Learning Communication Protocols from Scenarios

Benedikt Bollig\textsuperscript{1} Joost-Pieter Katoen\textsuperscript{2}
Carsten Kern\textsuperscript{2} Martin Leucker\textsuperscript{3}

\textsuperscript{1}Laboratoire Spécification et Vérification
\textsuperscript{2}Lehrstuhl für Informatik 2
\textsuperscript{3}Institut für Informatik

Aachen 2006, December 1\textsuperscript{st}
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
- standardized: ITU Z.120
- included in UML as sequence diagrams
Formally

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

- $P$: finite set of processes
- $E$: finite set of events ($E = \bigcup_{p \in P} E_p$)
- $l : E \rightarrow \text{Act}$: labeling function
- for $p \in P$: $\leq_p \subseteq E_p \times E_p$ 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 $E$
An MSC $M$ is a 5-tuple $M = \langle \mathcal{P}, E, \{\leq_p\}_{p \in \mathcal{P}}, <_{\text{msg}}, l \rangle$

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

A set of MSCs is called a **MSC language**

A **linearization** of an MSC is a total ordering of $E$
An MSC $M$ is a 5-tuple $M = \langle \mathcal{P}, E, \{\leq_p\}_{p \in \mathcal{P}}, <_{msg}, l \rangle$

- $\mathcal{P}$: finite set of processes
- $E$: finite set of events ($E = \bigcup_{p \in \mathcal{P}} E_p$)
- $l : E \rightarrow Act$: 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

- **Driver**: Encountering danger
  
  - $e_1$ break

- **Control Unit**:
  
  - $e_2$ getSensorData
  
  - $e_8$ checkData
  
  - $regulateBreakPower$

- **Wheel Sensor**: sendSensorData

- **Break**: $e_9$
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 $q$
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 \((\text{processes})\) 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 \(q\)
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 $\Sigma$)**
- learning a regular language $L(A) \subseteq \Sigma^*$ by constructing a minimal DFA $A$
- components:
  - **Learner:**
    - initially knows nothing about $A$
    - tries to learn $A$
    - proposes *hypothetical* automaton $H$
  - **Teacher:**
    - knows $A$
    - answers membership queries of Learner ($w \not\in L(A)$)
  - **Oracle:**
    - knows $A$
    - answers equivalence queries of Learner ($L(H) \not\equiv L(A)$)
Angluin’s algorithm

Idea:

- algorithm for learning DFA (over $\Sigma$)
- learning a regular language $L(A) \subseteq \Sigma^*$ by constructing a minimal DFA $A$

components:

Learner:
- initially knows nothing about $A$
- tries to learn $A$
- proposes hypothetical automaton $H$

Teacher:
- knows $A$
- answers membership queries of Learner ($w \in L(A)$)

Oracle:
- knows $A$
- answers equivalence queries of Learner ($L(H) \equiv L(A)$)
Angluin’s algorithm

Idea:
- algorithm for learning DFA (over \( \Sigma \))
- 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 \( (w \in L(\mathcal{A})) \)
  - **Oracle:**
    - knows \( \mathcal{A} \)
    - answers equivalence queries of Learner \( (L(\mathcal{H}) \not\equiv L(\mathcal{A})) \)
Angluin’s algorithm

Idea:

- algorithm for learning DFA (over $\Sigma$)
- learning a regular language $L(A) \subseteq \Sigma^*$ by constructing a minimal DFA $A$
- components:
  - $Learner$:
    - initially knows nothing about $A$
    - tries to learn $A$
    - proposes hypothetical automaton $H$
  - $Teacher$:
    - knows $A$
    - answers membership queries of $Learner$ ($w \notin L(A)$)
  - $Oracle$:
    - knows $A$
    - answers equivalence queries of $Learner$ ($L(H) \equiv L(A)$)
Angluin’s algorithm

Idea:

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

\[ w \in L(A) \]

\[ L(H) \neq L(A) \]

\[ \text{yes or counter example} \]

\[ \text{yes/no answer} \]
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 for 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 for 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 for learning regular word languages)
Learning MSCs

Classes of learnable regular MSC languages

Tool Presentation

Introduction

Learning

MSC \( ? \in L(A) \)

Learner

Teacher

yes/no answer

L(H) \( ? \equiv L(A) \)

Oracle

yes or counter example (given as MSC)
Learning MSCs

Classes of learnable regular MSC languages

Tool Presentation

Introduction

Learning

Learner

Teacher

MSC $\in L(A)$

yes/no answer

$L(H) \supseteq L(A)$

yes or counter example (given as MSC)

Oracle

computer

user
A simple Negotiation Protocol

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
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)

\[ A_1: \]
- 1!2(req)
- 1!2(req)

\[ A_2: \]
- 2?1(req)
- 2!1(ack)
- 2?1(req)

[Diagram showing transitions between states for MPA A1 and A2 with labeled transitions (req, ack) and bounds indicated by arrows.]
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 $B=1$)

$$A_1:\overset{1!2(req)}{\longrightarrow}A_2:\overset{2?1(req)}{\longrightarrow}$$

```
1       2
req     req
req     req
req     req
```
Definition: an MPA is a \textit{universally-bounded product} MPA if

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

A product MPA is \textit{safe/deadlock-free}, iff

- from any configuration that is reachable from the initial configuration you can arrive at a final configuration
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(ythesizing) 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
Thank you for your attention!