Applications of learning theory in verification

Madhavan Mukund

Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Formal Methods Update 2007, IIT Kanpur
14 April 2007
(Adapted from material contributed by P Madhusudan)

Motivation

- Abstraction is an important tool in verification
  - Build a coarse model M from a system description S
  - Every run of S is also a run of M
  - If M satisfies a safety property, so does S

Motivation

- Abstraction is an important tool in verification
  - Build a coarse model M from a system description S
  - Every run of S is also a run of M
  - If M satisfies a safety property, so does S
- Can we use learning to discover the abstraction?
  - S may have a complicated description ...
  - ... but abstraction M may be "small"
  - Circumvent complexity of verifying S directly

Motivation

- Abstraction is an important tool in verification
  - Build a coarse model M from a system description S
  - Every run of S is also a run of M
  - If M satisfies a safety property, so does S
- Can we use learning to discover the abstraction?
  - S may have a complicated description ...
  - ... but abstraction M may be "small"
  - Circumvent complexity of verifying S directly
- Other problems in verification can also benefit from this approach

Outline

- Two verification problems
  - Compositional verification of $P \parallel Q$
  - Deriving interface specification for a module

Outline

- Two verification problems
  - Compositional verification of $P \parallel Q$
  - Deriving interface specification for a module
- Learning regular languages
  - Active learner model [Angluin'86]
  - A tutorial introduction to the learning algorithm

Outline

- Two verification problems
  - Compositional verification of $P \parallel Q$
  - Deriving interface specification for a module
- Learning regular languages
  - Active learner model [Angluin'86]
  - A tutorial introduction to the learning algorithm
- How to apply learning for the two problems above
- Some pointers to other applications

Compositional verification

- Parallel composition $P \parallel Q$ of two modules

Compositional verification

- Parallel composition $P \parallel Q$ of two modules
- Does $P \parallel Q$ satisfy a safety specification φ?

Compositional verification

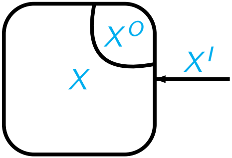
- Parallel composition $P \parallel Q$ of two modules
- Does $P \parallel Q$ satisfy a safety specification φ?
- Assume guarantee reasoning
  - Find R such that:
    - $P \parallel R \models \varphi$
    - Behaviours of Q are included in behaviours of R
  - R may be small compared to P and Q.

Compositional verification ...

Module P

State variables X

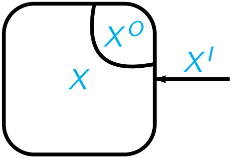
- output variables $X^O \subseteq X$,
  disjoint set of input variables X^I



Compositional verification ...

Module P

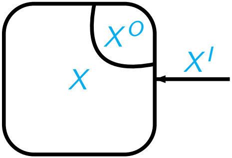
- State variables X
  output variables $X^O \subseteq X$,
  disjoint set of input variables X^I



- Assume we are working with boolean abstraction

Compositional verification . . .

Module P



- State variables X
  output variables $X^O \subseteq X$,
  disjoint set of input variables X^I
- Assume we are working with boolean abstraction
- State : $s : (X \uplus X^I) \rightarrow \{0, 1\}$

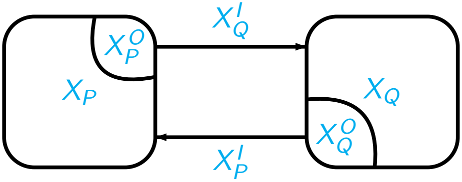
  Transition : $T \subseteq (S \setminus S^I) \times S^I \times (S \setminus S^I)$

  Behaviour : $s_1 s_2 \ldots$

  Visible Behaviour : $s_1^{I \cup O} s_2^{I \cup O} \ldots$

Module composition

$P \parallel Q$: Outputs of P are inputs to Q and vice versa



- $VisBeh(P \parallel Q) = VisBeh(P) \cap VisBeh(Q)$

Compositional verification of modules

Safety property φ: boolean formula over $X^I \cup X^O$

- $s_1 s_2 \ldots \models \varphi$ if for each i, $s_i^{I \cup O} \models \varphi$

Compositional verification of modules

Safety property φ: boolean formula over $X^I \cup X^O$

- $s_1 s_2 \ldots \models \varphi$ if for each i, $s_i^{I \cup O} \models \varphi$

When does $P \parallel Q \models \varphi$?

- For each $\sigma \in \mathit{VisBeh}(P \parallel Q)$, $\sigma \models \varphi$

Compositional verification of modules

Safety property φ: boolean formula over $X^I \cup X^O$

- $s_1 s_2 \ldots \models \varphi$ if for each i, $s_i^{I \cup O} \models \varphi$

When does $P \parallel Q \models \varphi$?

- For each $\sigma \in \mathit{VisBeh}(P \parallel Q)$, $\sigma \models \varphi$

Assume guarantee reasoning

- Find R such that:
  - $P \parallel R \models \varphi$
  - $\mathit{VisBeh}(Q) \subseteq \mathit{VisBeh}(R)$
- Learn a regular language R with small DFA?

Interface synthesis

- A class C with variables $V = \{v_1, v_2, \ldots\}$ and methods $M = \{m_1, m_2, \ldots\}$

Interface synthesis

- A class C with variables $V = \{v_1, v_2, \ldots\}$ and methods $M = \{m_1, m_2, \ldots\}$
- State of an object $s : V \to \{0, 1\}$—again we assume a boolean abstraction

Interface synthesis

- A class C with variables $V = \{v_1, v_2, \ldots\}$ and methods $M = \{m_1, m_2, \ldots\}$
- State of an object $s : V \to \{0, 1\}$—again we assume a boolean abstraction
- $V_R \subseteq V$—output variables

Interface synthesis

- A class C with variables $V = \{v_1, v_2, \ldots\}$ and methods $M = \{m_1, m_2, \ldots\}$
- State of an object $s : V \to \{0, 1\}$—again we assume a boolean abstraction
- $V_R \subseteq V$—output variables
- A call to method m nondeterministically transforms s to s' and returns s'_R

Interface synthesis

- A class C with variables $V = \{v_1, v_2, \ldots\}$ and methods $M = \{m_1, m_2, \ldots\}$
- State of an object $s : V \to \{0, 1\}$—again we assume a boolean abstraction
- $V_R \subseteq V$—output variables
- A call to method m nondeterministically transforms s to s' and returns s'_R
- A run is a sequence $(m_1, s_R^1), (m_2, s_R^2), \ldots$

Interface synthesis

- A class C with variables $V = \{v_1, v_2, \ldots\}$ and methods $M = \{m_1, m_2, \ldots\}$
- State of an object $s : V \to \{0, 1\}$—again we assume a boolean abstraction
- $V_R \subseteq V$—output variables
- A call to method m nondeterministically transforms s to s' and returns s'_R
- A run is a sequence $(m_1, s_R^1), (m_2, s_R^2), \ldots$
- Safety specification: Boolean formula φ on return variables

Interface synthesis

- A class C with variables $V = \{v_1, v_2, \ldots\}$ and methods $M = \{m_1, m_2, \ldots\}$
- State of an object $s : V \to \{0, 1\}$—again we assume a boolean abstraction
- $V_R \subseteq V$—output variables
- A call to method m nondeterministically transforms s to s' and returns s'_R
- A run is a sequence $(m_1, s^1_R), (m_2, s^2_R), \ldots$
- Safety specification: Boolean formula φ on return variables
- A run is safe if for each i, $s^i_R \models \varphi$

Interface synthesis

- A class C with variables $V = \{v_1, v_2, \ldots\}$ and methods $M = \{m_1, m_2, \ldots\}$
- State of an object $s : V \to \{0, 1\}$—again we assume a boolean abstraction
- $V_R \subseteq V$—output variables
- A call to method m nondeterministically transforms s to s' and returns s'_R
- A run is a sequence $(m_1, s^1_R), (m_2, s^2_R), \ldots$
- Safety specification: Boolean formula φ on return variables
- A run is safe if for each i, $s^i_R \models \varphi$
- Want to restrict runs of the class to permit only safe runs

Interface

An interface is a function $I : (M \times V_R)^* \rightarrow 2^M$

- After a run $\sigma = (m_1, s_R^1), (m_2, s_R^2), \ldots$, $I(\sigma)$ specifies which methods can be invoked

Interface

An interface is a function $I : (M \times V_R)^* \to 2^M$

- After a run $\sigma = (m_1, s_R^1), (m_2, s_R^2), \ldots,$ $I(\sigma)$ specifies which methods can be invoked

A run is consistent with an interface if,

- for every prefix $\rho = (m_1, s_R^1), (m_2, s_R^2), \ldots, (m_k, s_R^k)$, $m_{k+1} \in I(\rho)q$

Interface

An interface is a function $I : (M \times V_R)^* \to 2^M$

- After a run $\sigma = (m_1, s_R^1), (m_2, s_R^2), \ldots$, $I(\sigma)$ specifies which methods can be invoked

A run is consistent with an interface if,

- for every prefix $\rho = (m_1, s_R^1), (m_2, s_R^2), \ldots, (m_k, s_R^k)$, $m_{k+1} \in I(\rho)q$

An interface I is good if all runs consistent with I satisfy φ

Interface

An interface is a function $I : (M \times V_R)^* \rightarrow 2^M$

- After a run $\sigma = (m_1, s_R^1), (m_2, s_R^2), \ldots$, $I(\sigma)$ specifies which methods can be invoked

A run is consistent with an interface if,

- for every prefix $\rho = (m_1, s_R^1), (m_2, s_R^2), \ldots, (m_k, s_R^k)$, $m_{k+1} \in I(\rho) q$

An interface I is good if all runs consistent with I satisfy φ

I can be thought of as an automaton over $(M \times V_R)$

Interface

An interface is a function $I : (M \times V_R)^* \to 2^M$

- After a run $\sigma = (m_1, s_R^1), (m_2, s_R^2), \ldots$, $I(\sigma)$ specifies which methods can be invoked

A run is consistent with an interface if,

- for every prefix $\rho = (m_1, s_R^1), (m_2, s_R^2), \ldots, (m_k, s_R^k)$, $m_{k+1} \in I(\rho)q$

An interface I is good if all runs consistent with I satisfy φ

I can be thought of as an automaton over $(M \times V_R)$

Can we learn a maximal interface?

Learning Regular Languages

Fix a finite alphabet Σ.

- There is a learner and a teacher
- Teacher knows a regular language T
- Objective of the learner: To learn T by constructing an automaton for T.

Learning Regular Languages

Fix a finite alphabet Σ.

- There is a learner and a teacher
- Teacher knows a regular language T
- Objective of the learner: To learn T by constructing an automaton for T.

Complexity will be measured on the complexity of the language: the minimum number of states needed to capture T.

Active learning [Angluin'86]

- Learner asks questions:
- Membership: Is $w \in T$?
  - Yes or No
- Equivalence question: Is $T = L(C)$?
  - Yes or No+counterexample
  - Counterexample is in $(T \setminus L(C)) \cup (L(C) \setminus T)$.

Active learning [Angluin'86]

- Learner asks questions:
- Membership: Is $w \in T$?
  - Yes or No
- Equivalence question: Is $T = L(C)$?
  - Yes or No+counterexample
  - Counterexample is in $(T \setminus L(C)) \cup (L(C) \setminus T)$.

Theorem (Angluin, Rivest-Schapire, Kearns-Vazirani)

Regular languages can be learnt using at most $O(kn^2 + n \log m)$ membership and $O(n)$ equivalence queries.

- n — *size of the minimal DFA accepting target language* T
- m — *size of the largest counterexample*
- k — *size of the alphabet.*

Also, in time polynomial in $O(kn^2 + n \log m)$.

How do we learn T?

Key points

- How many states are there?
- How do we reach these states from the initial state?
- How do we build the transitions correctly?

When are states different?

Simple observation:

Let u and v be two strings.

> *If* $\exists w$ *such that* $uw \in T \iff vw \notin T$,
> *then* u *and* v *must lead to different states.*

When are states different?

Simple observation:

Let u and v be two strings.

If $\exists w$ *such that* $uw \in T \iff vw \notin T$,
then u *and* v *must lead to different states.*

If this condition holds, we say u and v are distinguishable

When are states different?

Simple observation:

Let u and v be two strings.

If $\exists w$ such that $uw \in T \iff vw \notin T$,
then u and v must lead to different states.

If this condition holds, we say u and v are distinguishable

If we find n strings $s_1, \ldots, s_n$, that are pairwise distinguishable, we know that automaton for T has (at least) n states.

Access strings

Access string to a state q

- Some string that gets you from q_0 to q.

Hence ε is an access string for q_0.

Access strings

Access string to a state q

- Some string that gets you from q_0 to q.

Hence ε is an access string for q_0.

If we have n access strings $s_1, s_2, \ldots, s_n$, that are pairwise distinguishable, then the states reached on these strings must *all* be different.

An observation pack

Access strings	s_1	s_2	$\dots$	$\dots$	s_k
Experiments	E_{s_1}	E_{s_2}	$\dots$	$\dots$	E_{s_k}

An observation pack for T has n access strings $S = \{s_1, \dots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

An observation pack

Access strings	s_1	s_2	$\dots$	$\dots$	s_k
Experiments	E_{s_1}	E_{s_2}	$\dots$	$\dots$	E_{s_k}

An observation pack for T has n access strings $S = \{s_1, \dots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
  - $(u, +) \in E_{s_i}$ implies $s_i.u_i \in T$
  - $(u, -) \in E_{s_i}$ implies $s_i.u_i \notin T$

An observation pack

Access strings	s_1	s_2	...	...	s_k
Experiments	E_{s_1}	E_{s_2}	...	...	E_{s_k}

An observation pack for T has n access strings $S = \{s_1, \ldots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
  - $(u, +) \in E_{s_i}$ implies $s_i.u_i \in T$
  - $(u, -) \in E_{s_i}$ implies $s_i.u_i \notin T$
- For any two access strings s_i and s_j, there is some experiment that distinguishes them.
  i.e., there is some u that figures in E_{s_i} and E_{s_j} with opposite polarity.

An observation pack

Access strings	s_1	s_2	...	...	s_k
Experiments	E_{s_1}	E_{s_2}	...	...	E_{s_k}

An observation pack for T has n access strings $S = \{s_1, \ldots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
  - $(u, +) \in E_{s_i}$ implies $s_i.u_i \in T$
  - $(u, -) \in E_{s_i}$ implies $s_i.u_i \notin T$
- For any two access strings s_i and s_j, there is some experiment that distinguishes them.
  i.e., there is some u that figures in E_{s_i} and E_{s_j} with opposite polarity.
- $\varepsilon \in S$, and $\varepsilon \in E_{s_i}$ for each i.

An observation pack

Access strings	s_1	s_2	...	...	s_k
Experiments	E_{s_1}	E_{s_2}	...	...	E_{s_k}

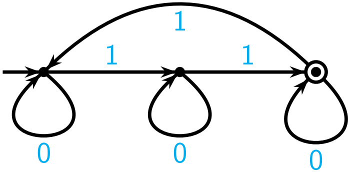
An observation pack for T has n access strings $S = \{s_1, \ldots, s_n\}$, and each $s \in S$ is associated with a set of experiments E_s such that:

- Each E_{s_i} consists of a set of pairs of the form $(u, +)$ or $(u, -)$:
  - $(u, +) \in E_{s_i}$ implies $s_i.u_i \in T$
  - $(u, -) \in E_{s_i}$ implies $s_i.u_i \notin T$
- For any two access strings s_i and s_j, there is some experiment that distinguishes them.
  i.e., there is some u that figures in E_{s_i} and E_{s_j} with opposite polarity.
- $\varepsilon \in S$, and $\varepsilon \in E_{s_i}$ for each i.

Note: If an observation pack with n access strings exists, then minimal automaton for T has *at least* n states.

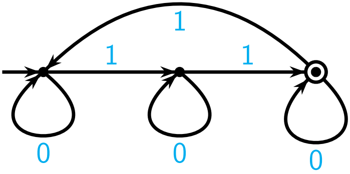
Example

Target language T: strings over $\{0, 1\}$ where $\#1's = 2 \bmod 3$



Example

Target language T: strings over $\{0,1\}$ where $\#1's = 2 \bmod 3$

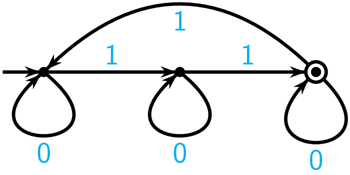


An observation pack:

Access strings	ε	010
Experiments	$(\varepsilon, -)$ $(10, -)$	$(\varepsilon, -)$ $(10, +)$

Example

Target language T: strings over $\{0,1\}$ where $\#1's = 2 \bmod 3$



An observation pack:

Access strings	ε	010
Experiments	$(\varepsilon, -)$ $(10, -)$	$(\varepsilon, -)$ $(10, +)$

$\varepsilon.\varepsilon \notin T$; $010.\varepsilon \notin T$
$\varepsilon.10 \notin T$; $010.10 \in T$

Likeness and escape

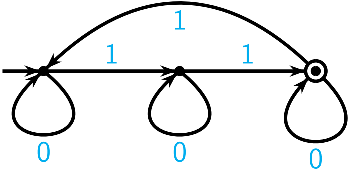
Let O be an observation pack.

A word w is like an access string s in O, if w agrees with s on all the experiments in E_s.
i.e. ,$\forall u \in E_s$, $wu \in T$ iff $su \in T$.

Note: No two access strings are alike $\Rightarrow w$ can be like *at most one* access string in O, since

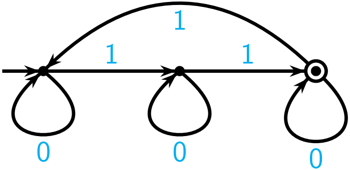
If w is *not* like any access string, we say it escapes the pack.

Example



Access strings	ε	110
Experiments	$(\varepsilon, -)$ $(10, -)$	$(\varepsilon, -)$ $(10, +)$

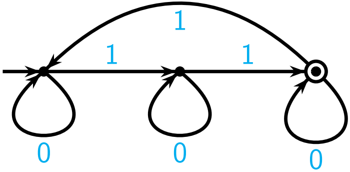
Example



Access strings	ε	110
Experiments	$(\varepsilon, -)$ $(10, -)$	$(\varepsilon, -)$ $(10, +)$

The word 001 is like 010 (since $001.\varepsilon \notin T$, $001.10 \in T$).

Example



Access strings	ε	110
Experiments	$(\varepsilon, -)$ $(10, -)$	$(\varepsilon, -)$ $(10, +)$

The word 001 is like 010 (since $001.\varepsilon \notin T$, $001.10 \in T$).

The word 11 is not like any access string in O (since $11.\varepsilon \in T$).
So 11 *escapes*.

Expanding a pack

If O is an observation pack, and w escapes O, then we can expand O to include w:

- Add w as a new access string
- For every access s string in O, there is some u in E_s that distinguishes w and s.
- Add this string to E_w

Expanding a pack

If O is an observation pack, and w escapes O, then we can expand O to include w:

- Add w as a new access string
- For every access s string in O, there is some u in E_s that distinguishes w and s.
- Add this string to E_w

The new pack is a proper observation pack ...

Expanding a pack

If O is an observation pack, and w escapes O, then we can expand O to include w:

- Add w as a new access string
- For every access s string in O, there is some u in E_s that distinguishes w and s.
- Add this string to E_w

The new pack is a proper observation pack ...

...and has one more access string.

Closure

An observation pack O is said to be closed if

- For every access string s in O and $a \in \Sigma$, $s.a$ is like some access string in O.

Closure

An observation pack O is said to be closed if

- For every access string s in O and $a \in \Sigma$, $s.a$ is like some access string in O.

If O is closed, we can build an automaton from it:

- States: The access strings in O: $\{s_1 \ldots, s_k\}$

Closure

An observation pack O is said to be closed if

- For every access string s in O and $a \in \Sigma$, $s.a$ is like some access string in O.

If O is closed, we can build an automaton from it:

- States: The access strings in O: $\{s_1 \ldots, s_k\}$
- From s on a, go to the state that is *like* sa.

Closure

An observation pack O is said to be closed if

- For every access string s in O and $a \in \Sigma$, $s.a$ is like some access string in O.

If O is closed, we can build an automaton from it:

- States: The access strings in O: $\{s_1 \ldots, s_k\}$
- From s on a, go to the state that is *like* sa.
- Mark a state s final iff $(\varepsilon, +) \in E_s$.

Automaton construction

Theorem

If the observation pack O has as many states as M_T, then the automaton constructed is isomorphic to M_T.

Automaton construction

Theorem

If the observation pack O has as many states as M_T, then the automaton constructed is isomorphic to M_T.

Proof.

- The number of states is correct.
- Initial state maps to initial state of M_T.
- On any letter, we move to the right state.
- Final states are marked correctly.

$\square$

Automaton construction

Theorem

If the observation pack O has as many states as M_T, then the automaton constructed is isomorphic to M_T.

Proof.

- The number of states is correct.
- Initial state maps to initial state of M_T.
- On any letter, we move to the right state.
- Final states are marked correctly.

$\square$

So, the whole problem reduces to finding an observation pack with n access strings!!

Learning from a false automaton

Let O be an observation pack.

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Phase II: If O is closed but has less access strings than $|M_T|$.

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Phase II: If O is closed but has less access strings than $|M_T|$.

- Then automaton constructed has too few states.
- How do we learn access strings to new states?

Learning from a false automaton

Let O be an observation pack.

Phase I: If O is *not* closed, expand pack using some new access string $s.a$.

Phase II: If O is closed but has less access strings than $|M_T|$.

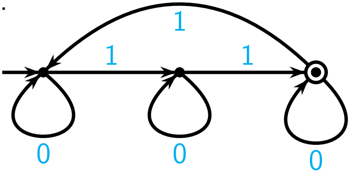
- Then automaton constructed has too few states.
- How do we learn access strings to new states?

Equivalence query:

- Build conjecture automaton C.
- Ask teacher "$L(C) = T$?"
- Use counterexample given by teacher to generate new access string.

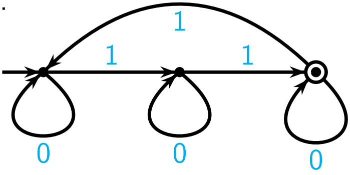
A learning example. . .

Target language T:



A learning example...

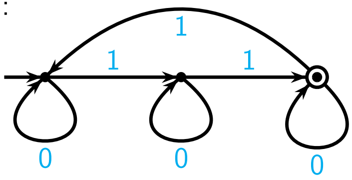
Target language T:



Access strings	$s_0 = \varepsilon$
Experiments	$(\varepsilon, -)$

A learning example...

Target language T:



Access strings	$s_0 = \varepsilon$
Experiments	$(\varepsilon, -)$

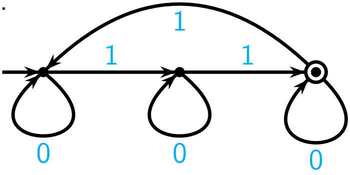
Check closure:

0 is like ε (since $0 \notin T$).

1 is like ε (since $1 \notin T$).

A learning example...

Target language T:



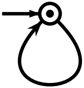
Access strings	$s_0 = \varepsilon$
Experiments	$(\varepsilon, -)$

Check closure:

0 is like ε (since $0 \notin T$).

1 is like ε (since $1 \notin T$).

A learning example . . .

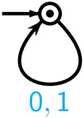


Counter-example: $101 \in T \setminus L(C)$

Run of 101 on C: $s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$

- $s_0 = \varepsilon$

A learning example . . .

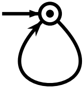


Counter-example: $101 \in T \setminus L(C)$

Run of 101 on C: $s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$

- $s_0 = \varepsilon$
- $s_0.101 \in T$

A learning example . . .

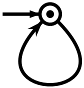


Counter-example: $101 \in T \setminus L(C)$

Run of 101 on C: $s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$

- $s_0 = \varepsilon$
- $s_0.101 \in T$
- $s_0.01 \notin T$.

A learning example . . .

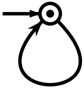


Counter-example: $101 \in T \setminus L(C)$

Run of 101 on C: $s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$

- $s_0 = \varepsilon$
- $s_0.101 \in T$
- $s_0.01 \notin T$.
- So we cannot go on 1 to s_0!
  (since 01 distinguishes 1 and s_0)

A learning example . . .



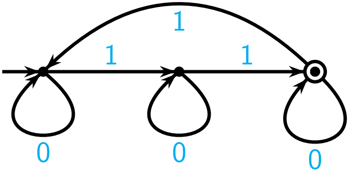
Counter-example: $101 \in T \setminus L(C)$

Run of 101 on C: $s_0 \xrightarrow{1} s_0 \xrightarrow{0} s_0 \xrightarrow{1} s_0$

- $s_0 = \varepsilon$
- $s_0.101 \in T$
- $s_0.01 \notin T$.
- So we cannot go on 1 to s_0!
  (since 01 distinguishes 1 and s_0)
- So let's add 01 as experiment string for s_0.

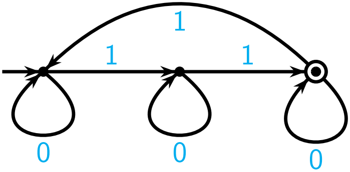
A learning example

T



A learning example

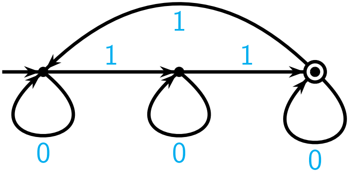
T



Access strings	$s_0 = \varepsilon$	$s_1 = 1$
Experiments	$(\varepsilon, -)$ $(01, -)$	$(\varepsilon, -)$ $(01, +)$

A learning example

T

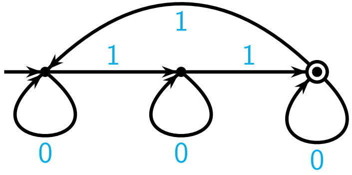


Access strings	$s_0 = \varepsilon$	$s_1 = 1$
Experiments	$(\varepsilon, -)$ $(01, -)$	$(\varepsilon, -)$ $(01, +)$

Check closure:
10 is like 1 (since $10 \notin T$ and $10.01 \in T$)

A learning example

T



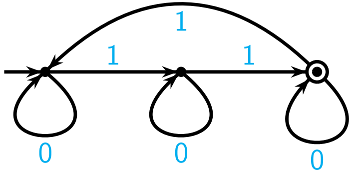
Access strings	$s_0 = \varepsilon$	$s_1 = 1$
Experiments	$(\varepsilon, -)$	$(\varepsilon, -)$
	$(01, -)$	$(01, +)$

Check closure:
10 is like 1 (since $10 \notin T$ and $10.01 \in T$)

But 11 is neither like ε nor like 1 (since $11 \in T$).

A learning example

T



Access strings	$s_0 = \varepsilon$	$s_1 = 1$
Experiments	$(\varepsilon, -)$	$(\varepsilon, -)$
	$(01, -)$	$(01,+)$

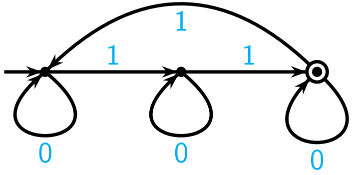
Check closure:
10 is like 1 (since $10 \notin T$ and $10.01 \in T$)

But 11 is neither like ε nor like 1 (since $11 \in T$).

So 11 escapes and forms a new access string.

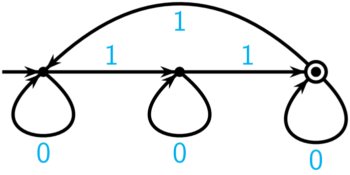
A learning example ...

T



A learning example . . .

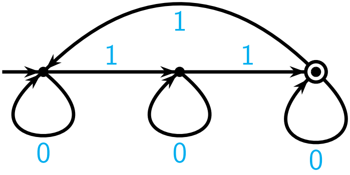
T



Access strings	$s_0 = \varepsilon$	$s_1 = 1$	$s_2 = 11$
Experiments	$(\varepsilon, -)$ $(01, -)$	$(\varepsilon, -)$ $(01, +)$	$(\varepsilon, +)$

A learning example . . .

T

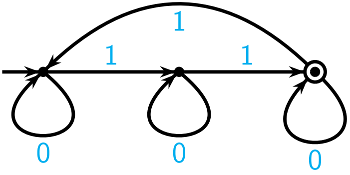


Access strings	$s_0 = \varepsilon$	$s_1 = 1$	$s_2 = 11$
Experiments	$(\varepsilon, -)$ $(01, -)$	$(\varepsilon, -)$ $(01, +)$	$(\varepsilon, +)$

Check closure:
0 is like s_0; 10 is like 1;
110 is like 11; 111 is like 0.

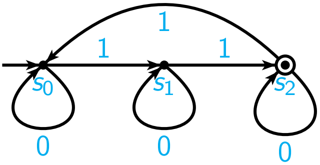
A learning example . . .

T



Access strings	$s_0 = \varepsilon$	$s_1 = 1$	$s_2 = 11$
Experiments	$(\varepsilon, -)$ $(01, -)$	$(\varepsilon, -)$ $(01, +)$	$(\varepsilon, +)$

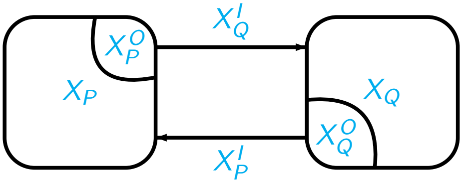
Check closure:
0 is like s_0; 10 is like 1;
110 is like 11; 111 is like 0.



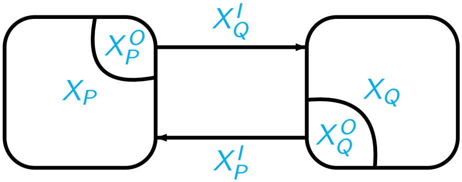
References

- Angluin
  Learning regular sets from queries and counterexamples
  Inf. and Comp. '87
- Rivest, Schapire
  Inference of finite automata using homing sequences
  Inf. and Comp. '95
- Kearns, Vazirani
  Introduction to Computational Learning Theory
  MIT Press
- Balcázar, Díaz, Gavalda, Watanabe
  Algorithms for Learning Finite Automata from queries: A Unified View
  Tech report, `http://citeseer.ist.psu.edu/67130.html`

Compositional verification of modules



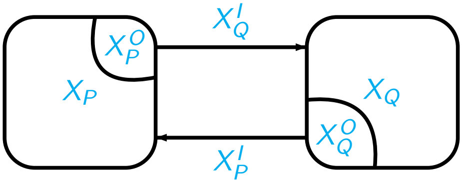
Compositional verification of modules



Safety property φ: boolean formula over $X^I \cup X^O$

- $s_1 s_2 \ldots \models \varphi$ if for each i, $s_i^{I \cup O} \models \varphi$

Compositional verification of modules



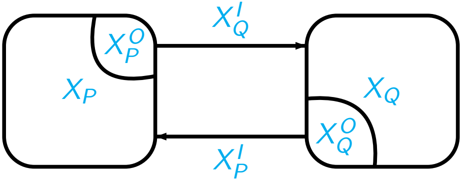
Safety property φ: boolean formula over $X^I \cup X^O$

- $s_1 s_2 \ldots \models \varphi$ if for each i, $s_i^{I \cup O} \models \varphi$

When does $P \parallel Q \models \varphi$?

- For each $\sigma \in \mathit{VisBeh}(P \parallel Q)$, $\sigma \models \varphi$

Compositional verification of modules



Safety property φ: boolean formula over $X^I \cup X^O$

- $s_1 s_2 \ldots \models \varphi$ if for each i, $s_i^{I \cup O} \models \varphi$

When does $P \parallel Q \models \varphi$?

- For each $\sigma \in VisBeh(P \parallel Q)$, $\sigma \models \varphi$

Assume guarantee reasoning

- Find R such that:
  - $P \parallel R \models \varphi$
  - $VisBeh(Q) \subseteq VisBeh(R)$
- Learn a regular language R with small DFA?

Compositional verification of modules . . .

Most permissive R

$$L_{\mathsf{max}} = \{\sigma \mid \sigma \in \mathit{VisBeh}(P) \Rightarrow \sigma \models \varphi\}$$

Compositional verification of modules . . .

Most permissive R

$$L_{\max} = \{\sigma \mid \sigma \in VisBeh(P) \Rightarrow \sigma \models \varphi\}$$

Lower bound for R

$$L_{\min} = VisBeh(Q)$$

Compositional verification of modules . . .

Most permissive R

$$L_{\max} = \{\sigma \mid \sigma \in \mathit{VisBeh}(P) \Rightarrow \sigma \models \varphi\}$$

Lower bound for R

$$L_{\min} = \mathit{VisBeh}(Q)$$

Note that both $L_{\max}$ and $L_{\min}$ are regular

Compositional verification of modules . . .

Most permissive R

$$L_{\max} = \{\sigma \mid \sigma \in \mathit{VisBeh}(P) \Rightarrow \sigma \models \varphi\}$$

Lower bound for R

$$L_{\min} = \mathit{VisBeh}(Q)$$

Note that both $L_{\max}$ and $L_{\min}$ are regular

Want to learn R, $L_{\min} \subseteq R \subseteq L_{\max}$

Compositional verification of modules . . .

Most permissive R

$$L_{\mathrm{max}} = \{\sigma \mid \sigma \in \mathit{VisBeh}(P) \Rightarrow \sigma \models \varphi\}$$

Lower bound for R

$$L_{\mathrm{min}} = \mathit{VisBeh}(Q)$$

Note that both L_{max} and L_{min} are regular

Want to learn R, $L_{\mathrm{min}} \subseteq R \subseteq L_{\mathrm{max}}$

Target language is unknown!

Compositional verification: Implementing the teacher ...

Recall that $L_{\min} \subseteq R \subseteq L_{\max}$

Compositional verification: Implementing the teacher . . .

Recall that $L_{\mathrm{min}} \subseteq R \subseteq L_{\mathrm{max}}$

Equivalence query, $L(C) = R$?

- Subset query $L(C) \subseteq L_{\mathrm{max}}$?
- Superset query $L(C) \supseteq L_{\mathrm{min}}$?

Compositional verification: Implementing the teacher . . .

Recall that $L_{\text{min}} \subseteq R \subseteq L_{\text{max}}$

Equivalence query, $L(C) = R$?

- Subset query $L(C) \subseteq L_{\text{max}}$?
- Superset query $L(C) \supseteq L_{\text{min}}$?

Membership query, $w \in R$?

- If $w \notin L_{\text{max}}$, answer No.
- If $w \in L_{\text{min}}$, answer Yes.
- If $w \in L_{\text{max}} \setminus L_{\text{min}}$, ambiguous!
  - Heuristic: Answer Yes (i.e., answer with respect to L_{max})
  - May result in larger R than required

Compositional verification: Implementing the teacher . . .

Recall that $L_{\text{min}} \subseteq R \subseteq L_{\text{max}}$

Equivalence query, $L(C) = R$?

- Subset query $L(C) \subseteq L_{\text{max}}$?
- Superset query $L(C) \supseteq L_{\text{min}}$?

Membership query, $w \in R$?

- If $w \notin L_{\text{max}}$, answer No.
- If $w \in L_{\text{min}}$, answer Yes.
- If $w \in L_{\text{max}} \setminus L_{\text{min}}$, ambiguous!
  - Heuristic: Answer Yes (i.e., answer with respect to L_{max})
  - May result in larger R than required

Practical note: Use BDDs to deal with large alphabet $X^I \cup X^O$

Learning interfaces

- A class with variables V and methods M. Each method call returns values over $V_R \subseteq V$

Learning interfaces

- A class with variables V and methods M. Each method call returns values over $V_R \subseteq V$
- A run is a sequence $(m_1, s_R^1), (m_2, s_R^2), \ldots$

Learning interfaces

- A class with variables V and methods M. Each method call returns values over $V_R \subseteq V$
- A run is a sequence $(m_1, s_R^1), (m_2, s_R^2), \ldots$
- Safety specification: Boolean formula φ on return variables

Learning interfaces

- A class with variables V and methods M. Each method call returns values over $V_R \subseteq V$
- A run is a sequence $(m_1, s_R^1), (m_2, s_R^2), \ldots$
- Safety specification: Boolean formula φ on return variables
- Want to restrict runs of the class to permit only safe runs

Learning interfaces

- A class with variables V and methods M. Each method call returns values over $V_R \subseteq V$
- A run is a sequence $(m_1, s_R^1), (m_2, s_R^2), \ldots$
- Safety specification: Boolean formula φ on return variables
- Want to restrict runs of the class to permit only safe runs
- An interface is a function $I : (M \times V_R)^* \to 2^M$

Learning interfaces

- A class with variables V and methods M. Each method call returns values over $V_R \subseteq V$
- A run is a sequence $(m_1, s_R^1), (m_2, s_R^2), \ldots$
- Safety specification: Boolean formula φ on return variables
- Want to restrict runs of the class to permit only safe runs
- An interface is a function $I : (M \times V_R)^* \rightarrow 2^M$
- An interface I is good if all runs consistent with I satisfy φ

Learning interfaces . . .

Given an class C and an interface I, interaction is a game over $C \parallel I$

- Given the history, I chooses a method m to execute
- Given the method m, C fixes the return state after m executes

Learning interfaces ...

Given an class C and an interface I, interaction is a game over $C \parallel I$

- Given the history, I chooses a method m to execute
- Given the method m, C fixes the return state after m executes

Observe that $L(I)$ is prefix closed. Hence, membership query $w \in L(I)$ can be converted into subset query $\mathit{Prefixes}(w) \subseteq L(I)$.

Learning interfaces . . .

Given an class C and an interface I, interaction is a game over $C \parallel I$

- Given the history, I chooses a method m to execute
- Given the method m, C fixes the return state after m executes

Observe that $L(I)$ is prefix closed. Hence, membership query $w \in L(I)$ can be converted into subset query $Prefixes(w) \subseteq L(I)$.

Checking $L(C) = L(I)$ is broken up into subset and superset queries, as before

Learning interfaces . . .

Given an class C and an interface I, interaction is a game over $C \parallel I$

- Given the history, I chooses a method m to execute
- Given the method m, C fixes the return state after m executes

Observe that $L(I)$ is prefix closed. Hence, membership query $w \in L(I)$ can be converted into subset query $\mathit{Prefixes}(w) \subseteq L(I)$.

Checking $L(C) = L(I)$ is broken up into subset and superset queries, as before

$L(C) \subseteq L(I)$: Build $C \parallel I$ and ask the CTL question $AG\varphi$

Learning interfaces . . .

Given an class C and an interface I, interaction is a game over $C \parallel I$

- Given the history, I chooses a method m to execute
- Given the method m, C fixes the return state after m executes

Observe that $L(I)$ is prefix closed. Hence, membership query $w \in L(I)$ can be converted into subset query $\mathit{Prefixes}(w) \subseteq L(I)$.

Checking $L(C) = L(I)$ is broken up into subset and superset queries, as before

$L(C) \subseteq L(I)$: Build $C \parallel I$ and ask the CTL question $AG\varphi$

$L(C) \supseteq L(I)$: More difficult, will not go into detail here.