

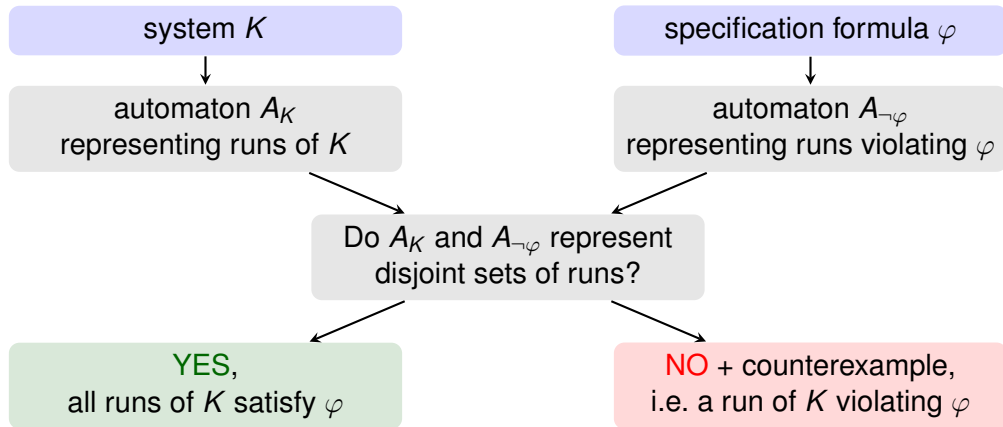
IA169 Model Checking

Automata-based LTL model checking

Jan Strejček

Faculty of Informatics
Masaryk University

Automata-based LTL model checking



agenda

- formalization of the state-based LTL model checking problem: (fair) Kripke structure and LTL
- Büchi automata (BA) and generalized Büchi automata (GBA)
- transformation of finite (fair) Kripke structures to (G)BA
- translation of LTL to BA via self-loop alternating automata
- algorithms checking disjointness of A_K and $A_{\neg\varphi}$
 - algorithm based on SCC decomposition
 - nested DFS algorithm
 - optimizations
- action-based version of LTL model checking

sources

- Chapter 7 of E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith: *Model Checking, Second Edition*, MIT, 2018.
- M. Y. Vardi: *An Automata-Theoretic Approach to Linear Temporal Logic*, LNCS 1043, Springer, 1995.

Formalization of the state-based LTL model checking problem

atomic propositions

- basic observable properties of each state of the system
- for example: $x \geq y + 10$, z is even, *gate is open*, *program is at line 10*
- the validity of each atomic proposition in each state of the system has to be fully determined by the state
- specification talks only about validity of atomic proposition during system runs
- AP denotes a countable set of atomic propositions

atomic propositions

- basic observable properties of each state of the system
 - for example: $x \geq y + 10$, *z is even*, *gate is open*, *program is at line 10*
 - the validity of each atomic proposition in each state of the system has to be fully determined by the state
 - specification talks only about validity of atomic proposition during system runs
 - *AP* denotes a countable set of atomic propositions
-
- basic formalism for state-based systems is a **Kripke structure**

Definition (Kripke structure)

A **Kripke structure** is a tuple $K = (S, T, S_0, L)$, where

- S is a set of **states**,
- $T \subseteq S \times S$ is a **transition relation**,
- $S_0 \subseteq S$ is a set of **initial states**,
- $L : S \rightarrow 2^{AP}$ is a **labeling function** associating to each state $s \in S$ the set of atomic propositions that are true in s .

Definition (Kripke structure)

A **Kripke structure** is a tuple $K = (S, T, S_0, L)$, where

- S is a set of **states**,
- $T \subseteq S \times S$ is a **transition relation**,
- $S_0 \subseteq S$ is a set of **initial states**,
- $L : S \rightarrow 2^{AP}$ is a **labeling function** associating to each state $s \in S$ the set of atomic propositions that are true in s .

- Kripke structures are typically described in an implicit way
- formats for implicit description typically offer
 - programs, processes, finite-state machines
 - synchronous or asynchronous composition
 - communication and synchronization mechanisms
 - nondeterminism or inputs

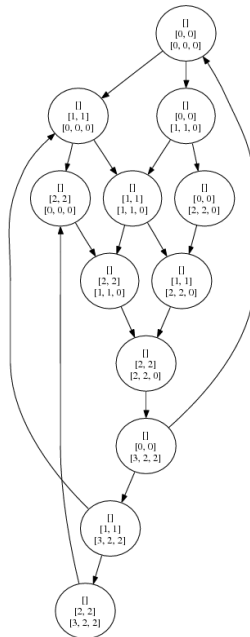
Example in the modelling language DVE

```
channel {byte} c[0];
```

```
process A {  
  byte a;  
  state q1,q2,q3;  
  init q1;  
  trans  
  q1→q2 { effect a=a+1; },  
  q2→q3 { effect a=a+1; },  
  q3→q1 { sync c!a; effect a=0; };  
}
```

```
process B {  
  byte b,x;  
  state p1,p2,p3,p4;  
  init p1;  
  trans  
  p1→p2 { effect b=b+1; },  
  p2→p3 { effect b=b+1; },  
  p3→p4 { sync c?x; },  
  p4→p1 { guard x==b; effect b=0, x=0; };  
}
```

```
system async;
```



Example of a simple mutual exclusion system

cobegin $P_0 \parallel P_1$ **coend**

$P_0::$ $l_0:$ **while true do**
 $NC_0:$ **wait** ($turn = 0$);
 $CR_0:$ $turn := 1$
 end while

$P_1::$ $l_1:$ **while true do**
 $NC_1:$ **wait** ($turn = 1$);
 $CR_1:$ $turn := 0$
 end while

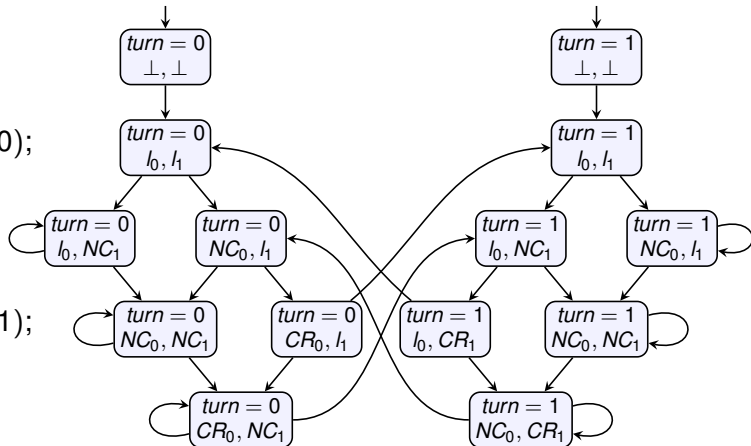
assume that $turn$ is initially 0 or 1

Example of a simple mutual exclusion system

cobegin $P_0 \parallel P_1$ **coend**

$P_0:: l_0:$ **while true do**
 $NC_0:$ **wait** ($turn = 0$);
 $CR_0:$ $turn := 1$
 end while

$P_1:: l_1:$ **while true do**
 $NC_1:$ **wait** ($turn = 1$);
 $CR_1:$ $turn := 0$
 end while



assume that $turn$ is initially 0 or 1

Definition (run)

Let $K = (S, T, S_0, L)$ be a Kripke structure. A **run** of K is an infinite sequence $\pi = s_0 s_1 s_2 \dots$ of states such that $s_0 \in S_0$ and $(s_i, s_{i+1}) \in T$ holds for each $i \geq 0$.

Definition (run)

Let $K = (S, T, S_0, L)$ be a Kripke structure. A **run** of K is an infinite sequence $\pi = s_0 s_1 s_2 \dots$ of states such that $s_0 \in S_0$ and $(s_i, s_{i+1}) \in T$ holds for each $i \geq 0$.

- linear time model checking decides whether all runs satisfy the specification
- the set of infinite sequences of states is denoted by S^ω
- to consider also **finite runs**, we can define a **run** as a maximal sequence $\pi = s_0 s_1 s_2 \dots \in S^+ \cup S^\omega$ of successive states starting in an initial state, where maximal means infinite or ending in a state without any successor
- it is usually assumed that there are no states without any successors: any system can be transformed to this form by adding self-loops to such states

Linear temporal logic (LTL)

Definition (linear temporal logic, LTL)

Formulae of **Linear Temporal Logic (LTL)** are defined by

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where \top stands for **true** and a ranges over a countable set AP .

Definition (linear temporal logic, LTL)

Formulae of **Linear Temporal Logic (LTL)** are defined by

$$\varphi ::= \top \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where \top stands for **true** and a ranges over a countable set AP .

abbreviations and alternative notation

$$\blacksquare \perp \equiv \neg\top$$

$$\blacksquare \varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$$

$$\blacksquare \varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$$

$$\blacksquare \varphi \Leftrightarrow \psi \equiv \varphi \Rightarrow \psi \wedge \psi \Leftarrow \varphi$$

$$\blacksquare \bigcirc\varphi \equiv X\varphi$$

$$\blacksquare F\varphi \equiv \diamond\varphi \equiv \top \mathbf{U} \varphi$$

$$\blacksquare G\varphi \equiv \square\varphi \equiv \neg F\neg\varphi$$

Intuitive semantic of LTL

	operator name	intuitive meaning
Xa	next	• <i>a</i> • • • ...
aUb	until	<i>a a ... a b</i> • • • ...
Fa	eventually	• • ... • <i>a</i> • • • ...
Ga	always or globally	<i>a a a a ...</i>

Semantics of LTL

- we interpret LTL on infinite words $w = w(0)w(1)\dots \in (2^{AP})^\omega$
- by w_i we denote the suffix of w of the form $w(i)w(i+1)w(i+2)\dots$

- we interpret LTL on infinite words $w = w(0)w(1)\dots \in (2^{AP})^\omega$
- by w_i we denote the suffix of w of the form $w(i)w(i+1)w(i+2)\dots$

Definition

The relation $w \models \varphi$, meaning that w **satisfies** φ , is defined inductively as follows.

$$w \models \top$$

$$w \models a \quad \text{iff} \quad a \in w(0)$$

$$w \models \neg\varphi \quad \text{iff} \quad w \not\models \varphi$$

$$w \models \varphi_1 \wedge \varphi_2 \quad \text{iff} \quad w \models \varphi_1 \wedge w \models \varphi_2$$

$$w \models \mathbf{X}\varphi \quad \text{iff} \quad w_1 \models \varphi$$

$$w \models \varphi_1 \mathbf{U} \varphi_2 \quad \text{iff} \quad \exists i \geq 0 . w_i \models \varphi_2 \wedge \forall 0 \leq j < i . w_j \models \varphi_1$$

- we interpret LTL on infinite words $w = w(0)w(1)\dots \in (2^{AP})^\omega$
- by w_i we denote the suffix of w of the form $w(i)w(i+1)w(i+2)\dots$

Definition

The relation $w \models \varphi$, meaning that w **satisfies** φ , is defined inductively as follows.

$$w \models \top$$

$$w \models a \quad \text{iff} \quad a \in w(0)$$

$$w \models \neg\varphi \quad \text{iff} \quad w \not\models \varphi$$

$$w \models \varphi_1 \wedge \varphi_2 \quad \text{iff} \quad w \models \varphi_1 \wedge w \models \varphi_2$$

$$w \models X\varphi \quad \text{iff} \quad w_1 \models \varphi$$

$$w \models \varphi_1 \mathbf{U} \varphi_2 \quad \text{iff} \quad \exists i \geq 0 . w_i \models \varphi_2 \wedge \forall 0 \leq j < i . w_j \models \varphi_1$$

By $AP(\varphi)$ we denote the set of atomic propositions appearing in φ .

The **language** of φ is defined as $L(\varphi) = \{w \in \Sigma^\omega \mid w \models \varphi\}$, where $\Sigma = 2^{AP(\varphi)}$.

Definition

Let $K = (S, T, S_0, L)$ be a Kripke structure and φ be an LTL formula.

A run $\pi = s_0 s_1 s_2 \dots$ of K **satisfies** φ , written $\pi \models \varphi$, if $L(s_0)L(s_1)L(s_2) \dots \models \varphi$.

K **satisfies** φ , written $K \models \varphi$, if $\pi \models \varphi$ holds for every run π of K .

The goal of LTL model checking

Definition

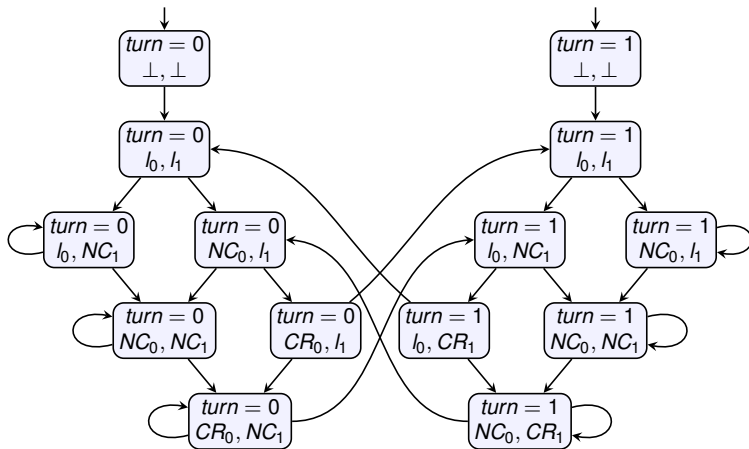
Let $K = (S, T, S_0, L)$ be a Kripke structure and φ be an LTL formula.

A run $\pi = s_0s_1s_2\dots$ of K **satisfies** φ , written $\pi \models \varphi$, if $L(s_0)L(s_1)L(s_2)\dots \models \varphi$.

K **satisfies** φ , written $K \models \varphi$, if $\pi \models \varphi$ holds for every run π of K .

Given a Kripke structure K and an LTL formula φ , the **goal of LTL model checking** is to decide whether $K \models \varphi$ or not. In the negative case, model checking should provide a **counterexample**, i.e., a run π of K such that $\pi \not\models \varphi$.

Example



which formulae are satisfied?

- $G\neg(CR_0 \wedge CR_1)$
- $GFturn = 0 \wedge GFturn = 1$

- fairness allows to add additional restrictions on the system runs
- can reflect properties of process schedulers

- fairness allows to add additional restrictions on the system runs
- can reflect properties of process schedulers

Definition (fair Kripke structure)

A **fair Kripke structure** is a tuple $K = (S, T, S_0, L, \mathcal{F})$, where (S, T, S_0, L) is a Kripke structure and $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$ is a finite set of **fairness constraints** such that $F_i \subseteq S$ for each $1 \leq i \leq n$.

A sequence $\pi = s_0 s_1 s_2 \in S^\omega$ is called a **fair run** of K if it is a run of (S, T, S_0, L) and it visits each $F_i \in \mathcal{F}$ infinitely often, i.e., $s_j \in F_i$ for infinitely many j .

K **fairly satisfies** an LTL formula φ , written $K \models_F \varphi$, if each fair run of K satisfies φ .

- fairness allows to add additional restrictions on the system runs
- can reflect properties of process schedulers

Definition (fair Kripke structure)

A **fair Kripke structure** is a tuple $K = (S, T, S_0, L, \mathcal{F})$, where (S, T, S_0, L) is a Kripke structure and $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$ is a finite set of **fairness constraints** such that $F_i \subseteq S$ for each $1 \leq i \leq n$.

A sequence $\pi = s_0 s_1 s_2 \in \mathcal{S}^\omega$ is called a **fair run** of K if it is a run of (S, T, S_0, L) and it visits each $F_i \in \mathcal{F}$ infinitely often, i.e., $s_j \in F_i$ for infinitely many j .

K **fairly satisfies** an LTL formula φ , written $K \models_F \varphi$, if each fair run of K satisfies φ .

- add reasonable fairness constraint to the mutual exclusion system

Büchi automata (BA) and generalized Büchi automata (GBA)

Definition (Büchi automaton, BA)

A **Büchi automaton (BA)** is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$, where

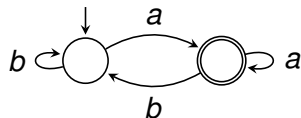
- Q is a finite set of **states**,
- Σ is a finite **alphabet**,
- $\delta \subseteq Q \times \Sigma \times Q$ is a **transition relation**,
- $Q_0 \subseteq Q$ is a set of **initial states**,
- $F \subseteq Q$ is a set of **accepting states**.

Definition (Büchi automaton, BA)

A **Büchi automaton (BA)** is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$, where

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

- we write $p \xrightarrow{a} q$ instead of $(p, a, q) \in \delta$



Büchi automaton (BA)

- for an arbitrary infinite sequence σ , by $\text{inf}(\sigma)$ we denote the set of its elements that appear infinitely often in σ

Büchi automaton (BA)

- for an arbitrary infinite sequence σ , by $\text{inf}(\sigma)$ we denote the set of its elements that appear infinitely often in σ

Definition (run, language)

Let $A = (Q, \Sigma, \delta, Q_0, F)$ be a BA.

A **run** of A over an infinite word $w = a_1 a_2 \dots \in \Sigma^\omega$ is a sequence of states $\pi = s_0 s_1 \dots \in Q^\omega$ satisfying $s_0 \in Q_0$ and $s_{i-1} \xrightarrow{a_i} s_i$ for each $i \geq 1$.

A run π is **accepting** if $\text{inf}(\pi) \cap F \neq \emptyset$.

A word $w \in \Sigma^\omega$ is **accepted** by A if there exists an accepting run of A over w .

A **language** represented by A is the set $L(A) \subseteq \Sigma^\omega$ of words accepted by A .

Büchi automaton (BA)

- for an arbitrary infinite sequence σ , by $\text{inf}(\sigma)$ we denote the set of its elements that appear infinitely often in σ

Definition (run, language)

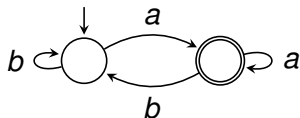
Let $A = (Q, \Sigma, \delta, Q_0, F)$ be a BA.

A **run** of A over an infinite word $w = a_1 a_2 \dots \in \Sigma^\omega$ is a sequence of states $\pi = s_0 s_1 \dots \in Q^\omega$ satisfying $s_0 \in Q_0$ and $s_{i-1} \xrightarrow{a_i} s_i$ for each $i \geq 1$.

A run π is **accepting** if $\text{inf}(\pi) \cap F \neq \emptyset$.

A word $w \in \Sigma^\omega$ is **accepted** by A if there exists an accepting run of A over w .

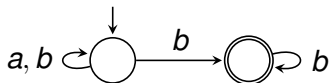
A **language** represented by A is the set $L(A) \subseteq \Sigma^\omega$ of words accepted by A .



$$L(A) = \{w \in \{a, b\}^\omega \mid a \in \text{inf}(w)\}$$

Properties of Büchi automata

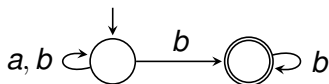
- languages represented by Büchi automata are called ω -regular
- the class of ω -regular languages is closed under \cup , \cap , and complement (though complementation of Büchi automata is highly non-trivial)
- deterministic Büchi automata are less expressive than nondeterministic ones: for example $\{a, b\}^* \cdot \{b\}^\omega$ cannot be described by any deterministic BA



$$L(A) = \{a, b\}^* \cdot \{b\}^\omega$$

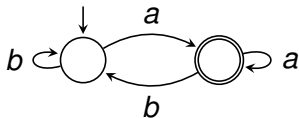
Properties of Büchi automata

- languages represented by Büchi automata are called ω -regular
- the class of ω -regular languages is closed under \cup , \cap , and complement (though complementation of Büchi automata is highly non-trivial)
- deterministic Büchi automata are less expressive than nondeterministic ones: for example $\{a, b\}^* \cdot \{b\}^\omega$ cannot be described by any deterministic BA



$$L(A) = \{a, b\}^* \cdot \{b\}^\omega$$

- the class of languages represented by deterministic Büchi automata is not closed under complement



$$L(B) = \{w \in \{a, b\}^\omega \mid a \in \text{inf}(w)\}$$
$$L(B) = \{a, b\}^\omega \setminus L(A)$$

Generalized Büchi automaton (GBA)

Definition (generalized Büchi automaton, GBA)

A **generalized Büchi automaton (GBA)** is a tuple $A = (Q, \Sigma, \delta, Q_0, \mathcal{F})$, where Q, Σ, δ, Q_0 have the same meaning as in BA and $\mathcal{F} = \{F_1, \dots, F_n\}$ is a finite set of **accepting sets** satisfying $F_i \subseteq Q$ for each $F_i \in \mathcal{F}$.

The definition of **run** is the same as for BA.

A run π is **accepting** if for each $F_i \in \mathcal{F}$ it holds $\text{inf}(\pi) \cap F_i \neq \emptyset$.

The definition of an **accepted word** and **language** is the same as for BA.

Definition (generalized Büchi automaton, GBA)

A **generalized Büchi automaton (GBA)** is a tuple $A = (Q, \Sigma, \delta, Q_0, \mathcal{F})$, where Q, Σ, δ, Q_0 have the same meaning as in BA and $\mathcal{F} = \{F_1, \dots, F_n\}$ is a finite set of **accepting sets** satisfying $F_i \subseteq Q$ for each $F_i \in \mathcal{F}$.

The definition of **run** is the same as for BA.

A run π is **accepting** if for each $F_i \in \mathcal{F}$ it holds $\text{inf}(\pi) \cap F_i \neq \emptyset$.

The definition of an **accepted word** and **language** is the same as for BA.

- each BA $(Q, \Sigma, \delta, Q_0, F)$ can be seen as a GBA $(Q, \Sigma, \delta, Q_0, \{F\})$
- each GBA can be transformed into a BA representing the same language
- GBAs can be more succinct

Transformation of finite (fair) Kripke structures to (G)BA

- since now on, we consider only Kripke structures K with finitely many states
- assume that we know the set $AP(\varphi)$, which is always finite
- when deciding $K \stackrel{?}{\models} \varphi$, we can ignore atomic propositions outside $AP(\varphi)$
- we transform K into a Büchi automaton A_K with alphabet $\Sigma = 2^{AP(\varphi)}$ representing the language

$$L_K^\Sigma = \{a_0 a_1 a_2 \dots \in \Sigma^\omega \mid \text{there exists a run } s_0 s_1 s_2 \dots \text{ of } K \text{ such that } a_i = L(s_i) \cap AP(\varphi) \text{ for each } i \geq 0\}$$

corresponding to runs of K projected to $AP(\varphi)$

Kripke structure \rightarrow BA

input: a set $AP(\varphi)$ and a Kripke structure $K = (S, T, S_0, L)$

output: a BA $A_K = (S, 2^{AP(\varphi)}, \delta, S_0, S)$ representing L_K^Σ , where $\Sigma = 2^{AP(\varphi)}$

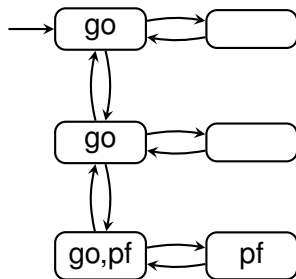
- $\delta = \{(p, a, q) \mid (p, q) \in T \text{ and } a = L(p) \cap AP(\varphi)\}$

Kripke structure \rightarrow BA

input: a set $AP(\varphi)$ and a Kripke structure $K = (S, T, S_0, L)$

output: a BA $A_K = (S, 2^{AP(\varphi)}, \delta, S_0, S)$ representing L_K^Σ , where $\Sigma = 2^{AP(\varphi)}$

$$\delta = \{(p, a, q) \mid (p, q) \in T \text{ and } a = L(p) \cap AP(\varphi)\}$$



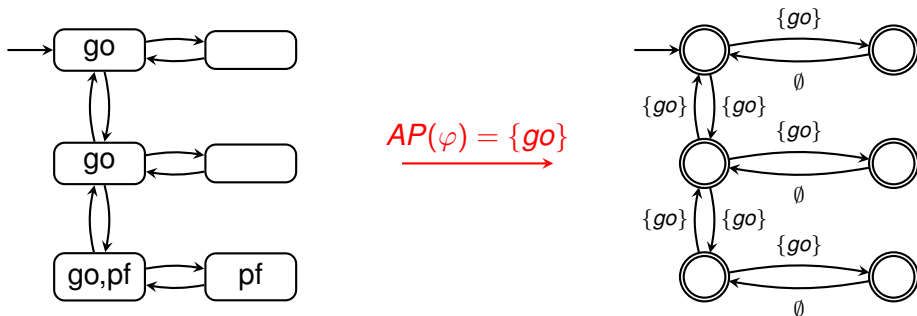
$$\underline{AP(\varphi) = \{go\}}$$

Kripke structure \rightarrow BA

input: a set $AP(\varphi)$ and a Kripke structure $K = (S, T, S_0, L)$

output: a BA $A_K = (S, 2^{AP(\varphi)}, \delta, S_0, S)$ representing L_K^Σ , where $\Sigma = 2^{AP(\varphi)}$

■ $\delta = \{(p, a, q) \mid (p, q) \in T \text{ and } a = L(p) \cap AP(\varphi)\}$



- similarly, we transform a fair Kripke structure K into a generalized Büchi automaton A_K with alphabet $\Sigma = 2^{AP(\varphi)}$ representing the language

$$L_K^\Sigma = \{a_0 a_1 a_2 \dots \in \Sigma^\omega \mid \text{there exists a fair run } s_0 s_1 s_2 \dots \text{ of } K \text{ such that } a_i = L(s_i) \cap AP(\varphi) \text{ for each } i \geq 0\}$$

- similarly, we transform a fair Kripke structure K into a generalized Büchi automaton A_K with alphabet $\Sigma = 2^{AP(\varphi)}$ representing the language

$$L_K^\Sigma = \{a_0 a_1 a_2 \dots \in \Sigma^\omega \mid \text{there exists a fair run } s_0 s_1 s_2 \dots \text{ of } K \text{ such that } a_i = L(s_i) \cap AP(\varphi) \text{ for each } i \geq 0\}$$

input: a set $AP(\varphi)$ and a fair Kripke structure $K = (S, T, S_0, L, \mathcal{F})$

output: a GBA $A_K = (S, 2^{AP(\varphi)}, \delta, S_0, \mathcal{F})$ representing L_K^Σ , where $\Sigma = 2^{AP(\varphi)}$

- $\delta = \{(p, a, q) \mid (p, q) \in T \text{ and } a = L(p) \cap AP(\varphi)\}$

Translation of LTL to BA via self-loop alternating automata

LTL \rightarrow BA translations in general

- translates an LTL formula φ into a BA A_φ accepting $L(\varphi)$
- many LTL \rightarrow BA translations
 - LTL \rightarrow GBA \rightarrow BA (Spin)
 - LTL \rightarrow transition-based GBA (TGBA) \rightarrow BA (Spot)
 - LTL \rightarrow self-loop alternating BA \rightarrow TGBA \rightarrow BA (LTL2BA, LTL3BA)
 - LTL \rightarrow self-loop alternating BA \rightarrow BA
 - ...
- translations via self-loop alternating automata offer
 - size-reducing optimizations of self-loop alternating automata
 - smaller resulting BA (in some cases)

Translation of LTL to BA via self-loop alternating automata

Alternating automata

Definition (positive boolean formulae)

Positive Boolean formulae over set Q , denoted with $\mathcal{B}^+(Q)$, are defined by

$$\varphi ::= \top \mid \perp \mid q \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$$

where \top stands for **true**, \perp stands for **false**, and q ranges over Q .

Definition (positive boolean formulae)

Positive Boolean formulae over set Q , denoted with $\mathcal{B}^+(Q)$, are defined by

$$\varphi ::= \top \mid \perp \mid q \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$$

where \top stands for **true**, \perp stands for **false**, and q ranges over Q .

$S \subseteq Q$ is a **model** of $\varphi \iff$ the valuation assigning true just to elements of S satisfies φ

S is a **minimal model** of φ (written $S \models \varphi$) \iff S is a model of φ and no proper subset of S is a model of φ

Examples of positive Boolean formulae

formulae of $\mathcal{B}^+(\{p, q, r\})$	(minimal) models
\perp	no model
\top	$\emptyset, \{p\}, \{q\}, \{r\}, \{p, q\}, \dots$
$p \wedge q$	$\{p, q\}, \{p, q, r\}$
$p \vee (q \wedge r)$	$\{p\}, \{p, q\}, \{p, r\}, \{q, r\}, \{p, q, r\}$
$p \wedge (q \vee r)$	$\{p, q\}, \{p, r\}, \{p, q, r\}$

Examples of positive Boolean formulae

formulae of $\mathcal{B}^+(\{p, q, r\})$	(minimal) models
\perp	no model
\top	$\emptyset, \{p\}, \{q\}, \{r\}, \{p, q\}, \dots$
$p \wedge q$	$\{p, q\}, \{p, q, r\}$
$p \vee (q \wedge r)$	$\{p\}, \{p, q\}, \{p, r\}, \{q, r\}, \{p, q, r\}$
$p \wedge (q \vee r)$	$\{p, q\}, \{p, r\}, \{p, q, r\}$

- minimal models correspond to clauses in disjunctive normal form (without superfluous clauses)

$$\varphi \equiv \bigvee_{S \models \varphi} \left(\bigwedge_{p \in S} p \right)$$

Definition (alternating Büchi automaton)

An **alternating Büchi automaton** is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$, where

- Q is a finite set of **states**,
- Σ is a finite **alphabet**,
- $\delta : Q \times \Sigma \rightarrow \mathcal{B}^+(Q)$ is a **transition function**,
- $Q_0 \subseteq Q$ is a set of **initial states**,
- $F \subseteq Q$ is a set of **accepting states**.

Definition (tree, Q -labeled tree)

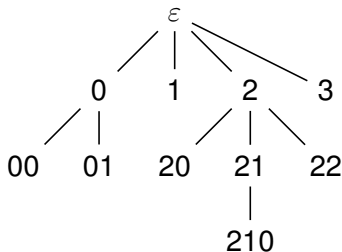
A **tree** is a set $T \subseteq \mathbb{N}_0^*$ such that if $xc \in T$, where $x \in \mathbb{N}_0^*$ and $c \in \mathbb{N}_0$, then also

- $x \in T$ and
- $xc' \in T$ for all $0 \leq c' < c$.

Definition (tree, Q -labeled tree)

A **tree** is a set $T \subseteq \mathbb{N}_0^*$ such that if $xc \in T$, where $x \in \mathbb{N}_0^*$ and $c \in \mathbb{N}_0$, then also

- $x \in T$ and
- $xc' \in T$ for all $0 \leq c' < c$.



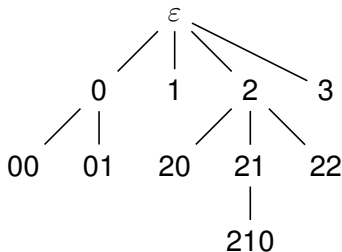
$$T = \{\varepsilon, 0, 1, 2, 3, 00, 01, 20, 21, 22, 210\}$$

Definition (tree, Q -labeled tree)

A **tree** is a set $T \subseteq \mathbb{N}_0^*$ such that if $xc \in T$, where $x \in \mathbb{N}_0^*$ and $c \in \mathbb{N}_0$, then also

- $x \in T$ and
- $xc' \in T$ for all $0 \leq c' < c$.

A **Q -labeled tree** is a pair (T, r) of a tree T and a labeling function $r : T \rightarrow Q$.



$$T = \{\varepsilon, 0, 1, 2, 3, 00, 01, 20, 21, 22, 210\}$$

Definition (run, language)

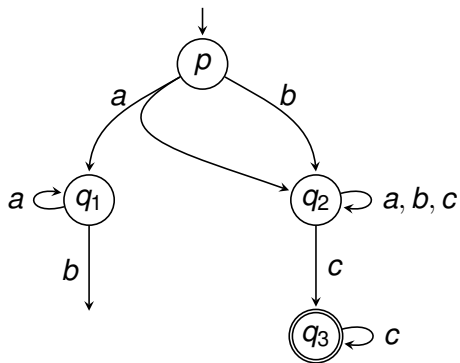
A **run** of an alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$ on word $w = a_0 a_1 \dots \in \Sigma^\omega$ is a Q -labeled tree (T, r) such that

- $r(\varepsilon) \in Q_0$ and
- for each $x \in T$: $\{r(xc) \mid c \in \mathbb{N}_0, xc \in T\} \models \delta(r(x), a_{|x|})$.

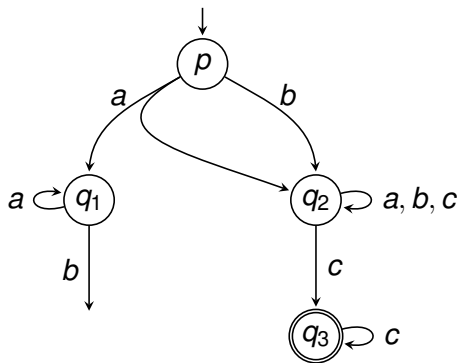
A run (T, r) is **accepting** iff for each infinite branch σ in T it holds that infinitely many nodes of the branch are labeled with a state in F .

A word $w \in \Sigma^\omega$ is **accepted** by A iff there exists an accepting run of A over w .
A **language** represented by A is the set $L(A) \subseteq \Sigma^\omega$ of words accepted by A .

Example of an alternating Büchi automaton



Example of an alternating Büchi automaton



$$L(A) = \{a\}^* \cdot \{b\} \cdot \{a, b, c\}^* \cdot \{c\}^\omega$$

Self-loop alternating Büchi automaton

Intuitively, an alternating BA is **self-loop** (or **1-weak** or **linear** or **very weak**, written **SLAA** or **A1W** or **VWAA**) if it contains no cycles except self-loops.

Self-loop alternating Büchi automaton

Intuitively, an alternating BA is **self-loop** (or **1-weak** or **linear** or **very weak**, written **SLAA** or **A1W** or **VWAA**) if it contains no cycles except self-loops.

Definition (self-loop alternating BA)

Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an alternating BA. For each $p \in Q$ we define the set of all **successors** of p as

$$Succ(p) = \{q \mid \exists a \in \Sigma, S \subseteq Q : S \cup \{q\} \models \delta(p, a)\}.$$

Automaton A is **self-loop** (or **1-weak** or **linear** or **very weak**) if there exists a partial order \leq on Q such that for all $p, q \in Q$ it holds:

$$q \in Succ(p) \implies q \leq p$$

- standard **Büchi automata** are alternating Büchi automata where each $\delta(p, a)$ is \perp or a disjunction of states
- self-loop alternating BA have the same expressive power as LTL

Translation of LTL to BA via self-loop alternating automata

LTL \rightarrow self-loop alternating BA

LTL \rightarrow self-loop alternating BA

input: an LTL formula φ

output: self-loop alternating BA $A = (Q, \Sigma, \delta, \{q_\varphi\}, F)$ accepting $L(\varphi)$

LTL \rightarrow self-loop alternating BA

input: an LTL formula φ

output: self-loop alternating BA $A = (Q, \Sigma, \delta, \{q_\varphi\}, F)$ accepting $L(\varphi)$

- $Q = \{q_\psi, q_{\neg\psi} \mid \psi \text{ is a subformula of } \varphi\}$
- $\Sigma = 2^{AP(\varphi)}$

LTL \rightarrow self-loop alternating BA

input: an LTL formula φ

output: self-loop alternating BA $A = (Q, \Sigma, \delta, \{q_\varphi\}, F)$ accepting $L(\varphi)$

- $Q = \{q_\psi, q_{\neg\psi} \mid \psi \text{ is a subformula of } \varphi\}$
- $\Sigma = 2^{AP(\varphi)}$
- δ is defined as follows (where $\bar{\alpha} \in \mathcal{B}^+(Q)$ satisfies $\bar{\alpha} \equiv \neg\alpha$)

$$\begin{aligned}\delta(q_\top, l) &= \top \\ \delta(q_a, l) &= \top \text{ if } a \in l, \perp \text{ otherwise} \\ \delta(q_{\neg\psi}, l) &= \overline{\delta(q_\psi, l)} \\ \delta(q_{\psi \wedge \rho}, l) &= \delta(q_\psi, l) \wedge \delta(q_\rho, l) \\ \delta(q_{\chi\psi}, l) &= q_\psi \\ \delta(q_{\psi \cup \rho}, l) &= \delta(q_\rho, l) \vee (\delta(q_\psi, l) \wedge q_{\psi \cup \rho})\end{aligned}$$

$$\begin{aligned}\overline{\top} &= \perp \\ \overline{\perp} &= \top \\ \overline{q_{\neg\psi}} &= q_\psi \\ \overline{q_\psi} &= q_{\neg\psi} \\ \overline{\beta \wedge \gamma} &= \overline{\beta} \vee \overline{\gamma} \\ \overline{\beta \vee \gamma} &= \overline{\beta} \wedge \overline{\gamma}\end{aligned}$$

LTL \rightarrow self-loop alternating BA

input: an LTL formula φ

output: self-loop alternating BA $A = (Q, \Sigma, \delta, \{q_\varphi\}, F)$ accepting $L(\varphi)$

- $Q = \{q_\psi, q_{\neg\psi} \mid \psi \text{ is a subformula of } \varphi\}$
- $\Sigma = 2^{AP(\varphi)}$
- δ is defined as follows (where $\bar{\alpha} \in \mathcal{B}^+(Q)$ satisfies $\bar{\alpha} \equiv \neg\alpha$)

$$\begin{aligned}\delta(q_\top, l) &= \top \\ \delta(q_a, l) &= \top \text{ if } a \in l, \perp \text{ otherwise} \\ \delta(q_{\neg\psi}, l) &= \overline{\delta(q_\psi, l)} \\ \delta(q_{\psi \wedge \rho}, l) &= \delta(q_\psi, l) \wedge \delta(q_\rho, l) \\ \delta(q_{\chi\psi}, l) &= q_\psi \\ \delta(q_{\psi \cup \rho}, l) &= \delta(q_\rho, l) \vee (\delta(q_\psi, l) \wedge q_{\psi \cup \rho})\end{aligned}$$

$$\begin{aligned}\overline{\top} &= \perp \\ \overline{\perp} &= \top \\ \overline{q_{\neg\psi}} &= q_\psi \\ \overline{q_\psi} &= q_{\neg\psi} \\ \overline{\beta \wedge \gamma} &= \overline{\beta} \vee \overline{\gamma} \\ \overline{\beta \vee \gamma} &= \overline{\beta} \wedge \overline{\gamma}\end{aligned}$$

- $F = \{q_{\neg(\psi \cup \rho)} \mid \psi \cup \rho \text{ is a subformula of } \varphi\}$

Note that every infinite path of a run of A has a suffix labeled with a state of the form $q_{\psi \cup \rho}$ or $q_{\neg(\psi \cup \rho)}$ (other states have no loops and can appear at most once on a path). F is defined to prevent the first case: $\psi \cup \rho$ is satisfied only if ρ eventually holds.

Theorem

Given an LTL formula φ , one can construct a self-loop alternating BA A accepting $L(\varphi)$ and such that the number of states of A is linear in the length of φ .

Translation of LTL to BA via self-loop alternating automata

Self-loop alternating BA \rightarrow BA

Self-loop alternating BA \rightarrow BA

input: a self-loop alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$

output: a BA $A' = (Q', \Sigma, \delta', Q'_0, F')$ accepting $L(A)$

Self-loop alternating BA \rightarrow BA

input: a self-loop alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$

output: a BA $A' = (Q', \Sigma, \delta', Q'_0, F')$ accepting $L(A)$

Intuitively, A' tracks states on each level of the computation tree of A . Moreover, A' has to divide the set of states into two sets: states labeling paths with recent occurrence of an accepting state, and states labeling the other paths.

Self-loop alternating BA \rightarrow BA

input: a self-loop alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$

output: a BA $A' = (Q', \Sigma, \delta', Q'_0, F')$ accepting $L(A)$

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

Self-loop alternating BA \rightarrow BA

input: a self-loop alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$

output: a BA $A' = (Q', \Sigma, \delta', Q'_0, F')$ accepting $L(A)$

- $Q' = 2^Q \times 2^Q$
- $Q'_0 = \{(\{q_0\}, \emptyset) \mid q_0 \in Q_0\}$

Self-loop alternating BA \rightarrow BA

input: a self-loop alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$

output: a BA $A' = (Q', \Sigma, \delta', Q'_0, F')$ accepting $L(A)$

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

- $Q'_0 = \{(\{q_0\}, \emptyset) \mid q_0 \in Q_0\}$

- $\delta'((U, V), l)$ is defined as:

- if $U \neq \emptyset$ then

$$\delta'((U, V), l) = \{(U', V') \mid \exists X, Y \subseteq Q \text{ such that}$$
$$X \models \bigwedge_{q \in U} \delta(q, l) \text{ and}$$
$$Y \models \bigwedge_{q \in V} \delta(q, l) \text{ and}$$
$$U' = X \setminus F \text{ and } V' = Y \cup (X \cap F)\}$$

- if $U = \emptyset$ then

$$\delta'((\emptyset, V), l) = \{(U', V') \mid \exists Y \subseteq Q \text{ such that}$$
$$Y \models \bigwedge_{q \in V} \delta(q, l) \text{ and}$$
$$U' = Y \setminus F \text{ and } V' = Y \cap F)\}$$

Self-loop alternating BA \rightarrow BA

input: a self-loop alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$

output: a BA $A' = (Q', \Sigma, \delta', Q'_0, F')$ accepting $L(A)$

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

- $Q'_0 = \{(\{q_0\}, \emptyset) \mid q_0 \in Q_0\}$

- $\delta'((U, V), l)$ is defined as:

- if $U \neq \emptyset$ then

$$\delta'((U, V), l) = \{(U', V') \mid \exists X, Y \subseteq Q \text{ such that}$$
$$X \models \bigwedge_{q \in U} \delta(q, l) \text{ and}$$
$$Y \models \bigwedge_{q \in V} \delta(q, l) \text{ and}$$
$$U' = X \setminus F \text{ and } V' = Y \cup (X \cap F)\}$$

- if $U = \emptyset$ then

$$\delta'((\emptyset, V), l) = \{(U', V') \mid \exists Y \subseteq Q \text{ such that}$$
$$Y \models \bigwedge_{q \in V} \delta(q, l) \text{ and}$$
$$U' = Y \setminus F \text{ and } V' = Y \cap F)\}$$

- $F' = \{\emptyset\} \times 2^Q$

Theorem

Given a self-loop alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$, one can construct a BA A' accepting $L(A)$ and such that the number of states of A' is $2^{\mathcal{O}(|Q|)}$.

Theorem

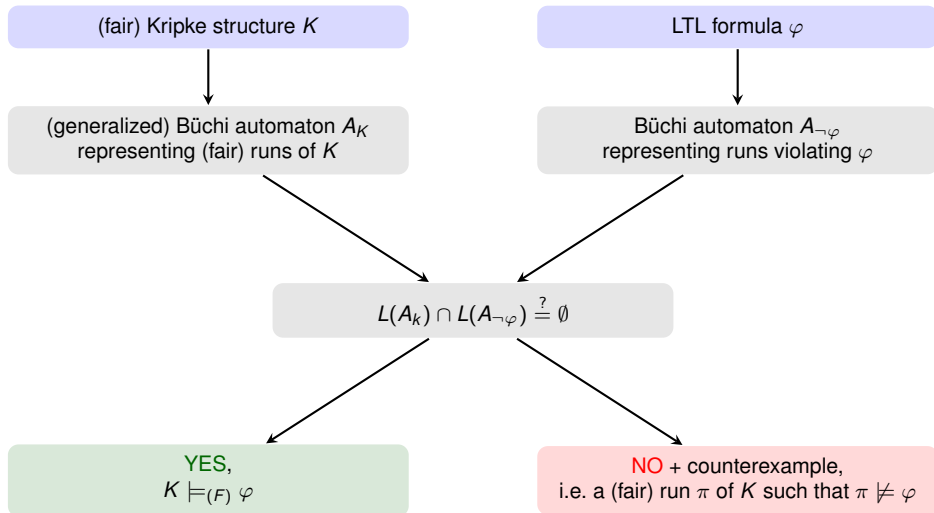
Given a self-loop alternating BA $A = (Q, \Sigma, \delta, Q_0, F)$, one can construct a BA A' accepting $L(A)$ and such that the number of states of A' is $2^{\mathcal{O}(|Q|)}$.

Corollary

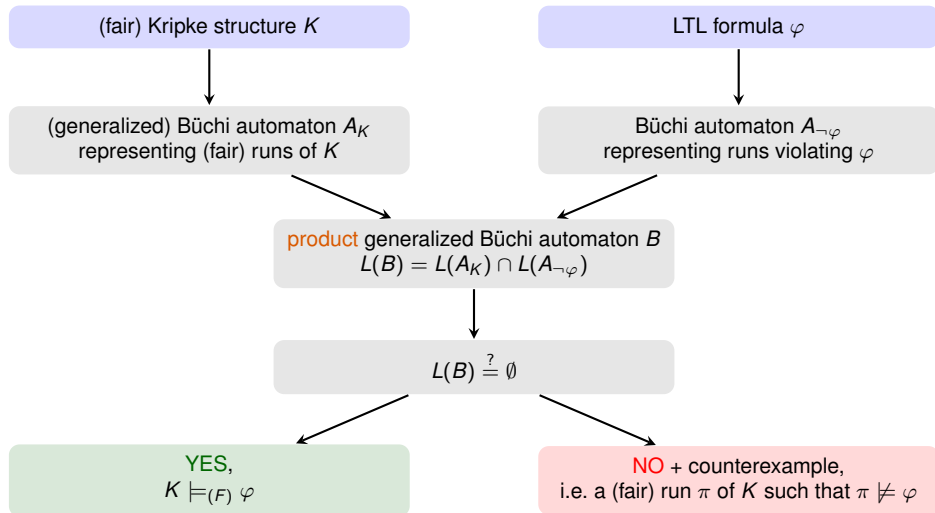
Given an LTL formula φ and an alphabet Σ , one can construct a BA A' accepting $L(\varphi)$ and such that the number of states of A' is $2^{\mathcal{O}(|\varphi|)}$.

Algorithms checking disjointness of A_K and $A_{\neg\varphi}$

Automata-based LTL model checking



Automata-based LTL model checking



Construction of product automaton

input: GBAs $A_1 = (Q_1, \Sigma, \delta_1, Q_{01}, \mathcal{F}_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, Q_{02}, \mathcal{F}_2)$

output: a GBA $B = (Q_1 \times Q_2, \Sigma, \delta, Q_{01} \times Q_{02}, \mathcal{F})$ representing $L(A_1) \cap L(A_2)$

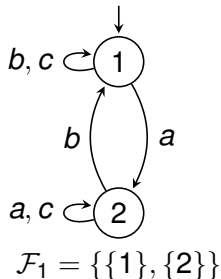
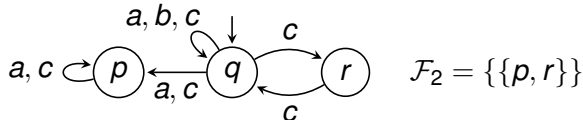
- $\delta = \{((p_1, p_2), a, (q_1, q_2)) \mid (p_1, a, q_1) \in \delta_1 \text{ and } (p_2, a, q_2) \in \delta_2\}$
- $\mathcal{F} = \{F_{1i} \times Q_2 \mid F_{1i} \in \mathcal{F}_1\} \cup \{Q_1 \times F_{2i} \mid F_{2i} \in \mathcal{F}_2\}$

Construction of product automaton

input: GBAs $A_1 = (Q_1, \Sigma, \delta_1, Q_{01}, \mathcal{F}_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, Q_{02}, \mathcal{F}_2)$

output: a GBA $B = (Q_1 \times Q_2, \Sigma, \delta, Q_{01} \times Q_{02}, \mathcal{F})$ representing $L(A_1) \cap L(A_2)$

- $\delta = \{((p_1, p_2), a, (q_1, q_2)) \mid (p_1, a, q_1) \in \delta_1 \text{ and } (p_2, a, q_2) \in \delta_2\}$
- $\mathcal{F} = \{F_{1i} \times Q_2 \mid F_{1i} \in \mathcal{F}_1\} \cup \{Q_1 \times F_{2i} \mid F_{2i} \in \mathcal{F}_2\}$



Construction of product automaton

input: GBAs $A_1 = (Q_1, \Sigma, \delta_1, Q_{01}, \mathcal{F}_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, Q_{02}, \mathcal{F}_2)$

output: a GBA $B = (Q_1 \times Q_2, \Sigma, \delta, Q_{01} \times Q_{02}, \mathcal{F})$ representing $L(A_1) \cap L(A_2)$

- $\delta = \{((p_1, p_2), a, (q_1, q_2)) \mid (p_1, a, q_1) \in \delta_1 \text{ and } (p_2, a, q_2) \in \delta_2\}$
- $\mathcal{F} = \{F_{1i} \times Q_2 \mid F_{1i} \in \mathcal{F}_1\} \cup \{Q_1 \times F_{2i} \mid F_{2i} \in \mathcal{F}_2\}$

Lemma

$$L(B) = L(A_1) \cap L(A_2).$$

Theorem

Let $B = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a GBA. The following conditions are equivalent.

- 1 $L(B) \neq \emptyset$
- 2 There exists a nontrivial SCC of B reachable from Q_0 and such that the SCC contains at least one state of each $F_i \in \mathcal{F}$.
- 3 There exists an accepting run of B of the form $\tau.\rho^\omega$ (so-called *lasso-shaped*).

Theorem

Let $B = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a GBA. The following conditions are equivalent.

- 1 $L(B) \neq \emptyset$
- 2 There exists a nontrivial SCC of B reachable from Q_0 and such that the SCC contains at least one state of each $F_i \in \mathcal{F}$.
- 3 There exists an accepting run of B of the form $\tau.\rho^\omega$ (so-called *lasso-shaped*).

Proof.

- 1 \implies 2 Assume that $L(B) \neq \emptyset$. Hence, there exists an accepting run π . The run has to contain an infinite suffix contained in a single nontrivial SCC of B reachable from Q_0 . As the run visits each $F_i \in \mathcal{F}$ infinitely often, this SCC has to contain at least one state of each $F_i \in \mathcal{F}$.



Theorem

Let $B = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a GBA. The following conditions are equivalent.

- 1 $L(B) \neq \emptyset$
- 2 There exists a nontrivial SCC of B reachable from Q_0 and such that the SCC contains at least one state of each $F_i \in \mathcal{F}$.
- 3 There exists an accepting run of B of the form $\tau.\rho^\omega$ (so-called *lasso-shaped*).

Proof.

- 2 \implies 3 Assume that B has a nontrivial SCC reachable from Q_0 and containing at least one state of each $F_i \in \mathcal{F}$. Let τ be a sequence of successive states starting in Q_0 and leading to a state q of the SCC. Due to the properties of the SCC, there exists a sequence ρ of states of the SCC which starts in some successor of q , ends in q , and contains some state of each $F_i \in \mathcal{F}$. Then $\tau.\rho^\omega$ is an accepting run. \square

Theorem

Let $B = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a GBA. The following conditions are equivalent.

- 1 $L(B) \neq \emptyset$
- 2 There exists a nontrivial SCC of B reachable from Q_0 and such that the SCC contains at least one state of each $F_i \in \mathcal{F}$.
- 3 There exists an accepting run of B of the form $\tau.\rho^\omega$ (so-called *lasso-shaped*).

Proof.

3 \implies 1 Obvious.



Algorithms checking disjointness of A_K and $A_{\neg\varphi}$

Algorithm based on SCC decomposition

Emptiness check by SCC decomposition

input : a GBA $B = (Q, \Sigma, \delta, Q_0, \mathcal{F})$

output: *true* if $L(B) = \emptyset$; *false* otherwise

procedure *isGBAempty*

 remove unreachable states from the automaton

 decompose the automaton into SCCs

if some nontrivial SCC contains at least one state of each $F_i \in \mathcal{F}$ **then**

return *false*

else

return *true*

Emptiness check by SCC decomposition

input : a GBA $B = (Q, \Sigma, \delta, Q_0, \mathcal{F})$

output: *true* if $L(B) = \emptyset$; *false* otherwise

procedure *isGBAempty*

remove unreachable states from the automaton

decompose the automaton into SCCs

if some nontrivial SCC contains at least one state of each $F_i \in \mathcal{F}$ **then**

return *false*

else

return *true*

- if $L(B) \neq \emptyset$, a **counterexample** accepted by a lasso-shaped run $\tau.\rho^\omega$ can be constructed such that τ reaches the found SCC from Q_0 and ρ is a loop containing all states of the SCC
- the corresponding accepted word $u.v^\omega \in L(B)$ is also lasso-shaped

Emptiness check by SCC decomposition

pros

- simple
- SCC decomposition can be done in time $\mathcal{O}(|Q| + |\delta|)$

Emptiness check by SCC decomposition

pros

- simple
- SCC decomposition can be done in time $\mathcal{O}(|Q| + |\delta|)$

cons

- the whole GBA has to be known before the procedure starts
- in model checking, GBA is a product of A_K and $A_{\neg\varphi}$, where A_K is typically very large and described implicitly

Emptiness check by SCC decomposition

pros

- simple
- SCC decomposition can be done in time $\mathcal{O}(|Q| + |\delta|)$

cons

- the whole GBA has to be known before the procedure starts
- in model checking, GBA is a product of A_K and $A_{\neg\varphi}$, where A_K is typically very large and described implicitly

on-the-fly model checking algorithms

- the emptiness check explores the product automaton gradually and can detect nonemptiness without knowing the whole product
- the states and transitions of the product are constructed from $A_{\neg\varphi}$ and the implicit description of A_K only on demand

Algorithms checking disjointness of A_K and $A_{\neg\varphi}$

Nested DFS algorithm

Nested DFS check

- also called **double DFS**
- allows on-the-fly model checking
- checks emptiness of a BA (not generalized)
- can be easily used for model checking of a (not fair) Kripke structure K
- such K is transformed into a BA A_K where all states are accepting

Nested DFS check

- also called **double DFS**
- allows on-the-fly model checking
- checks emptiness of a BA (not generalized)
- can be easily used for model checking of a (not fair) Kripke structure K
- such K is transformed into a BA A_K where all states are accepting

construction of product BA for a BA with all states accepting and another BA

input: a BA $A_1 = (Q_1, \Sigma, \delta_1, Q_{01}, Q_1)$ and a BA $A_2 = (Q_2, \Sigma, \delta_2, Q_{02}, F_2)$

output: a BA $B = (Q_1 \times Q_2, \Sigma, \delta, Q_{01} \times Q_{02}, F)$ representing $L(A_1) \cap L(A_2)$

- $\delta = \{((p_1, p_2), a, (q_1, q_2)) \mid (p_1, a, q_1) \in \delta_1 \text{ and } (p_2, a, q_2) \in \delta_2\}$
- $F = Q_1 \times F_2$

Theorem

Let $B = (Q, \Sigma, \delta, Q_0, F)$ be a BA. The $L(B) \neq \emptyset \iff$ there exist a run of the form $\tau.\rho^\omega$ where ρ starts with a state of F .

Theorem

Let $B = (Q, \Sigma, \delta, Q_0, F)$ be a BA. The $L(B) \neq \emptyset \iff$ there exist a run of the form $\tau.\rho^\omega$ where ρ starts with a state of F .

Proof.

\Leftarrow Follows directly from the fact that $\tau.\rho^\omega$ is an accepting run of B .

Theorem

Let $B = (Q, \Sigma, \delta, Q_0, F)$ be a BA. The $L(B) \neq \emptyset \iff$ there exist a run of the form $\tau.\rho^\omega$ where ρ starts with a state of F .

Proof.

- \Leftarrow Follows directly from the fact that $\tau.\rho^\omega$ is an accepting run of B .
- \Rightarrow Assume that $L(B) \neq \emptyset$. There exists an accepting run $\pi = s_0s_1 \dots \in Q^\omega$. As π is accepting, there exists a state $q \in \text{inf}(\pi) \cap F$. Let $i < j$ be such that s_i, s_j are the first two occurrences of q in π . Further, let $\tau = s_0s_1 \dots s_{i-1}$ and $\rho = s_is_{i+1} \dots s_{j-1}$. Then $\tau.\rho^\omega = s_0s_1 \dots s_{i-1}.(s_is_{i+1} \dots s_{j-1})^\omega$ is a run of B and ρ starts with $s_i \in F$.



Nested DFS algorithm

- the algorithm uses two nested instances of depth-first search
- the first DFS searches for reachable accepting states
- the nested DFS looks for a cycle from accepting states
- the algorithm terminates when a cycle from an accepting state is found
- all executions of the nested DFS share the information about visited states:
without this feature, the overall complexity of nested DFS executions would be $\mathcal{O}(|F| \cdot (|Q| + |\delta|))$

Nested DFS algorithm

input : a BA $B = (Q, \Sigma, \delta, Q_0, F)$

output: *true* if $L(B) = \emptyset$;
false otherwise

procedure *isBAempty*

visited1 $\leftarrow \emptyset$

visited2 $\leftarrow \emptyset$

onStack $\leftarrow \emptyset$

forall $q_0 \in Q_0$ **do**

 dfs1(q_0)

terminate *true*

procedure *dfs1*(q)

visited1 \leftarrow visited1 $\cup \{q\}$

onStack \leftarrow onStack $\cup \{q\}$

forall *successors* q' of q **do**

if $q' \notin$ visited1 **then** dfs1(q')

if $q \in F$ **then** dfs2(q)

onStack \leftarrow onStack $\setminus \{q\}$

procedure *dfs2*(q)

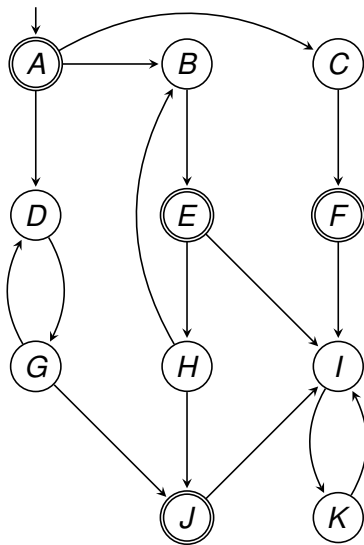
visited2 \leftarrow visited2 $\cup \{q\}$

forall *successors* q' of q **do**

if $q' \in$ onStack **then** **terminate** *false*

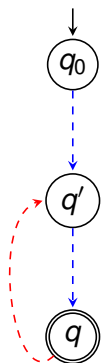
if $q' \notin$ visited2 **then** dfs2(q')

Example



Nested DFS algorithm

- if the algorithm returns *false*, it can produce a **counterexample** corresponding to the lasso-shaped accepting run given by the current content of DFS stacks
- let q be the accepting state from which the last nested DFS was executed
- let q' be the state on stack discovered by the nested DFS



--- stack of the first DFS

--- stack of the nested DFS

accepting lasso-shaped run:

$$q_0 \dashrightarrow q' \dashrightarrow (q \dashrightarrow q' \dashrightarrow)^\omega$$

Correctness of the nested DFS algorithm

Theorem

The nested DFS algorithm returns false $\iff L(B) \neq \emptyset$.

Correctness of the nested DFS algorithm

Theorem

The nested DFS algorithm returns *false* $\iff L(B) \neq \emptyset$.

Proof.

\implies is obvious. We prove \impliedby by contradiction. Assume that $L(B) \neq \emptyset$ and the algorithm returns *true*. As $L(B) \neq \emptyset$, there is a run $\tau.\rho^\omega$ where ρ starts with a state $q \in F$. When the nested DFS is started from q , there has to be a state q' on the stack of the first DFS reachable from q . Nested DFS has not found the cycle because q' is reachable only via $r \in \text{visited2}$. Assume that q is the first such a state and that r is added to visited2 during the nested DFS started from $q'' \in F$.

- 1 If q'' is reachable from q , then there is a cycle $q'' \dashrightarrow r \dashrightarrow q \dashrightarrow q''$ which is the contradiction with the assumption that q is the first such state.
- 2 If q'' is not reachable from q , then q is reachable from q'' via $q'' \dashrightarrow r \dashrightarrow q$. We have the contradiction with the fact that the first DFS backtracks from a state only after it backtracks from all states reachable from them and thus nested DFS from q'' cannot be executed before the nested DFS from q . \square

Complexity of the nested DFS algorithm

complexity of the **first DFS**

- time: $\mathcal{O}(|Q| + |\delta|)$
- space: $\mathcal{O}(|Q|)$

complexity of the **first DFS**

- time: $\mathcal{O}(|Q| + |\delta|)$
- space: $\mathcal{O}(|Q|)$

complexity of the **nested DFS** (all executions)

- time: $\mathcal{O}(|Q| + |\delta|)$
- space: $\mathcal{O}(|Q|)$

complexity of the **first DFS**

- time: $\mathcal{O}(|Q| + |\delta|)$
- space: $\mathcal{O}(|Q|)$

complexity of the **nested DFS** (all executions)

- time: $\mathcal{O}(|Q| + |\delta|)$
- space: $\mathcal{O}(|Q|)$

overall complexity

- time: $\mathcal{O}(|Q| + |\delta|)$
- space: $\mathcal{O}(|Q|)$

Algorithms checking disjointness of A_K and $A_{\neg\varphi}$

Optimizations

Definition (terminal BA, weak BA)

Let B be a Büchi automaton with alphabet Σ . A Büchi automaton is **terminal** if each accepting state has a loop transition under each $a \in \Sigma$.

A Büchi automaton is **weak** if each strongly connected component consists either of accepting states or of nonaccepting states.

Definition (terminal BA, weak BA)

Let B be a Büchi automaton with alphabet Σ . A Büchi automaton is **terminal** if each accepting state has a loop transition under each $a \in \Sigma$.

A Büchi automaton is **weak** if each strongly connected component consists either of accepting states or of nonaccepting states.

- many LTL properties translate to terminal or weak BA
- if this is the case, simpler emptiness checks can be used

- assume that $A_{\neg\varphi}$ is a **terminal** BA and each state of BA A_K is accepting and has a successor
- let B be the product BA of $A_{\neg\varphi}$ and A_K
- $L(B) \neq \emptyset$ iff B has a reachable accepting state
- instead of nested DFS, emptiness of $L(B)$ can be decided by a **single DFS** checking the reachability of an accepting state
- properties φ with terminal $A_{\neg\varphi}$ are called **safety** properties
- typical safety property: $G\neg err$

- assume that $A_{\neg\varphi}$ is a **weak** BA and each state of BA A_K is accepting
- let B be the product BA of $A_{\neg\varphi}$ and a BA A_K
- each cycle of B contains either only accepting states or no accepting state
- instead of nested DFS, emptiness of $L(B)$ can be decided by a **single DFS** that looks for a cycle and if a cycle is found, it checks whether the current state is accepting
- typical property φ with weak $A_{\neg\varphi}$: $G(a \implies Fb)$ (**responsivity**)

Extending LTL with release

- another derived LTL operator **release**: $\varphi R \psi \equiv \neg(\neg\varphi U \neg\psi)$
- equivalently: $\varphi R \psi \equiv G\psi \vee \psi U(\psi \wedge \varphi)$

aRb $b b b b \dots$ or $b b \dots b(ab) \dots$

- by adding \perp , \vee , and R to the basic syntax of LTL, we can push all negations towards atomic propositions using equivalences

$$\begin{aligned}\neg(\varphi U \psi) &\equiv \neg\varphi R \neg\psi \\ \neg(\varphi R \psi) &\equiv \neg\varphi U \neg\psi \\ \neg X\varphi &\equiv X\neg\varphi \\ \neg(\varphi \vee \psi) &\equiv \neg\varphi \wedge \neg\psi \\ \neg(\varphi \wedge \psi) &\equiv \neg\varphi \vee \neg\psi \\ \neg\neg a &\equiv a\end{aligned}$$

Definition (hierarchy of LTL classes)

- $\Sigma_0 = \Pi_0$ is the smallest set of LTL formulas containing all atomic propositions and closed under application of \wedge , \vee , \neg , and X .
- Σ_{i+1} is the smallest set of LTL formulas containing Π_i and closed under application of \wedge , \vee , X , and U .
- Π_{i+1} is the smallest set of LTL formulas containing Σ_i and closed under application of \wedge , \vee , X , and R .

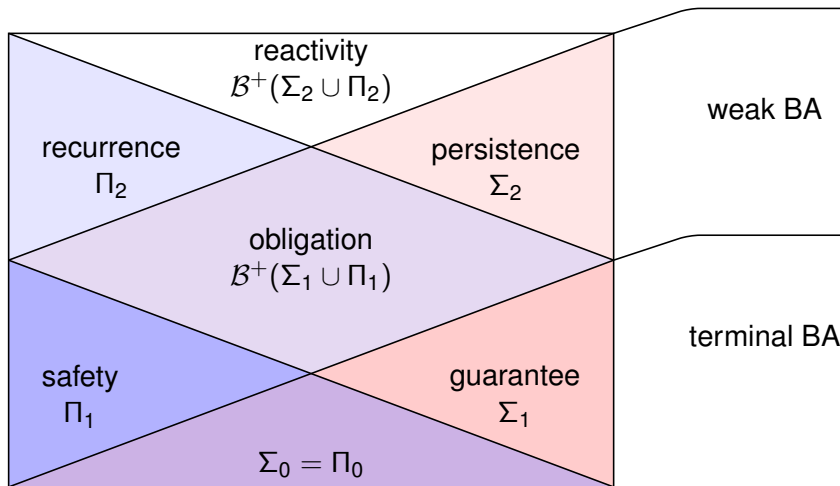
Definition (hierarchy of LTL classes)

- $\Sigma_0 = \Pi_0$ is the smallest set of LTL formulas containing all atomic propositions and closed under application of \wedge , \vee , \neg , and X .
 - Σ_{i+1} is the smallest set of LTL formulas containing Π_i and closed under application of \wedge , \vee , X , and U .
 - Π_{i+1} is the smallest set of LTL formulas containing Σ_i and closed under application of \wedge , \vee , X , and R .
-
- for each $\varphi \in \Pi_i$, $\neg\varphi$ can be transformed (by pushing negations towards atomic propositions) to an equivalent formula $\psi \in \Sigma_i$
 - for each $\varphi \in \Sigma_i$, $\neg\varphi$ can be transformed (by pushing negations towards atomic propositions) to an equivalent formula $\psi \in \Pi_i$

Properties corresponding to LTL classes

- Σ_1 describes **guarantee** properties
- Π_1 describes **safety** properties
- $\mathcal{B}^+(\Sigma_1 \cup \Pi_1)$ describes **obligation** properties
- Σ_2 describes **persistence** properties
- Π_2 describes **recurrence** (or **response**) properties
- $\mathcal{B}^+(\Sigma_2 \cup \Pi_2)$ describes **reactivity** properties
- the LTL classes are sometimes called guarantee, safety, ... formulae

Hierarchy of properties



- each language definable in LTL is definable in $B^+(\Sigma_2 \cup \Pi_2)$
- formulae of Σ_1 can be translated to terminal BA
- formulae of Σ_2 can be translated to weak BA

Fighting state-space explosion

state-space explosion problem

- Kripke structure (and thus also the product automaton) can have enormous number of states, often exponential in the size of its implicit description
- model checking algorithms often run out of memory

Fighting state-space explosion

state-space explosion problem

- Kripke structure (and thus also the product automaton) can have enormous number of states, often exponential in the size of its implicit description
- model checking algorithms often run out of memory

methods fighting the problem

- low-level techniques for saving memory
 - lossless compression of states in memory
 - heuristics based on lossy compression
 - forgetting some visited states

Fighting state-space explosion

state-space explosion problem

- Kripke structure (and thus also the product automaton) can have enormous number of states, often exponential in the size of its implicit description
- model checking algorithms often run out of memory

methods fighting the problem

- low-level techniques for saving memory
 - lossless compression of states in memory
 - heuristics based on lossy compression
 - forgetting some visited states
- on-the-fly approaches
 - can help to find a counterexample, but does not help for correct systems

Fighting state-space explosion

state-space explosion problem

- Kripke structure (and thus also the product automaton) can have enormous number of states, often exponential in the size of its implicit description
- model checking algorithms often run out of memory

methods fighting the problem

- low-level techniques for saving memory
 - lossless compression of states in memory
 - heuristics based on lossy compression
 - forgetting some visited states
- on-the-fly approaches
 - can help to find a counterexample, but does not help for correct systems
- state-space reduction methods
 - partial order reduction (only for LTL properties without X operators)
 - symmetry reduction (avoids exploration of symmetric parts)
 - abstraction

Fighting state-space explosion

state-space explosion problem

- Kripke structure (and thus also the product automaton) can have enormous number of states, often exponential in the size of its implicit description
- model checking algorithms often run out of memory

methods fighting the problem

- low-level techniques for saving memory
 - lossless compression of states in memory
 - heuristics based on lossy compression
 - forgetting some visited states
- on-the-fly approaches
 - can help to find a counterexample, but does not help for correct systems
- state-space reduction methods
 - partial order reduction (only for LTL properties without X operators)
 - symmetry reduction (avoids exploration of symmetric parts)
 - abstraction
- symbolic representation of sets of states (by formulae or BDDs)
- parallel and distributed algorithms

Action-based version of LTL model checking

actions

- basic observable information attached to each transition of the system
- for example: *gate opening*, *process P entered critical section*
- *Act* denotes a countable set of actions

actions

- basic observable information attached to each transition of the system
- for example: *gate opening, process P entered critical section*
- *Act* denotes a countable set of actions

- basic formalism for action-based systems is a **labeled transition system**

Definition (labeled transition system, LTS)

A **labeled transition systems (LTS)** is a tuple $M = (S, Act', \delta, S_0)$, where

- S is a set of **states**,
- $Act' \subseteq Act$ is a finite set of **actions**,
- $\delta \subseteq S \times Act' \times S$ is a **transition relation**,
- $S_0 \subseteq S$ is a set of initial states.

Definition (labeled transition system, LTS)

A **labeled transition system (LTS)** is a tuple $M = (S, Act', \delta, S_0)$, where

- S is a set of **states**,
- $Act' \subseteq Act$ is a finite set of **actions**,
- $\delta \subseteq S \times Act' \times S$ is a **transition relation**,
- $S_0 \subseteq S$ is a set of initial states.

Definition (run, trace)

Let $M = (S, Act', \delta, S_0)$ be an LTS. A **run** of M is an infinite sequence $\pi = (s_0, a_0, s_1)(s_1, a_1, s_2)(s_2, a_2, s_3) \dots \in \delta^\omega$ of adjacent transitions such that $s_0 \in S_0$.

The **trace** of π is then the infinite word $\sigma(\pi) = a_0 a_1 a_2 \dots$

modified **syntax** of LTL

- the only change is that a ranges over Act (instead of AP)

modified **syntax** of LTL

- the only change is that a ranges over Act (instead of AP)

modified **semantics** of LTL

- we interpret LTL on infinite words $w = w(0)w(1)\dots \in Act^\omega$
- the only change in the inductive definition of $w \models \varphi$ is the line

$$w \models a \text{ iff } a = w(0) \quad (\text{instead of } w \models a \text{ iff } a \in w(0))$$

Definition

Let $M = (S, Act', \delta, S_0)$ be an LTS and φ be an LTL formula.

A run π of M **satisfies** φ , written $\pi \models \varphi$, if $\sigma(\pi) \models \varphi$.

M **satisfies** φ , written $M \models \varphi$, if $\pi \models \varphi$ holds for every run π of M .

The goal of action-based LTL model checking

Definition

Let $M = (S, Act', \delta, S_0)$ be an LTS and φ be an LTL formula.

A run π of M **satisfies** φ , written $\pi \models \varphi$, if $\sigma(\pi) \models \varphi$.

M **satisfies** φ , written $M \models \varphi$, if $\pi \models \varphi$ holds for every run π of M .

The **goal of action-based LTL model checking** is to decide whether a given LTS M satisfies a given LTL formula φ . In the negative case, model checking should provide a **counterexample**, i.e., a run π of M such that $\pi \not\models \varphi$.

The automata-based approach to LTL model checking of finite LTS is basically identical as for finite Kripke structures.

changes

- Büchi automata use the alphabet $\Sigma = Act'$ given by the LTS
- we assume that φ contains only actions from Act' (otherwise, we extend Act')
- LTS $M = (S, Act', \delta, S_0)$ is transformed into a BA $A_M = (S, Act', \delta, S_0, S)$
- modification of LTL \rightarrow self-loop alternating BA translation

$\delta(q_a, l) = \top$ if $a = l$, \perp otherwise (instead of $\delta(q_a, l) = \top$ if $a \in l$, \perp otherwise)