System Specification, Verification and Synthesis (SSVS) – CS 4830/7485, Fall 2019

16: Formal Specification: Automata

Stavros Tripakis

Northeastern University
Khoury College of Computer Sciences

Outline

- Automata-based specifications
  - Finite automata: DFA and NFA
  - Omega automata (ω-automata): Büchi automata

FINITE AUTOMATA
(meaning both finite-state and finite-word)

Deterministic Finite Automata

A DFA is a tuple

$$(\Sigma, S, s_0, \delta, F)$$

- Σ: finite set of symbols, letters (the *alphabet*)
- S: finite set of states
- $s_0 \in S$: (unique) initial state
- δ: transition function (usually total but could also be partial)

$$\delta : S \times \Sigma \to S$$

- $F \subseteq S$: set of final/accepting states

Deterministic Finite Automata

A DFA is a tuple

$$(\Sigma, S, s_0, \delta, F)$$

- Σ: finite set of symbols, letters (the *alphabet*)
- S: finite set of states
- $s_0 \in S$: (unique) initial state
- δ: transition function (usually total but could also be partial)

$$\delta : S \times \Sigma \to S$$

- $F \subseteq S$: set of final/accepting states

A DFA can be seen as a special case of a Moore machine – how?

Deterministic Finite Automata

A DFA is a tuple

$$(\Sigma, S, s_0, \delta, F)$$

- Σ: finite set of symbols, letters (the *alphabet*)
- S: finite set of states
- $s_0 \in S$: (unique) initial state
- δ: transition function (usually total but could also be partial)

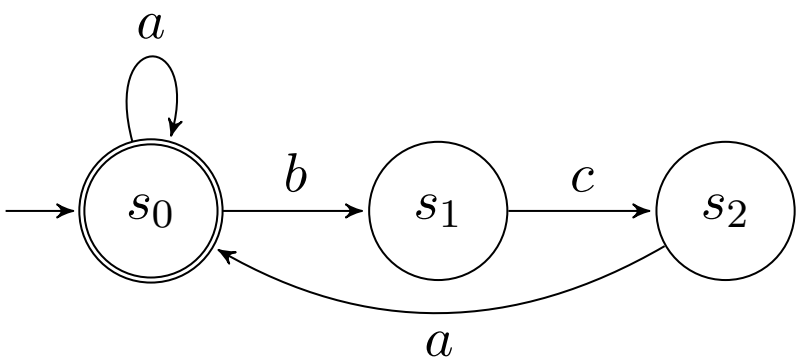
$$\delta : S \times \Sigma \to S$$

- $F \subseteq S$: set of final/accepting states

A DFA can be seen as a special case of a Moore machine – how?
Σ is the set of input symbols I. The set of output symbols O is binary: a state is either accepting or non-accepting.

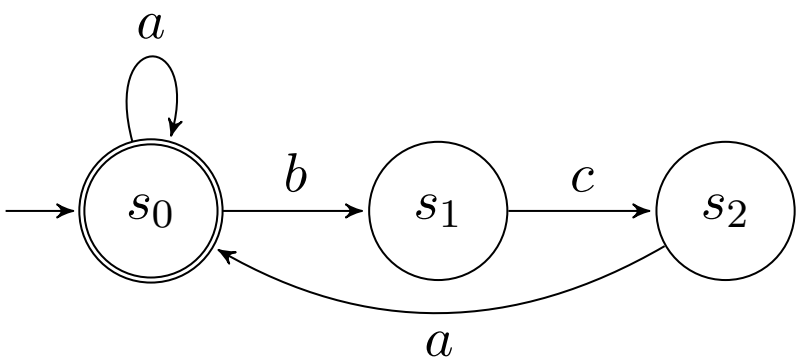
Example: DFA

Define the tuple $(\Sigma, S, s_0, \delta, F)$ for the automaton below:

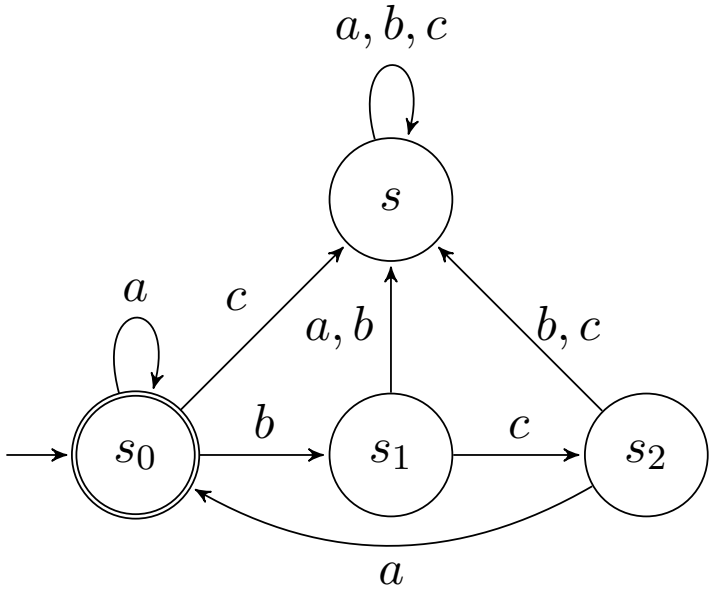


Example: DFA

Define the tuple $(\Sigma, S, s_0, \delta, F)$ for the automaton below:



The automaton above is **incomplete**: it's missing several transitions!
We complete it by adding a non-accepting sink state:



DFA: Informal Semantics

- A DFA $A = (\Sigma, S, s_0, \delta, F)$ defines a **language** $L(A) \subseteq \Sigma^*$, i.e., a set of finite words over Σ.

$$L(A) = \{ \text{ all words in } \Sigma^* \text{ accepted by } A \}$$

- A word is accepted if the automaton ends up in an accepting state after reading the word.
- What's the language defined by the automaton in the previous slide?

DFA: Informal Semantics

- A DFA $A = (\Sigma, S, s_0, \delta, F)$ defines a **language** $L(A) \subseteq \Sigma^*$, i.e., a set of finite words over Σ.

$$L(A) = \{ \text{ all words in } \Sigma^* \text{ accepted by } A\}$$

- A word is accepted if the automaton ends up in an accepting state after reading the word.
- What's the language defined by the automaton in the previous slide?
- Answer: $L(A) = \{\epsilon, a, aa, aaa, ..., bca, bcabca, ..., abca, aabca, ...\}$
- Answer (using regular expression notation): $L(A) = (a + (bca))^*$

DFA: Formal Semantics

- Let $A = (\Sigma, S, s_0, \delta, F)$.
- Let $\rho \in \Sigma^*$, with $\rho = a_1 a_2 \cdots a_n$, for $n \geq 0$. If $n = 0$ then $\rho = \epsilon$, the empty word.
- Define the function $\delta^* : S \times \Sigma^* \to S$ as follows:

$$\delta^*(s, \epsilon) = s \quad \text{and} \quad \delta^*(s, a \cdot \rho) = \delta^*(\delta(s, a), \rho)$$

- A word $\rho \in \Sigma^*$ is *accepted by* A if $\delta^*(s_0, \rho) \in F$.
  Otherwise ρ is *rejected by* A.
- The language of A (or the language *accepted* or *recognized* by A) is defined as:

$$L(A) = \{\rho \in \Sigma^* \mid \delta^*(s_0, \rho) \in F\}$$

Non-Deterministic Finite Automata

An NFA is a tuple

$$(\Sigma, S, S_0, \Delta, F)$$

- Σ: alphabet
- S: finite set of states
- $S_0 \subseteq S$: **set of** initial states
- Δ: transition **relation**

  $$\Delta \subseteq S \times \Sigma \times S$$

  or

  $$\Delta \subseteq S \times \Sigma \cup \{\epsilon\} \times S$$

  where ϵ is the "empty" symbol, or "silent" or "unobservable" or "internal" action.
- $F \subseteq S$: set of final/accepting states

NFA Semantics

A **run** of a NFA $(\Sigma, S, S_0, \Delta, F)$ is a finite sequence of states and transitions:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \cdots s_n$$

such that

- $s_0 \in S_0$
- $\forall i : (s_i, a_{i+1}, s_{i+1}) \in \Delta$

The run is **accepting** if it ends in an accepting state: $s_n \in F$.

The **word generated by this run** is the corresponding sequence of labels:

$$a_1 a_2 a_3 \cdots a_n$$

A word is **accepted** if there **exists** a run generating it.

The **language** of the automaton is the set of all accepted words.

Example

These two automata accept the same language:

Which language?

Example

These two automata accept the same language:

Which language?

All finite strings over $\Sigma = \{0, 1\}$ that end with 1:

$$L = \{1, 01, 11, 001, 011, ...\}$$

Determinization of Finite Automata

Theorem ([Hopcroft and Ullman, 1990])

Every NFA can be transformed into an equivalent DFA, i.e., a DFA that accepts the same language.

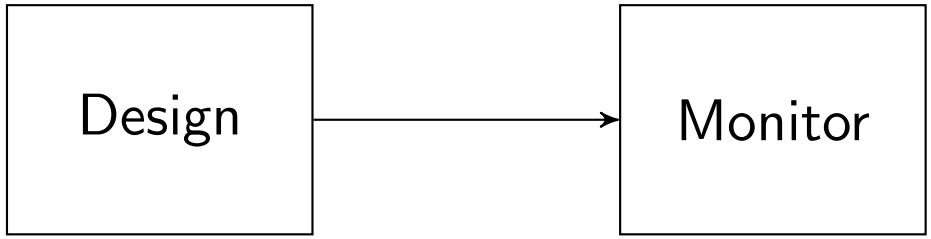
The proof uses the famous **powerset construction**, a very useful construction for several CS applications.

The languages accepted by NFA or DFA are called **regular**.

Finite Automata and Safety Properties

- We can use finite automata (DFA or NFA) to capture the **negation** of safety properties.
- The **bad prefixes** of a safety property are finite words.
- Assuming the set of bad prefixes is regular, it can be captured as a finite automaton.

Verification using safety monitors[1]



- **Design** is the transition system we want to check (implementation).
- **Monitor** captures $\neg\phi$, the negation of the specification.
- When ϕ is safety, the Monitor can simply be an NFA or DFA which accepts the bad prefixes of ϕ.
- The accepting states of that NFA/DFA are **error states**. The Monitor acts like a "watchdog": it observes the outputs of Design, and moves to an error state if ever Design does something wrong.
- Once the Monitor enters the error state, it cannot leave. Once a safety property is violated, there is no recovery.

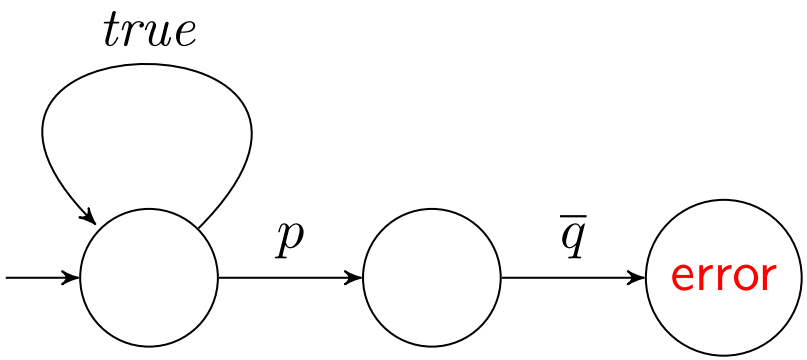
[1] This is related to the so-called *automata-theoretic verification* approach, which we will formalize later.

Example: safety monitor

What would a Monitor for $\mathbf{G}(p \rightarrow \mathbf{X}q)$ look like?

Example: safety monitor

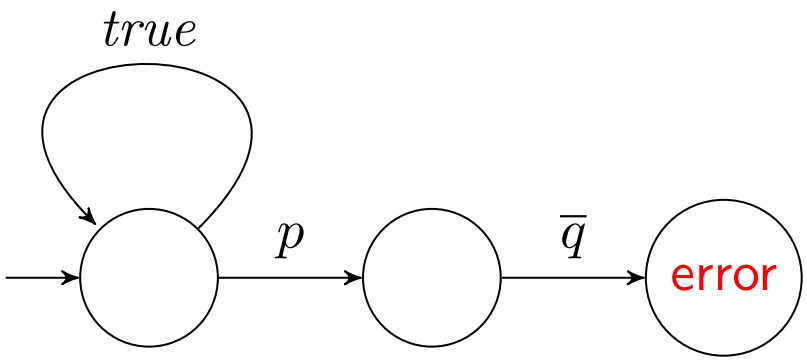
What would a Monitor for $\mathbf{G}(p \rightarrow \mathbf{X}q)$ look like?



What is the alphabet (set of input symbols) that this monitor reads?

Example: safety monitor

What would a Monitor for $\mathbf{G}(p \rightarrow \mathbf{X}q)$ look like?

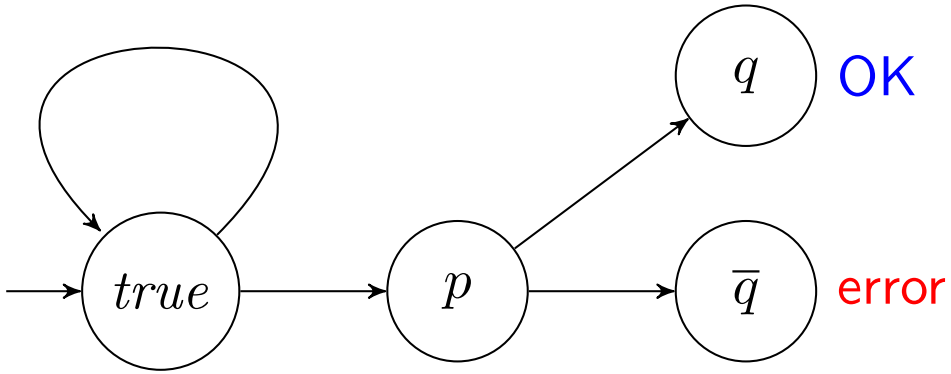


What is the alphabet (set of input symbols) that this monitor reads?

- 2^{AP}. Assume in this case $\mathsf{AP} = \{p, q\}$.
- A **guard** like p actually means $\{pq, p\overline{q}\}$.
- Monitor synchronizes with Design at every step.
- Specifically: Design $\times$ Monitor moves from product state (s, m) to (s', m') as follows:
  1. Monitor chooses a transition $m \xrightarrow{g} m'$ whose guard g is satisfied by $L(s)$ (the propositions holding at s).
  2. Design picks a move $s \longrightarrow s'$.

Safety monitor with guards on the states

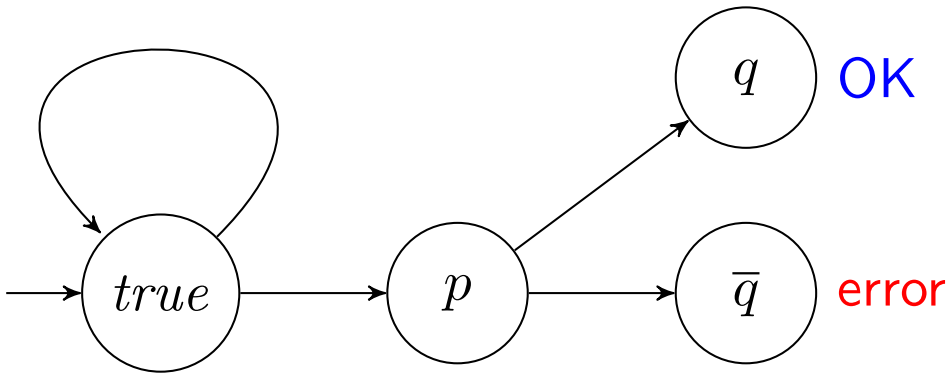
We could also put the guards on the states, instead of the transitions:



Again, Monitor synchronizes with Design at every step, but here the synchronization rules are slightly different:

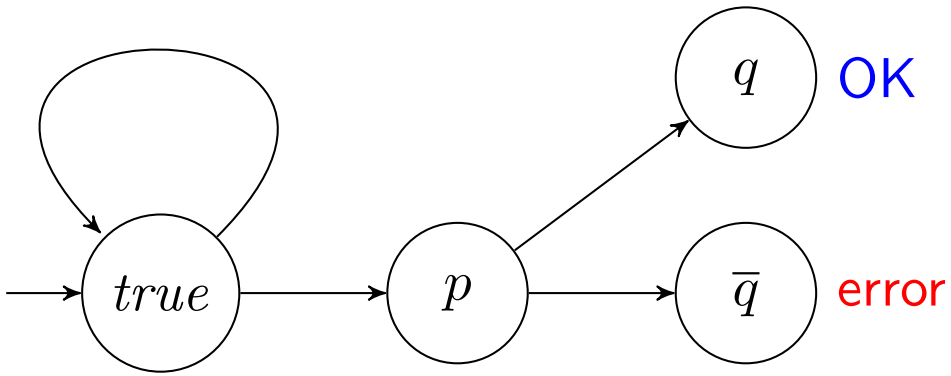
- Let $L(s) \subseteq \text{AP}$ be the labeling of a state s of Design, and $g_m \subseteq 2^{\text{AP}}$ be the guard of a state m of Monitor.
- Then a product state (s, m) is legal iff $L(s)$ satisfies g_m.
- During composition, only legal product states are generated.

Safety monitor with guards on the states



Is this really the correct Monitor for $\mathbf{G}(p \rightarrow \mathbf{X}q)$?

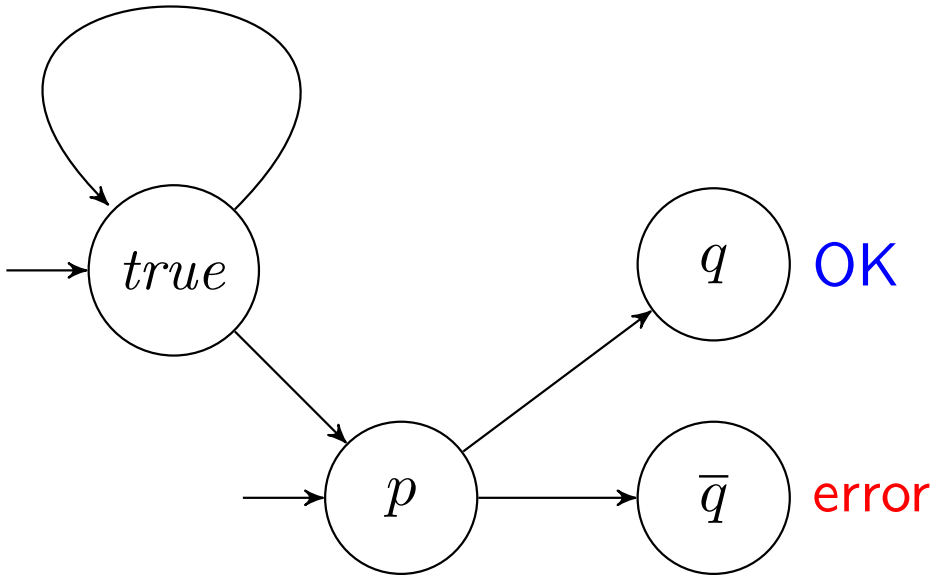
Safety monitor with guards on the states



Is this really the correct Monitor for $\mathbf{G}(p \rightarrow \mathbf{X}q)$?
What if a p happens right at the first step?
Above Monitor misses that case.

Safety monitor with guards on the states (corrected)

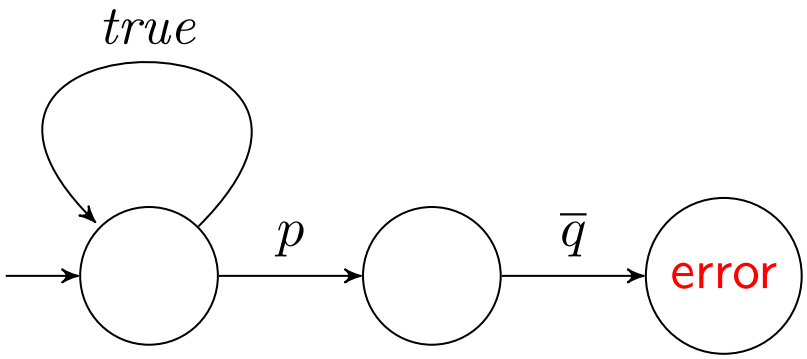
Monitor for $\mathbf{G}(p \rightarrow \mathbf{X}q)$ with labels on states:



Corrected by making p also an initial state.

Safety monitors can be non-deterministic

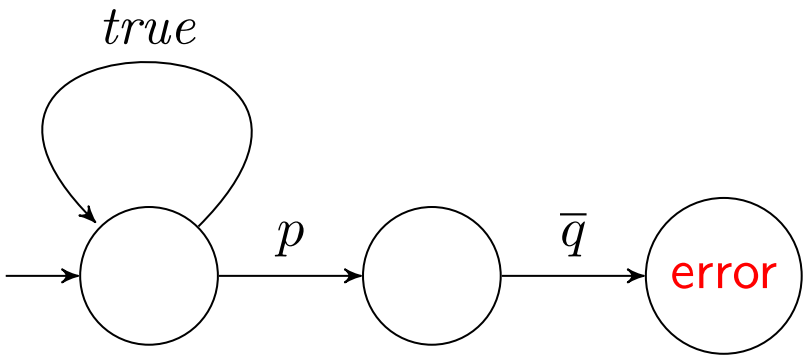
Monitor for $\mathbf{G}(p \rightarrow \mathbf{X}q)$:



This monitor is **non-deterministic**: **every** run must avoid the error state.

Safety monitors can be non-deterministic

Monitor for $\mathbf{G}(p \rightarrow \mathbf{X}q)$:



This monitor is **non-deterministic**: **every** run must avoid the error state.

This is OK, because we can explore **all** reachable states of the product Design $\times$ Monitor, and check that the error state is unreachable.

Here we can see an example of the advantage of non-determinism in specification.

ω-AUTOMATA

Non-Deterministic Büchi Automata (NBA)

NBA syntax = NFA syntax = $(\Sigma, S, S_0, \Delta, F)$.

But the interpretation is different (same syntax, different semantics):

- An NFA accepts **finite words**, that is, elements of Σ^*.
- An NBA accepts **infinite words**, that is, elements of Σ^ω.
  An element of Σ^ω is an infinite sequence

$$a_1 a_2 a_3 \cdots$$

  of letters in Σ, i.e., where for every i, $a_i \in \Sigma$.

Non-Deterministic Büchi Automata (NBA)

NBA syntax = NFA syntax = $(\Sigma, S, S_0, \Delta, F)$.

But the interpretation is different (same syntax, different semantics):

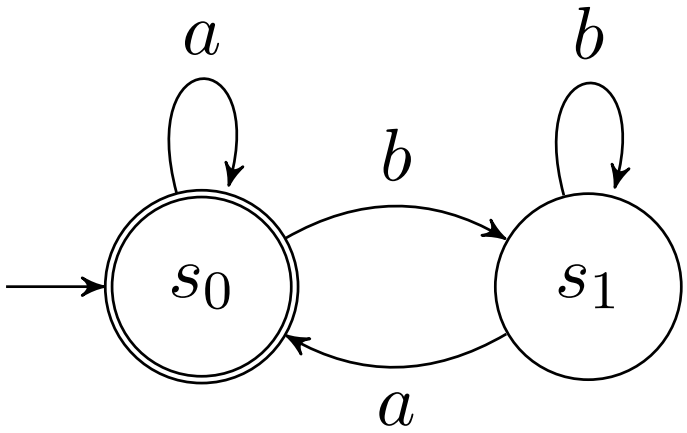
- An NFA accepts **finite words**, that is, elements of Σ^*.
- An NBA accepts **infinite words**, that is, elements of Σ^ω.
  An element of Σ^ω is an infinite sequence

$$a_1 a_2 a_3 \cdots$$

  of letters in Σ, i.e., where for every i, $a_i \in \Sigma$.

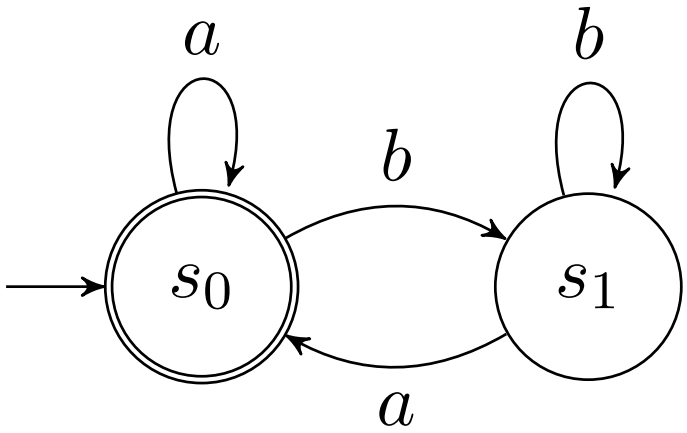
NBA acceptance condition: an infinite word is accepted if there **exists** an infinite run generating this word such that the run visits an accepting state (i.e., a state in F) **infinitely often**.

Büchi automaton: example



Which words does this Büchi automaton accept?

Büchi automaton: example



Which words does this Büchi automaton accept?

All infinite words where a appears infinitely often.

Deterministic and complete Büchi automata

- A *Deterministic Büchi Automaton* (DBA) is an NBA where every infinite word generates at most one run (it could also deadlock, i.e., generate zero runs).
- A sufficient condition for determinism is that the transition relation Δ is a function (it could be a partial function).

Deterministic and complete Büchi automata

- A *Deterministic Büchi Automaton* (DBA) is an NBA where every infinite word generates at most one run (it could also deadlock, i.e., generate zero runs).
- A sufficient condition for determinism is that the transition relation Δ is a function (it could be a partial function).
  Is this condition also necessary?

Deterministic and complete Büchi automata

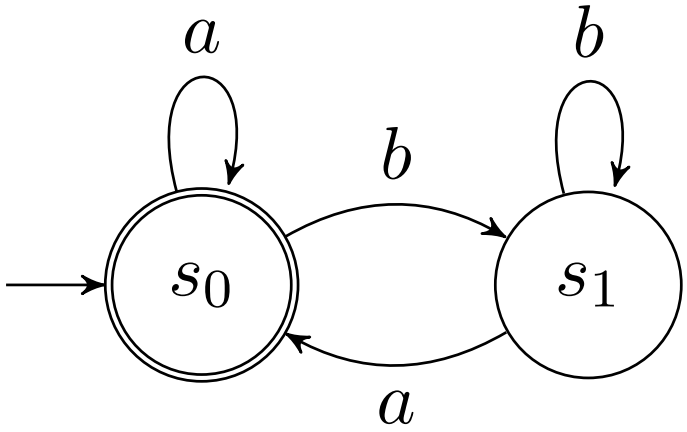
- A *Deterministic Büchi Automaton* (DBA) is an NBA where every infinite word generates at most one run (it could also deadlock, i.e., generate zero runs).
- A sufficient condition for determinism is that the transition relation Δ is a function (it could be a partial function).
  Is this condition also necessary?
  No: there could be unreachable states with non-deterministic outgoing transitions.

Deterministic and complete Büchi automata

- A *Deterministic Büchi Automaton* (DBA) is an NBA where every infinite word generates at most one run (it could also deadlock, i.e., generate zero runs).
- A sufficient condition for determinism is that the transition relation Δ is a function (it could be a partial function).
  Is this condition also necessary?
  No: there could be unreachable states with non-deterministic outgoing transitions.
- A (deterministic or non-deterministic) Büchi automaton is *complete* if every infinite word generates at least one run.
- A sufficient condition for completeness is
  $\forall s \in S : \forall a \in \Sigma : \exists s' \in S : (s, s') \in \Delta$, i.e., every state has at least one successor for every input letter.

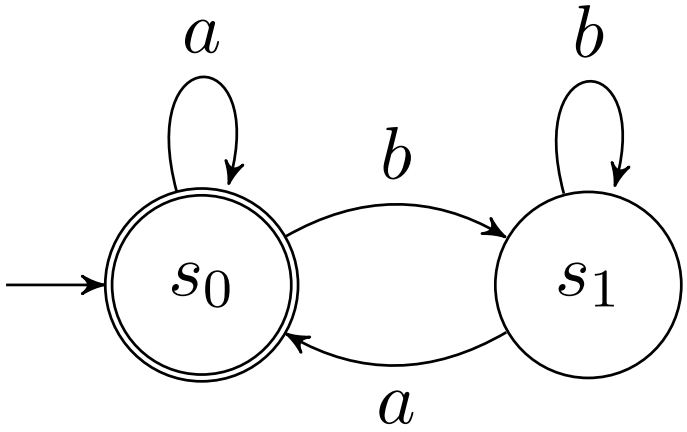
Deterministic and complete Büchi automata

Is this automaton deterministic?



Deterministic and complete Büchi automata

Is this automaton deterministic?

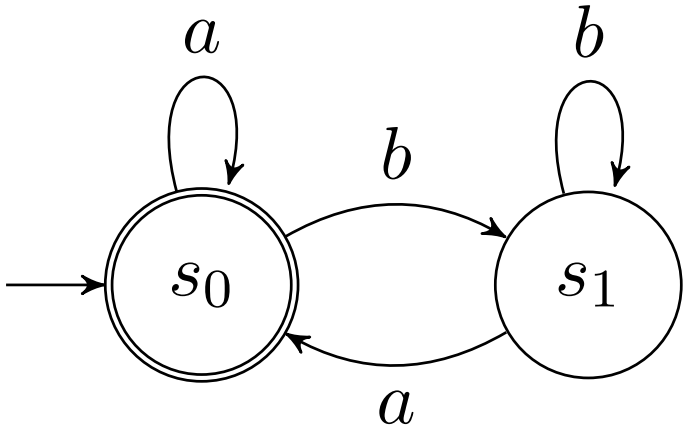


Yes.

Is it complete?

Deterministic and complete Büchi automata

Is this automaton deterministic?

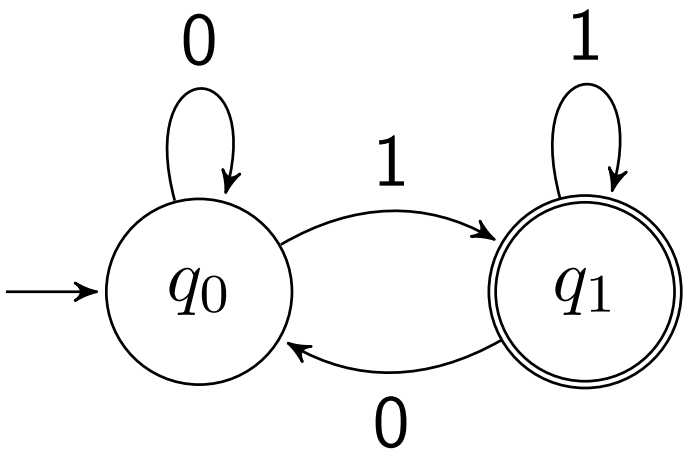


Yes.

Is it complete? Yes, assuming $\Sigma = \{a, b\}$.

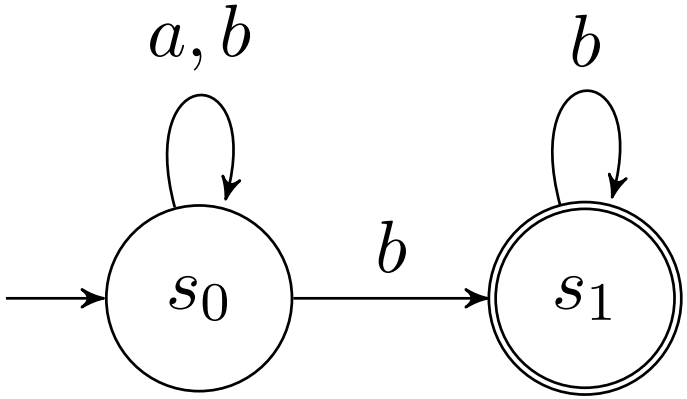
Parenthesis: Syntax vs Semantics

This could be a DFA or a DBA: no way to tell just by looking at the diagram!



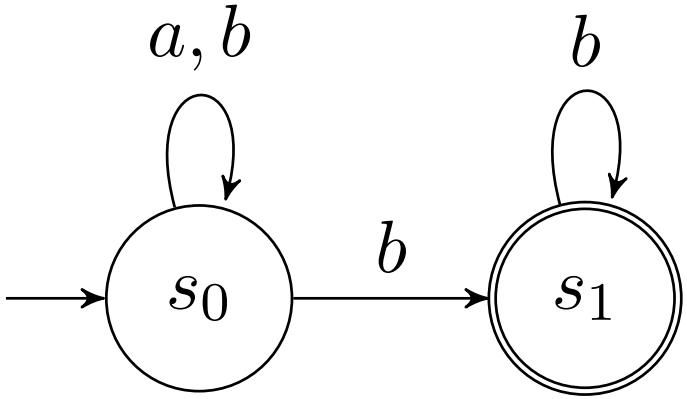
Same syntax, different semantics!

Non-Deterministic vs. Deterministic Büchi Automata



Which words does this Büchi automaton accept?

Non-Deterministic vs. Deterministic Büchi Automata

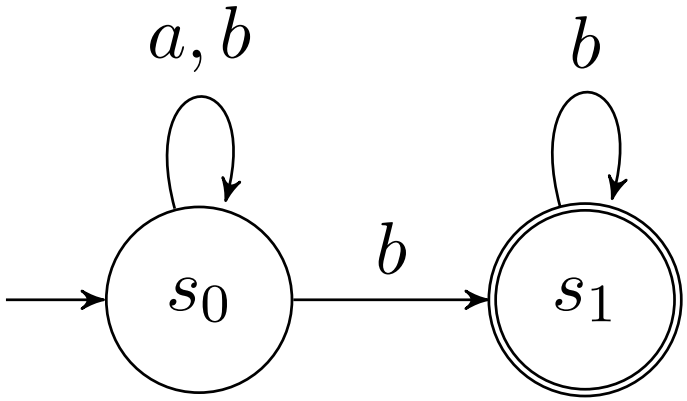


Which words does this Büchi automaton accept?

All infinite words ending with an infinite sequence of b's.

Omega-regular expression: $(a + b)^* \cdot b^\omega$

Non-Deterministic vs. Deterministic Büchi Automata

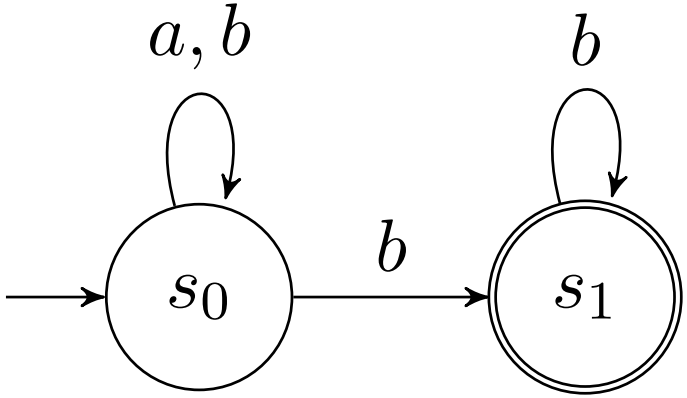


This Büchi automaton is **non-deterministic**. Why?

Can we determinize it?

Let's see what the standard subset construction gives:

Non-Deterministic vs. Deterministic Büchi Automata



Theorem

There is no deterministic Büchi automaton which is equivalent to the above (i.e., which accepts the same language).

Proof: on whiteboard.

NBA vs. DBA

Let:

- NBA denote the class of infinite-word languages accepted by non-deterministic Büchi automata.
  Therefore, $NBA \subseteq 2^{\Sigma^\omega}$.
- DBA denote the class of infinite-word languages accepted by deterministic Büchi automata.

The previous example shows that DBA is a **strict subset** of NBA:

$$DBA \subset NBA$$

In contrast, DFA and NFA accept the same class of (finite-word) languages.

NBA vs. DBA

Let:

- NBA denote the class of infinite-word languages accepted by non-deterministic Büchi automata.
  Therefore, $NBA \subseteq 2^{\Sigma^\omega}$.
- DBA denote the class of infinite-word languages accepted by deterministic Büchi automata.

The previous example shows that DBA is a **strict subset** of NBA:

$$DBA \subset NBA$$

In contrast, DFA and NFA accept the same class of (finite-word) languages.

What about LTL? We'll see that next.

Other types of ω-automata[2]

- Street automata
- Rabin automata
- Parity automata
- ...

These automata are **not** more powerful than non-deterministic Büchi automata: they all recognize the class of ω-regular languages.

But contrary to NBA, the above automata can be determinized.

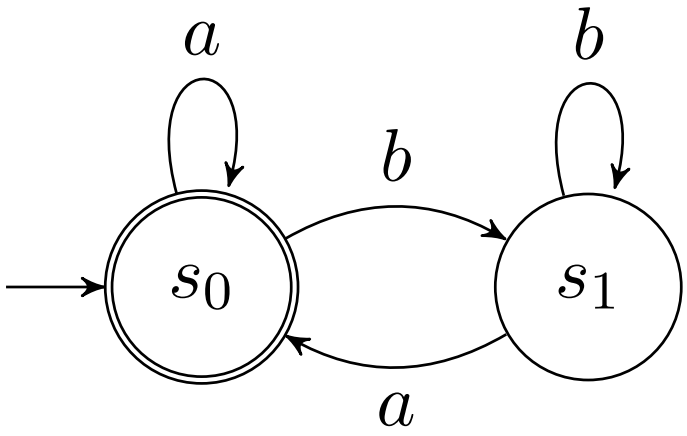
We will not look at these automata in this course. For more info, see *Automata-Theoretic Verification* by Esparza, Kupferman, Vardi: `https://www.cs.rice.edu/~vardi/papers/hba11.pdf`

[2] There's also tree automata, alternating automata, alternating tree automata, etc. We will not look at any of these.

LTL VS BUCHI AUTOMATA

LTL vs. DBA

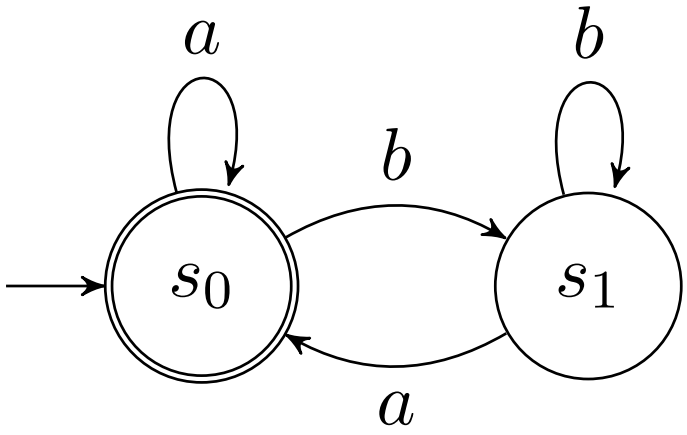
Let $\Sigma = \{a, b\}$.



Is there an LTL formula which "accepts" the same language as this DBA?
(We can define the language of an LTL formula as the set of infinite words that satisfy the formula.)

LTL vs. DBA

Let $\Sigma = \{a, b\}$.

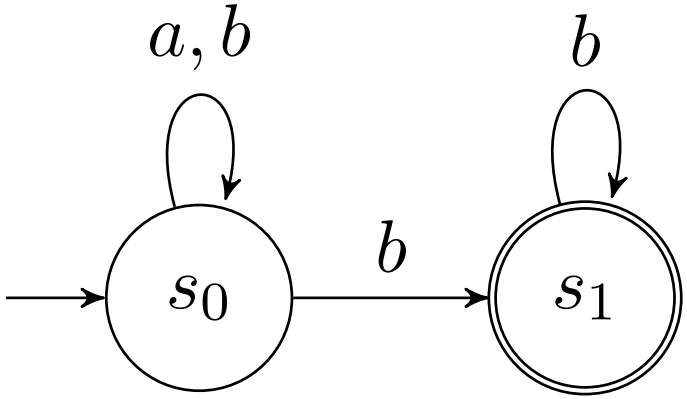


Is there an LTL formula which "accepts" the same language as this DBA?
(We can define the language of an LTL formula as the set of infinite words that satisfy the formula.)

Yes:

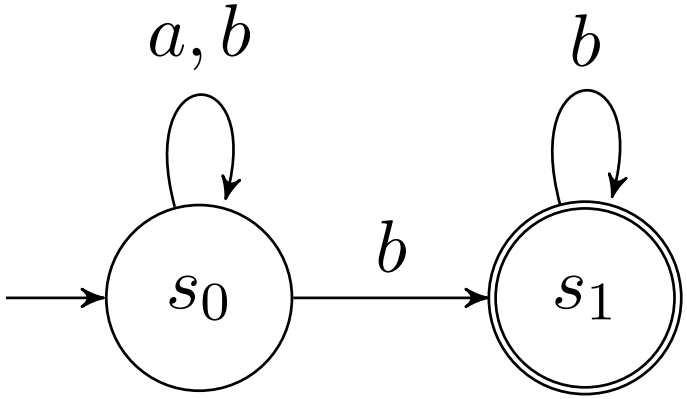
$$\mathbf{GF}a$$

LTL vs. DBA



Is there an LTL formula which "accepts" the same language as this NBA?

LTL vs. DBA



Is there an LTL formula which "accepts" the same language as this NBA?

Yes:

$$\mathbf{FG}b$$

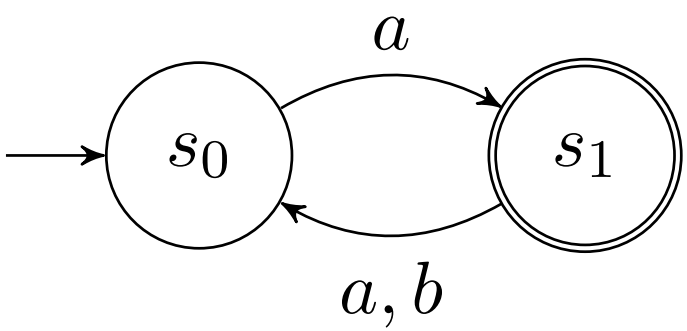
Let LTL and DBA be the class of properties expressible as LTL formulas and DBA, respectively. This example shows that

$$LTL \not\subseteq DBA$$

because as we have seen earlier, there is no DBA equivalent to the above NBA.

LTL vs. DBA

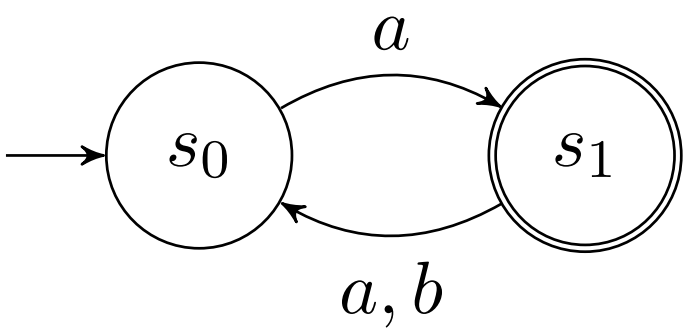
Consider the following DBA:



Which language does this automaton accept?

LTL vs. DBA

Consider the following DBA:



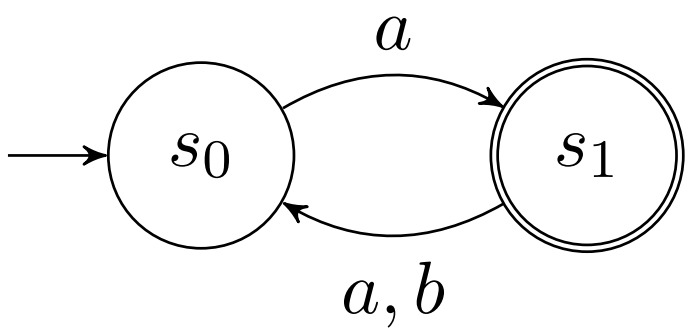
Which language does this automaton accept?

The set of all words where a appears (at least) in positions $1, 3, 5, \ldots$ (i.e., all odd positions).

Is there an equivalent LTL formula?

LTL vs. DBA

Consider the following DBA:



Which language does this automaton accept?

The set of all words where a appears (at least) in positions $1, 3, 5, \ldots$ (i.e., all odd positions).

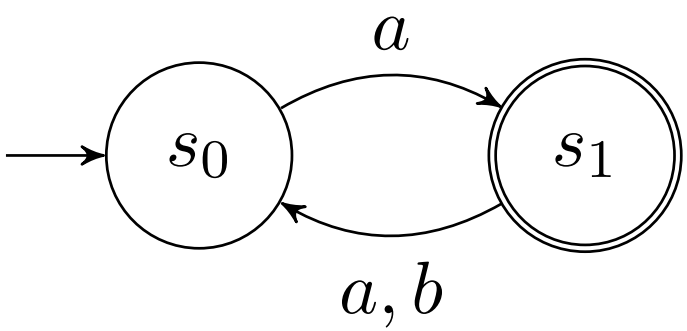
Is there an equivalent LTL formula?

No [Wolper, 1983]

In particular, formula $\mathbf{G}(a \rightarrow \mathbf{XX}a)$ doesn't work. Why?

LTL vs. DBA

Consider the following DBA:



Which language does this automaton accept?

The set of all words where a appears (at least) in positions $1, 3, 5, \ldots$ (i.e., all odd positions).

Is there an equivalent LTL formula?

No [Wolper, 1983]

In particular, formula $\mathbf{G}(a \rightarrow \mathbf{XX}a)$ doesn't work. Why?

This example shows that

$$DBA \not\subseteq LTL$$

LTL vs. NBA

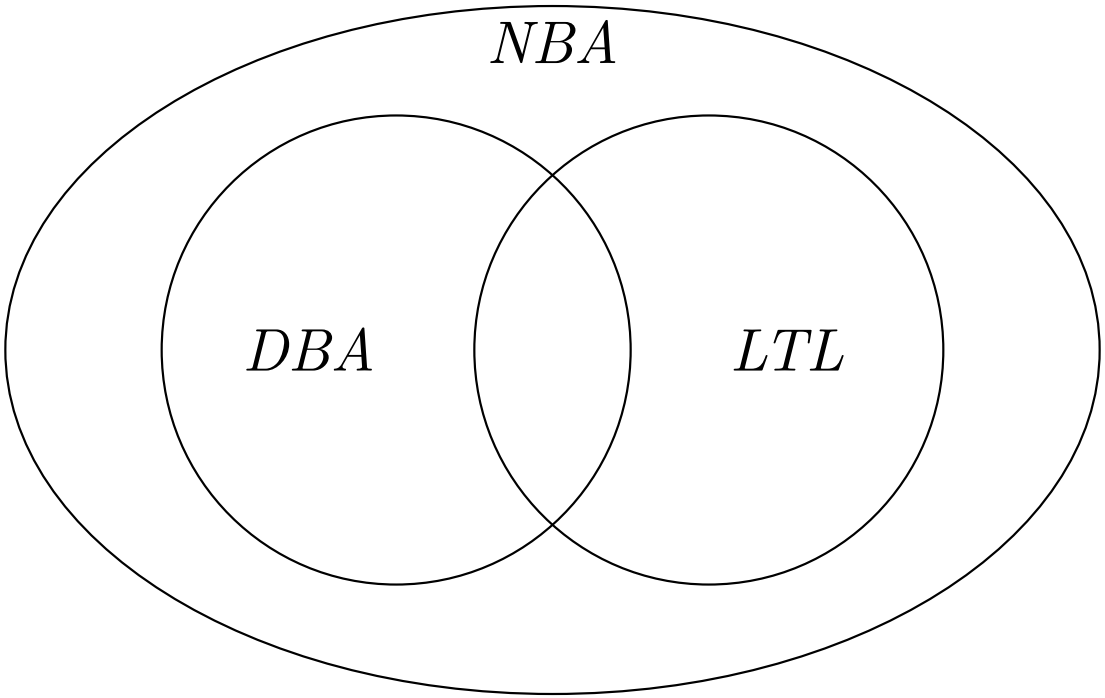
What about LTL vs. NBA?

We will later provide a translation from LTL formulas to NBA, therefore proving

$$LTL \subset NBA$$

The inclusion is strict because of the previous example (and the fact that every DBA is a special case of an NBA).

Summary: NBA vs. LTL vs. DBA



Parenthesis: LTL with quantifiers

Just like there is propositional logic (without quantifiers) and first-order logic (with quantifiers over variables), there is also **propositional** LTL (the one we've been studying) and LTL with quantifiers.

We call the latter **QLTL**.

QLTL is strictly more expressive than LTL.

For example, the property on slide 33 can be expressed in QLTL:

$$\exists s : (\mathbf{G}(s \rightarrow \mathbf{X}\overline{s})) \wedge (\mathbf{G}(\overline{s} \rightarrow \mathbf{X}s)) \wedge (\mathbf{G}(s \rightarrow a))$$

Note: s here is a **sequence** of boolean values, so $\exists s : \cdots$ is read as "*there exists an infinite sequence s such that ...*".

Parenthesis: LTL with quantifiers

Just like there is propositional logic (without quantifiers) and first-order logic (with quantifiers over variables), there is also **propositional** LTL (the one we've been studying) and LTL with quantifiers.

We call the latter **QLTL**.

QLTL is strictly more expressive than LTL.

For example, the property on slide 33 can be expressed in QLTL:

$$\exists s : (\mathbf{G}(s \rightarrow \mathbf{X}\overline{s})) \wedge (\mathbf{G}(\overline{s} \rightarrow \mathbf{X}s)) \wedge (\mathbf{G}(s \rightarrow a))$$

Note: s here is a **sequence** of boolean values, so $\exists s : \cdots$ is read as "*there exists an infinite sequence s such that ...*".

We will not study QLTL further in this course.

Bibliography


Baier, C. and Katoen, J.-P. (2008).
Principles of Model Checking.
MIT Press.


Clarke, E., Grumberg, O., and Peled, D. (2000).
Model Checking.
MIT Press.


Hopcroft, J. E. and Ullman, J. D. (1990).
Introduction To Automata Theory, Languages, And Computation.
Addison-Wesley.


Wolper, P. (1983).
Temporal logic can be more expressive.
Information and Control, 56(1-2):72 – 99.