Safety Proof Synthesis for Regular Transition Systems

Chih-Duo Hong

FLOLAC 2023

Transition system

- A *transition system* is a triple (S, I, T), where
  - S is the set of states
  - $I \subseteq S$ is the set of initial states
  - $T \subseteq S \times S$ is the set of transitions
- A *trace* of (S, I, T) is a sequence $\sigma = \sigma_0\ \sigma_1\ \sigma_2 \ldots \in S^{\omega}$ such that
  - $\sigma_0 \in I$
  - for all $i \geq 0$, $(\sigma_i, \sigma_{i+1}) \in T$
- That is, a trace is a finite/infinite sequence of consecutive transitions starting from an initial state.

Safety property

- Safety properties are concerned with the assurance that certain undesirable behaviors will never occur in a system
- Typical safety properties of software:
  1. **Division by zero:** A program will never divide a number by zero
  2. **Null dereference:** A program will never dereference a null or uninitialized pointer
  3. **Data race:** A shared variable will never be updated simultaneously
- Safety of a transition system
  - Does every trace never reach a bad state?
- Model checking a liveness property
  - Yes + proof
  - No + counterexample (a system trace that reaches a bad state)

Liveness property

- Liveness properties are concerned with the assurance that certain desirable behaviors will eventually occur in a system
- Typical liveness properties of software:
  1. **Termination:** A program will eventually terminate
  2. **Response:** A system will respond to an input event within a bounded time frame
  3. **Progress:** A thread will eventually make progress and not get stuck in a deadlock
- Liveness of a transition system
  - Does every trace eventually reach a good state?
- Model checking a liveness property
  - Yes + proof
  - No + counterexample (a system trace that never reaches a good state)

Symbolic transition system

- We usually specify and reason about a transition system using a *symbolic representation*

- In this lecture, we will introduce two symbolic representations for infinite-state transition systems:

  1. Logical formulas (over a background theory)
  2. Regular languages

Formulas as symbolic representation

- A *symbolic transition system* is a tuple (V, I, T), where
  - V is a set of variables,
  - I is a formula over variables V
  - T is a formula over variables $V \cup V'$

    (E.g., $i' = i + 1$ is a formula over $\{i\} \cup \{i'\}$ that increments i by 1)
- A *state* $\sigma \in A$ is a type-consistent assignment to variables in V
- A trace of (V, I, T) is a sequence $\sigma = \sigma_0\ \sigma_1\ \sigma_2 \ldots \in A^{\omega}$, where
  - $\sigma_0 \vDash I(V)$
  - $\sigma_i, \sigma'_{i+1} \vDash T(V, V')$ for all $i \geq 0$

Example: the Collatz transition system

- Consider the following operation on a natural number:
  - If the number is even, divide it by two.
  - If the number is odd, triple it and add one.
- Applying this operation to a number repeatedly will generate a sequence, for example: $21 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$
- The corresponding symbolic transition system is (V, I, T), where $V := \{x\}$, $I \coloneqq (x \geq 1)$, and T is defined in Presburger arithmetic as

$$(\exists k.\, x = 2k \wedge x' = k) \vee (\exists k.\, x = 2k + 1 \wedge x' = 3x + 1)$$

Example: the Collatz transition system

- Consider the following operation on a natural number:
  - If the number is even, divide it by two.
  - If the number is odd, triple it and add one.
- Applying this operation to a number repeatedly will generate a sequence, for example: $21 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

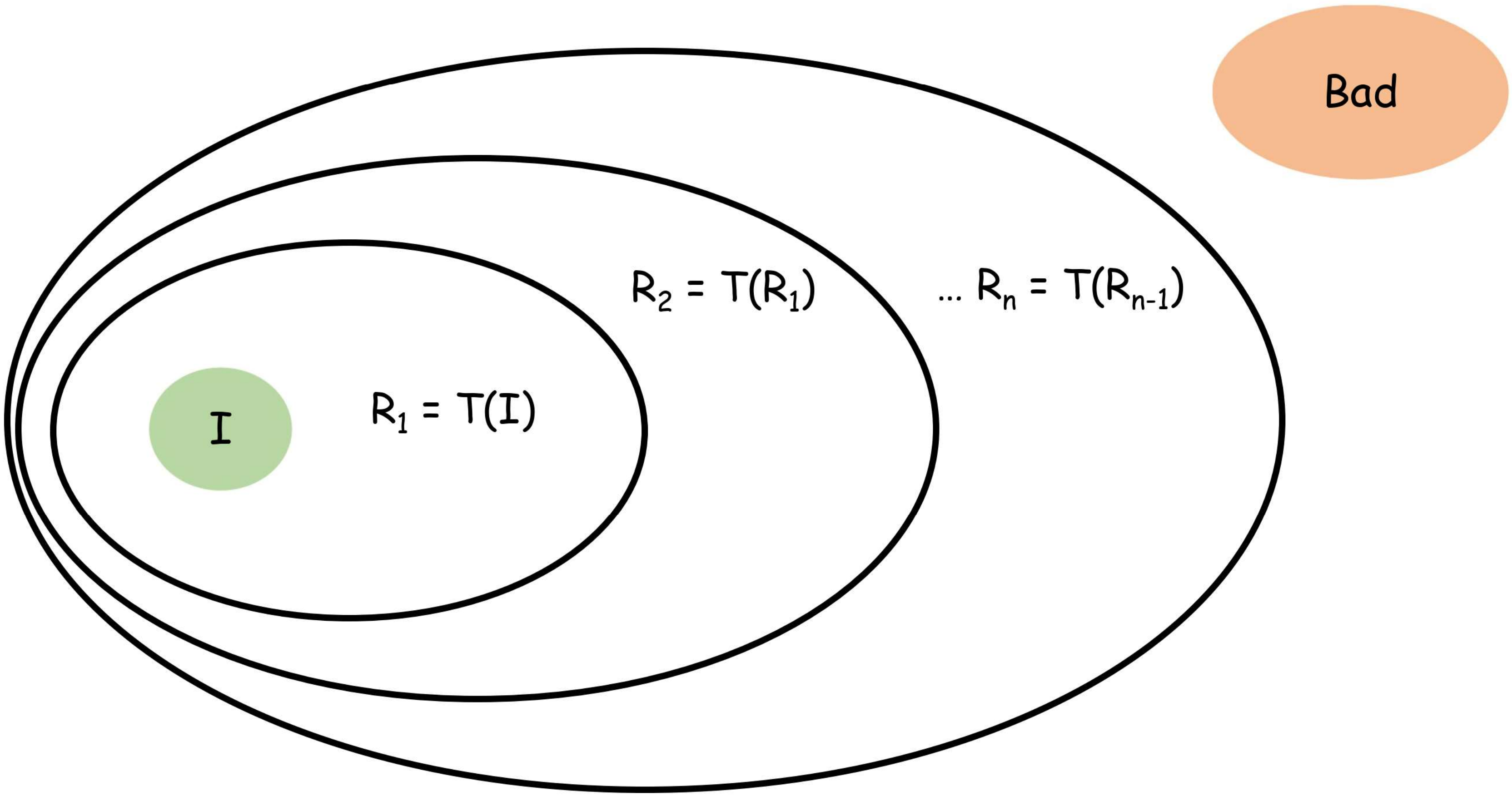
An example safety property:

"Every sequence starting from a power of 2 will reach no odd numbers but 1."

An example liveness property:

"Every sequence will eventually reach 1."

Forward reachability analysis



$$T(A) \coloneqq \{\, s' : s \in A \text{ and } (s, s') \in T \,\}$$

Inductive invariant

A set of states Inv is an **inductive invariant** if it satisfies the following three conditions:

- Initiation: $I \subseteq Inv$
- Consecution: $T(Inv) \subseteq Inv$
- Safety: $Inv \cap B = \emptyset$

When I, F, B, Inv are expressed in formulas, these conditions are equivalent to

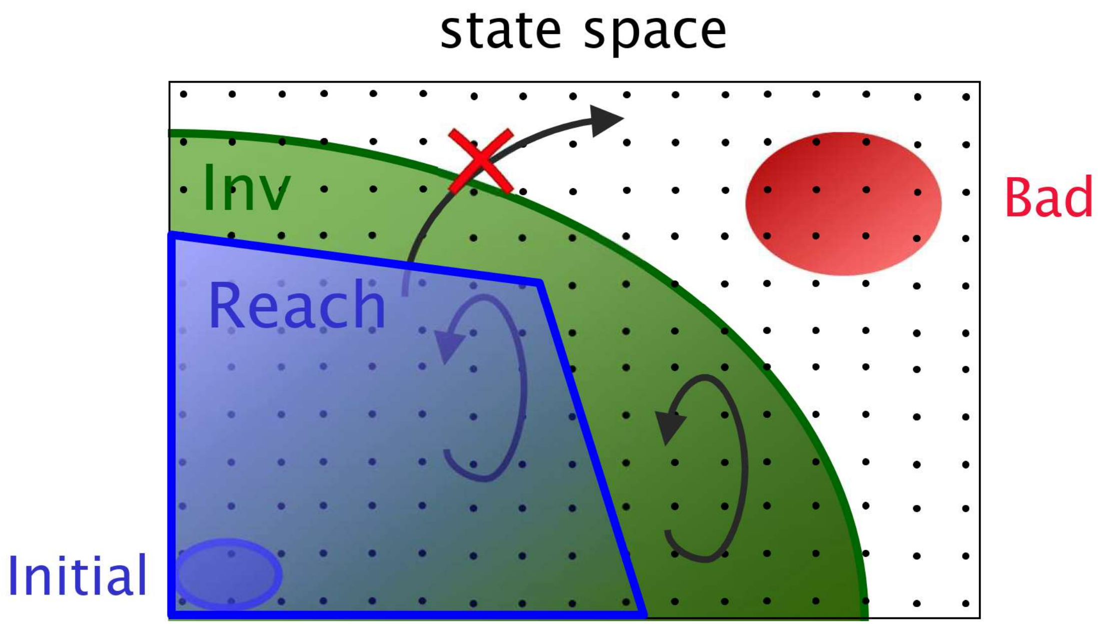
$$I(V) \Rightarrow Inv(V)$$

$$Inv(V) \land T(V, V') \Rightarrow Inv(V')$$

$$Inv(V) \Rightarrow \neg B(V)$$

Inductive invariant (cont'd)

- Initiation: I ⊆ Inv
- Consecution: T(Inv) ⊆ Inv
- Safety: Inv ∩ B = ∅



A system is safe iff it has an inductive invariant

Example: inductive invariant

- Consider a symbolic transition system (V, I, T), where

$$V := \{x, y\}$$
$$I \coloneqq x = 1 \land y = 1$$
$$T \coloneqq (x' = x + y) \land (y' = y + x)$$

- We want to prove the safety property $P \coloneqq y \geq 1$.

Example: inductive invariant (cont'd)

$P \coloneqq y \geq 1$ is not an inductive invariant

- $I \Rightarrow P$:
  - $(x = 1 \wedge y = 1) \Rightarrow y \geq 1$
- But $P \wedge T \not\Rightarrow P'$:
  - $y \geq 1 \wedge (x' = x + y \wedge y' = x + y) \not\Rightarrow y' \geq 1$

$V \coloneqq \{x, y\}$
$I \coloneqq x = 1 \wedge y = 1$
$T \coloneqq (x' = x + y) \wedge (y' = y + x)$
$P \coloneqq y \geq 1$

$I(V) \Rightarrow Inv(V)$

$Inv(V) \wedge T(V, V') \Rightarrow Inv(V')$

$Inv(V) \Rightarrow \neg B(V)$

Example: inductive invariant (cont'd)

$P := y \geq 1$ is not an inductive invariant

- I ⇒ P :
  - $(x = 1 \wedge y = 1) \Rightarrow y \geq 1$
- But $P \wedge T \not\Rightarrow P'$:
  - $y \geq 1 \wedge (x' = x + y \wedge y' = x + y) \not\Rightarrow y' \geq 1$

$V := \{x, y\}$
$I := x = 1 \wedge y = 1$
$T := (x' = x + y) \wedge (y' = y + x)$
$P := y \geq 1$

Consider $Inv := x \geq 0 \wedge y \geq 1$

- $(x = 1 \wedge y = 1) \Rightarrow x \geq 0 \wedge y \geq 1$
- $x \geq 0 \wedge y \geq 1 \wedge (x' = x + y \wedge y' = x + y) \Rightarrow x' \geq 0 \wedge y' \geq 1$
- $x \geq 0 \wedge y \geq 1 \Rightarrow y \geq 1$

Property proved!

I(V) ⇒ Inv(V)
Inv(V) ∧ T(V, V') ⇒ Inv(V')
Inv(V) ⇒ ¬B(V)

Example: inductive invariant (cont'd)

Induction hypothesis

$P \coloneqq y \geq 1$ is n… Base case …nvariant

$V \coloneqq \{x, y\}$
$I \coloneqq x = 1 \wedge y = 1$
$T \coloneqq (x' = x + y) \wedge (y' = y + x)$
$P \coloneqq y \geq 1$

- $I \Rightarrow P$:
  - $(x = 1 \wedge y = 1)$ … Induction step
- But $P \wedge T \nRightarrow P'$:
  - $y \geq 1 \wedge (x' = x + y \wedge y' = x + y)$ …

Strengthening the
induction hypothesis

Consider Inv $\coloneqq$ $\mathbf{x \geq 0} \wedge y \geq 1$

- $(x = 1 \wedge y = 1) \Rightarrow x \geq 0 \wedge y \geq 1$
- $x \geq 0 \wedge y \geq 1 \wedge (x' = x + y \wedge y' = x + y) \Rightarrow x' \geq 0 \wedge y' \geq 1$
- $x \geq 0 \wedge y \geq 1 \Rightarrow y \geq 1$

Property proved!

Symbolic transition system

- We usually specify and reason about a transition system using a *symbolic representation*

- In this lecture, we introduce two common symbolic representation for infinite transition systems:

  1. Logical formulas (over a background theory)

  2. Regular languages

Regular language as symbolic representation

- For a finite alphabet Σ, define $\Sigma_\# \coloneqq \Sigma \uplus \{\#\}$ with padding symbol $\#$.
- A regular language $L \subseteq \Sigma_\#^*$ encodes a set of words

$$[\![L]\!] \coloneqq \{w : w\#^k \in L \text{ for all } k \geq 0\} \subseteq \Sigma^*$$

Regular language as symbolic representation

- For a finite alphabet Σ, define $\Sigma_\# \coloneqq \Sigma \uplus \{\#\}$ with padding symbol $\#$.
- A regular language $L \subseteq \Sigma_\#^*$ encodes a set of words

$$[\![L]\!] \coloneqq \{w : w\#^k \in L \text{ for all } k \geq 0\} \subseteq \Sigma^*$$

- The *convolution* of two words u and v in Σ^* is defined as $u \otimes v \coloneqq \begin{bmatrix} u_1 \\ v_1 \end{bmatrix} \cdots \begin{bmatrix} u_n \\ v_n \end{bmatrix} \in (\Sigma_\# \times \Sigma_\#)^*$, where $n = \max\{|u|, |v|\}$ and

$$u_k = \begin{cases} u[k], & k < |u| \\ \#, & k \geq |u| \end{cases} \quad \text{and} \quad v_k = \begin{cases} v[k], & k < |v| \\ \#, & k \geq |v| \end{cases} \quad \text{for } 0 \leq k < n.$$

Regular language as symbolic representation

- For a finite alphabet Σ, define $\Sigma_\# \coloneqq \Sigma \uplus \{\#\}$ with padding symbol $\#$.
- A regular language $L \subseteq \Sigma_\#^*$ encodes a set of words

$$[\![L]\!] \coloneqq \{w : w\#^k \in L \text{ for all } k \geq 0\} \subseteq \Sigma^*$$

- The *convolution* of two words u and v in Σ^* is defined as $u \otimes v \coloneqq \begin{bmatrix} u_1 \\ v_1 \end{bmatrix} \cdots \begin{bmatrix} u_n \\ v_n \end{bmatrix} \in (\Sigma_\# \times \Sigma_\#)^*$, where $n = \max\{|u|, |v|\}$ and

$$u_k = \begin{cases} u[k], & k < |u| \\ \#, & k \geq |u| \end{cases} \quad \text{and} \quad v_k = \begin{cases} v[k], & k < |v| \\ \#, & k \geq |v| \end{cases} \quad \text{for } 0 \leq k < n.$$

- A regular language $L \subseteq (\Sigma_\# \times \Sigma_\#)^*$ encodes a binary relation

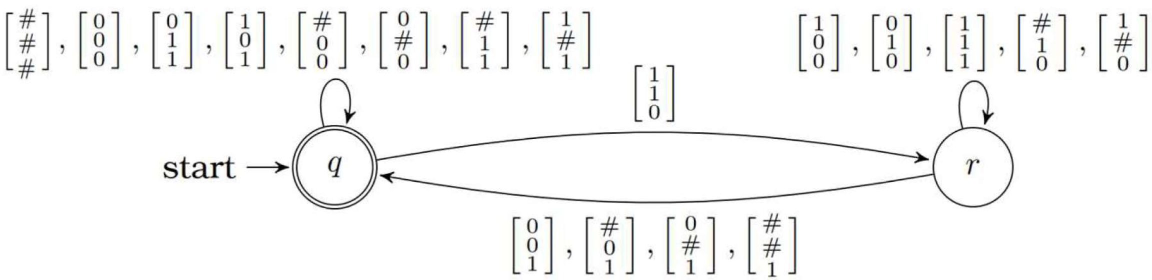
$$[\![L]\!] \coloneqq \{(u, v) : u \otimes v \begin{bmatrix} \# \\ \# \end{bmatrix}^k \in L \text{ for all } k \geq 0\} \subseteq \Sigma^* \times \Sigma^*$$

- We use L_E to denote the minimal language L satisfying $[\![L]\!] = E$

Example: regular language as symbolic representation

- We can encode the structure $\langle \mathbb{N}, 0, 1, +, < \rangle$ in regular languages by representing natural numbers in binary with the least significant bit first and without tailing zeros.

- We can define $L_{\mathbb{N}} := (\varepsilon + (0+1)^* 1)\#^*$, $L_{zero} := \#^*$, $L_{one} := 1\#^*$. The language $L_+ = L(\{(x, y, z) : x + y = z\})$ can be defined by intersecting $L_{\mathbb{N}} \times L_{\mathbb{N}} \times L_{\mathbb{N}}$ with the language of



- In fact, every relation definable in $\mathrm{FO}(\mathbb{N}, 0, 1, +, <)$, which is equivalent to Presburger arithmetic, can be represented by a regular language under this encoding!

Regular transition system

A **regular transition system** (RTS) is a triple (Σ, I, T), where I is a regular language over alphabet $\Sigma_{\#}$, and T is a regular language over alphabet $\Sigma_{\#} \times \Sigma_{\#}$.

An RTS (Σ, I, T) induces a transition system $(\Sigma^*, [\![I]\!], [\![T]\!])$:

- Each state is a finite word over the alphabet Σ
- The set of initial states is a regular set $[\![I]\!] \subseteq \Sigma^*$
- The transition relation is regular relation $[\![T]\!] \subseteq \Sigma^* \times \Sigma^*$

Regular transition system (cont'd)

A **regular transition system** (RTS) is a triple (Σ, I, T), where I is a regular language over alphabet $\Sigma_\#$, and T is a regular language over alphabet $\Sigma_\# \times \Sigma_\#$.

Example The Collatz transition system is (isomorphic to) an RTS.

Presburger definable relations can be encoded in regular languages.

Example The configuration graph of a Turing machine is an RTS.

A TM with a two-sided tape can be simulated by a TM with a one-sided tape.

A configuration of a one-sided TM can be encoded as a regular language $usaw\#^*$, where u is the tape content before the head, s is the control state, a is the tape symbol at the head position, and w is the tape content after the head.

Safety of regular transition systems

Fix an RTS (Σ, I, T). Let $B \subseteq \Sigma_{\#}^{*}$ denote the language representation of a set of bad states.

- The RTS (Σ, I, T) is *safe* if $[\![B]\!]$ cannot be reached from $[\![I]\!]$
- A **safety proof** is a regular language P satisfying
  - $I \subseteq P$
  - $P \cap B = \emptyset$
  - $T(P) \subseteq P$
- A regular transition system is safe iff it has a safety proof

Example: the Collatz transition system

The Collatz system applies the following operation on natural numbers:

- If the number is even, divide it by two.
- If the number is odd, triple it and add one.

We can specify the Collatz transition system as an RTS (Σ, I, T) by encoding natural numbers in binary with the least significant bit first without tailing zeros.

Consider the safety property: "Every sequence starting from a power of 2 will reach no odd numbers but 1."

We set $I \coloneqq 0^*1\#^*$ as the initial states and $B \coloneqq 1(0+1)(0+1)^*\#^*$ as the bad states. Observe that $T(\underbrace{0\cdots0}_{n \text{ zeros}}1\#^*) = \underbrace{0\cdots0}_{n-1 \text{ zeros}}1\#^*$ for each $n \geq 1$.

We therefore have $I \cap B = \emptyset$ and $T(I) \subseteq I$. Namely, I is itself a safety proof.

Regular model checking

- The **regular model checking** problem is to find a *regular* safety proof for a regular transition system
- A RTS may not have a regular proof even if it is safe!
- For some subclass of RTSs, a regular proof is guaranteed to exist when the system is safe
- For example, the set of reachable states is regular for RTSs like Petri nets, pushdown systems, and lossy-channel systems
- Such systems have a regular safety proof whenever they are safe

Regular model checking (cont'd)

- If an RTS has a regular proof when it is safe, then safety checking of the system is decidable. Idea: launch two procedures as follows at the same time

Procedure A:

```
while true do
  i := 1
  let A_i be the i-th DFA, and let P := L_A
  if I ⊆ P and P ∩ B = ∅ and T(P) ⊆ P then
    terminate and report "safe"
  i := i + 1
```

Procedure B:

```
while true do
  i := 0
  if B is reachable from I in i step then
    terminate and report "unsafe"
  i := i + 1
```

- Eventually one of the two procedures will terminate!

Learning proofs for regular model checking

- In the rest of this lecture, we will look at two methods to find a regular proof for an RTS:
  - SAT-based learning
  - L^*-based learning
- The SAT-based method is less scalable (i.e. it is not effective when all regular proofs are large). However, it has the same termination guarantee as brute-force enumeration.
- The L^*-based method is more scalable and is capable of finding very large regular proofs in practice. However, it is not guaranteed to find a regular proof even if one exists.

Learning proofs for regular model checking

- In the rest of this lecture, we will look at two methods to find a regular proof for an RTS:
  - SAT-based learning
  - L^*-based learning
- The SAT-based method is less scalable (i.e. it is not effective when all regular proofs are large). However, it has the same termination guarantee as brute-force enumeration.
- The L^*-based method is more scalable and is capable of finding very large regular proofs in practice. However, it is not guaranteed to find a regular proof even if one exists.

SAT-based learning for safety proofs

Fix a regular system (Σ, I, T) and a set of bad states B

For each $n \geq 1$, we construct a Boolean formula Φ_n such that a model of Φ_n corresponds to a DFA A of n states and vice versa

SAT-based learning of regular proofs:

$n := 1, C := \emptyset$
while true do
  construct Φ_n
  while $\Phi_n \wedge \Phi_C$ has a model α do
    construct a DFA A from α
    if L_{A} is a safety proof then
      return A
    let cex be a witness of the violation
    $C := C \cup \{cex\}$
  $n := n + 1$

SAT-based learning for safety proofs

Fix a regular system (Σ, I, T) and a set of bad states B

For each $n \geq 1$, we construct a Boolean formula Φ_n such that a model of Φ_n corresponds to a DFA A of n states and vice versa

SAT-based learning of regular proofs:

$n := 1, C \coloneqq \emptyset$
while true do
    construct Φ_n
    while $\Phi_n \wedge \Phi_C$ has a model α do
        construct a DFA A from α
        if $\boldsymbol{L_A}$ **is a safety proof** then
            return A
        let cex be a witness of the violation
        $C \coloneqq C \cup \{cex\}$
    $n \coloneqq n + 1$

$I \subseteq L_A$
$L_A \cap B = \emptyset$
$T(L_A) \subseteq L_A$

SAT-based learning for safety proofs

Fix a regular system (Σ, I, T) and a set of bad states B

For each $n \geq 1$, we construct a Boolean formula Φ_n such that a model of Φ_n corresponds to a DFA A

SAT-based learning

$\alpha \vDash \Phi_C$ iff for all $c \in C$,
c is not a witness of A_α
violating the proof rules

$n := 1, C := \emptyset$
while true do
  construct Φ_n
  while $\Phi_n \wedge \boldsymbol{\Phi_C}$ has a model α do
    construct a DFA A from α
    if L_A is a safety proof then
      return A
    let cex be a witness of the violation
    $\boldsymbol{C} \coloneqq \boldsymbol{C} \cup \{\boldsymbol{cex}\}$
  $n \coloneqq n + 1$

SAT encoding of DFA

Encoding of a DFA $(\Sigma, S, s_0, \delta, F)$

- Given Σ and S, it suffices to fix s_0 and define only δ and F.
- For each $i, j \in S$ and $a \in \Sigma$, we define a Boolean variable $t_{i,a,j}$ such that "$t_{i,a,j}$ is true" corresponds to "$\delta(i,a) = j$".
- For each $i \in S$, we define a Boolean variable f_i such that "f_i is true" corresponds to "$i \in F$".
- We use the following constraint to ensure that the DFA is deterministic and complete:

$$\left(\bigwedge_{i,j,k\in S,\ j\neq k,\ a\in\Sigma} \neg\left(t_{i,a,j} \wedge t_{i,a,k}\right)\right) \wedge \left(\bigwedge_{i\in S,\ a\in\Sigma} \bigvee_{j\in S} t_{i,a,j}\right)$$

SAT encoding of DFA (cont'd)

Encoding of a DFA $(\Sigma, S, s_0, \delta, F)$

- For each $n \geq 1$, we define a propositional formula $\phi^n_{\mathrm{DFA}}(\bar{t}, \bar{f})$ as

$$\left(\bigwedge_{1\leq i,j,k\leq n,\ j\neq k,\ a\in\Sigma} \neg(t_{i,a,j} \wedge t_{i,a,k})\right) \wedge \left(\bigwedge_{1\leq i\leq n,\ a\in\Sigma} \bigvee_{1\leq j\leq n} t_{i,a,j}\right)$$

  with free variables

$$\{\, t_{i,a,j} : 1 \leq i,j \leq n, a \in \Sigma \,\} \text{ and } \{\, f_i : 1 \leq i \leq n \,\}.$$

- Any $\alpha \vDash \phi^n_{\mathrm{DFA}}(\bar{t}, \bar{f})$ corresponds to a DFA $A_\alpha \coloneqq (\Sigma, S, s_0, \delta, F)$:
  - $S = \{1, \dots, n\},\ s_0 = 1$
  - For $i \in S$ and $a \in \Sigma$, $\delta(i, a) = j$ iff $\alpha(t_{i,a,j}) = \mathrm{true}$
  - $F = \{\, i : \alpha(f_i) = \mathrm{true} \,\}$

Counterexample refinement

SAT-based learning of regular proofs:

$n := 1, C \coloneqq \emptyset$
while true
  construct Φ_n
  while $\Phi_n \wedge \Phi_C$ has a model α
    construct a DFA A from α
    if L_A is a safety proof then
      return A
    let cex be a witness of the violation
    $C \coloneqq C \cup \{cex\}$
  $n \coloneqq n + 1$

Counterexample refinement (cont'd)

- **Positive counterexample**
  - A positive cex is a word supposed to be accepted by A.
  - We obtain a positive cex $w \in I \setminus L_A$ when $I \nsubseteq L_A$.
- **Negative counterexample**
  - A negative cex is a word not supposed to be accepted by A.
  - We obtain a negative cex $w \in L_A \cap B$ when $L_A \cap B \neq \emptyset$.
- **Implication counterexample**
  - An implication cex is a pair of words (w, w') such that "w is in L_A" implies "w' is in L_A"
  - We obtain an implication cex when $T(L_A) \nsubseteq L_A$. In such case, we can find a pair of words (w, w') such that $w \in L_A$ and $w' \in T(w) \setminus L_A$.

$I \subseteq L_A$

$L_A \cap B = \emptyset$

$T(L_A) \subseteq L_A$

SAT encoding of positive counterexample

Encoding the membership of a word

- Suppose we got a positive counterexample w
- We give a formula ϕ_w^n such that

$$\text{if } \alpha \vDash \phi_{\mathrm{DFA}}^n \wedge \phi_w^n, \text{ then } A_\alpha \text{ accepts } w$$

- We introduce variables $\{\, v_{k,i} : 0 \leq k \leq |w|,\ \ 1 \leq i \leq n \,\}$ and let

$$\phi_w^n \coloneqq v_{0,1} \wedge \left(\bigwedge_{1 \leq k \leq |w|} \bigvee_{1 \leq i \leq n} v_{k,i} \right) \wedge \left(\bigwedge_{1 \leq i \leq n} \left(v_{|w|,i} \Rightarrow f_i \right) \right)$$
$$\wedge \left(\bigwedge_{1 \leq k \leq |w|} \bigwedge_{1 \leq i,j \leq n} \left(v_{k-1,i} \wedge v_{k,j} \Rightarrow t_{i,a_k,j} \right) \right)$$

Intuitively, $\alpha(v_{k,i}) = \text{true}$ iff the DFA A_α reaches state i after reading the prefix $a_1 \cdots a_k$ of the word w.

SAT encoding of negative counterexample

Encoding the non-membership of a word

- Suppose we got a negative counterexample w
- We give a formula ψ_w^n such that

$$\text{if } \alpha \vDash \phi_{\mathrm{DFA}}^n \wedge \psi_w^n \text{, then } A_\alpha \text{ does not accept } w$$

- We introduce variables $\{\, u_{k,i} : 0 \le k \le |w|,\ \ 1 \le i \le n \,\}$ and let

$$\psi_w^n \coloneqq u_{0,1} \wedge \left(\bigwedge_{1 \le k \le |w|} \bigvee_{1 \le i \le n} u_{k,i} \right) \wedge \left(\bigwedge_{1 \le i \le n} \left(u_{|w|,i} \Rightarrow \neg f_i \right) \right)$$

$$\wedge \left(\bigwedge_{1 \le k \le |w|} \bigwedge_{1 \le i,j \le n} \left(u_{k-1,i} \wedge u_{k,j} \Rightarrow t_{i,a_k,j} \right) \right)$$

Intuitively, $\alpha(u_{k,i}) = \text{true}$ iff the DFA A_α reaches state i after reading the prefix $a_1 \cdots a_k$ of the word w.

SAT encoding of negative counterexample

Encoding the non-membership of a word

- Suppose we got a negative counterexample w
- We give a formula ψ_w^n such that

  if $\alpha \vDash$ … ot accept w

  works only if A_α is a complete DFA

- We introduce va… $|w|,\ \ 1 \leq i \leq n \}$ and let

$$\psi_w^n := u_{0,1} \wedge \left(\bigwedge_{1 \leq k \leq |w|} \bigvee_{1 \leq i \leq n} u_{k,i} \right) \wedge \left(\bigwedge_{1 \leq i \leq n} (u_{|w|,i} \Rightarrow \neg f_i) \right)$$

$$\wedge \left(\bigwedge_{1 \leq k \leq |w|} \bigwedge_{1 \leq i,j \leq n} (u_{k-1,i} \wedge u_{k,j} \Rightarrow t_{i,a_k,j}) \right)$$

Intuitively, $\alpha(u_{k,i}) = \text{true}$ iff the DFA A_α reaches state i after reading the prefix $a_1 \cdots a_k$ of the word w.

SAT-based learning for safety proofs

SAT-based learning with counterexample refinement:

$n := 1$, $\boldsymbol{Pos} \coloneqq \emptyset$, $\boldsymbol{Neg} \coloneqq \emptyset$, $\boldsymbol{Imp} \coloneqq \emptyset$
while true do
  while $\phi^n_{\mathrm{DFA}} \wedge \boldsymbol{\Gamma_n}$ has a satisfying assignment α do
    construct a DFA A from α
    if A is a safety proof then
      return A
    add a new counterexample to either $\boldsymbol{Pos}$, $\boldsymbol{Neg}$, or $\boldsymbol{Imp}$
  $n \coloneqq n + 1$

$$\boldsymbol{\Gamma}_n \coloneqq \left(\bigwedge_{w \in \boldsymbol{Pos}} \phi^n_w\right) \wedge \left(\bigwedge_{w \in \boldsymbol{Neg}} \psi^n_w\right) \wedge \left(\bigwedge_{(w,v) \in \boldsymbol{Imp}} \psi^n_w \vee \phi^n_v\right)$$

Learning proofs for regular model checking

- In the rest of this lecture, we will look at two methods to find a regular proof for an RTS:
  - SAT-based learning
  - L^*-based learning
- The SAT-based method is less scalable (i.e. it is not effective when all regular proofs are large). However, it has the same termination guarantee as the brute-force enumeration.
- The L^*-based method is more scalable and can find very large regular proofs in practice. However, it is not guaranteed to find a regular proof even if one exists.

Myhill-Nerode Theorem

Given a language $L \subseteq \Sigma^*$, we can define an equivalence relation $\equiv_L$ over Σ^* such that $x \equiv_L y$ if and only if

$$\forall z \in \Sigma^*,\ xz \in L \Leftrightarrow yz \in L.$$

We will call $\equiv_L$ the *Nerode congruence.*

<u>Fact</u> $x \not\equiv_L y$ if and only if there exists $z \in \Sigma^*$ such that either $\boldsymbol{xz \in L \land yz \notin L}$, or $\boldsymbol{xz \notin L \land yz \in L}$.

In such case, we say z is a *distinguishing word* for x and y.

Myhill-Nerode Theorem (cont'd)

Given a language $L \subseteq \Sigma^*$, we can define an equivalence relation $\equiv_L$ over Σ^* such that $x \equiv_L y$ if and only if

$$\forall z \in \Sigma^*,\ xz \in L \Leftrightarrow yz \in L.$$

We will call $\equiv_L$ the *Nerode congruence.*

Example 1

Consider $\Sigma := \{a, b\}$ and $L := (aa)^*$.

Is $\varepsilon \equiv_L aa$?

Is $a \equiv_L aa$?

Is $ab \equiv_L ba$?

What are the equivalence classes induced by $\equiv_L$?

Myhill-Nerode Theorem (cont'd)

Given a language $L \subseteq \Sigma^*$, we can define an equivalence relation $\equiv_L$ over Σ^* such that $x \equiv_L y$ if and only if

$$\forall z \in \Sigma^*,\ xz \in L \Leftrightarrow yz \in L.$$

We will call $\equiv_L$ the *Nerode congruence.*

Example 1

Consider $\Sigma \coloneqq \{a, b\}$ and $L \coloneqq (aa)^*$.

Is $\varepsilon \equiv_L aa$?

Is $a \equiv_L aa$?

Is $ab \equiv_L ba$?

What are the equivalence classes induced by $\equiv_L$?

Myhill-Nerode Theorem (cont'd)

Given a language $L \subseteq \Sigma^*$, we can define an equivalence relation $\equiv_L$ over Σ^* such that $x \equiv_L y$ if and only if

$$\forall z \in \Sigma^*,\ xz \in L \Leftrightarrow yz \in L.$$

We will call $\equiv_L$ the *Nerode congruence.*

<u>Example 2</u>

Consider $\Sigma \coloneqq \{a, b\}$ and $L \coloneqq \{a^n b^n : n \geq 0\}$.

What are the equivalence classes induced by $\equiv_L$?

Myhill-Nerode Theorem (cont'd)

Given a language $L \subseteq \Sigma^*$, we can define an equivalence relation $\equiv_L$ over Σ^* such that $x \equiv_L y$ if and only if

$$\forall z \in \Sigma^*,\ xz \in L \Leftrightarrow yz \in L.$$

We will call $\equiv_L$ the *Nerode congruence.*

Myhill-Nerode Theorem

L is regular iff $\equiv_L$ induces a finite number of equivalence classes.

Key observation

When L is regular, the set of the equivalence classes is isomorphic to the set of states of the minimal DFA that recognizes L.

Nerode congruence vs DFA

When L is regular, the set of equivalence classes induced by $\equiv_L$ is isomorphic to the set of states of the minimal DFA that recognizes L.

Nerode congruence vs DFA (cont'd)

When L is regular, the set of equivalence classes induced by $\equiv_L$ is isomorphic to the set of states of the minimal DFA that recognizes L.

DFA to equivalence classes

Suppose $A := (\Sigma, s_0, S, \delta, F)$ is the minimal DFA recognizing L.

Let $L_s \subseteq \Sigma^*$ be the language accepted by $A_s := (\Sigma, s_0, S, \delta, \{s\})$.
Then $\{L_s : s \in S\}$ is the set of equivalence classes induced by $\equiv_L$.

Nerode congruence vs DFA (cont'd)

When L is regular, the set of equivalence classes induced by $\equiv_L$ is isomorphic to the set of states of the minimal DFA that recognizes L.

DFA to equivalence classes

Suppose $A := (\Sigma, s_0, S, \delta, F)$ is the minimal DFA recognizing L.

Let $L_s \subseteq \Sigma^*$ be the language accepted by $A_s \coloneqq (\Sigma, s_0, S, \delta, \{s\})$.
Then $\{L_s : s \in S\}$ is the set of equivalence classes induced by $\equiv_L$.

$\{L_s : s \in S\}$ forms a partitioning of Σ^* (by determinism of A)

If $x, y \in L_s$ for some $s \in S$, then $x \equiv_L y$ (by def. of L_s and $\equiv_L$)

If $x \equiv_L y$, then $x, y \in L_s$ for some $s \in S$ (by minimality of A)

Nerode congruence vs DFA (cont'd)

When L is regular, the set of equivalence classes induced by $\equiv_L$ is isomorphic to the set of states of the minimal DFA that recognizes L.

DFA to equivalence classes

Suppose $A := (\Sigma, s_0, S, \delta, F)$ is the minimal DFA recognizing L.

Let $L_s \subseteq \Sigma^*$ be the language accepted by $A_s \coloneqq (\Sigma, s_0, S, \delta, \{s\})$.
Then $\{L_s : s \in S\}$ is the set of equivalence classes induced by $\equiv_L$.

Equivalence classes to DFA

Let $\{[x]_L : x \in \Sigma^*\}$ be the set of equivalence classes induced by $\equiv_L$.
Define an automaton $A_L := (\Sigma, s_0, S, \delta, F)$ as follows:

$s_0 \coloneqq [\varepsilon]_L$

$S \coloneqq \{[x]_L : x \in \Sigma^*\}$

$\delta \coloneqq \{([x]_L, a, [xa]_L) : x \in \Sigma^*,\ a \in \Sigma\}$

$F \coloneqq \{[x]_L : x \in L\}$

Nerode congruence vs DFA (cont'd)

When L is regular, the set of equivalence classes induced by $\equiv_L$ is isomorphic to the set of states of the minimal DFA that recognizes L.

DFA to equivalence classes

Suppose $A := (\Sigma, s_0, S, \delta, F)$ is the minimal DFA recognizing L.

Let $L_s \subseteq \Sigma^*$ be the language accepted by $A_s := (\Sigma, s_0, S, \delta, \{s\})$.
Then $\{L_s : s \in S\}$ is the set of equivalence classes induced by $\equiv_L$.

Equivalence classes to DFA

Let $\{[x]_L : x \in \Sigma^*\}$ be the set of equivalence classes induced by $\equiv_L$.
Define an automaton $A_L := (\Sigma, s_0, S, \delta, F)$ as follows:

$s_0 := [\varepsilon]_L$

$S := \{[x]_L : x \in \Sigma^*\}$

$\delta := \{([x]_L, a, [xa]_L) : x \in \Sigma^*,\ a \in \Sigma\}$

$F := \{[x]_L : x \in L\}$

A_L is finite

A_L is deterministic

A_L is minimal

Nerode congruence vs DFA (cont'd)

Let $L \subseteq \Sigma^*$ be a language. Define an automaton $A_L := (\Sigma, s_0, S, \delta, F)$ as:

$$s_0 := [\varepsilon]_L$$
$$S := \{[x]_L : x \in \Sigma^*\}$$
$$\delta := \{([x]_L, a, [xa]_L) : x \in \Sigma^*,\ a \in \Sigma\}$$
$$F := \{[x]_L : x \in L\}$$

<u>Key observation</u> When L is regular, **there is a finite $D \subseteq \Sigma^*$** such that A_L is isomorphic to a DFA $A_{D,L} := (\Sigma, s_0, S, \delta, F)$ defined as follows:

$$s_0 := [\varepsilon]_{D,L}$$
$$S := \{[x]_{D,L} : x \in \Sigma^*\}$$
$$\delta := \{([x]_{D,L}, a, [xa]_{D,L}) : x \in \Sigma^*,\ a \in \Sigma\}$$
$$F := \{[x]_{D,L} : x \in L\}$$

Here, $y \in [x]_{D,L}$ iff x, y cannot be distinguished by any word in D w.r.t. L.

Nerode congruence vs DFA (cont'd)

Let $L \subseteq \Sigma^*$ be a language. Define an automaton $A_L := (\Sigma, s_0, S, \delta, F)$ as:

$$s_0 := [\varepsilon]_L$$
$$S := \{[x]_L : x \in \Sigma^*\}$$
$$\delta := \{([x]_L, a, [xa]_L) : x \in \Sigma^*,\ a \in \Sigma\}$$
$$F := \{[x]_L : x \in L\}$$

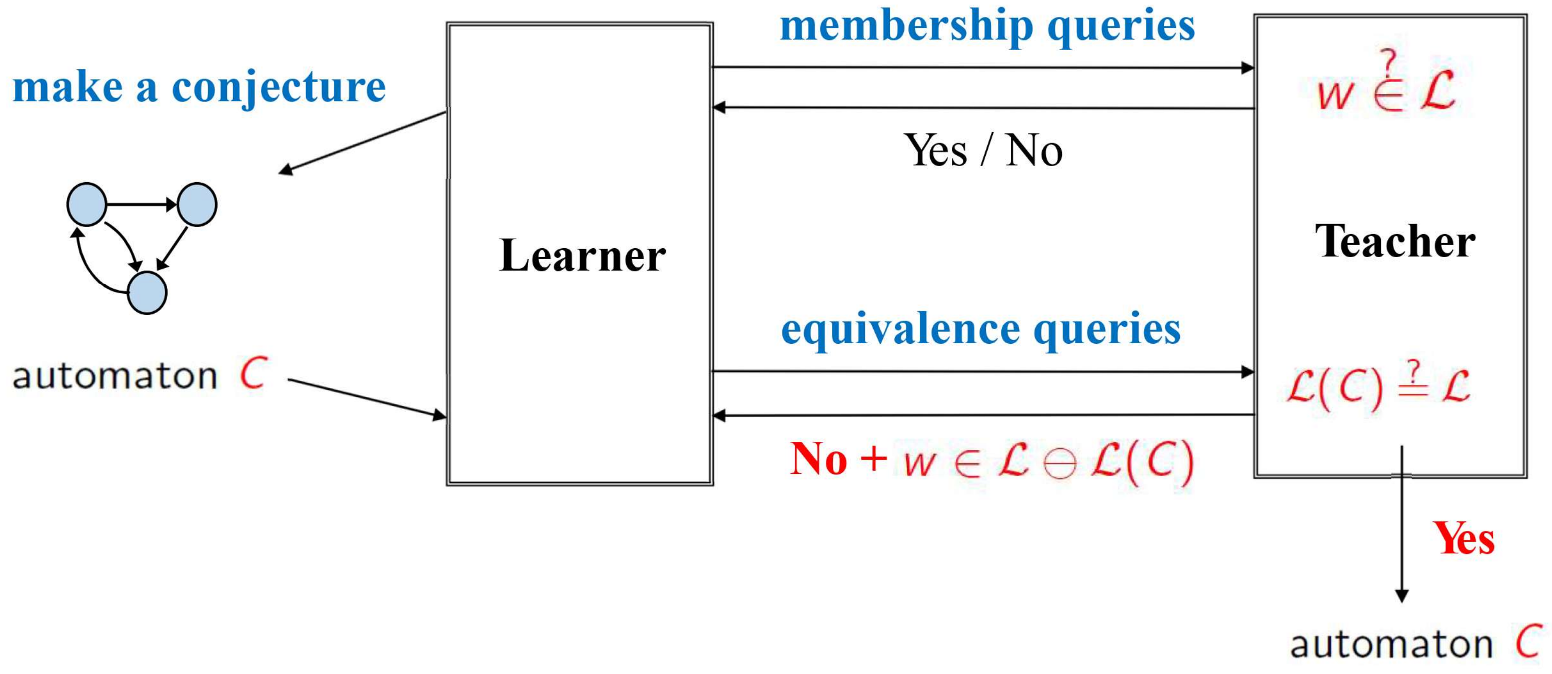
<u>Key observation</u> When L is regular, **there is a finite $D \subseteq \Sigma^*$** such that A_L is isomorphic to a DFA $A_{D,L} := (\Sigma, s_0, S, \delta, F)$ defined as follows:

$$s_0 := [\varepsilon]_{D,L}$$
$$S := \{[x]_{D,L} : x \in \Sigma^*\}$$
$$\delta := \{([x]_{D,L}, a, [xa]_{D,L}) : x \in \Sigma^*,\ a \in \Sigma\}$$
$$F := \{[x]_{D,L} : x \in L\}$$

Question: what's the relationship between $A_{D,L}$ and A_L in general?

L^* automata learning algorithm

L^* was proposed by Dana Angluin in 1987 and later improved by Rivest and Schapire in 1993. We will introduce R&S's version in this lecture.



L^* automata learning algorithm (cont'd)

Goal: learn a minimal DFA $A := (\Sigma, s_0, S, \delta, F)$ for a language L such that

$s_0 \coloneqq [\varepsilon]_L$

$S \coloneqq \{[x]_L : x \in \Sigma^*\}$

$\delta \coloneqq \{([x]_L, a, [xa]_L) : x \in \Sigma^*,\ a \in \Sigma\}$

$F \coloneqq \{[x]_L : x \in L\}$

L^* automata learning algorithm (cont'd)

Goal: learn a minimal DFA $A := (\Sigma, s_0, S, \delta, F)$ and $\boldsymbol{D} \subseteq \boldsymbol{\Sigma}^*$ for L such that

$s_0 \coloneqq [\varepsilon]_{D,L}$

$S \coloneqq \{[x]_{D,L} : x \in \Sigma^*\}$

$\delta \coloneqq \{([x]_{D,L}, a, [xa]_{D,L}) : x \in \Sigma^*,\ a \in \Sigma\}$

$F \coloneqq \{[x]_{D,L} : x \in L\}$

L^* automata learning algorithm (cont'd)

Goal: learn a minimal DFA $A := (\Sigma, s_0, S, \delta, F)$ and $\boldsymbol{D} \subseteq \boldsymbol{\Sigma}^*$ for L such that

$s_0 := [\varepsilon]_{D,L}$

$S := \{[x]_{D,L} : x \in \Sigma^*\}$

$\delta := \{([x]_{D,L}, a, [xa]_{D,L}) : x \in \Sigma^*,\ a \in \Sigma\}$

$F := \{[x]_{D,L} : x \in L\}$

The learner maintains an **observation table**:

Each w is a candidate representative of state $[w]_{D,L}$

Successors of the representatives: $[w]_{D,L} \xrightarrow{a} [wa]_{D,L}$

	u_1	$\cdots$	u_m
w_1	$w_1 u_1 \in_? L$	$\cdots$	$w_1 u_m \in_? L$
$\vdots$	$\vdots$		
w_n			
$w_1 a_1$	$w_1 a_1 u_1 \in_? L$		
$\vdots$	$\vdots$		
$w_n a_k$			

$D := \{u_1, \ldots, u_m\}$ is a set of distinguishing words for the representatives $w_1, \ldots, w_n$

L^* algorithm: the initial table

Fix $\Sigma \coloneqq \{a, b\}$ and suppose that the target language is $L \coloneqq (ab + aab)^*$.

The learner creates an initial table:

	$\boldsymbol{\varepsilon}$
$\boldsymbol{\varepsilon}$	
a	
b	

In the initial table, the column is indexed by ε, while the rows are indexed by $\{\varepsilon\} \cup \Sigma$.

L^* algorithm: the initial table

Fix $\Sigma \coloneqq \{a, b\}$ and suppose that the target language is $L \coloneqq (ab + aab)^*$.

The learner creates an initial table:

	$\boldsymbol{\varepsilon}$
$\boldsymbol{\varepsilon}$	T
a	F
b	F

The learner then fills the table by making membership queries.

Now we know that the state $[\varepsilon]_L$ differs from its successors $[a]_L$ and $[b]_L$.

We extend the table by adding a (or b) to the state space.

L^* algorithm: extending the table

Fix $\Sigma \coloneqq \{a, b\}$ and suppose that the target language is $L \coloneqq (ab + aab)^*$.

After extending the state space with a, we obtain the table

	ε
ε	T
a	F
b	F
aa	
ab	

The learner then extends the table with the successors of a, b and fills the table by making membership queries.

L^* algorithm: extending the table

Fix $\Sigma \coloneqq \{a, b\}$ and suppose that the target language is $L \coloneqq (ab + aab)^*$.

After extending the state space with a, we obtain the table

	$\boldsymbol{\varepsilon}$
$\boldsymbol{\varepsilon}$	T
a	F
b	F
aa	F
ab	T

Now every successor class has a representative in the table with respect to the current set of distinguishing words.

We say that the table is *closed*.

L^* algorithm: making a conjecture

Fix $\Sigma \coloneqq \{a, b\}$ and suppose that the target language is $L \coloneqq (ab + aab)^*$.

The table is closed. We construct a DFA $A_{D,L}$ from the table with $D = \{\varepsilon\}$.

	$\boldsymbol{\varepsilon}$
$\boldsymbol{\varepsilon}$	T
a	F
b	F
aa	F
ab	T

$s_0 \coloneqq [\varepsilon]_{D,L}$

$S \coloneqq \{[x]_{D,L} : x \in \Sigma^*\}$

$\delta \coloneqq \{([x]_{D,L}, a, [xa]_{D,L}) : x \in \Sigma^*,\ a \in \Sigma\}$

$F \coloneqq \{[x]_{D,L} : x \in L\}$

L^* algorithm: making a conjecture

Fix $\Sigma \coloneqq \{a, b\}$ and suppose that the target language is $L \coloneqq (ab + aab)^*$.

	ε
ε	T
a	F
b	F
aa	F
ab	T

The learner then makes an equivalence query $Eq(A_D)$ to the teacher.

The teacher replies “No” and provides a counterexample $w \in L_{A_D} \ominus L$.

Then this word w contains a suffix that is a valid distinguishing word.

L^* algorithm: making a conjecture

Fix $\Sigma \coloneqq \{a, b\}$ and suppose that the target language is $L \coloneqq (ab + aab)^*$.

	$\boldsymbol{\varepsilon}$
$\boldsymbol{\varepsilon}$	T
a	F
b	F
aa	F
ab	T

The learner then makes an equivalence query $Eq(A_D)$ to the teacher.

The teacher replies "No" and provides a counterexample $w \in L_{A_D} \ominus L$.

Suppose that the teacher returns bb. Then b is a distinguishing word for at least two D-equivalent states.

L^* algorithm: the 2nd iteration

Fix $\Sigma \coloneqq \{a, b\}$. Suppose that the language to learn is $L \coloneqq (ab + aab)^*$.

	$\boldsymbol{\varepsilon}$	$\boldsymbol{b}$
$\boldsymbol{\varepsilon}$	T	F
a	F	T
b	F	F
aa	F	T
ab	T	F

We include b in the state space and extend the table accordingly.

The representatives ***a* and *b*** are separated by the new distinguishing word!

L^* algorithm: the 2nd iteration

Fix $\Sigma \coloneqq \{a, b\}$. Suppose that the language to learn is $L \coloneqq (ab + aab)^*$.

	ε	b
ε	T	F
a	F	T
b	F	F
aa	F	T
ab	T	F
ba	F	F
bb	F	F

The table is now closed. The learner makes an equivalence query to the teacher. The teacher replies "No", and we obtain a new distinguishing word ab after analyzing the counterexample.

L^* algorithm: the 3rd iteration

Fix $\Sigma := \{a, b\}$. Suppose that the language to learn is $L := (ab + aab)^*$.

	ε	b	ab
ε	T	F	T
a	F	T	T
b	F	F	F
aa	F	T	F
ab	T	F	T
ba	F	F	F
bb	F	F	F
aaa	F	F	F
aab	T	F	T

The learner successfully learns a minimal DFA A for L in the 3rd iteration.

L^* algorithm: counterexample analysis

<u>Claim</u> If the teacher returns a counterexample $\boldsymbol{w \in L(A) \ominus L}$ for an equivalence query $\boldsymbol{Eq(A)}$, then one can make $\boldsymbol{\log|w|}$ membership queries to find a word that distinguish two states of $\boldsymbol{A}$.

Recall that a hypothesis automaton $A := (\Sigma, s_0, S, \delta, F)$ is defined as

$s_0 \coloneqq [\varepsilon]_{D,L}$

$S \coloneqq \{[x]_{D,L} : x \in \Sigma^*\}$

$\delta \coloneqq \{([x]_{D,L}, a, [xa]_{D,L}) : x \in \Sigma^*,\ a \in \Sigma\}$

$F \coloneqq \{[x]_{D,L} : x \in L\}$

Consider a counterexample word $w := a_1 \dots a_m$. Then A will reach state $[a_1 \dots a_k]_{D,L}$ after reading the prefix $a_1 \dots a_k$ of w.

If $w \in L(A) \ominus L$, then there exists $1 \le k \le m$ such that $a_{k+1} \dots a_m$ is a distinguishing word for some $x, y \in [a_1 \dots a_k]_{D,L}$.

We can locate this k using binary search with $\log|w|$ membership queries. Adding $a_{k+1} \dots a_m$ to D will identify at least one new state.

L^* algorithm: complexity

Complexity result of L^*

If the minimal DFA recognizing the target language has n states, then

1. The learner needs at most n equivalence queries
2. The learner needs $O(|\Sigma|n^2 + n \log m)$ membership queries

where m is the maximum size of counterexample returned by the teacher.

L^*-based learning for safety proofs

We introduce below how to use the L^* algorithm to learn a safety proof for a regular transition system (Σ, I, T).

- We need a target language for L^*. We cannot use the proof to learn as the target language since safety proof is not unique.
- Instead, we set (the language representation of) the reachable states $T^*(I)$ as the target language.
- Recall that $T^*(I)$ is unique, and is a proof when the system is safe.
- We will design a teacher for L^* such that when the system is safe and $T^*(I)$ is regular, the learner is guaranteed to find a proof.

L^*-based learning for safety proofs (cont'd)

- We set the reachable states $T^*(I)$ as the target language.
- We will design a teacher for L^* such that when the system is safe and $T^*(I)$ is regular, the learner is guaranteed to find a regular proof.

- **Resolving $\mathbf{Mem}(w)$:**
  - $w \in T^*(I)$ iff w is reachable from I.
- **Resolving $\mathbf{Eq}(A)$:**

  It suffices to check the proof rules for safety:

  - $I \subseteq L_A$
  - $L_A \cap B = \emptyset$
  - $T(L_A) \subseteq L_A$

L^*-based learning: resolving equivalence query

- We check the proof rules for safety to resolve $\mathrm{Eq}(A)$:
  - $I \subseteq L_A$
  - $L_A \cap B = \emptyset$
  - $T(L_A) \subseteq L_A$
- If any of the checks fails:
  - $I \nsubseteq L_A$: any $w \in I \setminus L_A$ is a **positive cex**
  - $L_A \cap B \neq \emptyset$: any $w \in L_A \cap B$ is a **negative cex**
  - $T(L_A) \nsubseteq L_A$: there is $w \in L_A$ and $T(w) \setminus L_A \neq \emptyset$.

    If $\mathrm{Mem}(w)$ is "no", then $w \notin T^*(I)$ and thus is a **negative cex**

    If $\mathrm{Mem}(w)$ is "yes", then any $w \in T(w) \setminus L_A$ is a **positive cex**

L^*-based learning: resolving equivalence query

- We check the proof rules for safety to resolve $\mathrm{Eq}(A)$:
  - $I \subseteq L_A$
  - $L_A \cap B = \emptyset$
  - $T(L_A) \subseteq L_A$

All $w \in L_A \ominus T^*(I)$ when the system is safe

- If any of the checks fails:
  - $I \nsubseteq L_A$: any $\boldsymbol{w} \in \boldsymbol{I} \setminus \boldsymbol{L_A}$ is a positive cex
  - $L_A \cap B \neq \emptyset$: any $\boldsymbol{w} \in \boldsymbol{L_A} \cap \boldsymbol{B}$ is a negative cex
  - $T(L_A) \nsubseteq L_A$: there is $w \in L_A$ and $T(w) \setminus L_A \neq \emptyset$.

    If $Mem(w)$ is "no", then $\boldsymbol{w} \notin \boldsymbol{T}^*(\boldsymbol{I})$ and thus is a negative cex

    If $Mem(w)$ is "yes", then any $\boldsymbol{w} \in \boldsymbol{T}(\boldsymbol{w}) \setminus \boldsymbol{L_A}$ is a positive cex

L^*-based learning: an example

Consider Israeli-Jalfon's leader election protocol.

1. Processes $1, \ldots, n$ are organized in a ring
2. At the beginning, at least *two* processes hold a token
3. At each step, a process can pass its token to the right or left
4. When a process receives two tokens, it discards one of them

Safety condition: there is at least one token in the ring.

L^*-based learning: an example

Consider Israeli-Jalfon's leader election protocol.

1. Processes $1, \ldots, n$ are organized in a ring
2. At the beginning, at least *two* processes hold a token
3. At each step, a process can pass its token to the right or left
4. When a process receives two tokens, it discards one of them

Safety condition: there is at least one token in the ring.

We model the protocol with an RTS (Σ, I, T) and bad states B, where

I: $(1+0)^*1(1+0)^*1(1+0)^*$

T: $id^* \begin{bmatrix}1\\0\end{bmatrix} \left(\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right) id^* + id^* \left(\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right) \begin{bmatrix}1\\0\end{bmatrix} id^* + \left(\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right) id^* \begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix} id^* \left(\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right)$

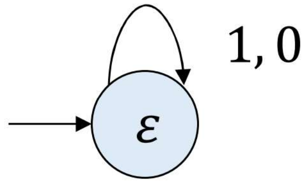
B: 0^*

$Id := \left(\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix}\right)$

L^*-based learning: an example (cont'd)

$I :\ (1+0)^*\,1\,(1+0)^*\,1\,(1+0)^*$

$B :\ 0^*$



The first closed table:

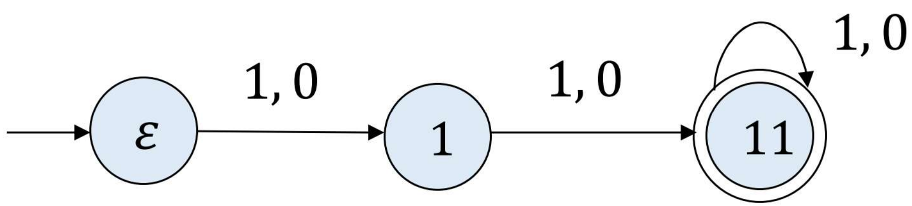
	ε
ε	F
1	F
0	F

Counterexample: $11 \in I \setminus L_A$. Add a new distinguishing word 1.

L^*-based learning: an example (cont'd)

I : $(1+0)^* \, 1 \, (1+0)^* \, 1 \, (1+0)^*$

B : 0^*



The second closed table:

	ε	**1**
ε	F	F
1	F	T
11	T	T
0	F	T
10	T	T
111	T	T
110	T	T

Counterexample: $000 \in L_A \cap B$. Add a new distinguishing word 0.

L^*-based learning: an example (cont'd)

I : $(1+0)^* \, 1 \, (1+0)^* \, 1 \, (1+0)^*$

B : 0^*

The third closed table leads to a regular proof. What is the DFA?

	ε	1	0
ε	F	F	F
1	F	T	T
0	F	T	F
11	T	T	T
10	T	T	T
01	T	T	T
00	F	T	F
111	T	T	T
110	T	T	T

Active learning algorithms for DFAs

	Algorithm	Publication
Angluins et al. 1987	Angluin's L^*	Learning regular sets from queries and counterexamples
Rivest and Schapire 1993	R & S 's Algorithm	Inference of Finite Automata Using Homing Sequences
Kearns and Vazirani 1994	K & V 's Algorithm	An introduction to computational learning theory
Parekh et al. 1997	ID and IID	A polynomial time incremental algorithm for regular grammar inference
Denis et al. 2001	DeLeTe2	Learning regular languages using RFSAs
Bongard et al. 2005	Estimation-Exploration	Active Coevolutionary Learning of Deterministic Finite Automata
Isberner et al. 2014	The TTT Algorithm	The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning
Volpato et al. 2015	LearnLTS	Approximate Active Learning of Nondeterministic Input Output Transition Systems