Replaying Play-In Play-Out:

Synthesis of Design Models from Scenarios by Learning

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

[1] Laboratoire Spécification et Vérification

[2] Lehrstuhl für Informatik 2

[3] Institut für Informatik

TACAS 2007, March 28^{th}

Outline

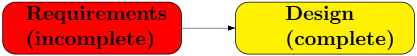
Presentation outline

Motivation

Requirements (incomplete)

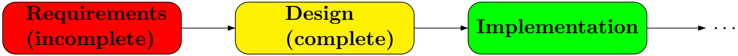
- initial phase: requirement elicitation
  - contradicting or incomplete system description
  - common description language: sequence diagrams
- goal: conforming design model
- closing gap between
  - requirement specification (usually incomplete) and
  - design model (complete description of system)
- similar to Harel's *play-in, play-out* approach

Motivation



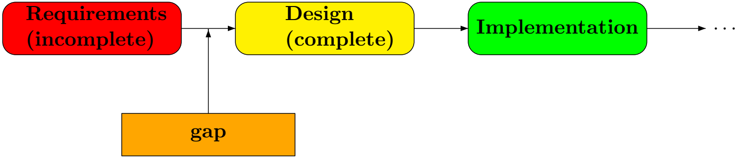
- initial phase: requirement elicitation
  - contradicting or incomplete system description
  - common description language: sequence diagrams
- goal: conforming design model
- closing gap between
  - requirement specification (usually incomplete) and
  - design model (complete description of system)
- similar to Harel's *play-in*, *play-out* approach

Motivation



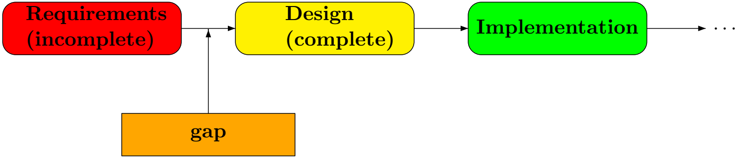
- initial phase: requirement elicitation
  - contradicting or incomplete system description
  - common description language: sequence diagrams
- goal: conforming design model
- closing gap between
  - requirement specification (usually incomplete) and
  - design model (complete description of system)
- similar to Harel's *play-in, play-out* approach

Motivation

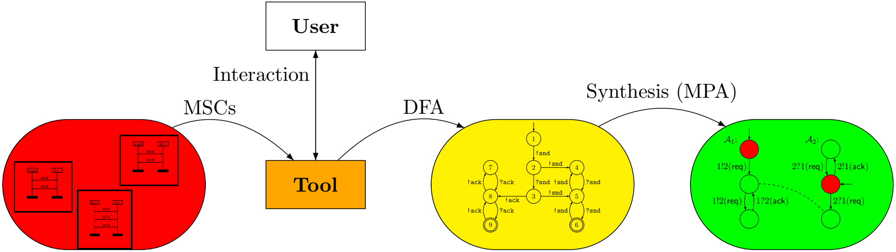


- initial phase: requirement elicitation
  - contradicting or incomplete system description
  - common description language: sequence diagrams
- goal: conforming design model
- closing gap between
  - requirement specification (usually incomplete) and
  - design model (complete description of system)
- similar to Harel's *play-in*, *play-out* approach

Motivation



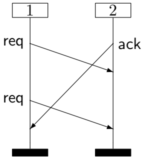
- initial phase: requirement elicitation
  - contradicting or incomplete system description
  - common description language: sequence diagrams
- goal: conforming design model
- closing gap between
  - requirement specification (usually incomplete) and
  - design model (complete description of system)
- similar to Harel's *play-in*, *play-out* approach



Our Approach

- use learning algorithms to synthesize models for communication protocols
- **Input:** set of Message Sequence Charts
  - standardized: ITU Z.120
  - included in UML as sequence diagrams
- **Output:** MPA fulfilling the specification
  - model is close to implementation

Message Sequence Chart



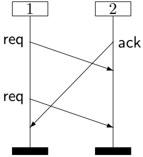
An MSC $M = \langle \mathcal{P}, E, \{\leq_p\}_{p \in \mathcal{P}}, <_{\mathsf{msg}}, l \rangle$

- $\mathcal{P}$: finite set of processes
- E: finite set of events ($E = \bigcup_{p \in \mathcal{P}} E_p$)
- $l : E \to Act = \{p!q(\mathsf{req}), p?q(\mathsf{ack}), \ldots\}$
- for $p \in \mathcal{P}$: $<_p \subseteq E_p \times E_p$ is a total order on E_p
- $<_{msg}$ relates sending and receiving events
- $\leq = \left(<_{msg} \cup \bigcup_{p \in \mathcal{P}} <_p \right)^*$

A set of MSCs is called an *MSC language*

A *linearization* of an MSC is a total ordering of E subsuming $\leq$

Message Sequence Chart



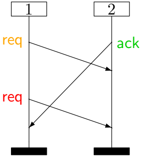
An MSC $M = \langle \mathcal{P}, E, \{\leq_p\}_{p\in\mathcal{P}}, <_{\text{msg}}, l\rangle$

- $\mathcal{P}$: finite set of processes
- E: finite set of events ($E = \bigcup_{p\in\mathcal{P}} E_p$)
- $l : E \to Act = \{p!q(\mathsf{req}), p?q(\mathsf{ack}), \dots\}$
- for $p \in \mathcal{P}$: $<_p \subseteq E_p \times E_p$ is a total order on E_p
- $<_{msg}$ relates sending and receiving events
- $\leq = \left(<_{msg} \cup \bigcup_{p\in\mathcal{P}} <_p\right)^*$

A set of MSCs is called an *MSC language*

A *linearization* of an MSC is a total ordering of E subsuming $\leq$

MSCs and Linearizations



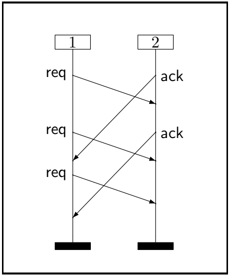
Some linearizations

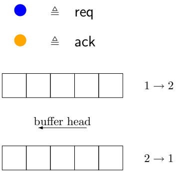
- 1!2(req) 1!2(req) 2!1(ack) 1?2(ack) 2?1(req) 2?1(req)
- 1!2(req) 2!1(ack) 1!2(req) 1?2(ack) 2?1(req) 2?1(req)
- 2!1(ack) 1!2(req) 2?1(req) 1!2(req) 2?1(req) 1?2(ack)
- ...

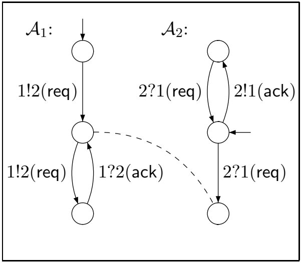
An MSC M is uniquely determined by its linearizations $Lin(M)$

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

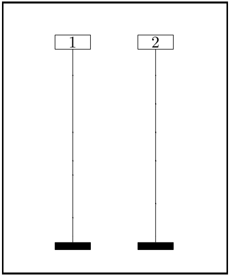


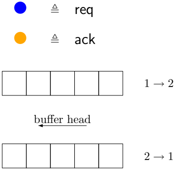


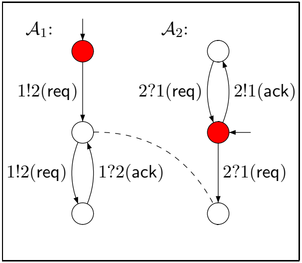


Message Passing Automata

- 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

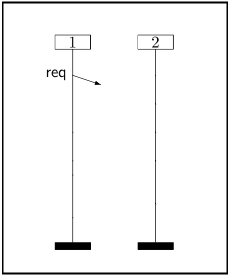


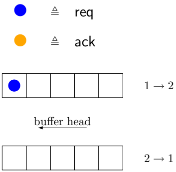


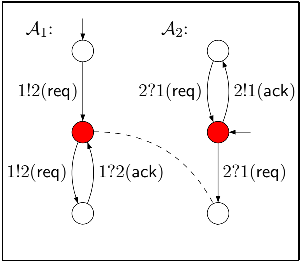


Message Passing Automata

- 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

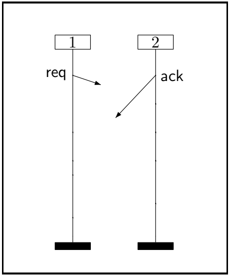


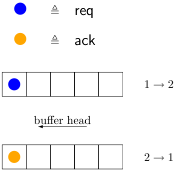


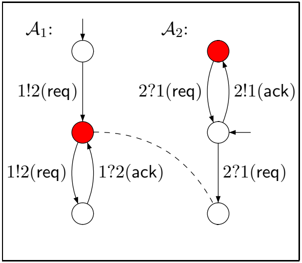


Message Passing Automata

- 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

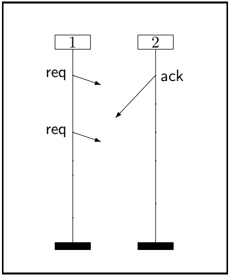


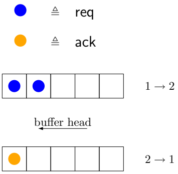


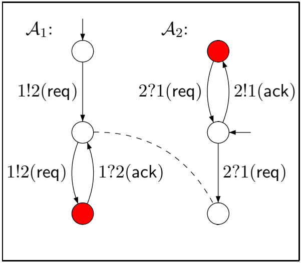


Message Passing Automata

- 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

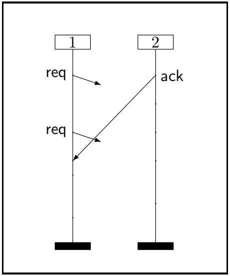


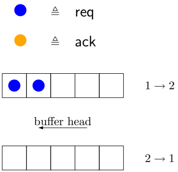


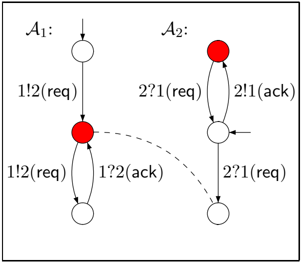


Message Passing Automata

- 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

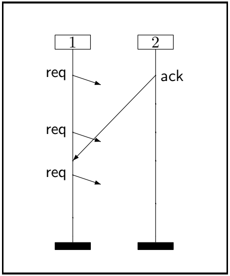


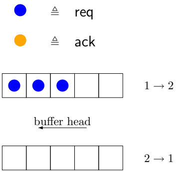


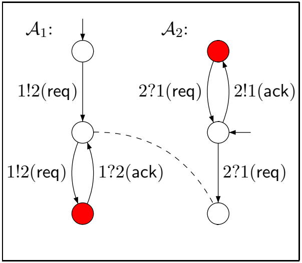


Message Passing Automata

- 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

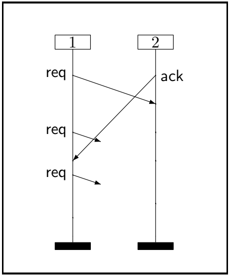


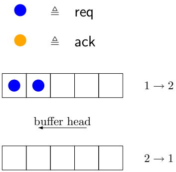


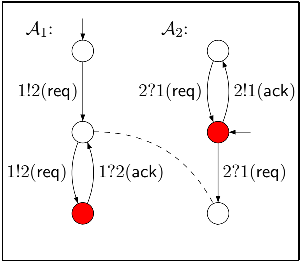


Message Passing Automata

- 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

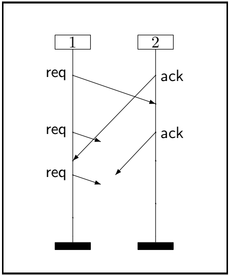


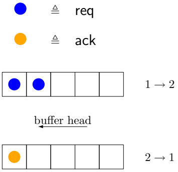


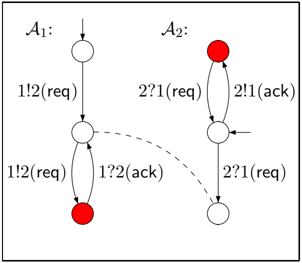


Message Passing Automata

- 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

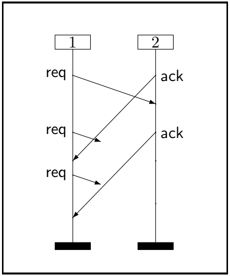


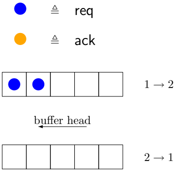


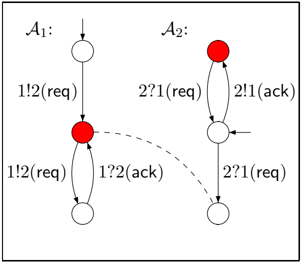


Message Passing Automata

- 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

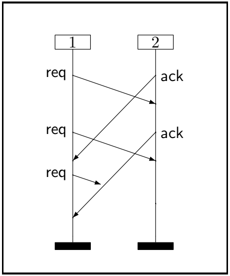


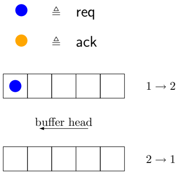


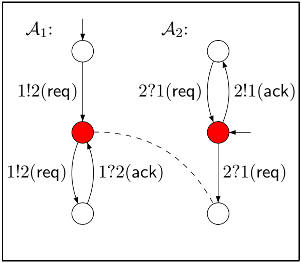


Message Passing Automata

- 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

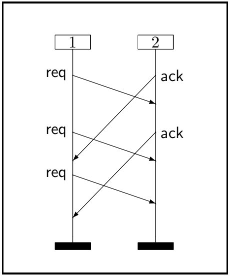


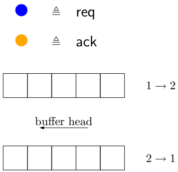


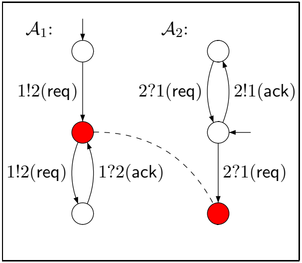


Message Passing Automata

- 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







Current State

Current State

- **goal:** *learning MPA*
- **given:** *learning DFA* [Angluin]

Presentation outline

Angluin's algorithm

Idea:

- learning regular language $L(\mathcal{A}) \subseteq \Sigma^*$ in terms of 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:

- learning regular language $L(\mathcal{A}) \subseteq \Sigma^*$ in terms of 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* $(w \stackrel{?}{\in} L(\mathcal{A}))$
  - *Oracle*:
    - knows $\mathcal{A}$
    - answers equivalence queries of *Learner* $(L(\mathcal{H}) \stackrel{?}{=} L(\mathcal{A}))$

Angluin's algorithm

Idea:

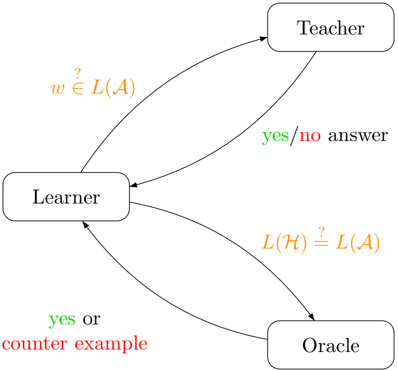
- learning regular language $L(\mathcal{A}) \subseteq \Sigma^*$ in terms of a minimal DFA $\mathcal{A}$
- components:
  - *Learner*:
    - initially knows nothing about $\mathcal{A}$
    - tries to learn $\mathcal{A}$
    - proposes *hypothetical* automaton $\mathcal{H}$
  - *Teacher*:
    - knows $\mathcal{A}$
    - 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:

- learning regular language $L(\mathcal{A}) \subseteq \Sigma^*$ in terms of 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* $\left(w \overset{?}{\in} L(\mathcal{A})\right)$
  - *Oracle*:
    - knows $\mathcal{A}$
    - answers equivalence queries of *Learner* $\left(L(\mathcal{H}) \overset{?}{=} L(\mathcal{A})\right)$

Angluin's algorithm



- query complexity: polynomial

Presentation outline

Goal

- learning MPA from examples (MSCs)

Approach

- 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 needed

Goal

- learning MPA from examples (MSCs)

Approach

- 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 needed

Goal

- learning MPA from examples (MSCs)

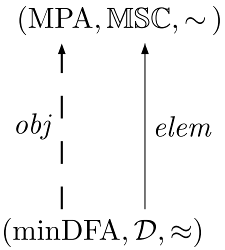
Approach

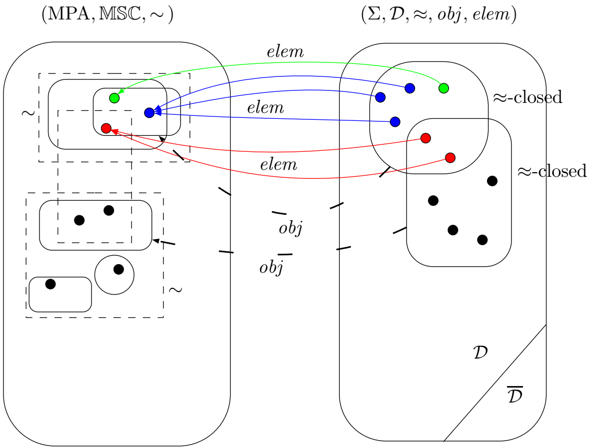
- extending Angluin's algorithm
- **Input:** linearizations of MSCs
  - positive scenarios are included in the language to learn
  - negative scenarios must not be contained
- positive and **negative** scenarios form system behavior

Problem

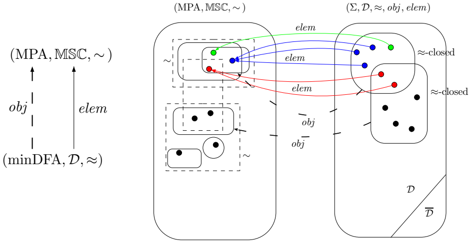
- correspondence between MPA and regular word languages needed

Defining a *Learning Setup*



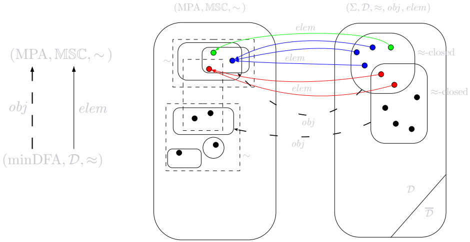


Defining a *Learning Setup*



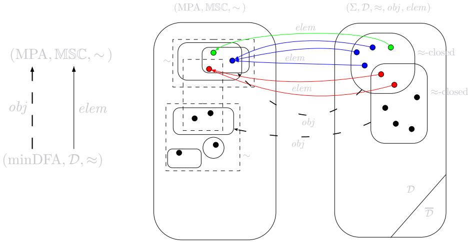
- membership queries for equiv. words need to be answered equivalently
- having found a hypothesis DFA $\mathcal{H}$:
  1. if $L(\mathcal{H}) \not\subseteq \mathcal{D}$, compute counterexample $w \in L(\mathcal{H}) \setminus \mathcal{D}$
  2. else if $L(\mathcal{H}) \subseteq \mathcal{D}$ but $L(\mathcal{H})$ not $\approx$-closed:
     - compute $w \approx w'$, $w \in L(\mathcal{H})$, $w' \notin L(\mathcal{H})$ and
     - perform membership queries for $[w]_{\approx}$

Defining a *Learning Setup*

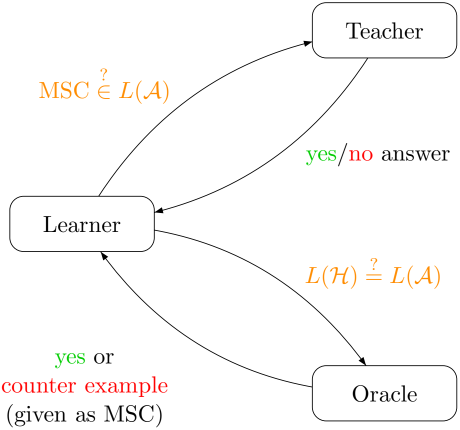


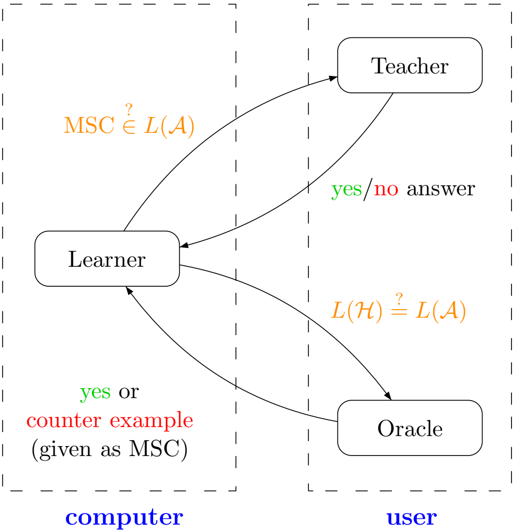
- membership queries for equiv. words need to be answered equivalently
- having found a hypothesis DFA $\mathcal{H}$:
  1. if $L(\mathcal{H}) \not\subseteq \mathcal{D}$, compute counterexample $w \in L(\mathcal{H}) \setminus \mathcal{D}$
  2. else if $L(\mathcal{H}) \subseteq \mathcal{D}$ but $L(\mathcal{H})$ not ≈-closed:
     - compute $w \approx w'$, $w \in L(\mathcal{H})$, $w' \notin L(\mathcal{H})$ and
     - perform membership queries for $[w]_\approx$

Defining a *Learning Setup*



- membership queries for equiv. words need to be answered equivalently
- having found a hypothesis DFA $\mathcal{H}$:
  1. if $L(\mathcal{H}) \not\subseteq \mathcal{D}$, compute counterexample $w \in L(\mathcal{H}) \setminus \mathcal{D}$
  2. else if $L(\mathcal{H}) \subseteq \mathcal{D}$ but $L(\mathcal{H})$ not $\approx$-closed:
     - compute $w \approx w'$, $w \in L(\mathcal{H})$, $w' \notin L(\mathcal{H})$ and
     - perform membership queries for $[w]_{\approx}$

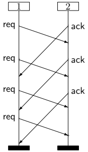

Teacher
MSC $\stackrel{?}{\in} L(\mathcal{A})$
yes/no answer
Learner
$L(\mathcal{H}) \stackrel{?}{=} L(\mathcal{A})$
yes or
counter example
(given as MSC)
Oracle


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

Classes of MSCs

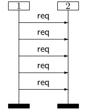
M is $\forall B$-bounded if

all linearizations of M do not exceed buffer bound B



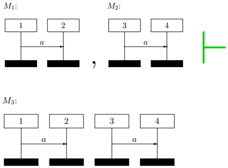
M is $\exists B$-bounded ($B \in \mathbb{N}$) if

events of M can be scheduled s.t. B is not exceeded



Definition: Inference relation $\vdash$

- process sets of M_1 and M_2 are distinct
- M_3 is inferred from two MSCs M_1, M_2



Results

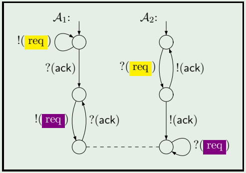
Learnable classes of MPA:

- $\forall$-bounded MPA
- $\exists B$-bounded MPA (for all $B \in \mathbb{N}$)
- $\forall$-bounded safe product MPA

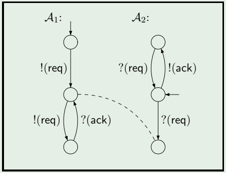
Not learnable

- $\forall$-bounded product MPA

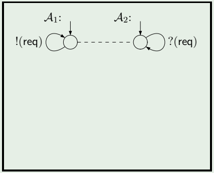
Example



not $\exists B$-bounded
no product MPA
not safe



$\forall$-bounded
product MPA
safe



not $\forall$-bounded
$\exists 1$-bounded
product MPA
safe

Learning Message-Passing Automata

Theorem: [Genest,Kuske,Muscholl], [Henriksen et al.]

- for any ∃-regular MSC language $\mathcal{L}$ one can compute an MPA $\mathcal{A}$, s.t. $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
- for any ∀-regular MSC language $\mathcal{L}$ one can compute a **deterministic** MPA $\mathcal{A}$, s.t. $\mathcal{L}(\mathcal{A}) = \mathcal{L}$

Theorem:

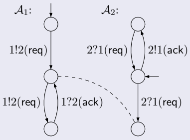
The ∀-regular safe product MSC languages are exactly the languages accepted by ∀-bounded safe product MPA

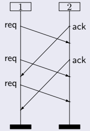
- ∃/∀-regular is treated with $\approx$
- ∀-regular safe product is handled by $\approx$, $\vdash$

universally-bounded MPA

A *universally-bounded* MPA

- Example of a universally-bounded MPA and ∀3-bounded MSC





- $\sim$: language equivalence of ∀-bounded MPA
- $\approx$: linearization equivalence
- *obj* : mapping a minimal DFA to a ∀-bounded MPA
- *elem* : mapping a linearization to its corresponding MSC

Algorithm for ∀-bounded MPA

Let $\mathcal{H}$ be a minimal DFA (hypothesis)

The problems $L(\mathcal{H}) \subseteq \mathcal{D}$ and $L(\mathcal{H})$ is $\approx$-closed are *constructively decidable*

- successively mark the states of $\mathcal{H}$ with channel contents
  - sending an event adds a message to the corresponding channel
  - receiving an event removes the message from the channel head
- check *diamond property* for independent σ, τ
- if problems in labeling the states are encountered, a counter example can be constructed and the learning algorithm continues

Complexity: linear in the size of $\mathcal{H}$

Algorithm for ∀-bounded MPA

Let $\mathcal{H}$ be a minimal DFA (hypothesis)

The problems $L(\mathcal{H}) \subseteq \mathcal{D}$ and $L(\mathcal{H})$ is ≈-closed are *constructively decidable*

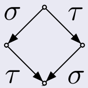
- successively mark the states of $\mathcal{H}$ with channel contents
  - sending an event adds a message to the corresponding channel
  - receiving an event removes the message from the channel head
- check *diamond property* for independent σ, τ
- if problems in labeling the states are encountered, a counter example can be constructed and the learning algorithm continues

Complexity: linear in the size of $\mathcal{H}$

Algorithm for ∀-bounded MPA

Let $\mathcal{H}$ be a minimal DFA (hypothesis)

The problems $L(\mathcal{H}) \subseteq \mathcal{D}$ and $L(\mathcal{H})$ is $\approx$-closed are *constructively decidable*

- successively mark the states of $\mathcal{H}$ with channel contents
  - sending an event adds a message to the corresponding channel
  - receiving an event removes the message from the channel head
- check *diamond property*  for independent σ, τ

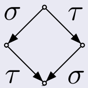
- if problems in labeling the states are encountered, a counter example can be constructed and the learning algorithm continues

Complexity: linear in the size of $\mathcal{H}$

Algorithm for ∀-bounded MPA

Let $\mathcal{H}$ be a minimal DFA (hypothesis)

The problems $L(\mathcal{H}) \subseteq \mathcal{D}$ and $L(\mathcal{H})$ is $\approx$-closed are *constructively decidable*

- successively mark the states of $\mathcal{H}$ with channel contents
  - sending an event adds a message to the corresponding channel
  - receiving an event removes the message from the channel head
- check *diamond property* 

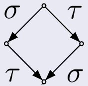
 for independent σ, τ
- if problems in labeling the states are encountered, a counter example can be constructed and the learning algorithm continues

Complexity: linear in the size of $\mathcal{H}$

Algorithm for ∀-bounded MPA

Let $\mathcal{H}$ be a minimal DFA (hypothesis)

The problems $L(\mathcal{H}) \subseteq \mathcal{D}$ and $L(\mathcal{H})$ is $\approx$-closed are *constructively decidable*

- successively mark the states of $\mathcal{H}$ with channel contents
  - sending an event adds a message to the corresponding channel
  - receiving an event removes the message from the channel head
- check *diamond property* 

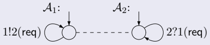
 for independent σ, τ
- if problems in labeling the states are encountered, a counter example can be constructed and the learning algorithm continues

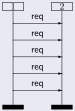
Complexity: linear in the size of $\mathcal{H}$

existentially B-bounded MPA

An *existentially B-bounded* MPA

- Example of an $\exists B$-bounded MPA (bound $B = 1$)





- $\sim$: language equivalence of $\exists B$-bounded MPA
- $\approx$: linearization equivalence for $\exists B$-bounded MSCs
- obj : mapping a minimal DFA to a $\exists B$-bounded MPA
- $elem$: mapping a linearization to its corresponding MSC

Algorithm for ∀-bounded safe product MPA

Let $\mathcal{H}$ be a ≈-closed minimal DFA

The problem if a regular ≈-closed set of MSC linearizations is recognized by some safe product MPA is *constructively decidable* (based on *EXPSPACE* algorithm by [Alur, Etessami, Yannakakis])

- construct deterministic MPA by projecting $\mathcal{H}$ to Act_p for any $p \in \mathcal{P}$ (determinizing and minimizing the resulting components): results in $(\mathcal{H}|_p)_{p \in \mathcal{P}}$
- $L(\mathcal{H})$ is recognized by some safe product MPA $\iff$ $(\mathcal{H}|_p)_{p \in \mathcal{P}}$ is safe and recognizes $L(\mathcal{H})$

- deadlocks contained?
- buffer bound exceeded?
- $L(\mathcal{H}) \stackrel{?}{=} L((\mathcal{H}|_p)_{p \in \mathcal{P}})$

Algorithm for ∀-bounded safe product MPA

Let $\mathcal{H}$ be a ≈-closed minimal DFA

The problem if a regular ≈-closed set of MSC linearizations is recognized by some safe product MPA is *constructively decidable* (based on *EXPSPACE* algorithm by [Alur, Etessami, Yannakakis])

- construct deterministic MPA by projecting $\mathcal{H}$ to Act_p for any $p \in \mathcal{P}$ (determinizing and minimizing the resulting components): results in $(\mathcal{H}|_p)_{p\in\mathcal{P}}$
- $L(\mathcal{H})$ is recognized by some safe product MPA $\iff$ $(\mathcal{H}|_p)_{p\in\mathcal{P}}$ is safe and recognizes $L(\mathcal{H})$

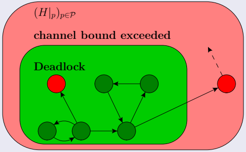
- deadlocks contained?
- buffer bound exceeded?
- $L(\mathcal{H}) \stackrel{?}{=} L((\mathcal{H}|_p)_{p\in\mathcal{P}})$

Algorithm for ∀-bounded safe product MPA

Let $\mathcal{H}$ be a ≈-closed minimal DFA

The problem if a regular ≈-closed set of MSC linearizations is recognized by some safe product MPA is *constructively decidable* (based on *EXPSPACE* algorithm by [Alur, Etessami, Yannakakis])

- construct deterministic MPA by projecting $\mathcal{H}$ to Act_p for any $p \in \mathcal{P}$ (determinizing and minimizing the resulting components): results in $(\mathcal{H}|_p)_{p\in\mathcal{P}}$
- $L(\mathcal{H})$ is recognized by some safe product MPA $\Longleftrightarrow$ $(\mathcal{H}|_p)_{p\in\mathcal{P}}$ is safe and recognizes $L(\mathcal{H})$



- deadlocks contained?
- buffer bound exceeded?
- $L(\mathcal{H}) \stackrel{?}{=} L((\mathcal{H}|_p)_{p\in\mathcal{P}})$

Results

Learnable classes of MPA:

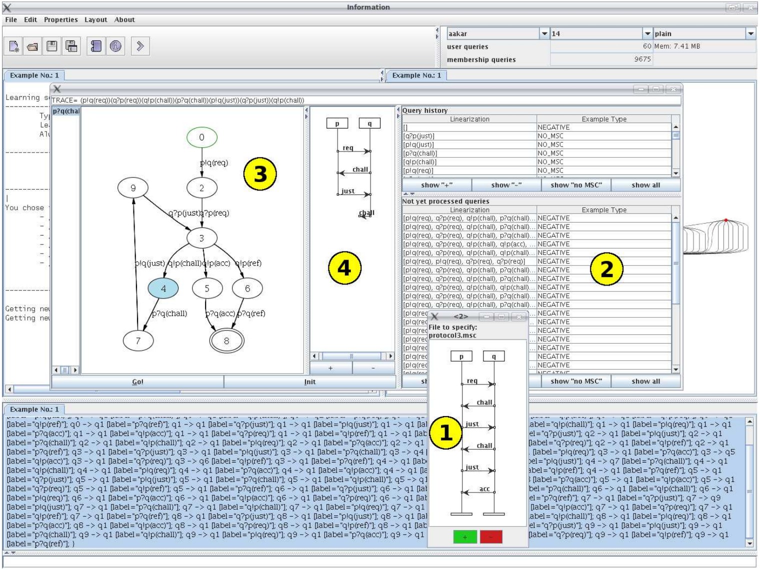
- $\forall$-bounded MPA
- $\exists B$-bounded MPA (for all $B \in \mathbb{N}$)
- $\forall$-bounded safe product MPA

Not learnable

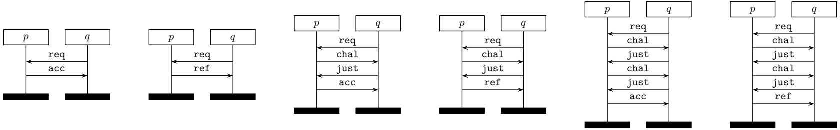
- $\forall$-bounded product MPA

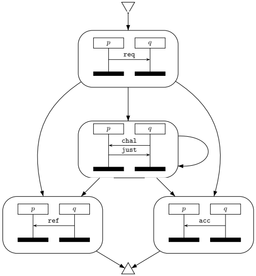
Presentation outline

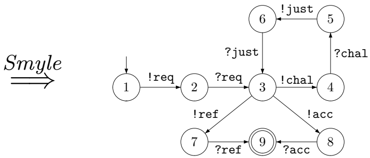
Tool Demo


Information
File Edit Properties Layout About
aakar
14
plain
user queries
60
Mem: 7.41 MB
membership queries
9675
Example No.: 1
TRACE= (p!q(req))(q?p(req))(q!p(chall))(p?q(chall))(p!q(just))(q?p(just))(q!p(chall))
Query history
Linearization
Example Type
show "+"
show "-"
show "no MSC"
show all
Not yet processed queries
Go!
Init
File to specify:
protocol3.msc
1
2
3
4

A Negotiation Protocol

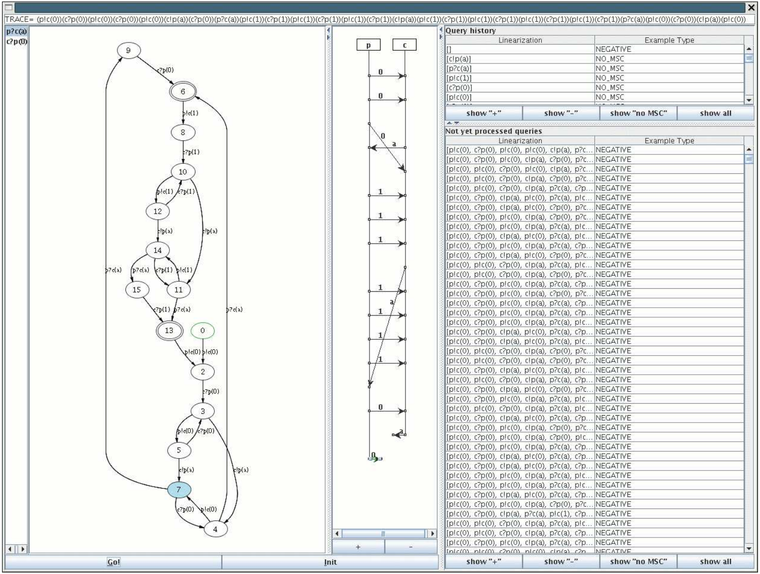






membership queries:	9675
user queries:	60

Alternating Bit Protocol (after 105 user queries)



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)
- **Smyle homepage:** `http://smyle.in.tum.de`

Presentation outline

Related Work

Similar Approaches

- *Play-In/Play-Out* approach [Harel et al.]
  - use the more expressive language of LSCs
  - more involved treatment of negative scenarios
- MAS (Minimally Adequate Synthesizer) [Mäkinen et al.]
  - based on Angluin's learning approach
  - only synchronous/sequential behavior
  - implementation model is not distributed

Outlook

Current Work

- more efficient partial order treatment
- dealing with *don't know* answers
- discover new/broader classes of learnable MPA
- case studies
- ...

http://smyle.in.tum.de

Thank you for your attention!