyset)\}$

Miyano-Hayashi - the actual construction

Description

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be an alternating Büchi automaton. We define its translation to a non-deterministic Büchi automaton by $\mathcal{A}' = (Q', \Sigma, \delta', Q_0', \mathcal{F}')$ with:

- $Q' = 2^Q \times 2^Q$
- $Q_0' = \{(\{q_0\}, \emptyset)\}$
- $\mathcal{F} = \{(S, S') \mid S' = \emptyset\}$

Miyano-Hayashi - the actual construction

Description

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be an alternating Büchi automaton. We define its translation to a non-deterministic Büchi automaton by $\mathcal{A}' = (Q', \Sigma, \delta', Q_0', \mathcal{F}')$ with:

- $Q' = 2^Q \times 2^Q$
- $Q_0' = \{(\{q_0\}, \emptyset)\}$
- $\mathcal{F} = \{(S, S') \mid S' = \emptyset\}$
- $\delta((S, S'), x) = \{(S'', S''') \mid \forall q \in Q : (q \in S) \rightarrow (S'' \models \delta(q, x)) \wedge (q \in S') \rightarrow ((S''' \cup (S'' \cap \mathcal{F})) \models \delta(q, x)), S'' \supseteq S'''\}$ for all $S, S' \subseteq Q$ with $S' \neq \emptyset$ and $x \in \Sigma$

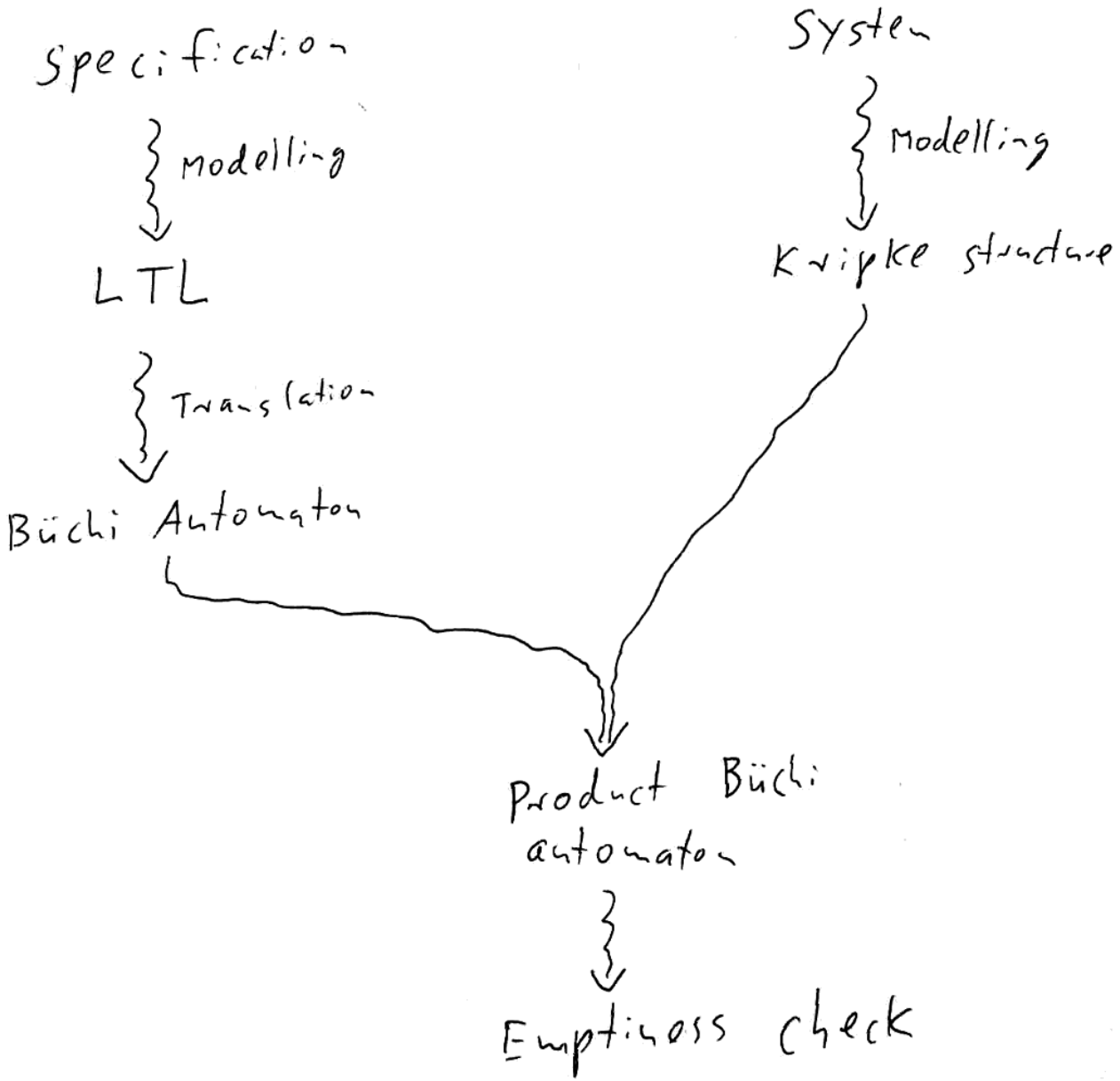
Miyano-Hayashi - the actual construction

Description

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be an alternating Büchi automaton. We define its translation to a non-deterministic Büchi automaton by $\mathcal{A}' = (Q', \Sigma, \delta', Q_0', \mathcal{F}')$ with:

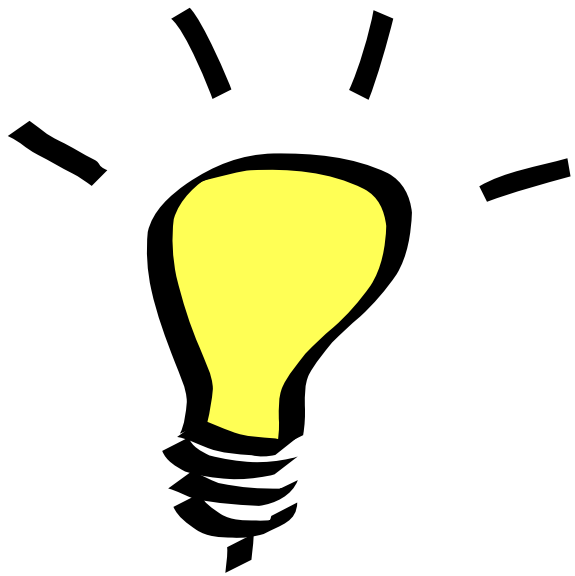
- $Q' = 2^Q \times 2^Q$
- $Q_0' = \{(\{q_0\}, \emptyset)\}$
- $\mathcal{F} = \{(S, S') \mid S' = \emptyset\}$
- $\delta((S, S'), x) = \{(S'', S''') \mid \forall q \in Q : (q \in S) \rightarrow (S'' \models \delta(q, x)) \wedge (q \in S') \rightarrow ((S''' \cup (S'' \cap \mathcal{F})) \models \delta(q, x)), S'' \supseteq S'''\}$ for all $S, S' \subseteq Q$ with $S' \neq \emptyset$ and $x \in \Sigma$
- $\delta((S, \emptyset), x) = \{(S'', S''') \mid \forall q \in Q, (q \in S) \rightarrow (S'' \models \delta(q, x)), S''' = S'' \setminus \mathcal{F}\}$ for all $S \subseteq Q$ and $x \in \Sigma$

Overview


Specification
modelling
LTL
Translation
Büchi Automaton
System
Modelling
Kripke structure
Product Büchi automaton
Emptiness check

Summary / List of Concepts

- Linear temporal logic
- Non-deterministic Büchi automata
- Product construction for model checking
- (Efficient) emptiness checking of non-deterministic Büchi automata
- Alternating automata
- Miyano-Hayashi construction



References I

J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proc. 1960 Internat. Congr .)*, pages 1–11. Stanford Univ. Press, Stanford, Calif., 1962.