Generating Models of Black-Box Components using Abstraction

Bengt Jonsson
Uppsala University
Joint work with
Fides Aarts\textsuperscript{1}, Falk Howar\textsuperscript{2}, Bernhard Steffen\textsuperscript{2}, Johan Uijen\textsuperscript{1}
1: Radboud University, Nijmegen
2: TU Dortmund
Modeling in (Component-Based) Design and Verification

Models are cornerstone of system development

- **Model Driven Development**
- **Model Based Testing**
  - tests generated as (abstract) executions
  - Tools: Qtronic, TGV, GOTCHA, TorX, ...
- **Model Checking**
  - Models of software, and of environment
Modeling Gap

- Typically, models are not available

- “Modeling SUT [system under test] is among biggest obstacles in Model Based Testing” [Hartmanis]

What to do if there is no model?
   (the norm in practice)
How to support generation of models?

• Model Behavior of existing implementation
  - By observations gained during extensive testing
    (Source code analysis: sometimes not feasible)

• Potential Applications:
  - Regression testing
  - Migrating from manual to model-based testing
  - Modeling environment of SUT, libraries
  - Verifying properties (Black Box Checking [Peled, Yannakakis])

• Other use of such techniques
  - For requirements capture
  - “Programming by Scenarios” (PlayIn-PlayOut) [Harel et al]
Model Generation by Inference

General Scheme:
• Given a set of instances:
  - Traces, Message charts, System states
• Produce a “simplest” specification which is consistent with these instances.

Applications:
• Behavioral Models from Behaviors
• Requirement Specifications from Scenarios
• Invariants from Sets of reachable/unreachable states
Outline

• Principles of Regular inference (automata learning)
• Extension to include data manipulation in protocols
• Abstraction techniques
• Experiments on communication protocols
• Further thoughts and future work
Regular Inference (Automata Learning)

• Construct Regular Language (as a DFA) from sample of accepted and rejected words.
• Developed since 1970’s. Applications in, e.g.,
  - Natural Language Processing,
  - Testing/Verification (more recently),
Regular Inference (Automata Learning)

• Construct Regular Language (as a DFA) from sample of accepted and rejected words.
• Developed since 1970’s. Applications in, e.g.,
  - Natural Language Processing,
  - Testing/Verification (more recently),
• off-line inference:
  - sample of words fixed a priori.
  - Problem is to construct “good enough” DFA.
  - Constructing minimal DFA is NP-complete [Gold 78]
• on-line inference:
  - words chosen dynamically, on the basis of previous information.
  - Easier to construct “good enough”/minimal DFA by extending sample with “interesting” words
  - Most well known algorithm: L* [Angluin 87]
Setup for inferring $A$

**Membership query:** is $w$ accepted or rejected?

**Equivalence query:** is $H$ equivalent to $A$?
Mealy Machines

- Finite State Machines w. input & output
- \(S\) states
- \(\delta: S \times I \rightarrow S\) transition function
- \(\lambda: S \times I \rightarrow O\) output function
- Often used for protocol modeling, for protocol testing techniques,

Assumptions:
- Deterministic
- Completely specified
Regular inference

- System viewed as Black box
- Membership query:
  - Supply input, observe output
- Record and Collect traces
- Construct protocol model

![Diagram of a state transition graph]

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a/0</td>
<td>a/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/1</td>
</tr>
<tr>
<td>b/1</td>
<td>b/0</td>
</tr>
<tr>
<td>b/0</td>
<td>b/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/1</td>
</tr>
<tr>
<td>b/1</td>
<td>b/0</td>
</tr>
<tr>
<td>b/0</td>
<td>b/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/1</td>
</tr>
<tr>
<td>b/1</td>
<td>b/0</td>
</tr>
<tr>
<td>b/0</td>
<td>b/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/1</td>
</tr>
<tr>
<td>b/1</td>
<td>b/0</td>
</tr>
<tr>
<td>b/0</td>
<td>b/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/0</td>
</tr>
<tr>
<td>a/0</td>
<td>a/1</td>
</tr>
<tr>
<td>b/1</td>
<td>b/0</td>
</tr>
<tr>
<td>b/0</td>
<td>b/0</td>
</tr>
</tbody>
</table>
Constructing Model from Traces

- Organize traces into tree
- Identify “equivalent” nodes

\[\begin{aligned}
a/0 \ a/1 \ b/1 \\
a/0 \ b/0 \ a/0 \\
b/1 \ a/0 \ b/0 \\
b/1 \ b/0 \ a/0
\end{aligned}\]
Constructing Model from Traces

- Organize traces into tree
- Identify “equivalent” nodes
- Merge nodes
- Form automaton

(a/0 a/1 b/1)
(a/0 b/0 a/0)
(b/1 a/0 b/0)
(b/1 b/0 a/0)
Which model?

• Many ways to identify nodes
• Finding “smallest” model is NP-complete [Gold78]
• Allow to ask for more information to get more traces
Which model?

• Many ways to identify nodes

• Finding “smallest” model is NP-complete [Gold74]

• Allow to ask for more information to get more traces
Which model?

- Many ways to identify nodes
- Finding “smallest” model is NP-complete [Gold74]
- Allow to ask for more information to get more traces
- Resolves ambiguities
- Constructing “smallest” model becomes simple [Angluin 87]
Algorithms for On-Line Inference

- Exist several variations, most well-known: $L^*$ [Angluin 87]
- Traces divided into prefix-suffix,
- Organized into Observation Table

<table>
<thead>
<tr>
<th>Prefixes</th>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>ε</td>
<td>0 1</td>
</tr>
<tr>
<td>a</td>
<td>1 0</td>
</tr>
<tr>
<td>b</td>
<td>0 0</td>
</tr>
<tr>
<td>aa</td>
<td>1</td>
</tr>
<tr>
<td>ab</td>
<td>0</td>
</tr>
<tr>
<td>ba</td>
<td>0</td>
</tr>
<tr>
<td>bb</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Output:

- Tree diagram with nodes labeled with prefixes and suffixes.
What about Protocols w. Data?
SIP Protocol (part of Server)

Variables: From, CurId, CurSeq

Constants: Me

\[
\text{INVITE}(\text{from}, \text{to}, \text{cid}, \text{cseq}) \ [\text{to} == \text{Me}] / \\
\text{From} = \text{from}; \ \text{CurId} = \text{cid}; \ \text{CurSeq} = \text{cseq}; \ \\
\text{100(From, to, CurId, CurSeq)}
\]

\[
\text{PRACK}(\text{from}, \text{to}, \text{cid}, \text{cseq}) \ [\text{from} == \text{From} \\
\land \ \text{to} == \text{Me} \land \ \text{cid} == \text{CurId} \\
\land \ \text{cseq} == \text{CurSeq}+1] / \text{200(From, to, CurId, CurSeq+1)}
\]

\[
\text{ACK}(\text{from}, \text{to}, \text{cid}, \text{cseq}) \ [\text{from} == \text{From} \\
\land \ \text{to} == \text{Me} \land \ \text{cid} == \text{CurId} \\
\land \ \text{cseq} == \text{CurSeq}] / \varepsilon
\]
Adapting to Automata Learning

Learner

INVITE(new,new)

Transducer

auxiliary variables:
CurId = ... CurSeq = ...

SIP (SUT)

INVITE(558,1)

100(current,next)

100(558,1)
Mapping parameters of input messages

\[
\begin{array}{llll}
\text{new} & \text{current} & \text{other} \\
\text{cid} & \text{CurId} = \text{"undef"} & = \text{CurId} & != \text{CurId} \\
\text{cseq} & \text{CurSeq} = \text{"undef"} & = \text{CurSeq} & = \text{CurSeq} + 1
\end{array}
\]

Maintaining auxiliary variables

\[
\begin{array}{llll}
\text{new} & \text{current} & \text{next} & \text{other} \\
\text{CurId} & = \text{cid} & <\text{unchanged}> & <\text{unchanged}> \\
\text{CurSeq} & = \text{cseq} & <\text{unchanged}> & <\text{unchanged}> & <\text{unchanged}>
\end{array}
\]
Inference by Abstraction

INVITE((new, new))

INVITE(558, 1)

auxiliary variables:
CurId = 558
CurSeq = 1

100(current, next)

100(558, 1)
Abstraction Mappings

Learner -> Transducer

INVITE(new,new)

auxiliary variables:

CurId = undef
CurSeq = undef

Transducer -> SIP (SUT)

INVITE(558,1)

100(current,next)

100(558,1)

Input-abstr
Abstraction Mappings

Learner \( \xrightarrow{\text{INVITE}(\text{new, new})} \) Transducer

auxiliary variables:
- \( \text{CurId} = 558 \)
- \( \text{CurSeq} = 1 \)

SIP (SUT) \( \xleftarrow{\text{100}(\text{current, next})} \) Learner

\( \xrightarrow{\text{100}(558, 1)} \) Transducer

\( \xleftarrow{\text{Output-abstr}} \)
Model inferred by Learner (part)

\[ S_0 \xrightarrow{\text{INVITE}(\text{new,new})/\text{200}(\text{current, current})} S_1 \]

\[ S_2 \xrightarrow{\text{PRACK}(\text{current,next})/\text{200}(\text{current,next})} S_1 \]

\[ S_2 \xrightarrow{\text{ACK}(\text{current, current})/\varepsilon} S_3 \]
What the SUT must have done:

Variables: \textit{CurId}, \textit{CurSeq}

\textbf{INVITE}(cid, cseq) [\textit{CurId == CurSeq == undef}] / \\
\quad \textit{CurId = cid}; \textit{CurSeq = cseq}; \\
\quad \textit{100(CurId,CurSeq)}

\textbf{PRACK}(cid, cseq) [\textit{cid == CurId} \\
\quad \land \textit{cseq == CurSeq+1}] / \textit{200(CurId,CurSeq+1)}

\textbf{ACK}(cid, cseq) [\textit{cid == CurId} \\
\quad \land \textit{cseq == CurSeq}] / \varepsilon
Healthiness condition:

Sufficiently Distinguishing input abstraction

INVITE(new,new)

INVITE(558,1)

INVITE(413,1)

auxiliary variables:
CurId = undef
CurSeq = undef
Healthiness condition:

Sufficiently Distinguishing input abstraction

INVITE(new,new)
Healthiness condition:

This does NOT guarantee that Learner will infer Finite Machine
Experiments

• Learner: the LearnLib tool (developed at TU Dortmund)
  - Efficient implementation of L*
  - Several equivalence oracles, e.g., controllable-size random test suite.

• SUT: ns-2 protocol simulator
  - Provides implementations of many standard protocols
  - Rather convenient C++ interface (no packet analyzer necessary)

• Transducer
  - Bridges asynchronous interface of LearnLib w. synchronous interface of ns-2
  - Implements instantiation of input symbols, and abstraction of output symbols
Session Initiation Protocol (SIP)

- Creating and Managing Multimedia protocol sessions
- SUT is ns-2 implementation of SIP Server
- Input messages have 7 parameters
- Each parameter abstracted to 2 or 3 values
- Inference: about 2 million membership queries
- Model w. 7 states and 41 transitions
SIP

• Model of behavior of SIP in ns-2
• SIP in ns-2 seems not to distinguish connected and unconnected state
Transport Control Protocol (TCP)

- Only connection establishment and termination
- SUT is ns-2 implementation of TCP
- Consider 2 sequence number parameters
- Each parameter abstracted to 2 or 3 values
- Model w. 33 states and 203 transitions
TCP

- Model of behavior of TCP in ns-2
- Only transitions with “accepted” values of input parameters are shown.
- Values of parameters not displayed
Conditions for Success:

INVITE(558, 1)

Finite control

predicates, e.g.,
cseq == CurSeq

functions, e.g.,
CurSeq + 1

100(558, 1)

Auxiliary
Variables:
CurId, CurSeq
Conditions for Success:

INVITE(558, 1)

Mealy machine

Precicates, e.g.,
cseq == CurSeq

Functions, e.g.,
CurSeq + 1

Variables:
CurId, CurSeq

INVITE(558, 1)

100(558, 1)
Conditions for Success:

If auxiliary variables in Transducer are more expressive than in SUT, i.e., all predicates and functions can be imitated, then finite control of Transducer can be a finite Mealy machine.

How find appropriate auxiliary variables, predicates, and functions?
Towards Automated Algorithm

• Infer input alphabet by successive refinement
• Library of commonly occurring alphabets,
• Adapt regular inference algorithm to dynamically changing alphabet
Timed Automata

- Based on standard automata
- **Clocks** give upper and lower bounds on distance in time between occurrences of symbols.
- Temporal properties of Timed Automata (reachability, LTL, ...) can be model-checked
- Implemented in tools (UPPAAL, IF/Kronos)

```
put ;
\[ x \leq 2 \]
\[ x := 0 \]

get ;
\[ x \geq 10 \]
\[ x := 0 \]
```
Inference of Event-Recording Automata [w. Olga Grinchtein]

- Timed Automata can not be determinized in general
- **Event-Recording Automata (ERA):** Each clock associated with particular symbol.
- ERA can be determinized

**Assumption:**
Inference algorithm can precisely control and record timing of symbols.

- put $x_{\text{get}} \leq 2$
- $x_{\text{put}} \geq 10$

\[s_0 \rightarrow s_1\]
Inference of ERAs

Problems:

- Determine guards
- Can be seen as inferring the input alphabet
- Done by refinement from observations of nondeterminism

\[ s_0 \quad \text{put} ; \quad x_{\text{get}} \leq 2 \quad \text{get} ; \quad x_{\text{put}} \geq 10 \quad s_1 \]
Refinement of guards

Start from untimed alphabet

Guards refined from nondeterminism

• get @0 put @2 accepted
• get @3 put @7 rejected

Determine the reason for difference by investigating other traces

• (binary) search procedure
• Finds “explaining pair”, e.g.,
  - get @2 put @4 accepted
  - get @2 put @4.5 rejected
Refinement of guards

Start from untimed alphabet

Guards refined from nondeterminism

- \texttt{get \texttt{@0} put \texttt{@2} accepted}
- \texttt{get \texttt{@3} put \texttt{@7} rejected}

Determine the reason for difference by investigating other traces

- (binary) search procedure
- Finds “explaining pair”, e.g.,
  - \texttt{get \texttt{@2} put \texttt{@4} accepted}
  - \texttt{get \texttt{@2} put \texttt{@4.5} rejected}
- Suggests guard $x_{\text{get}} \leq 2$ on put transition
Conclusions

• State machine models of communication protocols can be inferred
  - Using a priori knowledge about primitives for data manipulation

• The primitives can be inferred, given constraints on their form
Future work

- Library of common data structures
- Automatic generation of transducers
- Automated inference of input and output symbols
- Adapted Learning algorithm and implementation
- Incorporating nondeterminism
- Systematic coverage of possible concretizations of abstract symbols (= test input selection)