Automated Program Verification and Testing
15414/15614 Fall 2016
Lecture 20:
Explicit-State Model Checking, Part 2

Matt Fredrikson
mfredrik@cs.cmu.edu

November 17, 2016
LTL Model Checking
Today’s Lecture

LTL Model Checking

Algorithm based on automata operations
LTL Model Checking

Algorithm based on automata operations

▶ Refresher on basic automata theory
Today’s Lecture

LTL Model Checking

Algorithm based on automata operations
  ▶ Refresher on basic automata theory
  ▶ Introduce automata for languages of infinite words
Today’s Lecture

LTL Model Checking

Algorithm based on automata operations

- Refresher on basic automata theory
- Introduce automata for languages of infinite words
- See how to apply them to model checking
A Nondeterministic Finite Automaton (NFA) is a tuple \((Q, \Sigma, \delta, Q_0, F)\):
A Nondeterministic Finite Automaton (NFA) is a tuple \((Q, \Sigma, \delta, Q_0, F)\): 
- \(Q\) is a finite set of states; \(Q_0\) initial states, \(F\) final.
A Nondeterministic Finite Automaton (NFA) is a tuple \((Q, \Sigma, \delta, Q_0, F)\):
- \(Q\) is a finite set of states; \(Q_0\) initial states, \(F\) final
- \(\Sigma\) is a finite alphabet
Finite Automata: Refresher

A Nondeterministic Finite Automaton (NFA) is a tuple \((Q, \Sigma, \delta, Q_0, F)\):

- \(Q\) is a finite set of states; \(Q_0\) initial states, \(F\) final
- \(\Sigma\) is a finite alphabet
- \(\delta \subseteq Q \times \Sigma \times Q\) is the transition relation

Example:

\(Q_0 = \{q_0\}\)
\(Q = \{q_0, q_1\}\)
\(\Sigma = \{a, b\}\)

\(\delta\) is the transition relation,
\(F = \{q_1\}\)
A Nondeterministic Finite Automaton (NFA) is a tuple \((Q, \Sigma, \delta, Q_0, F)\):

- \(Q\) is a finite set of states; \(Q_0\) initial states, \(F\) final
- \(\Sigma\) is a finite alphabet
- \(\delta \subseteq Q \times \Sigma \times Q\) is the transition relation

An automaton is **deterministic** (a DFA) if:

\[
\forall a \in \Sigma. (q, a, q') \in \delta \land (q, a, q'') \in \delta \Rightarrow q = q''
\]
Finite Automata: Refresher

A Nondeterministic Finite Automaton (NFA) is a tuple \((Q, \Sigma, \delta, Q_0, F)\):

- \(Q\) is a finite set of states; \(Q_0\) initial states, \(F\) final
- \(\Sigma\) is a finite alphabet
- \(\delta \subseteq Q \times \Sigma \times Q\) is the transition relation

An automaton is **deterministic** (a DFA) if:

\[
\forall a \in \Sigma. (q, a, q') \in \delta \land (q, a, q'') \in \delta \Rightarrow q = q''
\]

Example:

\[
\begin{aligned}
Q &= \{q_0, q_1\}, \Sigma &= \{a, b\}, Q_0 &= \{q_0\}, F &= \{q_1\}
\end{aligned}
\]
Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA, $w = a_0 \ldots a_n \in \Sigma^*$ a finite word
Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA, $w = a_0 \ldots a_n \in \Sigma^*$ a finite word

A run for $w$ in $A$ is a finite sequence of states $q_0 \ldots q_{n-1}$ where:
Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA, $w = a_0 \ldots a_n \in \Sigma^*$ a finite word.

A run for $w$ in $A$ is a finite sequence of states $q_0 \ldots q_{n-1}$ where:

- $q_0 \in Q_0$
Let \( A = (Q, \Sigma, \delta, Q_0, F) \) be an NFA, \( w = a_0 \ldots a_n \in \Sigma^* \) a finite word

A \textbf{run} for \( w \) in \( A \) is a finite sequence of states \( q_0 \ldots q_{n-1} \) where:

\begin{itemize}
  \item \( q_0 \in Q_0 \)
  \item \((q_i, a_i, q_{i+1}) \in \delta\) for all \( 0 \leq i \leq n \)
\end{itemize}
Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA, $w = a_0 \ldots a_n \in \Sigma^*$ a finite word.

A **run** for $w$ in $A$ is a finite sequence of states $q_0 \ldots q_{n-1}$ where:

- $q_0 \in Q_0$
- $(q_i, a_i, q_{i+1}) \in \delta$ for all $0 \leq i \leq n$

A run is **accepting** if it ends in a final state, e.g., $q_n \in F$. 

The word $w$ is **accepted** by $A$ if it has an accepting run.

The language of $A$, denoted $L(A)$, is the subset of $\Sigma^*$ it accepts:

$$L(A) = \{w \in \Sigma^* | \exists \text{ accepting run for } w \}$$

Every NFA can be converted to a DFA accepting the same language.
Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA, $w = a_0 \ldots a_n \in \Sigma^*$ a finite word.

A **run** for $w$ in $A$ is a finite sequence of states $q_0 \ldots q_{n-1}$ where:

- $q_0 \in Q_0$
- $(q_i, a_i, q_{i+1}) \in \delta$ for all $0 \leq i \leq n$

A run is **accepting** if it ends in a final state, e.g., $q_n \in F$

The word $w$ is **accepted** by $A$ if it has an accepting run.
Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA, $w = a_0 \ldots a_n \in \Sigma^*$ a finite word.

A run for $w$ in $A$ is a finite sequence of states $q_0 \ldots q_{n-1}$ where:

- $q_0 \in Q_0$
- $(q_i, a_i, q_{i+1}) \in \delta$ for all $0 \leq i \leq n$

A run is accepting if it ends in a final state, e.g., $q_n \in F$.

The word $w$ is accepted by $A$ if it has an accepting run.

The language of $A$, denoted $L(A)$, is the subset of $\Sigma^*$ it accepts:

$$L(A) = \{ w \in \Sigma^* \mid \exists \text{ accepting run for } w \}$$
Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA, $w = a_0 \ldots a_n \in \Sigma^*$ a finite word

A **run** for $w$ in $A$ is a finite sequence of states $q_0 \ldots q_{n-1}$ where:

- $q_0 \in Q_0$
- $(q_i, a_i, q_{i+1}) \in \delta$ for all $0 \leq i \leq n$

A run is **accepting** if it ends in a final state, e.g., $q_n \in F$

The word $w$ is **accepted** by $A$ if it has an accepting run

The **language of** $A$, denoted $L(A)$, is the subset of $\Sigma^*$ it accepts:

$$L(A) = \{ w \in \Sigma^* \mid \exists \text{ accepting run for } w \}$$

Every NFA can be converted to a DFA accepting the same language
Example

\[ Q = \{ q_0, q_1 \}, \Sigma = \{ a, b \}, Q_0 = \{ q_0 \}, F = \{ q_1 \} \]
Example

\[ Q = \{q_0, q_1\}, \Sigma = \{a, b\}, Q_0 = \{q_0\}, F = \{q_1\} \]

aaaaaaa is accepted

abababa is accepted

aaaaaaa is rejected

The language of this automaton is:

\[ L(A) = \text{contains arbitrary sequence of } a; b \text{ ending with } a \]
Example

$$Q = \{q_0, q_1\}, \Sigma = \{a, b\}, Q_0 = \{q_0\}, F = \{q_1\}$$

aaaaaaa is accepted
Example

\[ Q = \{q_0, q_1\}, \Sigma = \{a, b\}, Q_0 = \{q_0\}, F = \{q_1\} \]

aaaaaaa is accepted

abababaa is
Example

\[ Q = \{q_0, q_1\}, \Sigma = \{a, b\}, Q_0 = \{q_0\}, F = \{q_1\} \]

aaaaaa is accepted

abababa is accepted
Example

$$Q = \{q_0, q_1\}, \Sigma = \{a, b\}, Q_0 = \{q_0\}, F = \{q_1\}$$

aaaaaaaaa is **accepted**

abababaa is **accepted**

aaaaaaab is
Example

\[ Q = \{ q_0, q_1 \}, \Sigma = \{ a, b \}, Q_0 = \{ q_0 \}, F = \{ q_1 \} \]

aaaaaa is accepted
abababaa is accepted
aaaaaaab is rejected
Example

$Q = \{q_0, q_1\}, \Sigma = \{a, b\}, Q_0 = \{q_0\}, F = \{q_1\}$

aaaaaaaaaa is **accepted**

abababaa is **accepted**

aaaaaaab is **rejected**

The language of this automaton is:

$L(A) = \{w \mid w \text{contains arbitrary sequence of } a, b \text{ ending with } a\}$
When two NFAs represent the same language, we say they’re **equivalent**.
Equivalence & Emptiness

When two NFAs represent the same language, we say they’re equivalent.

A central issue in automata theory is the Emptiness Problem.
When two NFAs represent the same language, we say they’re equivalent.

A central issue in automata theory is the ***emptiness problem***

Given an NFA $A$, decide whether $L(A) = \emptyset$
When two NFAs represent the same language, we say they’re equivalent.

A central issue in automata theory is the **emptiness problem**

Given an NFA $A$, decide whether $L(A) = \emptyset$

This is equivalent to reachability:

$$L(A) \neq \emptyset \text{ iff } \exists q_0, q_f. q_f \text{ reachable from } q_0$$
When two NFAs represent the same language, we say they’re equivalent.

A central issue in automata theory is the emptiness problem.

Given an NFA $A$, decide whether $L(A) = \emptyset$.

This is equivalent to reachability:

$$L(A) \neq \emptyset \text{ iff } \exists q_0, q_f. q_f \text{ reachable from } q_0$$

This can be decided in $O(|A|)$ by depth-first search.
The languages recognized by NFAs are called **regular**
The languages recognized by NFAs are called **regular**

Regular languages contain **finite words**
The languages recognized by NFAs are called regular.

Regular languages contain finite words.

Regular languages are also represented by regular expressions:
The languages recognized by NFAs are called regular.

Regular languages contain finite words.

Regular languages are also represented by regular expressions:

- $\emptyset$ is the RE denoting the empty language.
Regular Languages

The languages recognized by NFAs are called regular

Regular languages contain finite words

Regular languages are also represented by regular expressions:

- $\emptyset$ is the RE denoting the empty language
- $\epsilon$ is the RE denoting the language with the empty word
Regular Languages

The languages recognized by NFAs are called **regular**

Regular languages contain **finite words**

Regular languages are also represented by **regular expressions**:

- $\emptyset$ is the RE denoting the empty language
- $\epsilon$ is the RE denoting the language with the empty word
- If $E$ is an RE, then $E^*$ denotes the finite repetitions of $E$
The languages recognized by NFAs are called **regular**

Regular languages contain **finite words**

Regular languages are also represented by **regular expressions**:

- $\emptyset$ is the RE denoting the empty language
- $\epsilon$ is the RE denoting the language with the empty word
- If $E$ is an RE, then $E^*$ denotes the finite repetitions of $E$
- If $E_1, E_2$ are REs, then $E_1 + E_2$ denotes union of their languages
The languages recognized by NFAs are called **regular**

Regular languages contain **finite words**

Regular languages are also represented by **regular expressions**:

- \( \emptyset \) is the RE denoting the empty language
- \( \epsilon \) is the RE denoting the language with the empty word
- If \( E \) is an RE, then \( E^* \) denotes the finite repetitions of \( E \)
- If \( E_1, E_2 \) are REs, then \( E_1 + E_2 \) denotes union of their languages
- If \( E_1, E_2 \) are REs, then \( E_1E_2 \) denotes their concatenation
Example

The language of this automaton is:

$$L(A) = (a + b)a$$
The language of this automaton is:

\[ L(A) = (a + b)^* a \]
Properties of Regular Languages

The syntax of regular expressions implies several useful facts
The syntax of regular expressions implies several useful facts

- $E^*$: closed under finite repetition
- $E_1 + E_2$: closed under union
- $E_1 E_2$: closed under concatenation

They're also closed under intersection and complement.

If $L; L_1; L_2$ are regular languages, so are $L_1 \setminus L_2$; $n L$.
Properties of Regular Languages

The syntax of regular expressions implies several useful facts

- $E^*$: closed under finite repetition

- $E_1 + E_2$: closed under union

- $E_1 E_2$: closed under concatenation

They're also closed under intersection and complement

If $L_1, L_2$ are regular languages, so are $L_1 \setminus L_2$ and $L_1 \cap L_2$.
Properties of Regular Languages

The syntax of regular expressions implies several useful facts

- $E^*$: closed under finite repetition
- $E_1 + E_2$: closed under union
- $E_1 \cdot E_2$: closed under concatenation

They're also closed under intersection and complement.
The syntax of regular expressions implies several useful facts

- $E^*$: closed under finite repetition
- $E_1 + E_2$: closed under union
Properties of Regular Languages

The syntax of regular expressions implies several useful facts

- \( E^* \): closed under finite repetition
- \( E_1 + E_2 \): closed under union
- \( E_1 E_2 \):
The syntax of regular expressions implies several useful facts

- $E^*$: closed under finite repetition
- $E_1 + E_2$: closed under union
- $E_1E_2$: closed under concatenation
The syntax of regular expressions implies several useful facts:

- $E^*$: closed under finite repetition
- $E_1 + E_2$: closed under union
- $E_1 E_2$: closed under concatenation

They’re also closed under intersection and complement:

If $L, L_1, L_2$ are regular languages, so are $L_1 \cap L_2, \Sigma^* \setminus L$.
The syntax of regular expressions implies several useful facts:

- $E^*$: closed under finite repetition
- $E_1 + E_2$: closed under union
- $E_1 E_2$: closed under concatenation

They’re also closed under intersection and complement:

If $L, L_1, L_2$ are regular languages, so are $L_1 \cap L_2, \Sigma^* \setminus L$.

Given NFAs representing a language, we can construct NFAs corresponding to the application of these operations.
NFAs and REs describe languages containing finite words
Languages of Infinite Words

NFAs and REs describe languages containing finite words

Our transition systems describe infinite behaviors
Languages of Infinite Words

NFAs and REs describe languages containing finite words

Our transition systems describe infinite behaviors

We’ll describe such behaviors using $\omega$-regular languages
Languages of Infinite Words

NFAs and REs describe languages containing finite words

Our transition systems describe infinite behaviors

We’ll describe such behaviors using \( \omega \)-regular languages

These can be described by \( \omega \)-regular expressions of the form:

\[
E_1 F_1^\omega + \cdots + E_n F_n^\omega
\]
Languages of Infinite Words

NFAs and REs describe languages containing finite words

Our transition systems describe infinite behaviors

We’ll describe such behaviors using $\omega$-regular languages

These can be described by $\omega$-regular expressions of the form:

$$E_1 F_1^\omega + \cdots + E_n F_n^\omega$$

- $E_i$ and $F_i$ are regular expressions, $\epsilon \notin L(F_i)$
Languages of Infinite Words

NFAs and REs describe languages containing finite words

Our transition systems describe infinite behaviors

We’ll describe such behaviors using \( \omega \)-regular languages

These can be described by \( \omega \)-regular expressions of the form:

\[
E_1 F_1^\omega + \cdots + E_n F_n^\omega
\]

- \( E_i \) and \( F_i \) are regular expressions, \( \epsilon \notin L(F_i) \)
- Union and concatenation work as they did before
Languages of Infinite Words

NFAs and REs describe languages containing finite words.

Our transition systems describe infinite behaviors.

We’ll describe such behaviors using $\omega$-regular languages.

These can be described by $\omega$-regular expressions of the form:

$$E_1 F_1^\omega + \cdots + E_n F_n^\omega$$

- $E_i$ and $F_i$ are regular expressions, $\epsilon \not\in L(F_i)$
- Union and concatenation work as they did before
- $\omega$ denotes infinite repetition
Languages of Infinite Words

NFAs and REs describe languages containing finite words.

Our transition systems describe infinite behaviors.

We’ll describe such behaviors using \( \omega \)-regular languages.

These can be described by \( \omega \)-regular expressions of the form:

\[
E_1 F_1^\omega + \cdots + E_n F_n^\omega
\]

- \( E_i \) and \( F_i \) are regular expressions, \( \epsilon \notin L(F_i) \)
- Union and concatenation work as they did before
- \( \omega \) denotes \textit{infinite repetition}
- Like Kleene \( * \), but ad infinitum
For a word $ab$, we know that $(ab)^*$ denotes the set
\[ \{ab, abab, ababab, \ldots\} \]
For a word $ab$, we know that $(ab)^*$ denotes the set
\[ \{ab, abab, ababab, \ldots \} \]

What does $(ab)^\omega$ denote?
For a word \( ab \), we know that \((ab)^*\) denotes the set 
\[
\{ab, abab, ababab, \ldots\}
\]

What does \((ab)^\omega\) denote?
\[
(ab)^\omega = \{ababababab \ldots\}
\]
Infinite Repetition $\omega$

For a word $ab$, we know that $(ab)^*$ denotes the set
$$\{ab, abab, ababab, \ldots\}$$

What does $(ab)^\omega$ denote?
$$(ab)^\omega = \{ababababab \ldots\}$$

What about the empty word $\epsilon$?
For a word $ab$, we know that $(ab)^*$ denotes the set

$$\{ab, abab, ababab, \ldots\}$$

What does $(ab)^\omega$ denote?

$$(ab)^\omega = \{ababababab \ldots\}$$

What about the empty word $\epsilon$? $\epsilon^\omega = \epsilon$
Infinite Repetition $\omega$

For a word $ab$, we know that $(ab)^*$ denotes the set
\[ \{ab, abab, ababab, \ldots\} \]

What does $(ab)^\omega$ denote?
\[ (ab)^\omega = \{ababababab \ldots\} \]

What about the empty word $\epsilon$? $\epsilon^\omega = \epsilon$

Given an infinite word $w$, $w^\omega =$
For a word $ab$, we know that $(ab)^*$ denotes the set
\[ \{ab, abab, ababab, \ldots \} \]

What does $(ab)^\omega$ denote?

\[ (ab)^\omega = \{ababababab \ldots \} \]

What about the empty word $\epsilon$? $\epsilon^\omega = \epsilon$

Given an infinite word $w$, $w^\omega = w$
For a word \( ab \), we know that \((ab)^*\) denotes the set
\[
\{ab, abab, ababab, \ldots\}
\]

What does \((ab)^\omega\) denote?
\[
(ab)^\omega = \{ababababab \ldots\}
\]

What about the empty word \( \epsilon \)? \( \epsilon^\omega = \epsilon \)

Given an infinite word \( w \), \( w^\omega = w \)

We’ll lift \( \omega \) to finite languages \( L \subseteq \Sigma^* \) as well:
\[
L^\omega = \{w_1w_2w_3 \ldots \mid w_i \in L\}\]
Infinite Repetition $\omega$

For a word $ab$, we know that $(ab)^*$ denotes the set
$$\{ab, abab, ababab, \ldots\}$$

What does $(ab)^\omega$ denote?
$$ (ab)^\omega = \{ababababab\ldots\} $$

What about the empty word $\epsilon$? $\epsilon^\omega = \epsilon$

Given an infinite word $w$, $w^\omega = w$

We’ll lift $\omega$ to finite languages $L \subseteq \Sigma^*$ as well:
$$ L^\omega = \{w_1w_2w_3\ldots \mid w_i \in L\} $$

If $L$ doesn’t contain $\epsilon$, $L^\omega$ is an infinite language
How do we write mutual exclusion as an $\omega$-regular expression?
Example

How do we write mutual exclusion as an $\omega$-regular expression?

Recall, this was the safety property (invariant):

$$G \neg crit_1 \lor \neg crit_2$$
Example

How do we write mutual exclusion as an $\omega$-regular expression?

Recall, this was the safety property (invariant):

$$G \neg crit_1 \lor \neg crit_2$$

First, we need to define the alphabet
How do we write mutual exclusion as an $\omega$-regular expression?

Recall, this was the safety property (invariant):

$$G \neg crit_1 \lor \neg crit_2$$

First, we need to define the alphabet

- Need to reason about the set of all propositions that might hold
How do we write mutual exclusion as an \( \omega \)-regular expression?

Recall, this was the safety property (invariant):

\[
\mathbf{G} \neg crit_1 \lor \neg crit_2
\]

First, we need to define the alphabet

- Need to reason about the set of all propositions that might hold
- Setting \( \Sigma = 2^P \) (the atomic propositions) seems reasonable
Example

How do we write mutual exclusion as an $\omega$-regular expression?

Recall, this was the safety property (invariant):

$$G \neg \text{crit}_1 \lor \neg \text{crit}_2$$

First, we need to define the alphabet

- Need to reason about the set of all propositions that might hold
- Setting $\Sigma = 2^P$ (the atomic propositions) seems reasonable
- In this case, $P = \{\text{crit}_1, \text{crit}_2\}$
Example

How do we write mutual exclusion as an $\omega$-regular expression?

Recall, this was the safety property (invariant):

$$G \neg crit_1 \lor \neg crit_2$$

First, we need to define the alphabet

- Need to reason about the set of all propositions that might hold
- Setting $\Sigma = 2^P$ (the atomic propositions) seems reasonable
- In this case, $P = \{crit_1, crit_2\}$

Then symbols are $\emptyset$, $\{crit_1\}$, $\{crit_1, crit_2\}$, $\ldots$
Example

How do we write mutual exclusion as an $\omega$-regular expression?

Recall, this was the safety property (invariant):

$$G \neg crit_1 \lor \neg crit_2$$

First, we need to define the alphabet

- Need to reason about the set of all propositions that might hold
- Setting $\Sigma = 2^P$ (the atomic propositions) seems reasonable
- In this case, $P = \{crit_1, crit_2\}$

Then symbols are $\emptyset, \{crit_1\}, \{crit_1, crit_2\}, \ldots$

Our expression is:
How do we write mutual exclusion as an $\omega$-regular expression?

Recall, this was the safety property (invariant):

$$G \neg crit_1 \lor \neg crit_2$$

First, we need to define the alphabet

- Need to reason about the set of all propositions that might hold
- Setting $\Sigma = 2^P$ (the atomic propositions) seems reasonable
- In this case, $P = \{crit_1, crit_2\}$

Then symbols are $\emptyset, \{crit_1\}, \{crit_1, crit_2\}, \ldots$

Our expression is:

$$\left(\emptyset + \{crit_1\} + \{crit_2\}\right)^\omega$$
Automata on Infinite Words

NFA : Regular ::

Nondeterministic Buchi Automaton (NBA)

\[ A_{\text{NBA}} = (\Sigma; Q; Q_0; F; \delta) \]

- \( \Sigma \) is an alphabet
- \( Q \) is a finite set of states
- \( Q_0 \subseteq Q \) is the set of initial states
- \( F \subseteq Q \) is the set of accepting states
- \( \delta : Q \times \Sigma \rightarrow 2^Q \) is the transition function

The "syntax" is the same as NFAs; obviously the semantics is different.

\[ : \omega \text{-Regular} \]
Nondeterministic Buchi Automaton (NBA)

A NBA $M$ is a tuple $(\Sigma, Q, Q_0, F, \delta)$, where:

- $\Sigma$ is an alphabet
- $Q$ is a finite set of states
- $Q_0 \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is the set of accepting states
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition function
NFA : Regular :: Non deterministic Buchi Automata : $\omega$-Regular

**Non deterministic Buchi Automaton (NBA)**

A NBA $M$ is a tuple $(\Sigma, Q, Q_0, F, \delta)$, where:

- $\Sigma$ is an alphabet
- $Q$ is a finite set of states
- $Q_0 \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is the set of accepting states
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition function

The “syntax” is the same as NFAs; obviously the semantics is different
Let $w = a_0 a_1 \ldots$ be an infinite word in $\Sigma^\omega$
Let $w = a_0a_1 \ldots$ be an infinite word in $\Sigma^\omega$

A run for $w$ in $A$ is an infinite sequence of states $q_0 \ldots q_{n-1}$ where:
Let $w = a_0a_1 \ldots$ be an infinite word in $\Sigma^\omega$

A **run** for $w$ in $A$ is an infinite sequence of states $q_0 \ldots q_{n-1}$ where:

- $q_0 \in Q_0$
Let \( w = a_0a_1 \ldots \) be an infinite word in \( \Sigma^\omega \)

A **run** for \( w \) in \( A \) is an infinite sequence of states \( q_0 \ldots q_{n-1} \) where:

- \( q_0 \in Q_0 \)
- \( (q_i, a_i, q_{i+1}) \in \delta \) for all \( 0 \leq i \leq n \)
Let $w = a_0a_1 \ldots$ be an infinite word in $\Sigma^\omega$

A run for $w$ in $A$ is an infinite sequence of states $q_0 \ldots q_{n-1}$ where:

▶ $q_0 \in Q_0$

▶ $(q_i, a_i, q_{i+1}) \in \delta$ for all $0 \leq i \leq n$

A run is accepting if $q_i \in F$ for infinitely many indices $i$:

$$\{ q \in Q \mid \forall i \geq 0, \exists j \geq i. q_j = q \} \cap F \neq \emptyset$$
Let $w = a_0 a_1 \ldots$ be an infinite word in $\Sigma^\omega$

A run for $w$ in $A$ is an infinite sequence of states $q_0 \ldots q_{n-1}$ where:

- $q_0 \in Q_0$
- $(q_i, a_i, q_{i+1}) \in \delta$ for all $0 \leq i \leq n$

A run is accepting if $q_i \in F$ for infinitely many indices $i$:

$$\{q \in Q \mid \forall i \geq 0, \exists j \geq i. q_j = q \} \cap F \neq \emptyset$$

A language is $\omega$-regular language iff it is recognizable by an NBA
What runs does the word \( c \) have?

What about \( ab \)?

Is \((cabb)\) accepted? What is its run?
What runs does the word $c^\omega$ have?
What runs does the word $c^\omega$ have?

$q_1^\omega$
Example

What runs does the word \( c^\omega \) have?

\[ q_1^\omega \]

What about \( ab^\omega \)?
What runs does the word $c^\omega$ have?

$q_1^\omega$

What about $ab^\omega$?

$q_1 q_2 q_3^\omega$
What runs does the word $c^\omega$ have?

$q_1^\omega$

What about $ab^\omega$?

$q_1 q_2 q_3^\omega$

Is $(cabb)^\omega$ accepted?
What runs does the word $c^\omega$ have?

$q_1^\omega$

What about $ab^\omega$?

$q_1 q_2 q_3^\omega$

Is $(cabb)^\omega$ accepted? What is its run?
Example

What runs does the word $c^\omega$ have?

$q_1^\omega$

What about $ab^\omega$?

$q_1 q_2 q_3^\omega$

Is $(cabb)^\omega$ accepted? What is its run?

$(q_1 q_1 q_2 q_3)^\omega$
What $\omega$-regular expression does this accept?
What $\omega$-regular expression does this accept?

$$(a + b)^* b^\omega$$
What $\omega$-regular expression does this accept?

$$(a + b)^* b^\omega$$

What does it mean?
What \( \omega \)-regular expression does this accept?
\[
(a + b)^* b^\omega
\]

What does it mean? \( a \) occurs only finitely many times
Example: No send after read

Suppose we want to describe a safety property:

*The client must never send a packet after reading a classified file*
Example: No send after read

Suppose we want to describe a safety property:

*The client must never send a packet after reading a classified file*

Let \( P = \{\text{Send}, \text{Read}\} \)
Example: No send after read

Suppose we want to describe a safety property:

*The client must never send a packet after reading a classified file*

Let $P = \{\text{Send}, \text{Read}\}$

Technically, our $\Sigma$ should be: $\{\emptyset\}, \{\text{Send}\}, \{\text{Read}\}, \{\text{Send}, \text{Read}\}$
Suppose we want to describe a safety property:

_The client must never send a packet after reading a classified file_

Let $P = \{\text{Send}, \text{Read}\}$

Technically, our $\Sigma$ should be: $\emptyset, \{\text{Send}\}, \{\text{Read}\}, \{\text{Send, Read}\}$

We’ll be a bit sloppy, and let $\Sigma$ be formulas over $\text{Send, Read}$
Example: No send after read

Then we can write an $\omega$-regular expression:
Then we can write an $\omega$-regular expression:

$$(\neg \text{Read})^\omega + (\text{Read})(\neg \text{Send})^\omega$$
Example: No send after read

Then we can write an $\omega$-regular expression:

$$(\neg Read)^{\omega} + (Read)(\neg Send)^{\omega}$$

And we can encode this as an NBA:
Example: No send after read

Then we can write an $\omega$-regular expression:

$$(\neg \text{Read})^\omega + (\text{Read})(\neg \text{Send})^\omega$$

And we can encode this as an NBA:

$$\begin{array}{c}
\rightarrow q_0 \\
q_1
\end{array}$$
Example: No send after read

Then we can write an \( \omega \)-regular expression:

\[
(\neg \text{Read})^\omega + (\text{Read})(\neg \text{Send})^\omega
\]

And we can encode this as an NBA:
Example: No send after read

Then we can write an $\omega$-regular expression:

$$(\neg Read)^\omega + (Read)(\neg Send)^\omega$$

And we can encode this as an NBA:
Example: No send after read

Then we can write an $\omega$-regular expression:

$$(\neg \text{Read})^\omega + (\text{Read})(\neg \text{Send})^\omega$$

And we can encode this as an NBA:
Then we can write an $\omega$-regular expression:

$$\lnot \text{Read}^\omega + \text{Read} \cdot \lnot \text{Send}^\omega$$

And we can encode this as an NBA:
Then we can write an $\omega$-regular expression:

$$(\neg \text{Read})^\omega + (\text{Read})(\neg \text{Send})^\omega$$

And we can encode this as an NBA:
Example: No send after read

Then we can write an $\omega$-regular expression:

$$(\neg \text{Read})^\omega + (\text{Read})(\neg \text{Send})^\omega$$

And we can encode this as an NBA:
Example: No send after read

Then we can write an $\omega$-regular expression:

$$(\neg \text{Read})^\omega + (\text{Read})(\neg \text{Send})^\omega$$

And we can encode this as an NBA:
Example: Partial correctness

Now a more complicated example:

*Whenever the precondition is satisfied and the program terminates, the postcondition must be satisfied*
Now a more complicated example:

Whenever the precondition is satisfied and the program terminates, the postcondition must be satisfied

Our alphabet: formulas over \{Pre, Post, Done\}
Now a more complicated example:

Whenever the precondition is satisfied and the program terminates, the postcondition must be satisfied

Our alphabet: formulas over \{Pre, Post, Done\}

What’s our $\omega$-regular expression?
Now a more complicated example:

*Whenever the precondition is satisfied and the program terminates, the postcondition must be satisfied*

Our alphabet: formulas over \{Pre, Post, Done\}

What’s our \(\omega\)-regular expression?

\[
\neg Pre \omega + Pre \neg Done \omega + Pre \neg Done^* (Done \land Post) \omega
\]
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg \text{Pretrue}^\omega + \text{Pre}\neg \text{Done}^\omega + \text{Pre}\neg \text{Done}^* (\text{Done} \land \text{Post})^\omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pre_{true}^\omega + Pre_{\neg Done}^\omega + Pre_{\neg Done}^* (Done \land Post)^\omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pre\text{true}^\omega + Pre\neg Done^\omega + Pre\neg Done^*(Done \land Post)^\omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our \( \omega \)-regular expression?

\[ \neg Pre true^\omega + Pre \neg Done^\omega + Pre \neg Done^* (Done \wedge Post)^\omega \]

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pre \omega + Pre \neg Done^\omega + Pre \neg Done^* (Done \land Post)^\omega$$

And a corresponding NBA:
What’s our $\omega$-regular expression?

$$\neg Pre_{true}\omega + Pre\neg Done\omega + Pre\neg Done^*(Done \land Post)\omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pre \omega + Pre \neg Done^\omega + Pre \neg Done^* (Done \land Post)^\omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pre \omega + Pre \neg Done \omega + Pre \neg Done^* (Done \land Post) \omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pre true^\omega + Pre \neg Done^\omega + Pre \neg Done^*(Done \land Post)^\omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pre true^\omega + Pre \neg Done^\omega + Pre \neg Done^* (Done \land Post)^\omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg \text{Pre} \text{true}^\omega + \text{Pre} \neg \text{Done}^\omega + \text{Pre} \neg \text{Done}^* (\text{Done} \land \text{Post})^\omega$$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pre_{true}^\omega + Pre \neg Done^\omega + Pre \neg Done^* (Done \land Post)^\omega$$

And a corresponding NBA:
What’s our $\omega$-regular expression?

$\neg Pretrue^\omega + Pre\neg Done^\omega + Pre\neg Done^*(Done \land Post)^\omega$

And a corresponding NBA:
Example: Partial correctness

What’s our $\omega$-regular expression?

$$\neg Pretrue^\omega + Pre\neg Done^\omega + Pre\neg Done^*(Done \land Post)^\omega$$

And a corresponding NBA:
Like regular languages, $\omega$-regular enjoy closure properties
Like regular languages, \( \omega \)-regular enjoy closure properties

- Union
Like regular languages, $\omega$-regular enjoy closure properties

- Union
- Intersection

Emptiness is decidable in linear time

This is important for model checking, as we'll see
Like regular languages, \( \omega \)-regular enjoy closure properties

- Union
- Intersection
- Complement
Like regular languages, \(\omega\)-regular enjoy closure properties

- Union
- Intersection
- Complement
- Each of these corresponds to operations on NBA
Like regular languages, $\omega$-regular enjoy closure properties

- Union
- Intersection
- Complement
- Each of these corresponds to operations on NBA

But these aren’t necessarily the same operations as for NFAs
Like regular languages, \( \omega \)-regular enjoy closure properties

- Union
- Intersection
- Complement
- Each of these corresponds to operations on NBA

But these aren’t necessarily the same operations as for NFAs

- E.g., for intersection, word needs to go through both sets of accepting states infinitely often
Like regular languages, \(\omega\)-regular enjoy closure properties

- Union
- Intersection
- Complement
- Each of these corresponds to operations on NBA

But these aren’t necessarily the same operations as for NFAs

- E.g., for intersection, word needs to go through both sets of accepting states infinitely often
- Complement is tricky: NBAs aren’t closed under determinization
Like regular languages, \( \omega \)-regular enjoy closure properties

- Union
- Intersection
- Complement
- Each of these corresponds to operations on NBA

But these aren’t necessarily the same operations as for NFAs

- E.g., for intersection, word needs to go through both sets of accepting states infinitely often
- Complement is tricky: NBAs aren’t closed under determinization

Emptiness is decidable in linear time
Like regular languages, \( \omega \)-regular enjoy closure properties

- Union
- Intersection
- Complement
- Each of these corresponds to operations on NBA

But these aren’t necessarily the same operations as for NFAs

- E.g., for intersection, word needs to go through both sets of accepting states infinitely often
- Complement is tricky: NBAs aren’t closed under determinization

Emptiness is decidable in linear time

- This is important for model checking, as we’ll see
Let $A$ be an NBA representing some computation.
Let $A$ be an NBA representing some computation

Let $A_\phi$ be an NBA representing the specification
Let $A$ be an NBA representing some computation

Let $A_\phi$ be an NBA representing the specification

$\quad \Rightarrow$ $A_\phi$ describes the **allowed traces**
Let $A$ be an NBA representing some computation

Let $A_{\phi}$ be an NBA representing the specification
- $A_{\phi}$ describes the **allowed traces**
- Its language corresponds to “good” computations
Let $A$ be an NBA representing some computation

Let $A_\phi$ be an NBA representing the specification

- $A_\phi$ describes the allowed traces
- Its language corresponds to “good” computations

Then $A$ satisfies the specification $A_\phi$ exactly when:
Let $A$ be an NBA representing some computation

Let $A_\phi$ be an NBA representing the specification
  ▶ $A_\phi$ describes the \textbf{allowed traces}
  ▶ Its language corresponds to “good” computations

Then $A$ satisfies the specification $A_\phi$ exactly when:

$$L(A) \subseteq L(A_\phi)$$
Let $A$ be an NBA representing some computation

Let $A_\phi$ be an NBA representing the specification
- $A_\phi$ describes the **allowed traces**
- Its language corresponds to “good” computations

Then $A$ satisfies the specification $A_\phi$ exactly when:

$$L(A) \subseteq L(A_\phi)$$

The set of traces in $A$ is contained in the set of “good” computations
How do we check that $L(A) \subseteq L(A_\phi)$?

$L(A) \subseteq L(S) \iff L(A) \cap L(A_\phi) = \emptyset$

In other words, $A$ satisfies $A_\phi$ if none of its traces is prohibited.
How do we check that $L(A) \subseteq L(A_\phi)$?

$$L(A) \subseteq L(S) \iff L(A) \cap \overline{L(A_\phi)} = \emptyset$$

In other words, $A$ satisfies $A_\phi$ if none of its traces is prohibited

We can use closed NBA operations + emptiness check to do MC.
How do we check that $L(A) \subseteq L(A_\phi)$?

$L(A) \subseteq L(S) \iff L(A) \cap \overline{L(A_\phi)} = \emptyset$

In other words, $A$ satisfies $A_\phi$ if none of its traces is prohibited

We can use closed NBA operations + emptiness check to do MC

What about counterexamples?
How do we check that $L(A) \subseteq L(A_\phi)$?

$$L(A) \subseteq L(S) \iff L(A) \cap \overline{L(A_\phi)} = \emptyset$$

In other words, $A$ satisfies $A_\phi$ if none of its traces is prohibited.

We can use closed NBA operations + emptiness check to do MC.

What about counterexamples?

- $L(A) \cap \overline{L(A_\phi)} \neq \emptyset$ gives an $\omega$-regular language.
How do we check that $L(A) \subseteq L(A_\phi)$?

$$L(A) \subseteq L(S) \iff L(A) \cap \overline{L(A_\phi)} = \emptyset$$

In other words, $A$ satisfies $A_\phi$ if none of its traces is prohibited.

We can use closed NBA operations + emptiness check to do MC.

What about counterexamples?

- $L(A) \cap \overline{L(A_\phi)} \neq \emptyset$ gives an $\omega$-regular language.
- Any word in this language is a prohibited trace.
Automata-Theoretic Model Checking

How do we check that $L(A) \subseteq L(A_\phi)$?

$$L(A) \subseteq L(S) \iff L(A) \cap \overline{L(A_\phi)} = \emptyset$$

In other words, $A$ satisfies $A_\phi$ if none of its traces is prohibited

We can use closed NBA operations + emptiness check to do MC

What about counterexamples?

- $L(A) \cap \overline{L(A_\phi)} \neq \emptyset$ gives an $\omega$-regular language
- Any word in this language is a prohibited trace
- We pick an arbitrary word, find an appropriate prefix
We would like to solve the LTL model checking problem:

Given a Kripke structure $M$ and LTL formula $\phi$, decide whether $M, \pi \models \phi$ for each $\pi$ starting in an initial state.
We would like to solve the LTL model checking problem:

*Given a Kripke structure $M$ and LTL formula $\phi$, decide whether $M, \pi \models \phi$ for each $\pi$ starting in an initial state.*

To do this, we’ll need to represent $M$ and $\phi$ as NBAs.
We would like to solve the LTL model checking problem:

*Given a Kripke structure $M$ and LTL formula $\phi$, decide whether $M, \pi \models \phi$ for each $\pi$ starting in an initial state.*

To do this, we’ll need to represent $M$ and $\phi$ as NBAs.

Intuitively, this should pose no problem.
We would like to solve the LTL model checking problem:

Given a Kripke structure $M$ and LTL formula $\phi$, decide whether $M, \pi \models \phi$ for each $\pi$ starting in an initial state.

To do this, we’ll need to represent $M$ and $\phi$ as NBAs.

Intuitively, this should pose no problem:
- $M$ is a nondeterministic system over infinite paths.
We would like to solve the LTL model checking problem:

*Given a Kripke structure $M$ and LTL formula $\phi$, decide whether $M, \pi \models \phi$ for each $\pi$ starting in an initial state.*

To do this, we’ll need to represent $M$ and $\phi$ as NBAs

Intuitively, this should pose no problem

- $M$ is a nondeterministic system over infinite paths
- We’ve seen NBAs that “look like” LTL properties
We would like to solve the LTL model checking problem:

Given a Kripke structure $M$ and LTL formula $\phi$, decide whether $M, \pi \models \phi$ for each $\pi$ starting in an initial state.

To do this, we’ll need to represent $M$ and $\phi$ as NBAs

Intuitively, this should pose no problem

- $M$ is a nondeterministic system over infinite paths
- We’ve seen NBAs that “look like” LTL properties

However, this is the source of complexity in LTL model checking
**Kripke structure**

A Kripke structure $M = (P, S, I, L, R)$ consists of:

- Set of *atomic propositions* $P$
- States $S$
- Initial states $I \subseteq S$
- Labeling $L : S \mapsto 2^P$
- Transition relation $R \subseteq S \times S$
A Kripke structure $M = (P, S, I, L, R)$ consists of:

- Set of atomic propositions $P$
- States $S$
- Initial states $I \subseteq S$
- Labeling $L : S \rightarrow 2^P$
- Transition relation $R \subseteq S \times S$

Recalling this definition, the main difference seems to be:

- Transitions have no labels
- The "natural" alphabet $P$ labels states, not transitions
- There are no accepting states
### Kripke structure

A Kripke structure \( M = (P, S, I, L, R) \) consists of:

- Set of *atomic propositions* \( P \)
- States \( S \)
- Initial states \( I \subseteq S \)
- Labeling \( L : S \mapsto 2^P \)
- Transition relation \( R \subseteq S \times S \)

Recalling this definition, the main difference seems to be:

- Transitions have no labels
Recalling this definition, the main difference seems to be:

- Transitions have no labels
- The “natural” alphabet \( P \) labels states, not transitions
A Kripke structure $M = (P, S, I, L, R)$ consists of:

- Set of *atomic propositions* $P$
- States $S$
- Initial states $I \subseteq S$
- Labeling $L : S \mapsto 2^P$
- Transition relation $R \subseteq S \times S$

Recalling this definition, the main difference seems to be:

- Transitions have no labels
- The “natural” alphabet $P$ labels states, not transitions
- There are no accepting states
We’re given a Kripke structure
\[ M = (P, S, I, L, R) \]

We want NBA \( A = (\Sigma, Q, Q_0, F, \delta) \)
where:

\[ \overset{\text{▶}}{\delta} (q; q') \overset{\text{▶}}{\in} P \]

if:
1. \( (q; q') \overset{\text{▶}}{\in} R \) and \( L(q') \overset{\text{▶}}{=} q \)
2. \( q = \ell \) and \( L(q') \overset{\text{▶}}{=} q' \)

So \( Q = S \)

\[ \text{distinguished initial state} \]

What about \( F \)?

Every execution “accepted” by the system, so \( F = Q \)
**Kripke Structure \(\mapsto\) NBA**

We’re given a Kripke structure
\[ M = (P, S, I, L, R) \]

We want NBA \(A = (\Sigma, Q, Q_0, F, \delta)\)

where:
- \(\Sigma = 2^P\)

![Diagram of NBA with states and transitions]
We’re given a Kripke structure
\[ M = (P, S, I, L, R) \]

We want NBA \( A = (\Sigma, Q, Q_0, F, \delta) \) where:

- \( \Sigma = 2^P \)
- \( (q, \alpha, q') \in \delta \) if:
  1. \( (q, q') \in R \) and \( L(q') = \alpha \)
We’re given a Kripke structure
\[ M = (P, S, I, L, R) \]

We want NBA \( A = (\Sigma, Q, Q_0, F, \delta) \) where:

- \( \Sigma = 2^P \)
- \( (q, \alpha, q') \in \delta \) if:
  1. \( (q, q') \in R \) and \( L(q') = \alpha \)
  2. \( q = \ell, q' \in I \) and \( L(q') = \alpha \)
We’re given a Kripke structure
\[ M = (P, S, I, L, R) \]

We want NBA \( A = (\Sigma, Q, Q_0, F, \delta) \) where:

- \( \Sigma = 2^P \)
- \( (q, \alpha, q') \in \delta \) if:
  1. \( (q, q') \in R \) and \( L(q') = \alpha \)
  2. \( q = \ell, q' \in I \) and \( L(q') = \alpha \)
- So \( Q = S \cup \{\ell\} \), a distinguished initial state
We’re given a Kripke structure 
\[ M = (P, S, I, L, R) \]

We want NBA 
\[ A = (\Sigma, Q, Q_0, F, \delta) \]

where:

- \( \Sigma = 2^P \)
- \( (q, \alpha, q') \in \delta \) if:
  1. \( (q, q') \in R \) and \( L(q') = \alpha \)
  2. \( q = \ell, q' \in I \) and \( L(q') = \alpha \)
- So \( Q = S \cup \{\ell\} \), a distinguished initial state
- What about \( F \)?
We’re given a Kripke structure 
\[ M = (P, S, I, L, R) \]

We want NBA \( A = (\Sigma, Q, Q_0, F, \delta) \) where:

- \( \Sigma = 2^P \)
- \( (q, \alpha, q') \in \delta \) if:
  1. \( (q, q') \in R \) and \( L(q') = \alpha \)
  2. \( q = \ell, q' \in I \) and \( L(q') = \alpha \)

- So \( Q = S \cup \{\ell\} \), a distinguished initial state
- What about \( F \)?
- Every execution “accepted” by the system, so \( F = Q \)
We’re given a Kripke structure
\[ M = (P, S, I, L, R) \]

We want NBA \( A = (\Sigma, Q, Q_0, F, \delta) \) where:

- \( \Sigma = 2^P \)
- \( (q, \alpha, q') \in \delta \) if:
  1. \( (q, q') \in R \) and \( L(q') = \alpha \)
  2. \( q = \ell, q' \in I \) and \( L(q') = \alpha \)
- So \( Q = S \cup \{\ell\} \), a distinguished initial state
- What about \( F \)?
- Every execution “accepted” by the system, so \( F = Q \)
The final piece: converting LTL to NBA
The final piece: converting LTL to NBA

The “leaves” of LTL formulas are propositional formulas over $P$.
The final piece: converting LTL to NBA

The “leaves” of LTL formulas are propositional formulas over $P$

\[
\begin{align*}
G\ F\ (p \lor q) & \quad G\ (\neg c_1 \lor \neg c_2) & \quad G\ (p \rightarrow F\ q)
\end{align*}
\]
The final piece: converting LTL to NBA

The “leaves” of LTL formulas are propositional formulas over $P$

$$G F (p \lor q) \quad G (\neg c_1 \lor \neg c_2) \quad G (p \rightarrow F q)$$

We’ll use formulas over $P$ to represent alphabet symbolically.
NBA for LTL Formulas

The final piece: converting LTL to NBA

The “leaves” of LTL formulas are propositional formulas over $P$

$$\mathbf{G} \mathbf{F} (p \lor q) \quad \mathbf{G} (\neg c_1 \lor \neg c_2) \quad \mathbf{G} (p \rightarrow \mathbf{F} q)$$

We’ll use formulas over $P$ to represent alphabet symbolically

For example, if we have a transition:

```
\begin{tikzpicture}
  \node (q0) at (0,0) {$q_0$};
  \node (q1) at (1,0) {$q_1$};
  \draw[->] (q0) edge node [above] {$p_0 \lor p_1$} (q1);
\end{tikzpicture}
```
The final piece: converting LTL to NBA

The “leaves” of LTL formulas are propositional formulas over $P$

\[
\mathbf{G} \mathbf{F} (p \lor q) \quad \mathbf{G} (\neg c_1 \lor \neg c_2) \quad \mathbf{G} (p \rightarrow \mathbf{F} q)
\]

We’ll use formulas over $P$ to represent alphabet symbolically

For example, if we have a transition:

Then this is shorthand for:
LTL to NBA: Example ($\mathbf{X}$ operator)

Let’s start with the next operator

\[ \mathbf{X} p \rightarrow \text{any} ightarrow p \rightarrow \text{any} \rightarrow \text{any} \rightarrow \text{any} \rightarrow \cdots \]
LTL to NBA: Example ($\mathbf{X}$ operator)

Let’s start with the next operator

$\mathbf{X} \ p$

What is the corresponding NBA?
LTL to NBA: Example (X operator)

Let’s start with the next operator

\[ X \ p \]

What is the corresponding NBA?
LTL to NBA: Example ($\mathbf{X}$ operator)

Let’s start with the next operator

\[ \mathbf{X} p \]

What is the corresponding NBA?

\[ \mathbf{X} p \]

- It doesn’t matter what the current state is
LTL to NBA: Example ($\mathbf{X}$ operator)

Let’s start with the next operator

\[ \mathbf{X} p \]

What is the corresponding NBA?

\[ \mathbf{X} p \]

- It doesn’t matter what the current state is
- The next state must satisfy $p$
Let’s start with the next operator

\[ \text{X } p \]

What is the corresponding NBA?

\[ \text{X } p \]

- It doesn’t matter what the current state is
- The next state must satisfy \( p \)
- After that, any path suffices for acceptance
Now the until operator

$p_1 \text{ U } p_2$

What is the corresponding NBA?

$q_0 \quad q_1 \quad p_2 \quad p_1 \quad p_2 \quad \text{true} \quad p_1 \quad U \quad p_2 \quad \rightarrow \quad p_1 \quad \rightarrow \quad p_1 \quad \rightarrow \quad p_1 \quad \rightarrow \quad p_2 \quad \rightarrow \quad \text{any} \quad \rightarrow \quad \cdots
Now the until operator

$p_1 \mathbf{U} p_2$

What is the corresponding NBA?
LTL to NBA: Example (**U** operator)

Now the until operator

\[ p_1 \text{ U } p_2 \]

What is the corresponding NBA?

\[ p_1 \text{ U } p_2 \]

\[ q_0 \rightarrow p_2 \rightarrow q_1 \]
Now the until operator

$p_1 \mathbf{U} p_2$

What is the corresponding NBA?

$p_1 \mathbf{U} p_2$

$p_1 \mathbf{U} p_2$

$p_1$ holds arbitrarily long in the beginning
Now the until operator

$p_1 \mathbf{U} p_2 \quad \rightarrow \quad p_1 \quad \rightarrow \quad p_1 \quad \rightarrow \quad p_1 \quad \rightarrow \quad p_2 \quad \rightarrow \quad \text{any} \quad \rightarrow \quad \cdots$

What is the corresponding NBA?

$p_1 \mathbf{U} p_2 \quad \rightarrow \quad q_0 \quad \rightarrow \quad p_2 \quad \rightarrow \quad q_1$

- $p_1$ holds arbitrarily long in the beginning
- To pass into accepting, $p_2$ must hold at some point
LTL to NBA: Example (U operator)

Now the until operator

$p_1 U p_2$

What is the corresponding NBA?

- $p_1$ holds arbitrarily long in the beginning
- To pass into accepting, $p_2$ must hold at some point
- Afterwards, anything goes
X and U are sufficient to express F, G, R

However, composing temporal operators is expensive in general. In the worst case, the size of the NBA is exponential in $|\varphi|$. This is the source of complexity in LTL model checking.
$X$ and $U$ are sufficient to express $F$, $G$, $R$

- $F^p \Leftrightarrow true \ U^p$

However, composing temporal operators is expensive in general. In the worst case, the size of the NBA is exponential in $|\phi|!$. This is the source of complexity in LTL model checking.
X and U are sufficient to express F, G, R

- $F p \iff true U p$
- $G p \iff \neg F \neg p$

However, composing temporal operators is expensive in general. In the worst case, the size of the NBA is exponential in $|\phi|$! This is the source of complexity in LTL model checking.
X and U are sufficient to express F, G, R

► \( F \, p \iff \text{true} \, U \, p \)

► \( G \, p \iff \neg F \, \neg p \)

► \( p_1 \, R \, p_2 \iff \neg (\neg p_1 \, U \, \neg p_2) \)
X and U are sufficient to express F, G, R

- $F \ p \iff \text{true} \ U \ p$
- $G \ p \iff \neg F \ \neg p$
- $p_1 \ R \ p_2 \iff \neg (\neg p_1 \ U \ \neg p_2)$

However, composing temporal operators is expensive in general
\[ X \text{ and } U \text{ are sufficient to express } F, G, R \]

- \[ F p \iff \text{true} U p \]
- \[ G p \iff \neg F \neg p \]
- \[ p_1 R p_2 \iff \neg (\neg p_1 U \neg p_2) \]

However, composing temporal operators is expensive in general.

In the worst case, the size of the NBA is exponential in \(|\phi|\)!
\( X \) and \( U \) are sufficient to express \( F, G, R \)

- \( F \ p \iff true \ U \ p \)
- \( G \ p \iff \neg F \ \neg p \)
- \( p_1 \ R \ p_2 \iff \neg (\neg p_1 \ U \ \neg p_2) \)

However, composing temporal operators is expensive in general.

In the worst case, the size of the NBA is exponential in \(|\phi|\).

This is the source of complexity in LTL model checking.
Given a Kripke structure $M$ and LTL $\phi$: 

1. Convert $M$ into Buchi automaton $A$ and $\phi$ into $A\phi$.
2. Negate $\phi$ by building complement $A\phi$.  
   ▶ Note: Complement can blow up exponentially!
   ▶ In practice, negate $\phi$ before building NBA.
3. Check emptiness of $L(A \setminus A\phi)$.
4. If not empty, return a word (prefix) $wL(A \setminus A\phi)$.

Worst case complexity: $O(|M|^2 |\phi|)$.
Summary: Automata-Based LTL Model Checking

Given a Kripke structure $M$ and LTL $\phi$:
1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_{\phi}$
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_{\phi}$
2. Negate $\phi$ by building complement $\overline{A_{\phi}}$

Worst case complexity: $O(j_M j_{\phi}^2)$

Intersection $A_1 \setminus A_2$ produces automaton of size $j_A j_{A_1} j_{A_2}$

LTL to NBA produces $A_{\phi}$ of size $2^j_{\phi}$

Emptiness check is depth-first search – linear time
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_\phi$

2. Negate $\phi$ by building complement $\overline{A_\phi}$
   - **Note**: Complement can blow up exponentially!
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_{\phi}$

2. Negate $\phi$ by building complement $\overline{A_{\phi}}$
   - **Note**: Complement can blow up exponentially!
   - In practice, negate $\phi$ before building NBA

Worst case complexity: $O(j_M^2 j_{\phi})$
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_\phi$

2. Negate $\phi$ by building complement $\overline{A_\phi}$
   - **Note**: Complement can blow up exponentially!
   - In practice, negate $\phi$ before building NBA

3. Check emptiness of $L(A \cap \overline{A_\phi})$
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_\phi$

2. Negate $\phi$ by building complement $\overline{A_\phi}$
   - **Note:** Complement can blow up exponentially!
   - In practice, negate $\phi$ before building NBA

3. Check emptiness of $L(A \cap \overline{A_\phi})$

4. If not empty, return a word (prefix) $w \in L(A \cap \overline{A_\phi})$
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_{\phi}$

2. Negate $\phi$ by building complement $\overline{A_{\phi}}$
   - **Note**: Complement can blow up exponentially!
   - In practice, negate $\phi$ before building NBA

3. Check emptiness of $L(A \cap \overline{A_{\phi}})$

4. If not empty, return a word (prefix) $w \in L(A \cap \overline{A_{\phi}})$

Worst case complexity: $O(|M| \cdot 2^{\phi})$
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_\phi$
2. Negate $\phi$ by building complement $\overline{A_\phi}$
   - **Note**: Complement can blow up exponentially!
   - In practice, negate $\phi$ before building NBA
3. Check emptiness of $L(A \cap \overline{A_\phi})$
4. If not empty, return a word (prefix) $w \in L(A \cap \overline{A_\phi})$

Worst case complexity: $O(|M| \cdot 2^{|\phi|})$
- Intersection $A_1 \cap A_2$ produces automaton of size $|A_1| \cdot |A_2|$
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_\phi$

2. Negate $\phi$ by building complement $\overline{A_\phi}$
   - **Note**: Complement can blow up exponentially!
   - In practice, negate $\phi$ before building NBA

3. Check emptiness of $L(A \cap \overline{A_\phi})$

4. If not empty, return a word (prefix) $w \in L(A \cap \overline{A_\phi})$

Worst case complexity: $O(|M| \cdot 2^{|\phi|})$

- Intersection $A_1 \cap A_2$ produces automaton of size $|A_1| \cdot |A_2|$
- LTL to NBA produces $A_\phi$ of size $2^{|\phi|}$
Given a Kripke structure $M$ and LTL $\phi$:

1. Convert $M$ into Buchi automaton $A$, $\phi$ into $A_\phi$

2. Negate $\phi$ by building complement $\overline{A_\phi}$
   - **Note**: Complement can blow up exponentially!
   - In practice, negate $\phi$ before building NBA

3. Check emptiness of $L(A \cap \overline{A_\phi})$

4. If not empty, return a word (prefix) $w \in L(A \cap \overline{A_\phi})$

Worst case complexity: $O(|M| \cdot 2^{|\phi|})$

- Intersection $A_1 \cap A_2$ produces automaton of size $|A_1| \cdot |A_2|$
- LTL to NBA produces $A_\phi$ of size $2^{|\phi|}$
- Emptiness check is depth-first search – linear time
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A}_\phi$.

1. Construct property automaton $A \phi$
2. Begin taking intersection at initial states of $A$
3. Perform DFS incrementally at each step
4. Whenever DFS needs a state that hasn’t been built, add it

In many cases, counterexamples are found early before DFS backtracks too much. This works because bugs are often easy to find – software is buggy!
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A_{\phi}}$

Once we have the NBA, all we do is depth-first search
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A_\phi}$.

Once we have the NBA, all we do is depth-first search.

In practice, the search can proceed with the construction.
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A_\phi}$.

Once we have the NBA, all we do is depth-first search.

In practice, the search can proceed with the construction:

1. Construct property automaton $A_\phi$ first.
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A}_\phi$

Once we have the NBA, all we do is depth-first search

In practice, the search can proceed with the construction

1. Construct property automaton $A_\phi$ first
2. Begin taking intersection at initial states of $A$

This works because bugs are often easy to find – software is buggy!
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A_\phi}$

Once we have the NBA, all we do is depth-first search

In practice, the search can proceed with the construction

1. Construct property automaton $A_\phi$ first
2. Begin taking intersection at initial states of $A$
3. Perform DFS incrementally at each step
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A}_\phi$

Once we have the NBA, all we do is depth-first search

In practice, the search can proceed with the construction

1. Construct property automaton $A_\phi$ first
2. Begin taking intersection at initial states of $A$
3. Perform DFS incrementally at each step
4. Whenever DFS needs a state that hasn’t been built, add it
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A_\phi}$

Once we have the NBA, all we do is depth-first search

In practice, the search can proceed with the construction

1. Construct property automaton $A_\phi$ first
2. Begin taking intersection at initial states of $A$
3. Perform DFS incrementally at each step
4. Whenever DFS needs a state that hasn’t been built, add it

In many cases, counterexamples are found early before DFS backtracks too much
On-the-fly model checking

The expensive part of this algorithm is in constructing $A \cap \overline{A_\phi}$

Once we have the NBA, all we do is depth-first search

In practice, the search can proceed with the construction
1. Construct property automaton $A_\phi$ first
2. Begin taking intersection at initial states of $A$
3. Perform DFS incrementally at each step
4. Whenever DFS needs a state that hasn’t been built, add it

In many cases, counterexamples are found early before DFS backtracks too much

This works because bugs are often easy to find – software is buggy!
Next Lecture

- Symbolic model checking
- If time: introduce a model-checking tool