Learning Communication Protocols from Scenarios

Benedikt Bollig[1] Joost-Pieter Katoen[2]
Carsten Kern[2] Martin Leucker[3]

ENS CACHAN [1]
Laboratoire Spécification
et Vérification

RWTH AACHEN UNIVERSITY [2]
Lehrstuhl für Informatik 2

TUM TECHNISCHE UNIVERSITÄT MÜNCHEN [3]
Institut für Informatik

TDI 2.0 06

Aachen 2006, December 1st

RWTH AACHEN UNIVERSITY

Outline

1. Introduction
2. Learning
3. Learning MSCs
4. Classes of learnable regular MSC languages
5. Tool Presentation

Presentation outline

1. Introduction
2. Learning
3. Learning MSCs
4. Classes of learnable regular MSC languages
5. Tool Presentation

Software Development

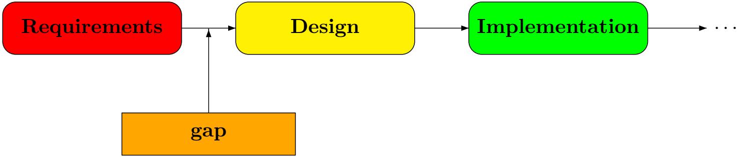
Initial software development phases

- initial phase: requirement elicitation
  - contradicting or incomplete system description
- goal: conforming design model

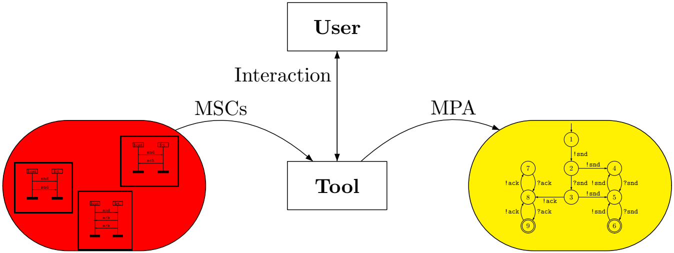
Problem

- gap between requirement specification and design phase i.e., *How to obtain an initial design model from a set of requirements*

Motivation

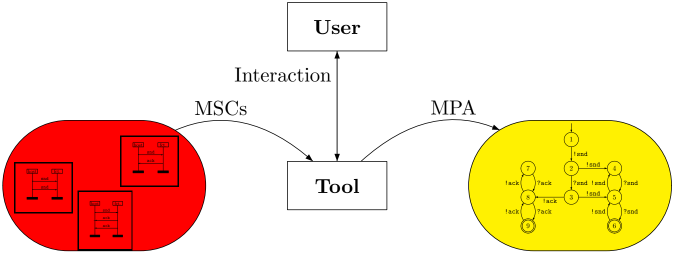


- closing gap between
  - requirement specification (possibly inconsistent) and
  - design model (complete description of system)
- similar to Harel's *play-in, play-out* approach
- novel aspect: use learning algorithms for synthesizing systems from scenario-based specifications



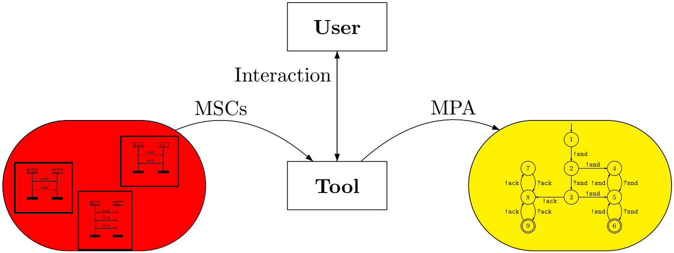
Idea:

- Use learning algorithms to synthesize models for communication protocols
- **Input:** set of MSCs (i.e., specification)
- **Output:** MPA fulfilling the specification



Idea:

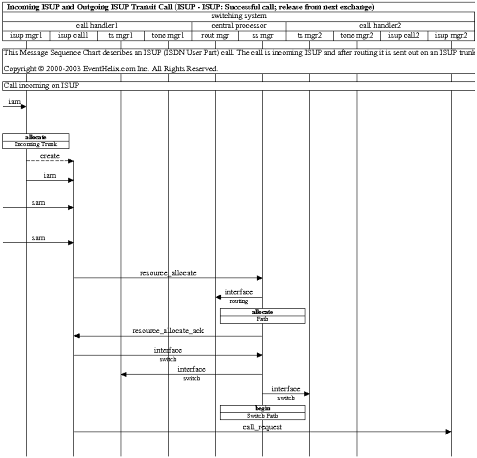
- Use learning algorithms to synthesize models for communication protocols
- **Input:** set of MSCs (i.e., specification)
- **Output:** MPA fulfilling the specification



Idea:

- Use learning algorithms to synthesize models for communication protocols
- **Input:** set of MSCs (i.e., specification)
- **Output:** MPA fulfilling the specification

A Message Sequence Chart



- standardized: ITU Z.120
- included in UML as sequence diagrams

Formally

An MSC M is a 5-tuple $M = \langle \mathcal{P}, E, \{\leq_p\}_{p\in\mathcal{P}}, <_{\mathsf{msg}}, l \rangle$

- $\mathcal{P}$: finite set of processes
- E: finite set of events ($E = \bigcup_{p\in\mathcal{P}} E_p$)
- $l : E \longrightarrow Act$: labeling function
- for $p \in \mathcal{P}$: $\leq_p \subseteq E_p \times E_p$ is a total order on E_p
- $<_{msg}$ describes the message order of M (partial order)

A set of MSCs is called an *MSC language*

A *linearization* of an MSC is a total ordering of E

Formally

An MSC M is a 5-tuple $M = \langle \mathcal{P}, E, \{\leq_p\}_{p \in \mathcal{P}}, <_{\mathsf{msg}}, l \rangle$

- $\mathcal{P}$: finite set of processes
- E: finite set of events ($E = \bigcup_{p \in \mathcal{P}} E_p$)
- $l : E \longrightarrow Act$: labeling function
- for $p \in \mathcal{P}$: $\leq_p \subseteq E_p \times E_p$ is a total order on E_p
- $<_{msg}$ describes the message order of M (partial order)

A set of MSCs is called an *MSC language*

A *linearization* of an MSC is a total ordering of E

Formally

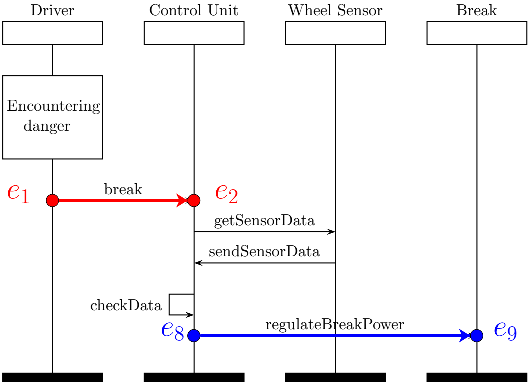
An MSC M is a 5-tuple $M = \langle \mathcal{P}, E, \{\leq_p\}_{p \in \mathcal{P}}, <_{\textsf{msg}}, l \rangle$

- $\mathcal{P}$: finite set of processes
- E: finite set of events ($E = \bigcup_{p \in \mathcal{P}} E_p$)
- $l : E \longrightarrow Act$: labeling function
- for $p \in \mathcal{P}$: $\leq_p \subseteq E_p \times E_p$ is a total order on E_p
- $<_{msg}$ describes the message order of M (partial order)

A set of MSCs is called an *MSC language*

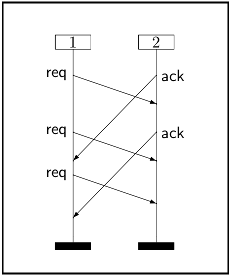
A *linearization* of an MSC is a total ordering of E

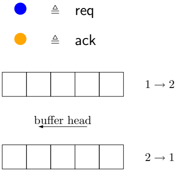
Scenario of the Antiblock System

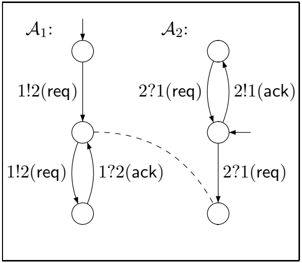


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

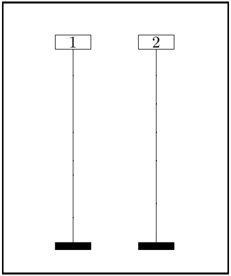


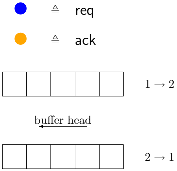


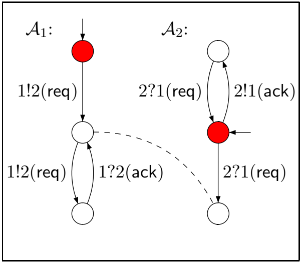


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

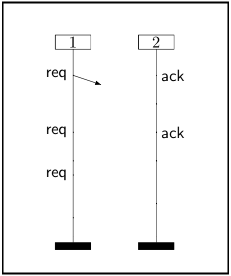


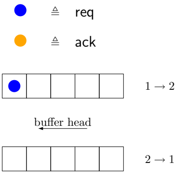


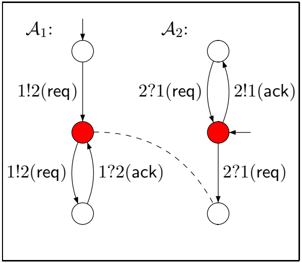


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

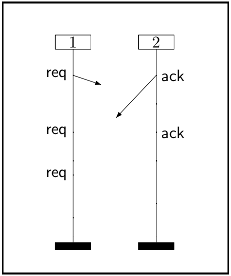


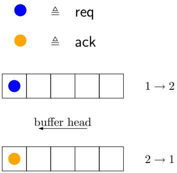


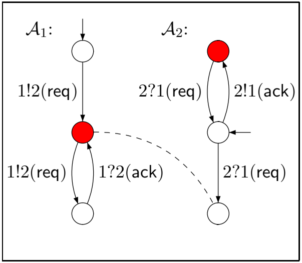


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

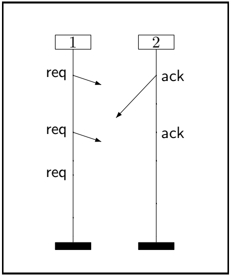


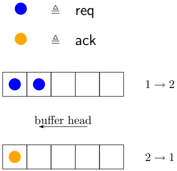


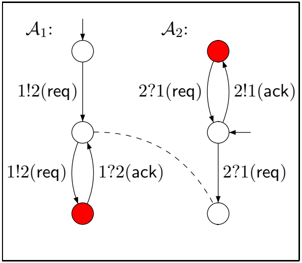


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

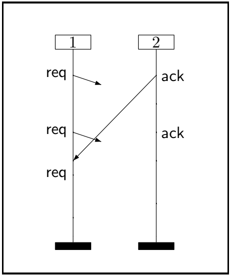


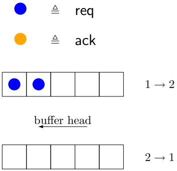


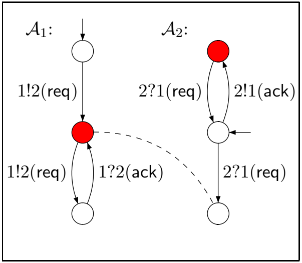


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

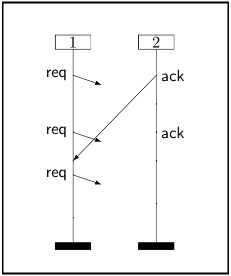


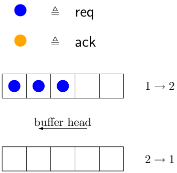


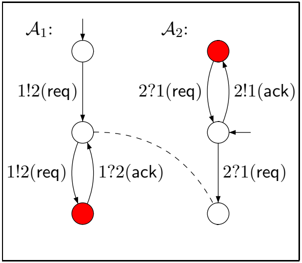


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

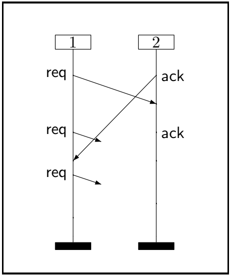


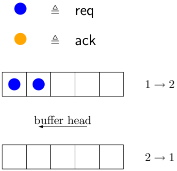


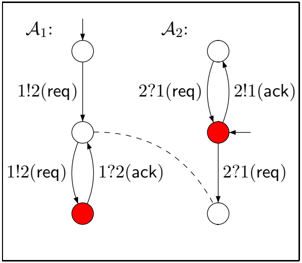


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

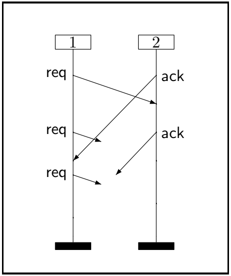


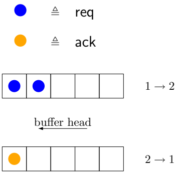


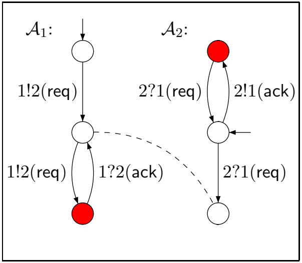


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

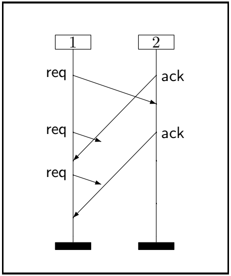


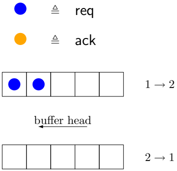


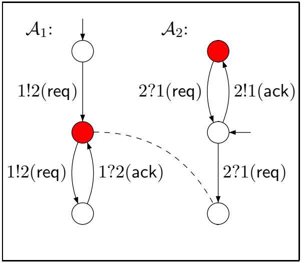


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

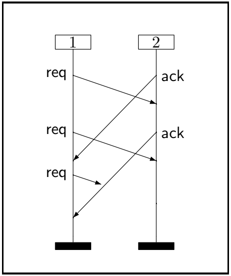


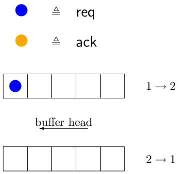


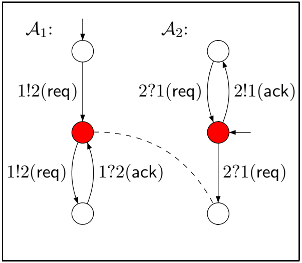


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q

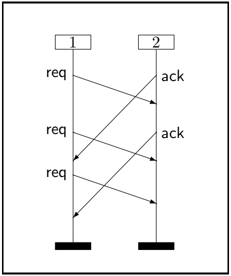


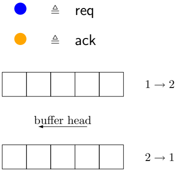


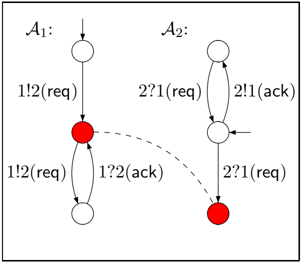


Message Passing Automata

- A set of finite-state automata (*processes*) with
  - common global initial state
  - set of global final states
- communication between automata through (reliable) FIFO channels
  - $p!q(a)$ appends message a to buffer between p and q
  - $q?p(a)$ removes message a from buffer between p and q







Presentation outline

1. Introduction
2. Learning
3. Learning MSCs
4. Classes of learnable regular MSC languages
5. Tool Presentation

Angluin's algorithm

Idea:

- algorithm for learning DFA (over Σ)
- learning a regular language $L(\mathcal{A}) \subseteq \Sigma^*$ by constructing a minimal DFA $\mathcal{A}$
- components:
  - *Learner*:
    - initially knows nothing about $\mathcal{A}$
    - tries to learn $\mathcal{A}$
    - proposes *hypothetical* automaton $\mathcal{H}$
  - *Teacher*:
    - knows $\mathcal{A}$
    - answers membership queries of *Learner* $(w \overset{?}{\in} L(\mathcal{A}))$
  - *Oracle*:
    - knows $\mathcal{A}$
    - answers equivalence queries of *Learner* $(L(\mathcal{H}) \overset{?}{=} L(\mathcal{A}))$

Angluin's algorithm

Idea:

- algorithm for learning DFA (over Σ)
- learning a regular language $L(\mathcal{A}) \subseteq \Sigma^*$ by constructing a minimal DFA $\mathcal{A}$
- components:
  - *Learner*:
    - initially knows nothing about $\mathcal{A}$
    - tries to learn $\mathcal{A}$
    - proposes *hypothetical* automaton $\mathcal{H}$
  - *Teacher*:
    - knows $\mathcal{A}$
    - answers membership queries of *Learner* $(w \overset{?}{\in} L(\mathcal{A}))$
  - *Oracle*:
    - knows $\mathcal{A}$
    - answers equivalence queries of *Learner* $(L(\mathcal{H}) \overset{?}{=} L(\mathcal{A}))$

Angluin's algorithm

Idea:

- algorithm for learning DFA (over Σ)
- learning a regular language $L(\mathcal{A}) \subseteq \Sigma^*$ by constructing a minimal DFA $\mathcal{A}$
- components:
  - *Learner*:
    - initially knows nothing about $\mathcal{A}$
    - tries to learn $\mathcal{A}$
    - proposes *hypothetical* automaton $\mathcal{H}$
  - *Teacher*:
    - knows $\mathcal{A}$
    - answers membership queries of *Learner* $(w \overset{?}{\in} L(\mathcal{A}))$
  - *Oracle*:
    - knows $\mathcal{A}$
    - answers equivalence queries of *Learner* $(L(\mathcal{H}) \overset{?}{=} L(\mathcal{A}))$

Angluin's algorithm

Idea:

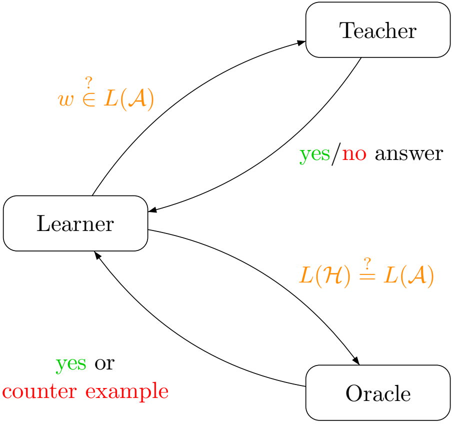
- algorithm for learning DFA (over Σ)
- learning a regular language $L(\mathcal{A}) \subseteq \Sigma^*$ by constructing a minimal DFA $\mathcal{A}$
- components:
  - *Learner*:
    - initially knows nothing about $\mathcal{A}$
    - tries to learn $\mathcal{A}$
    - proposes *hypothetical* automaton $\mathcal{H}$
  - *Teacher*:
    - knows $\mathcal{A}$
    - answers membership queries of *Learner* $(w \overset{?}{\in} L(\mathcal{A}))$
  - *Oracle*:
    - knows $\mathcal{A}$
    - answers equivalence queries of *Learner* $(L(\mathcal{H}) \overset{?}{=} L(\mathcal{A}))$

Angluin's algorithm

Idea:

- algorithm for learning DFA (over Σ)
- learning a regular language $L(\mathcal{A}) \subseteq \Sigma^*$ by constructing a minimal DFA $\mathcal{A}$
- components:
  - *Learner*:
    - initially knows nothing about $\mathcal{A}$
    - tries to learn $\mathcal{A}$
    - proposes *hypothetical* automaton $\mathcal{H}$
  - *Teacher*:
    - knows $\mathcal{A}$
    - answers membership queries of *Learner* $(w \overset{?}{\in} L(\mathcal{A}))$
  - *Oracle*:
    - knows $\mathcal{A}$
    - answers equivalence queries of *Learner* $(L(\mathcal{H}) \overset{?}{=} L(\mathcal{A}))$

Angluin's algorithm


Teacher
$w \overset{?}{\in} L(\mathcal{A})$
yes/no answer
Learner
$L(\mathcal{H}) \overset{?}{=} L(\mathcal{A})$
yes or
counter example
Oracle

Presentation outline

1 Introduction

2 Learning

3 Learning MSCs

4 Classes of learnable regular MSC languages

5 Tool Presentation

Goal

- Learning MPA from examples (MSCs)

Solution

- extending Angluin's algorithm
- **Input:** linearizations of MSCs
    - positive scenarios are included in the language to learn
    - negative scenarios must not be contained
- positive and **negative** scenarios form system behavior

Problem

- correspondence between MPA and regular word languages is needed (because Angluin's algorithm is designed form learning regular word languages)

Goal

- Learning MPA from examples (MSCs)

Solution

- extending Angluin's algorithm
- **Input:** linearizations of MSCs
  - positive scenarios are included in the language to learn
  - negative scenarios must not be contained
- positive and **negative** scenarios form system behavior

Problem

- correspondence between MPA and regular word languages is needed (because Angluin's algorithm is designed form learning regular word languages)

Goal

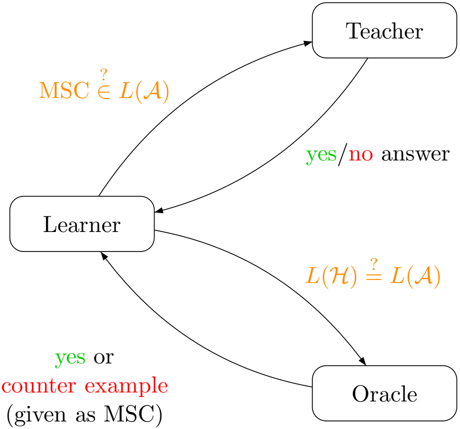
- Learning MPA from examples (MSCs)

Solution

- extending Angluin's algorithm
- **Input:** linearizations of MSCs
  - positive scenarios are included in the language to learn
  - negative scenarios must not be contained
- positive and **negative** scenarios form system behavior

Problem

- correspondence between MPA and regular word languages is needed (because Angluin's algorithm is designed form learning regular word languages)



Teacher

$\mathrm{MSC} \overset{?}{\in} L(\mathcal{A})$

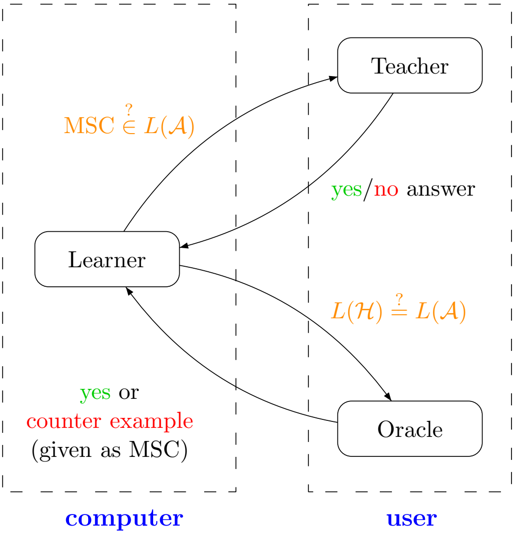
yes/no answer

Learner

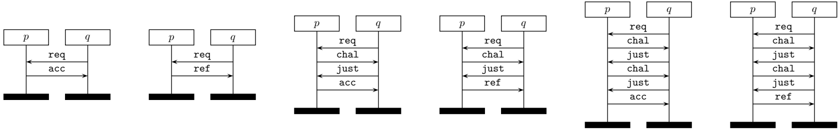
$L(\mathcal{H}) \overset{?}{=} L(\mathcal{A})$

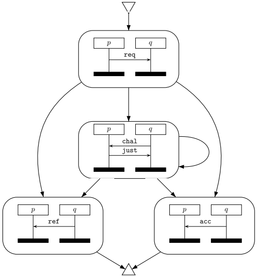
yes or
counter example
(given as MSC)

Oracle

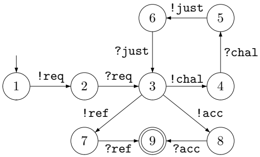

Teacher
$\text{MSC} \overset{?}{\in} L(\mathcal{A})$
yes/no answer
Learner
$L(\mathcal{H}) \overset{?}{=} L(\mathcal{A})$
yes or
counter example
(given as MSC)
Oracle
computer
user

A simple Negotiation Protocol





Smyle
⟹



membership queries:	9675
user queries:	65

Presentation outline

1 Introduction

2 Learning

3 Learning MSCs

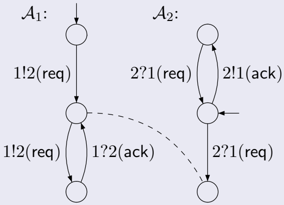
4 Classes of learnable regular MSC languages

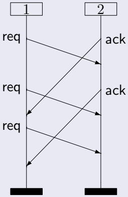
5 Tool Presentation

universally-bounded MPA

Definition: an MPA is *universally-bounded* iff

- its MSC language is universally-bounded
- informally: there is no run needing a buffer of infinite size
- Example of a universally-bounded MPA (bound: 2)

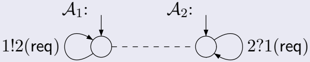


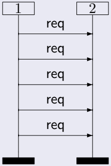


existentially-bounded MPA

Definition: an MPA is *existentially-bounded* iff

- its MSC language is existentially-bounded (buffer size B)
- informally: there is a run which needs a buffer of size $\leq B$
- Example of an existentially-bounded MPA (bound B=1)





universally-bounded *product* MPA

Definition: an MPA is a *universally-bounded product* MPA if

- acceptance condition is *local* (i.e., each process decides on its own when to halt)

A product MPA is safe/deadlock-free, iff

- from any configuration that is reachable from the initial configuration you can arrive at a final configuration

Theoretical results

Learnable classes: (channel size a priori fixed)

- universally-bounded MPA
- existentially-bounded MPA
- universally-bounded *safe product MPA*

Not learnable

- universally-bounded *product MPA*

Presentation outline

1. Introduction

2. Learning

3. Learning MSCs

4. Classes of learnable regular MSC languages

5. Tool Presentation

Algorithm

The learning chain (very coarse description)

```
1 Teacher specifies learning setup ($\forall/\exists$ and bound $B$)
2 Teacher provides set of positive and negative MSCs
3 while (Teacher not satisfied)
4         Learner asks set of membership queries
5         Teacher specifies them (as positive or negative)
6         Learner provides hypothesis automaton $\mathcal{H}$
7         Teacher is satisfied or provides counter example
8 Success: model was found
```

Summary

- synthesis of **design models** from scenario-based requirement specifications **using learning**

Advantages

- incremental generation of design models
- counterexamples for inconsistent requirements
- generation of minimal model

Disadvantages

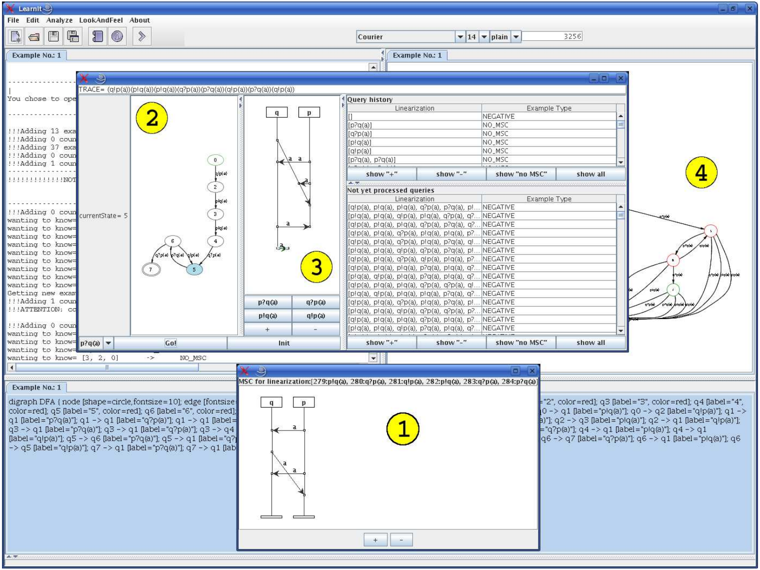
- for some protocols: huge memory requirements due to enormous number of linearizations

Implementation of learning approach: Smyle

S(ynthesizing) M(odels) (b)Y L(earning from) E(xamples)

- written in Java 1.5
- uses LearnLib library from University of Dortmund (Lehrstuhl 5 Prof. Dr. Bernhard Steffen)
- **Tool homepage:**
  `http://smyle.in.tum.de`
- **More concise information in: AIB-2006-12**
  Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

Tool Demo



```
----------------
|
You chose to ope

----------------

!!!Adding 13 exa
!!!Adding 0 coun
!!!Adding 37 exa
!!!Adding 0 coun
!!!Adding 1 coun
----------------
!!!!!!!!!!!!!NOT


----------------
!!!Adding 0 coun
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
wanting to know=
Getting new exam
!!!Adding 1 coun
!!!ATTENTION: co

!!!Adding 0 coun
wanting to know=
wanting to know=
wanting to know=
wanting to know= [3, 2, 0]          ->         NO_MSC
```

TRACE= (q!p(a))(p!q(a))(p!q(a))(q?p(a))(p?q(a))(q!p(a))(p?q(a))(q!p(a))

currentState= 5

p?q(a)

Query history

Linearization	Example Type
[]	NEGATIVE
[p?q(a)]	NO_MSC
[q?p(a)]	NO_MSC
[p!q(a)]	NO_MSC
[q!p(a)]	NO_MSC
[p?q(a), p?q(a)]	NO_MSC

Not yet processed queries

Linearization	Example Type
[q!p(a), p!q(a), p!q(a), q?p(a), p?q(a), p!...	NEGATIVE
[p!q(a), p!q(a), q!p(a), p!q(a), q?p(a), q?...	NEGATIVE
[q!p(a), p!q(a), p!q(a), q?p(a), p?q(a), q?...	NEGATIVE
[q!p(a), p!q(a), q?p(a), p!q(a), p!q(a), p?...	NEGATIVE
[q!p(a), p!q(a), q?p(a), p!q(a), p?q(a), q!...	NEGATIVE
[p!q(a), p!q(a), q!p(a), q?p(a), p?q(a), p!...	NEGATIVE
[q!p(a), p!q(a), q?p(a), q!p(a), p!q(a), p?...	NEGATIVE
[q!p(a), p!q(a), p!q(a), p?q(a), p!q(a), q?...	NEGATIVE
[p!q(a), q!p(a), p!q(a), p?q(a), p!q(a), q?...	NEGATIVE
[q!p(a), p!q(a), p!q(a), q?p(a), q?p(a), q!...	NEGATIVE
[p!q(a), q!p(a), p!q(a), p!q(a), p?q(a), q?...	NEGATIVE
[q!p(a), p!q(a), q?p(a), p!q(a), p?q(a), p!...	NEGATIVE
[p!q(a), p!q(a), q!p(a), q?p(a), q?p(a), p?...	NEGATIVE
[q!p(a), p!q(a), p!q(a), q?p(a), p!q(a), p?...	NEGATIVE
[p!q(a), p!q(a), q!p(a), p?q(a), p!q(a), q?...	NEGATIVE

MSC for linearization:[279:p!q(a), 280:q?p(a), 281:q!p(a), 282:p!q(a), 283:q?p(a), 284:p?q(a)]

```
digraph DFA { node [shape=circle,fontsize=10]; edge [fontsize
color=red]; q5 [label="5", color=red]; q6 [label="6", color=red];
q1 [label="p?q(a)"]; q1 -> q1 [label="q?p(a)"]; q1 -> q1 [label=
q3 -> q1 [label="p?q(a)"]; q3 -> q1 [label="q?p(a)"]; q3 -> q4
[label="q!p(a)"]; q5 -> q6 [label="p?q(a)"]; q5 -> q1 [label="q?
-> q5 [label="q!p(a)"]; q7 -> q1 [label="p?q(a)"]; q7 -> q1 [lab
```

```
="2", color=red]; q3 [label="3", color=red]; q4 [label="4",
0 -> q1 [label="p!q(a)"]; q0 -> q2 [label="q!p(a)"]; q1 ->
a)"]; q2 -> q3 [label="p!q(a)"]; q2 -> q1 [label="q!p(a)"];
="q?p(a)"]; q4 -> q1 [label="p!q(a)"]; q4 -> q1
q6 -> q7 [label="q?p(a)"]; q6 -> q1 [label="p!q(a)"]; q6
```

Thank you for your attention!