Inferring Automata with State-local Alphabet Abstractions

Malte Isberner[1,2], Falk Howar[2], Bernhard Steffen[1]

[1] TU Dortmund, Germany

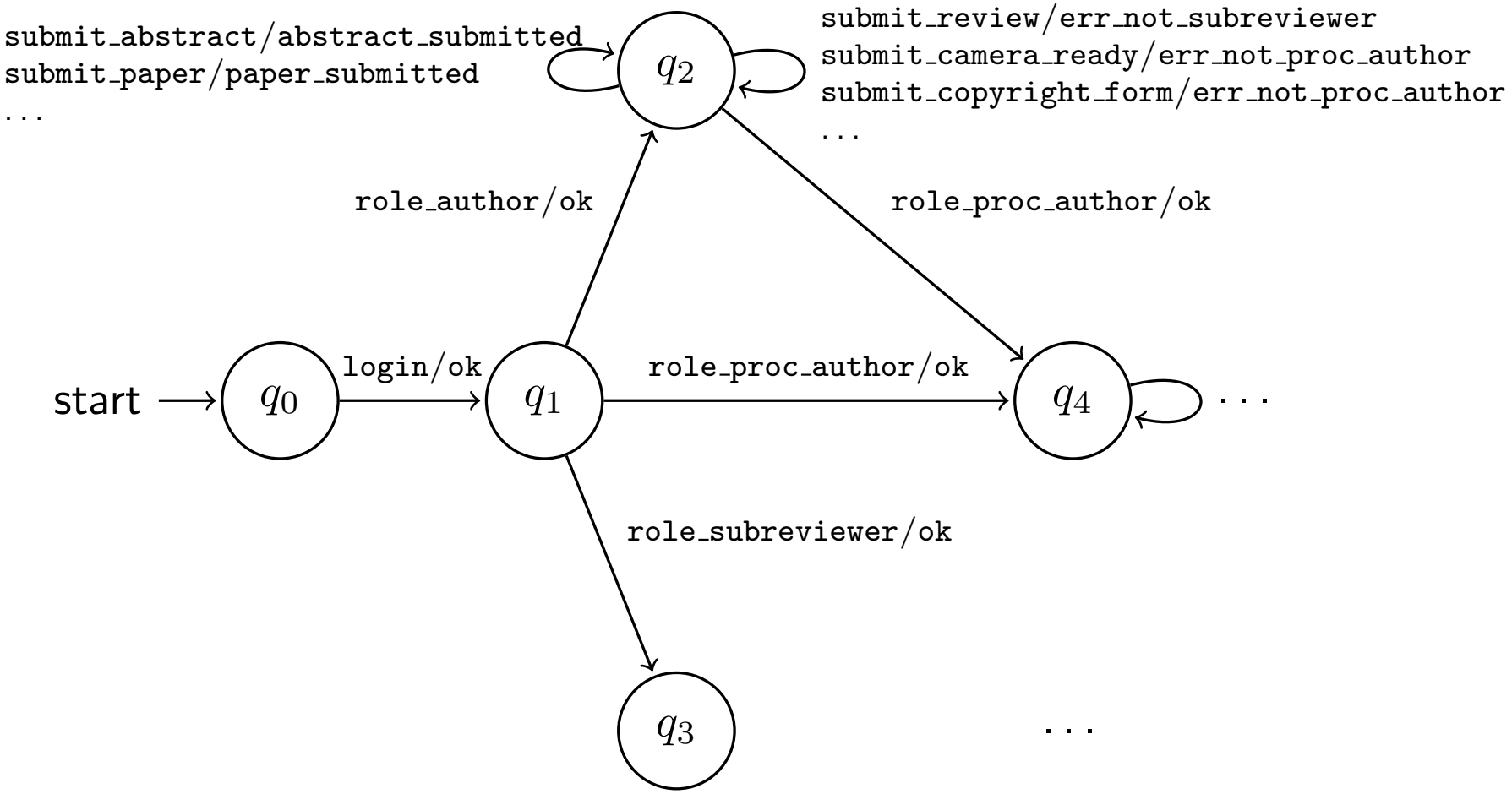
[2] CMU Silicon Valley, Mountain View, USA

technische universität dortmund

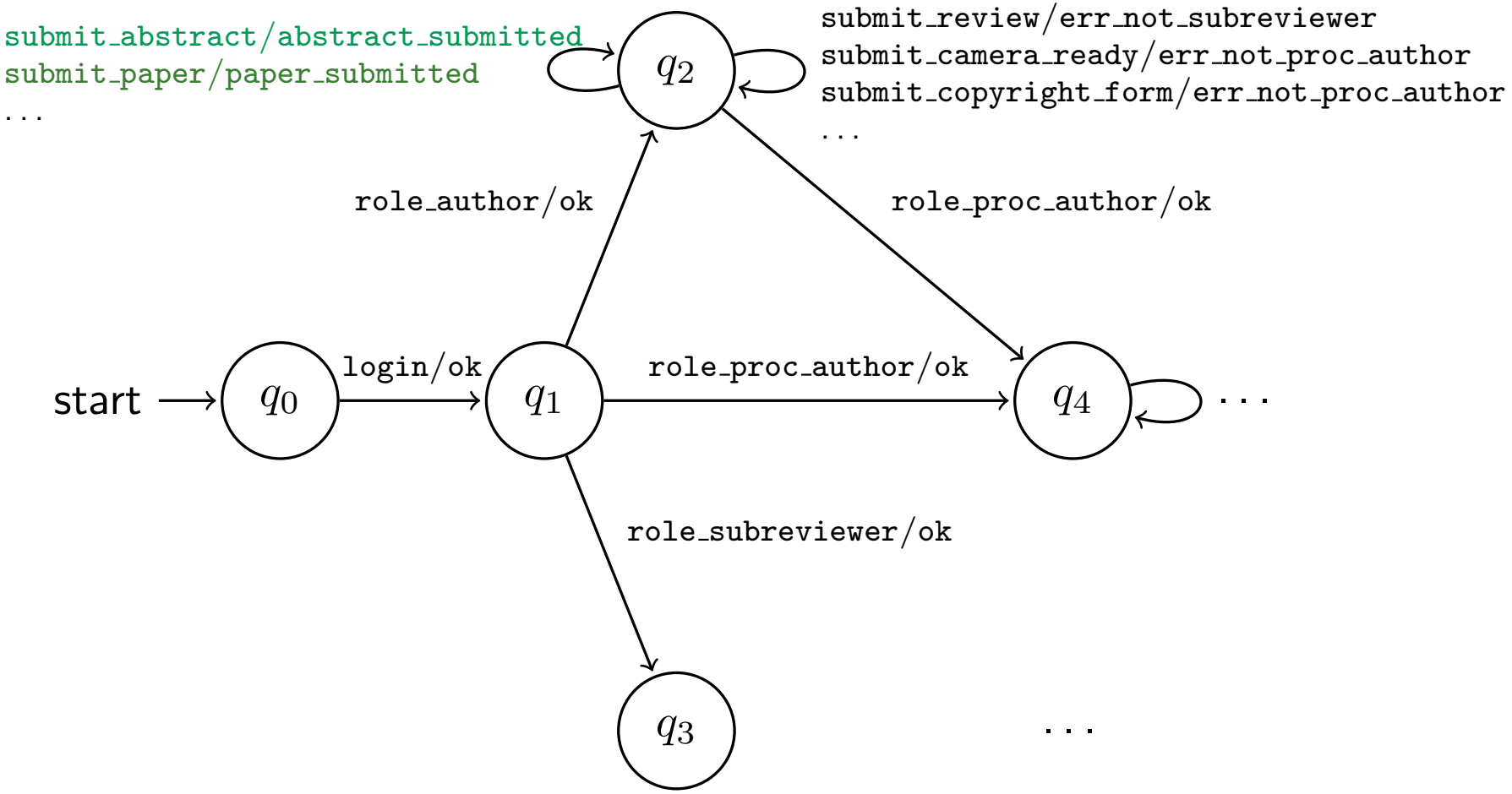
fakultät für informatik
lehrstuhl für programmiersysteme

NFM 2013 @ NASA ARC

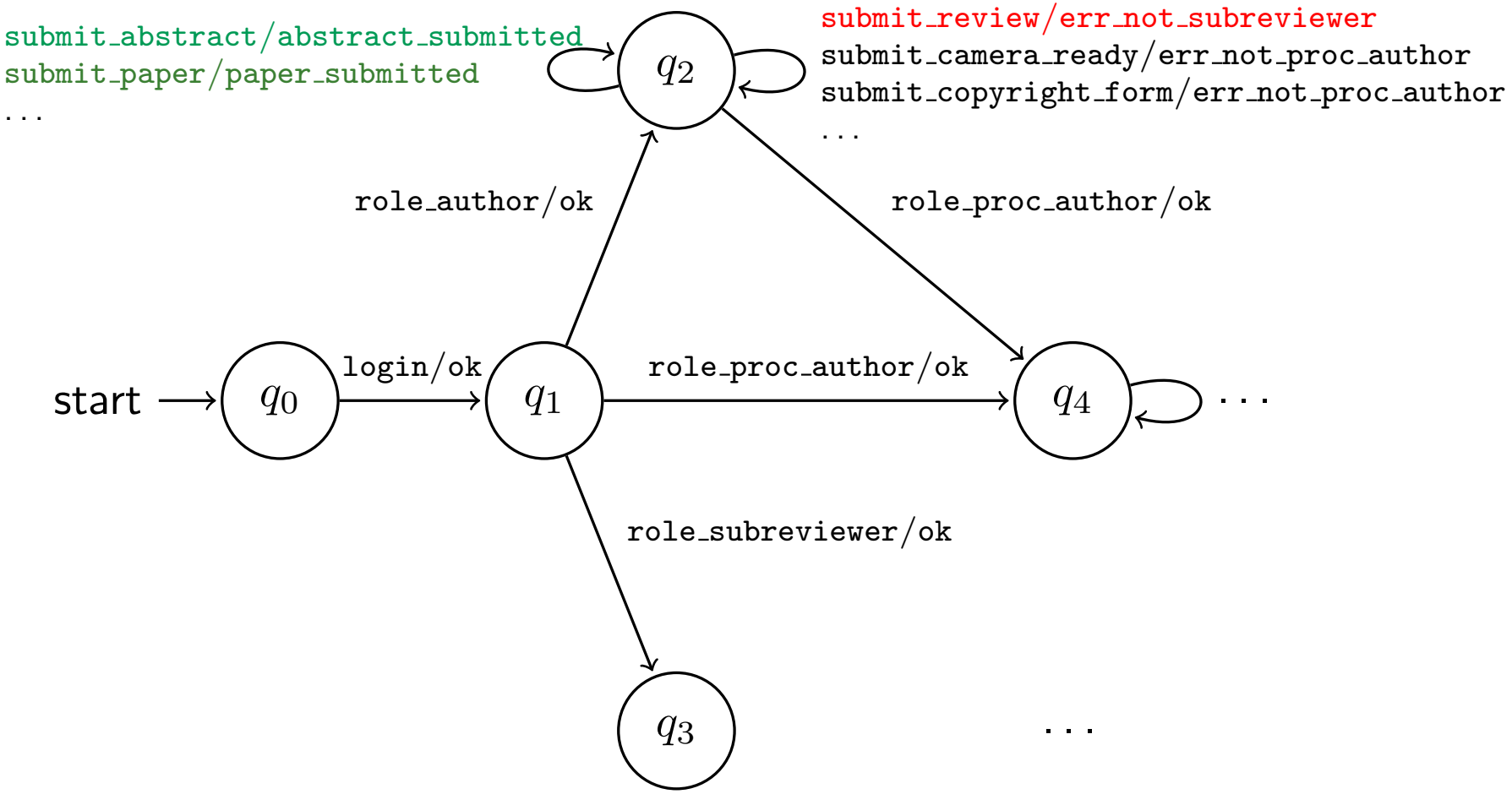
Example: A Conference Service



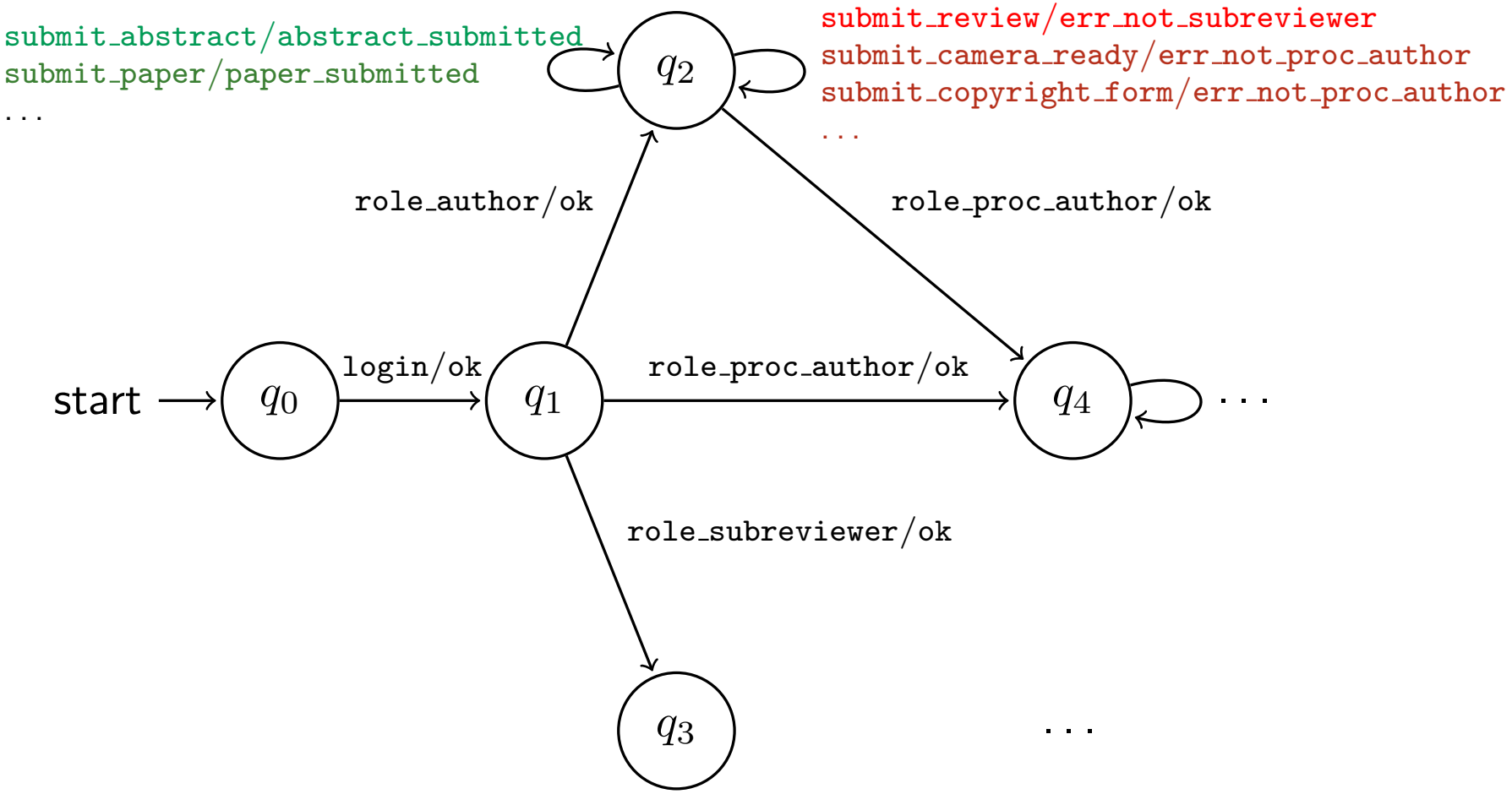
Example: A Conference Service



Example: A Conference Service

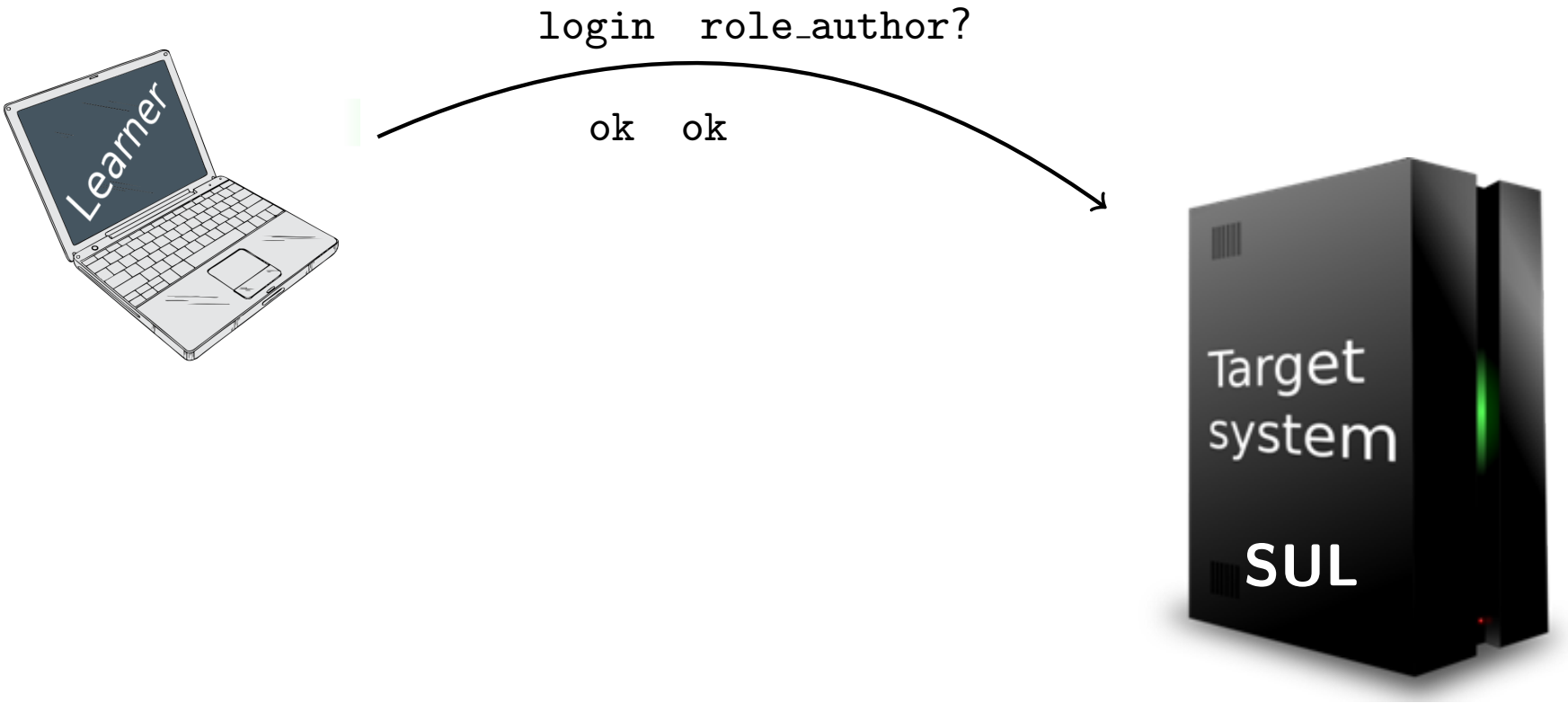


Example: A Conference Service

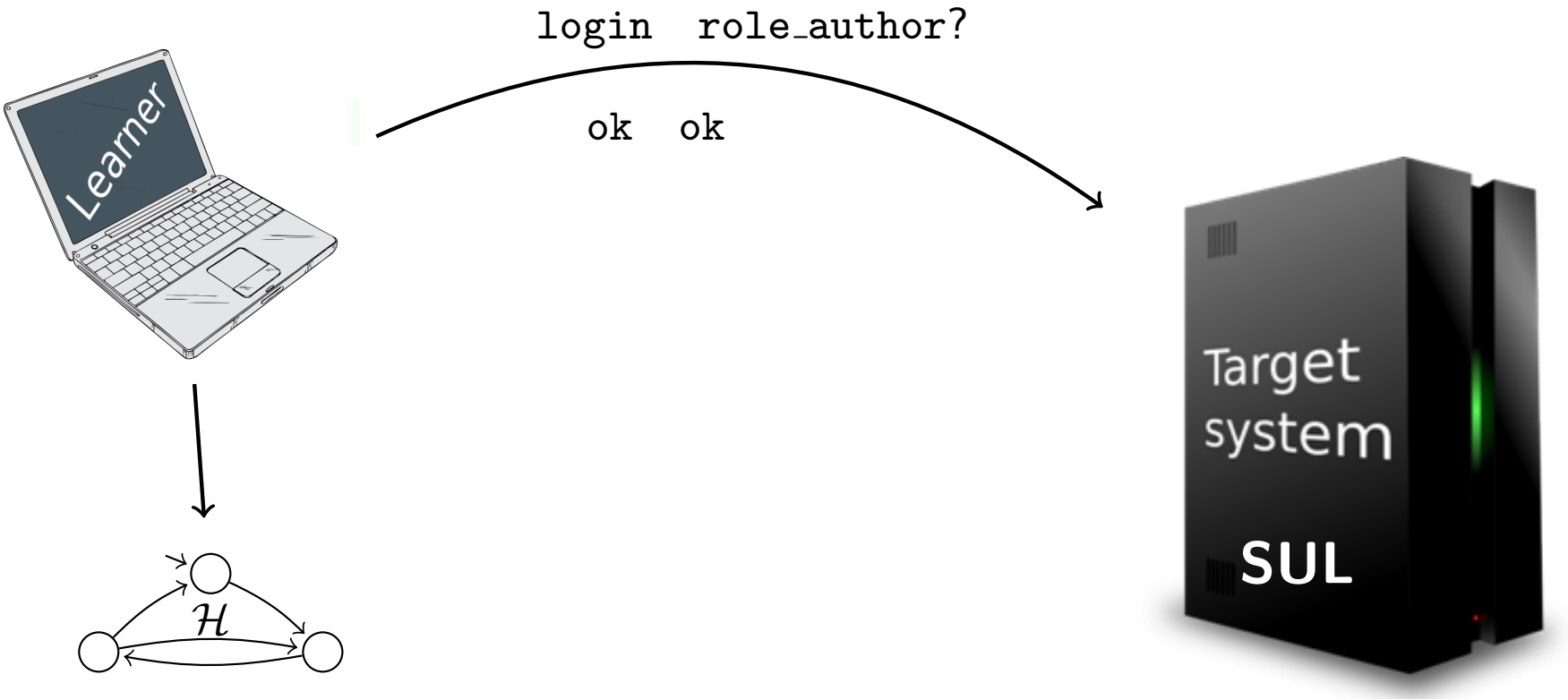


Active Automata Learning Illustrated

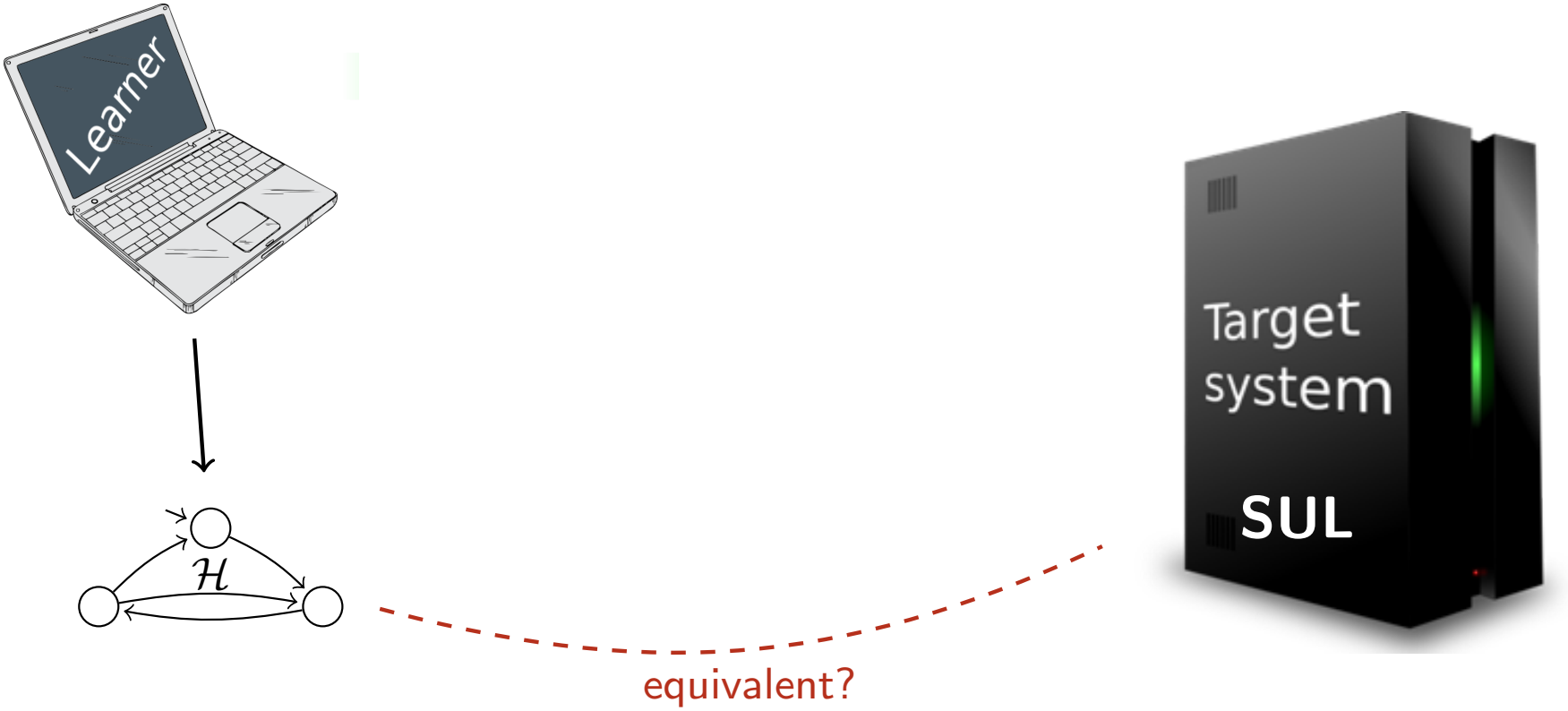
Active Automata Learning Illustrated



Active Automata Learning Illustrated

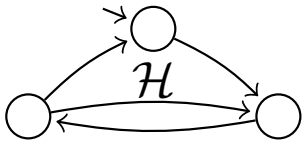


Active Automata Learning Illustrated



Active Automata Learning Illustrated

Yes: done

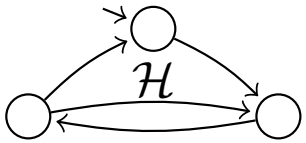


equivalent?

Active Automata Learning Illustrated

Yes: done

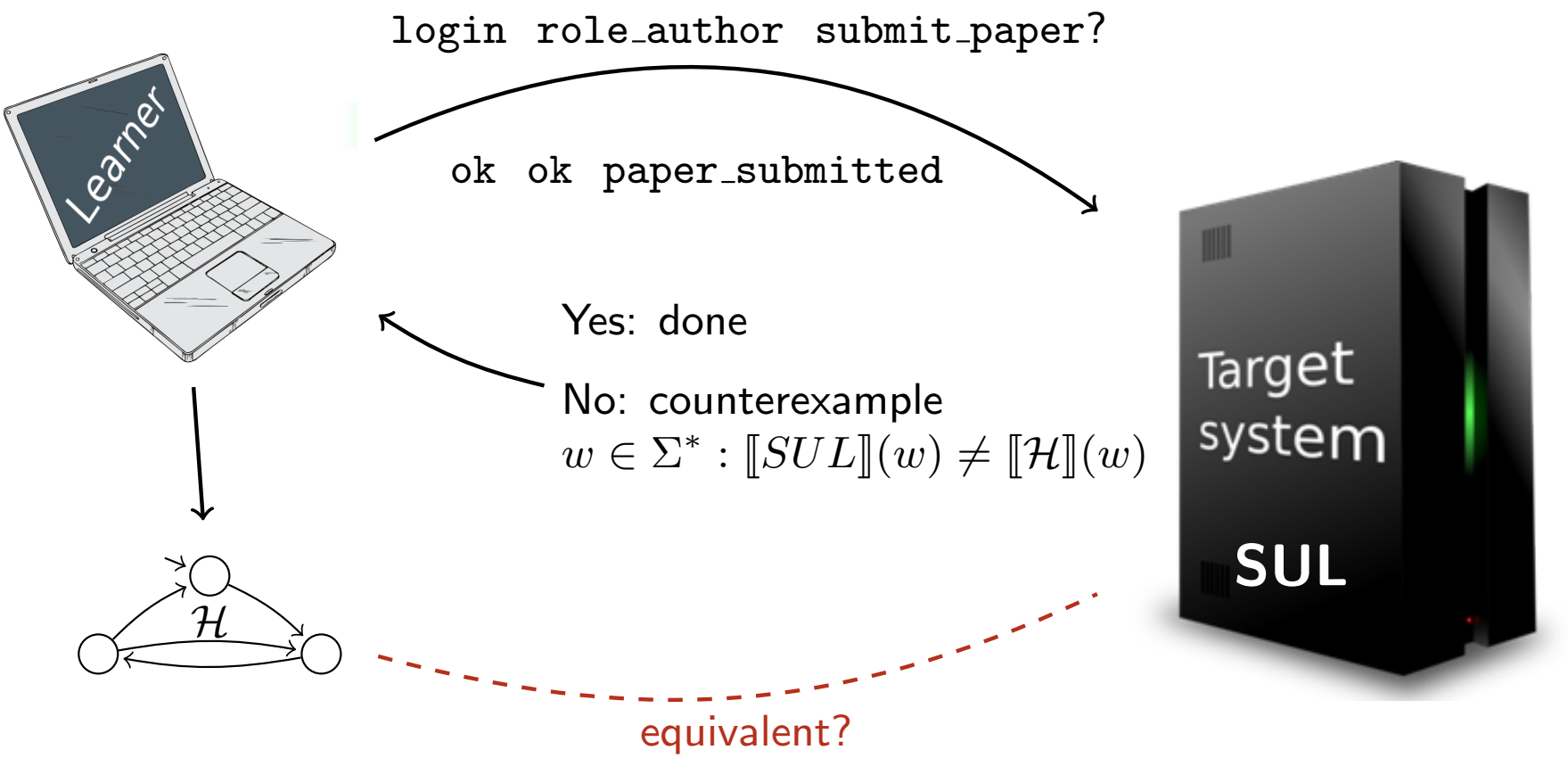
No: counterexample
$w \in \Sigma^* : [\![SUL]\!](w) \neq [\![\mathcal{H}]\!](w)$



equivalent?

Note: $[\![\cdot]\!]$ maps an **input** word to the respective system's **output**

Active Automata Learning Illustrated



Note: $[\![\cdot]\!]$ maps an **input** word to the respective system's **output**

Active Automata Learning Formalized

- Originally for learning regular sets (DFAs) [Angluin, 1987]; adapted to reactive systems (Mealy machines) [Niese, 2003]
- Input alphabet Σ, output alphabet Ω
- Interaction with "teacher" via
  - Membership Queries (MQs): What is the response to some input $w \in \Sigma^*$?
  - Equivalence Queries (EQs): Is the conjectured hypothesis correct?
- Formally: $MQ\colon \Sigma^* \to \Omega$, $EQ\colon S \to \Sigma^* \cup \{\text{yes}\}$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

- Rows:
  - Upper half: Short prefixes identify **states**

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)

The L^* Algorithm

- Central data structure: Observation Table

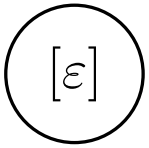
	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

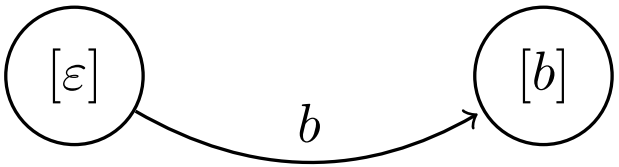


- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

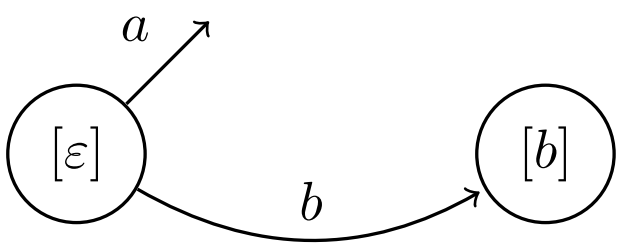


- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

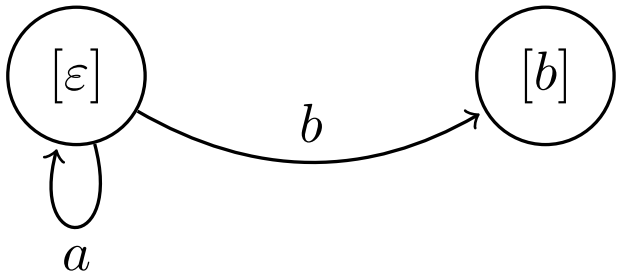


- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

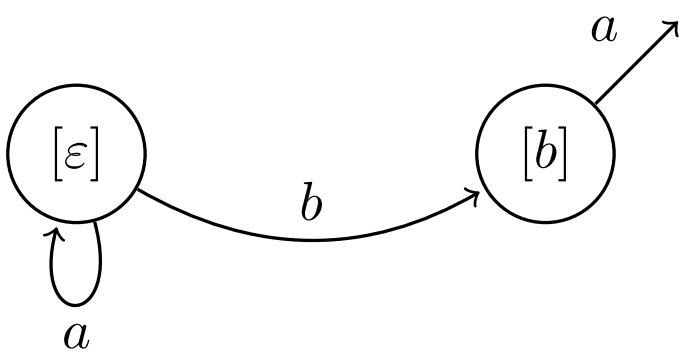


- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

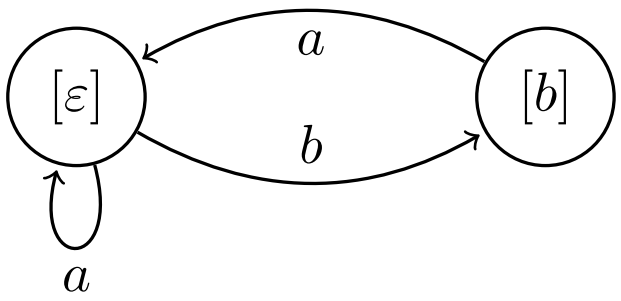


- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

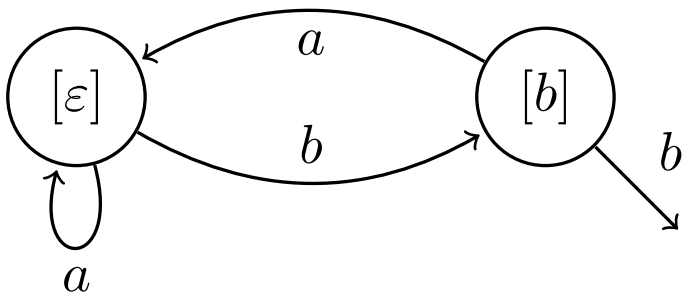


- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

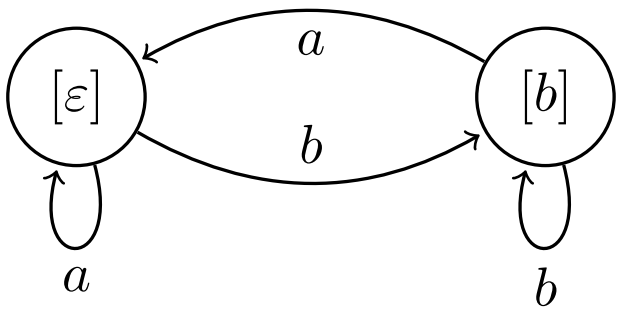


- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y

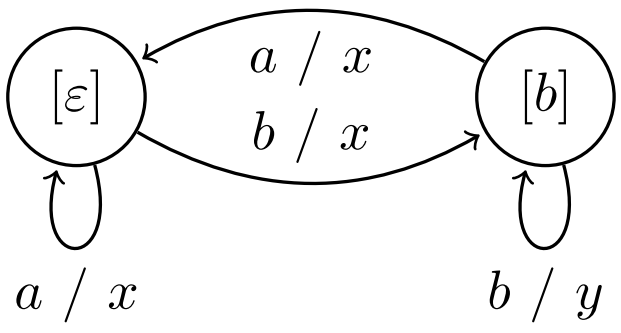


- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm

- Central data structure: Observation Table

	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y



- Rows:
  - Upper half: Short prefixes identify **states**
  - Lower half: Long prefixes identify **transitions** (one-letter extensions)
- Columns: Distinguishing suffixes
  - distinguish states (short prefixes)
  - match transition targets (long prefixes) with states (short prefixes)
- Cell contents (row u, col. v): $MQ(u \cdot v)$

The L^* Algorithm – Counterexamples

- Essential for defining transitions: Matching short prefix for each long prefix
  - Closed observation table

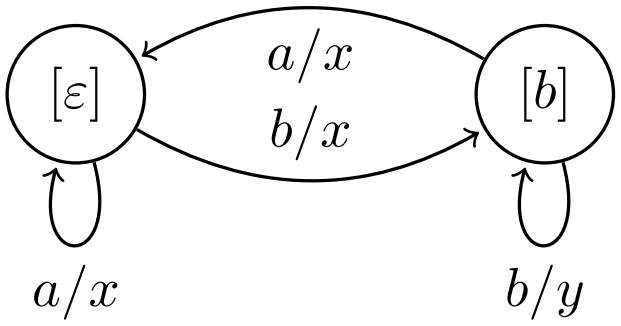
The L^* Algorithm – Counterexamples

- Essential for defining transitions: Matching short prefix for each long prefix
  - Closed observation table
- Counterexample: Word $w \in \Sigma^*$ such that $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$
- $\Rightarrow$ New distinguishing suffix v such that table is no longer closed

The L^* Algorithm – Counterexamples

- Essential for defining transitions: Matching short prefix for each long prefix
  - Closed observation table
- Counterexample: Word $w \in \Sigma^*$ such that $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$
- $\Rightarrow$ New distinguishing suffix v such that table is no longer closed

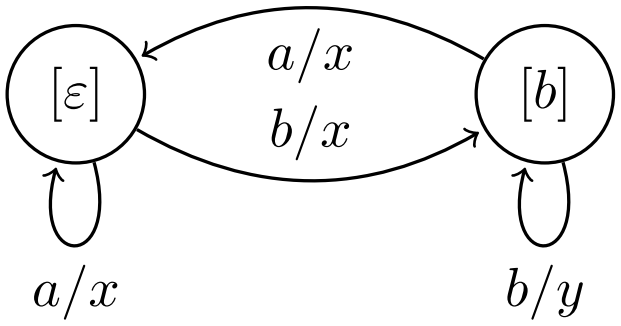
	a	b
ε	x	x
b	x	y
a	x	x
ba	x	x
bb	x	y



The L^* Algorithm – Counterexamples

- Essential for defining transitions: Matching short prefix for each long prefix
  - Closed observation table
- Counterexample: Word $w \in \Sigma^*$ such that $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$
- $\Rightarrow$ New distinguishing suffix v such that table is no longer closed

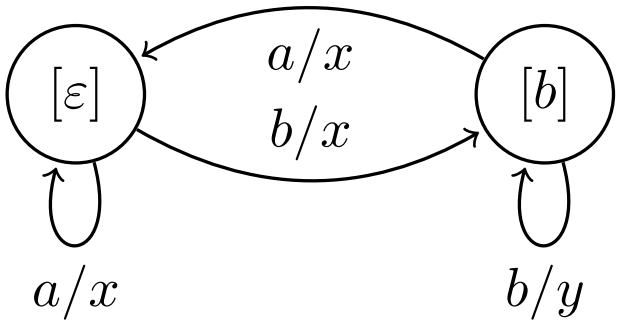
	a	b	ab
ε	x	x	x
b	x	y	x
a	x	x	x
ba	x	x	x
bb	x	y	y



The L^* Algorithm – Counterexamples

- Essential for defining transitions: Matching short prefix for each long prefix
  - Closed observation table
- Counterexample: Word $w \in \Sigma^*$ such that $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$
- $\Rightarrow$ New distinguishing suffix v such that table is no longer closed

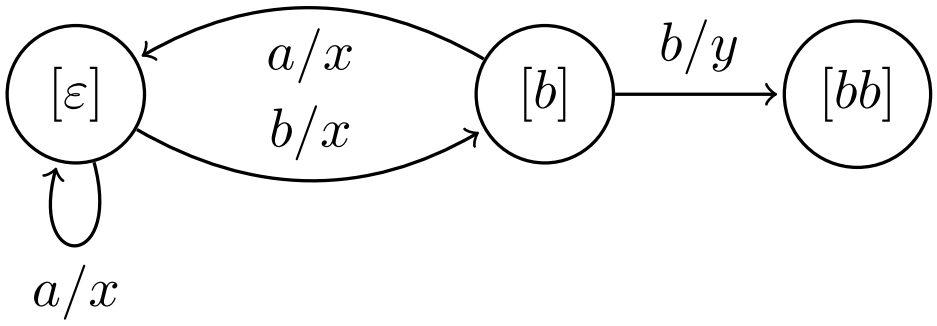
	a	b	ab
ε	x	x	x
b	x	y	x
a	x	x	x
ba	x	x	x
bb	x	y	y



The L^* Algorithm – Counterexamples

- Essential for defining transitions: Matching short prefix for each long prefix
  - Closed observation table
- Counterexample: Word $w \in \Sigma^*$ such that $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$
- $\Rightarrow$ New distinguishing suffix v such that table is no longer closed

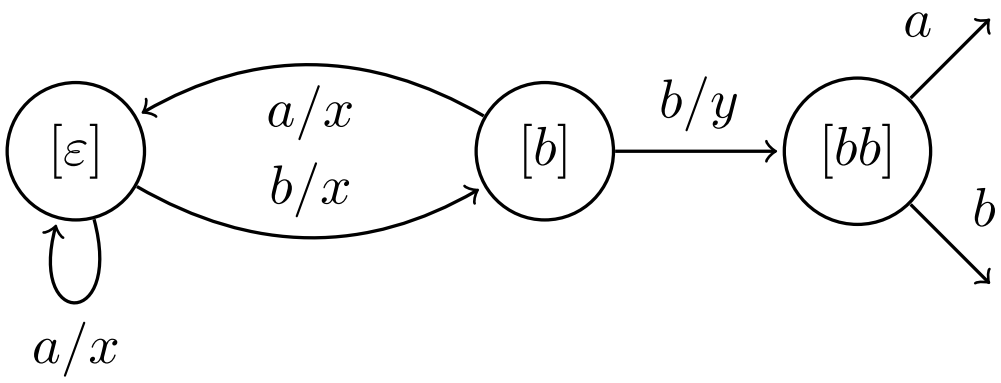
	a	b	ab
ε	x	x	x
b	x	y	x
bb	x	y	y
a	x	x	x
ba	x	x	x



The L^* Algorithm – Counterexamples

- Essential for defining transitions: Matching short prefix for each long prefix
  - Closed observation table
- Counterexample: Word $w \in \Sigma^*$ such that $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$
- $\Rightarrow$ New distinguishing suffix v such that table is no longer closed

	a	b	ab
ε	x	x	x
b	x	y	x
bb	x	y	y
a	x	x	x
ba	x	x	x
bba		$\dots$	
bbb		$\dots$	



Automata Learning in Practice

Automata Learning in Practice

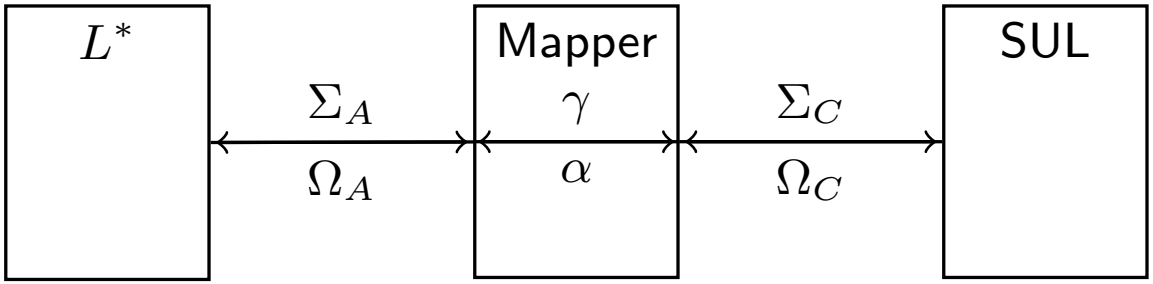
- Alphabet Σ too big (or even infinite) for many practical systems

Automata Learning in Practice

- Alphabet Σ too big (or even infinite) for many practical systems
- Solution: Alphabet abstraction
  - Concrete alphabet Σ_C and abstract alphabet Σ_A
  - **Mapper**: abstraction $\alpha \colon \Sigma_C \to \Sigma_A$, concretization $\gamma : \Sigma_A \to \Sigma_C$
  - Learner operates on Σ_A, mapper translates to Σ_C (and backwards)

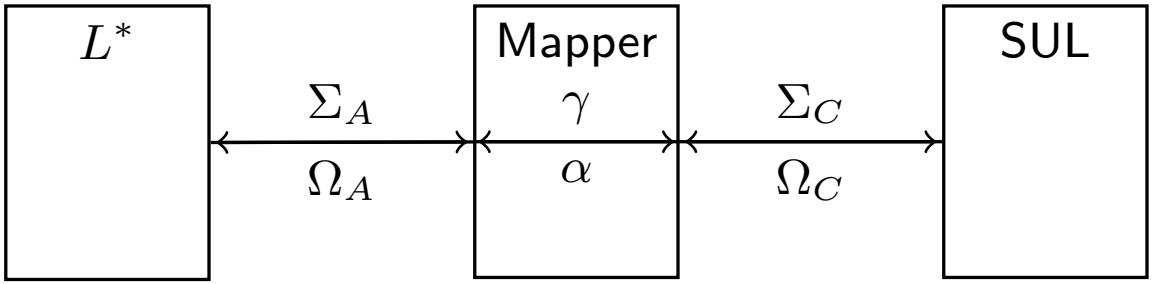
Automata Learning in Practice

- Alphabet Σ too big (or even infinite) for many practical systems
- Solution: Alphabet abstraction
  - Concrete alphabet Σ_C and abstract alphabet Σ_A
  - **Mapper**: abstraction $\alpha\colon \Sigma_C \to \Sigma_A$, concretization $\gamma : \Sigma_A \to \Sigma_C$
  - Learner operates on Σ_A, mapper translates to Σ_C (and backwards)



Automata Learning in Practice

- Alphabet Σ too big (or even infinite) for many practical systems
- Solution: Alphabet abstraction
  - Concrete alphabet Σ_C and abstract alphabet Σ_A
  - **Mapper**: abstraction $\alpha\colon \Sigma_C \to \Sigma_A$, concretization $\gamma : \Sigma_A \to \Sigma_C$
  - Learner operates on Σ_A, mapper translates to Σ_C (and backwards)



- Case-study: Inferring model of the EU biometric passport [Aarts et al., 2010]
  - Requires a-priori knowledge $\Rightarrow$ conflicts with black-box setting

Determinism-Preserving Abstractions

Common problems with manually tailored alphabet abstractions:

Determinism-Preserving Abstractions

Common problems with manually tailored alphabet abstractions:

- Abstraction too fine
  - blows up model, increases learning costs

Determinism-Preserving Abstractions

Common problems with manually tailored alphabet abstractions:

- Abstraction too fine
  - blows up model, increases learning costs
- Abstraction too coarse
  - Apparent non-determinism (though system itself is deterministic)

$$MQ\ (\ \texttt{submit_review}\ ) = \texttt{err_not_subreviewer}$$

Determinism-Preserving Abstractions

Common problems with manually tailored alphabet abstractions:

- Abstraction too fine
  - blows up model, increases learning costs
- Abstraction too coarse
  - Apparent non-determinism (though system itself is deterministic)

$$MQ\ (\ \texttt{submit_review}\ ) = \texttt{err_not_subreviewer}$$

$$\texttt{submit_review} \xrightarrow{\alpha} \text{invalidOp}$$

Determinism-Preserving Abstractions

Common problems with manually tailored alphabet abstractions:

- Abstraction too fine
  - blows up model, increases learning costs
- Abstraction too coarse
  - Apparent non-determinism (though system itself is deterministic)

$$
\begin{array}{ccccc}
MQ & (& \texttt{submit_review} &) & = & \texttt{err_not_subreviewer} \\
 & & \downarrow{\scriptstyle \alpha} & & & \\
 & & \text{invalidOp} & & & \\
 & & \downarrow{\scriptstyle \gamma} & & & \\
MQ & (& \texttt{submit_camera_ready} &) & = & \texttt{err_not_proc_author}
\end{array}
$$

Determinism-Preserving Abstractions

Common problems with manually tailored alphabet abstractions:

- Abstraction too fine
  - blows up model, increases learning costs
- Abstraction too coarse
  - Apparent non-determinism (though system itself is deterministic)

$$
\begin{array}{ccccc}
MQ & (& \texttt{submit_review} &) & = & \texttt{err_not_subreviewer} \\
 & & \downarrow \alpha & & & \\
 & & \text{invalidOp} & & & \nparallel \\
 & & \downarrow \gamma & & & \\
MQ & (& \texttt{submit_camera_ready} &) & = & \texttt{err_not_proc_author}
\end{array}
$$

Determinism-Preserving Abstractions

Common problems with manually tailored alphabet abstractions:

- Abstraction too fine
  - blows up model, increases learning costs
- Abstraction too coarse
  - Apparent non-determinism (though system itself is deterministic)

$$
\begin{array}{ccccc}
MQ & (& \texttt{submit_review} &) & = & \texttt{err_not_subreviewer} \\
 & & \downarrow{\scriptstyle \alpha} & & & \\
 & & \text{invalidOp} & & & \nparallel \\
 & & \downarrow{\scriptstyle \gamma} & & & \\
MQ & (& \texttt{submit_camera_ready} &) & = & \texttt{err_not_proc_author}
\end{array}
$$

$\Rightarrow$ Abstraction not determinism-preserving

Determinism-Preserving Abstractions

Common problems with manually tailored alphabet abstractions:

- Abstraction too fine
  - blows up model, increases learning costs
- Abstraction too coarse
  - Apparent non-determinism (though system itself is deterministic)

$$
\begin{array}{ccccc}
MQ & (& \texttt{submit_review} &) & = & \texttt{err_not_subreviewer} \\
 & & \downarrow \alpha & & & \\
 & & \text{invalidOp} & & & \nparallel \\
 & & \downarrow \gamma & & & \\
MQ & (& \texttt{submit_camera_ready} &) & = & \texttt{err_not_proc_author}
\end{array}
$$

  $\Rightarrow$ Abstraction not determinism-preserving
- Goal: Find coarsest determinism-preserving abstraction (DPA)

Automated Alphabet Abstraction Refinement

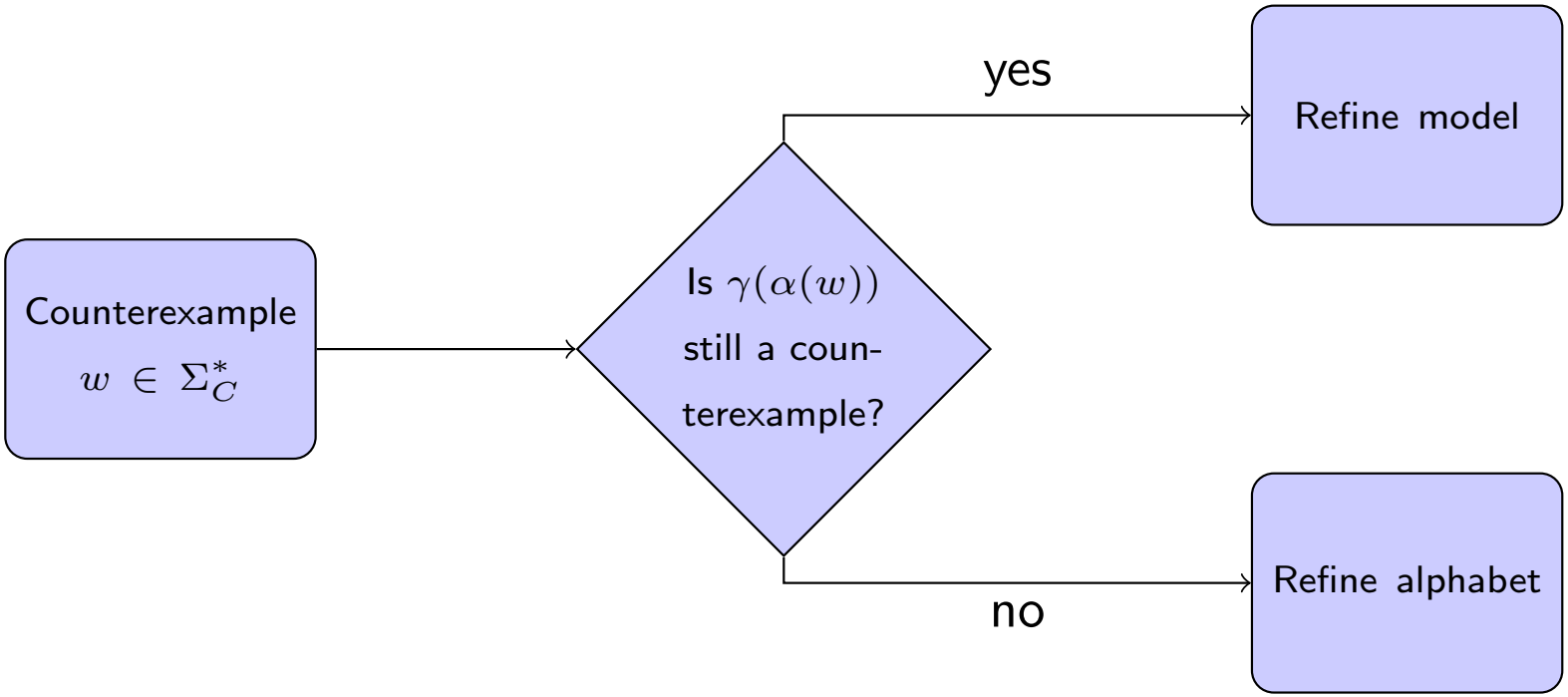
- Main idea: Augment black-box learning to also learn alphabet abstraction [Howar et al., 2011]

Automated Alphabet Abstraction Refinement

- Main idea: Augment black-box learning to also learn alphabet abstraction [Howar et al., 2011]
- Use counterexamples (EQs) for:
  - alphabet abstraction refinement in case of apparent non-determinism
  - model refinement otherwise

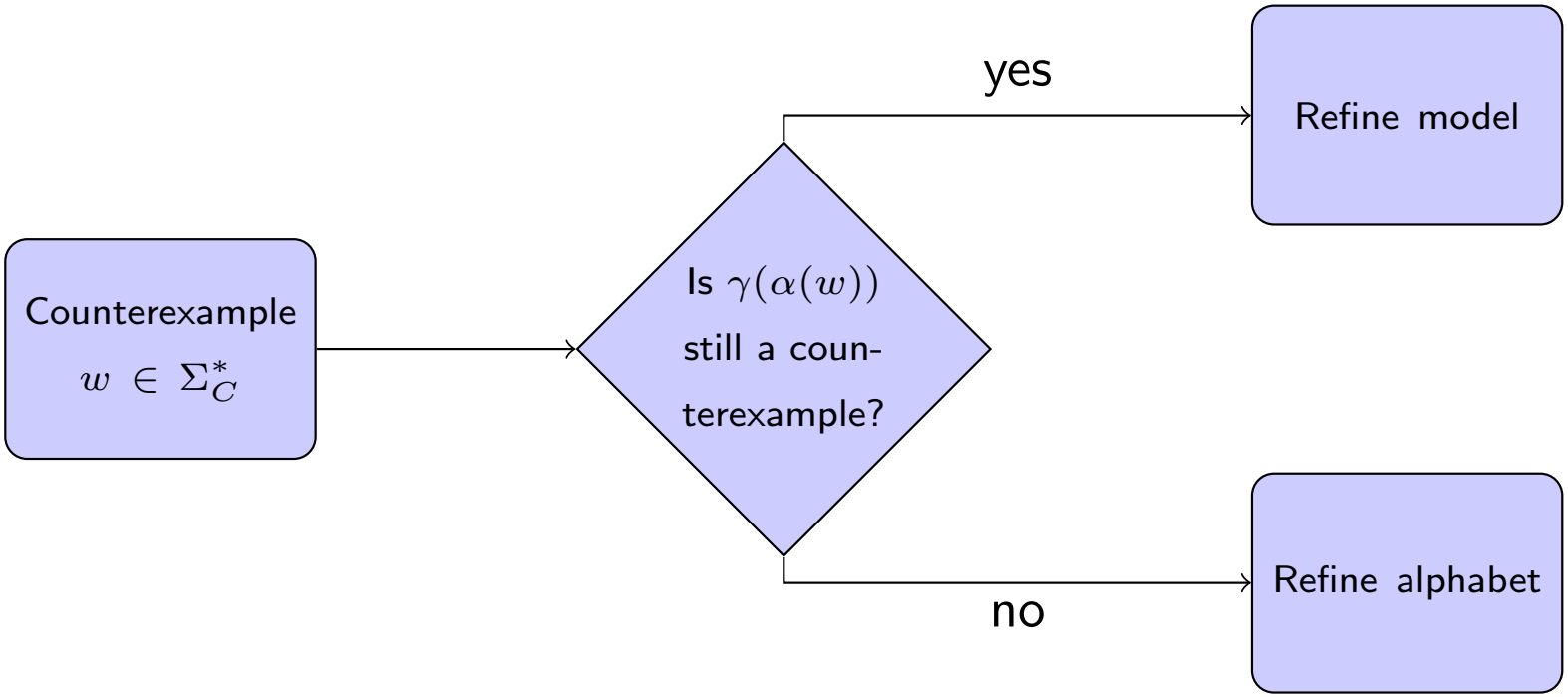
Automated Alphabet Abstraction Refinement

- Main idea: Augment black-box learning to also learn alphabet abstraction [Howar et al., 2011]
- Use counterexamples (EQs) for:
  - alphabet abstraction refinement in case of apparent non-determinism
  - model refinement otherwise



Automated Alphabet Abstraction Refinement

- Main idea: Augment black-box learning to also learn alphabet abstraction [Howar et al., 2011]
- Use counterexamples (EQs) for:
  - alphabet abstraction refinement in case of apparent non-determinism
  - model refinement otherwise



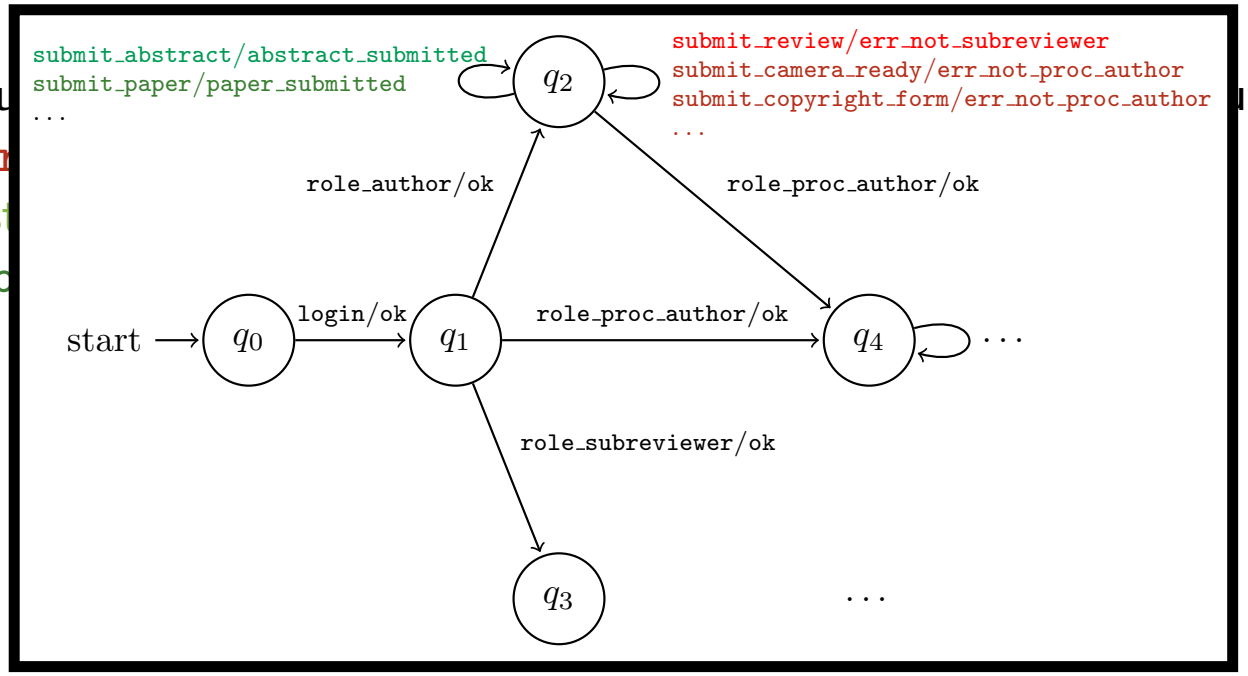
- Terminates with globally coarsest DPA

Local Alphabet Abstraction Refinement

- Problem: Nature of abstraction not captured adequately at global level
  - "author" role actions can be grouped in non-author state (output `err_not_author`)
  - distinguishable in author state (`abstract_submitted`, `paper_submitted`, ...)

Local Alphabet Abstraction Refinement

- Problem: Nature of abstraction not captured adequately at global level
  - "au … utput
    err
  - dist
    (ab



Local Alphabet Abstraction Refinement

- Problem: Nature of abstraction not captured adequately at global level
  - "author" role actions can be grouped in non-author state (output `err_not_author`)
  - distinguishable in author state (`abstract_submitted`, `paper_submitted`, ...)

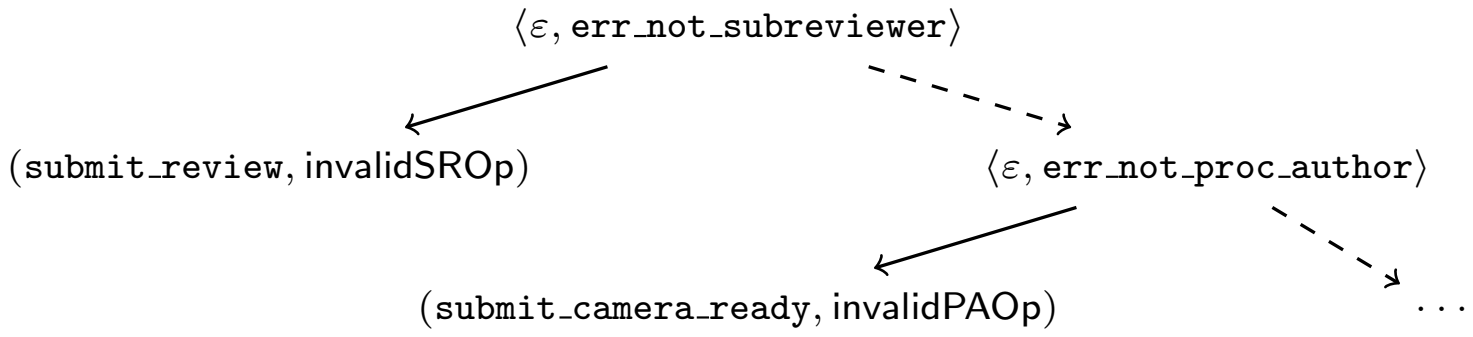
- Idea: Abstraction at a state-local level (this paper)
- Challenges:
  - (How to distinguish states w/o a global (abstract) alphabet?)
  - How to organize per-state black-box alphabet abstractions?
  - How to refine models and/or abstractions from counterexamples?

Organizing Abstractions – Abstraction Trees

- (Strict) black-box setting: SUL stimulus/reaction only source of information (also for alphabet abstraction)
- Abstraction trees: decision trees based on membership query outcomes (one for each state)

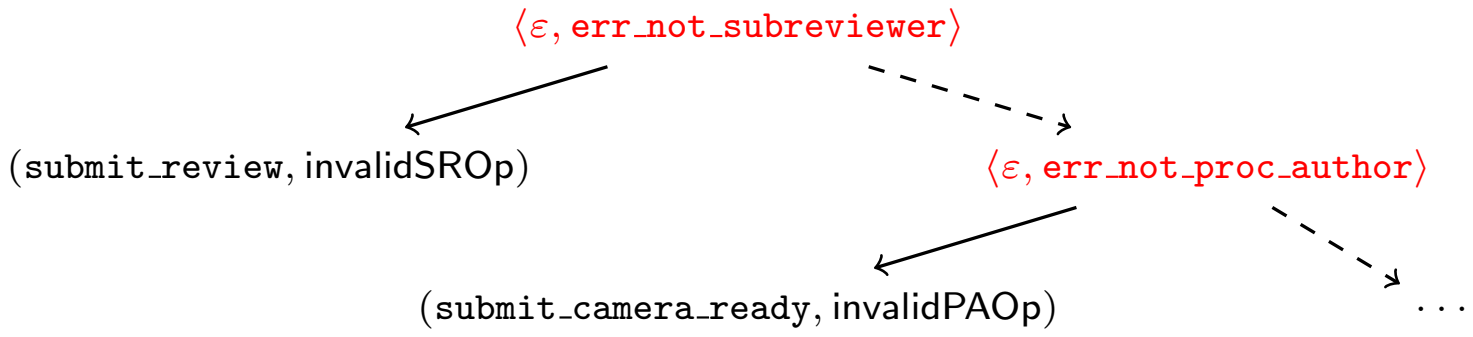
Organizing Abstractions – Abstraction Trees

- (Strict) black-box setting: SUL stimulus/reaction only source of information (also for alphabet abstraction)
- Abstraction trees: decision trees based on membership query outcomes (one for each state)
- $\mathcal{T}_{\texttt{login role_author}}$:



Organizing Abstractions – Abstraction Trees

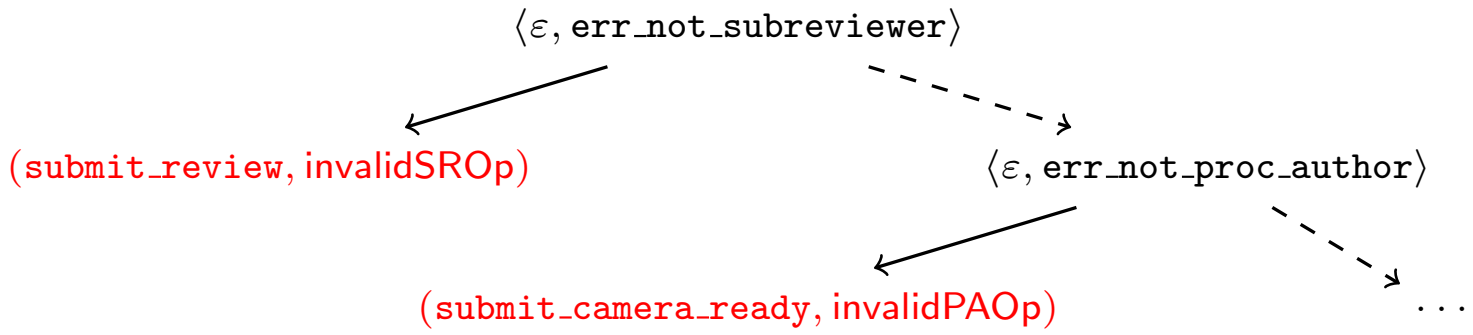
- (Strict) black-box setting: SUL stimulus/reaction only source of information (also for alphabet abstraction)
- Abstraction trees: decision trees based on membership query outcomes (one for each state)
- $\mathcal{T}_{\texttt{login role_author}}$:



- Inner nodes (decision nodes): classifiers; suffix v, output o

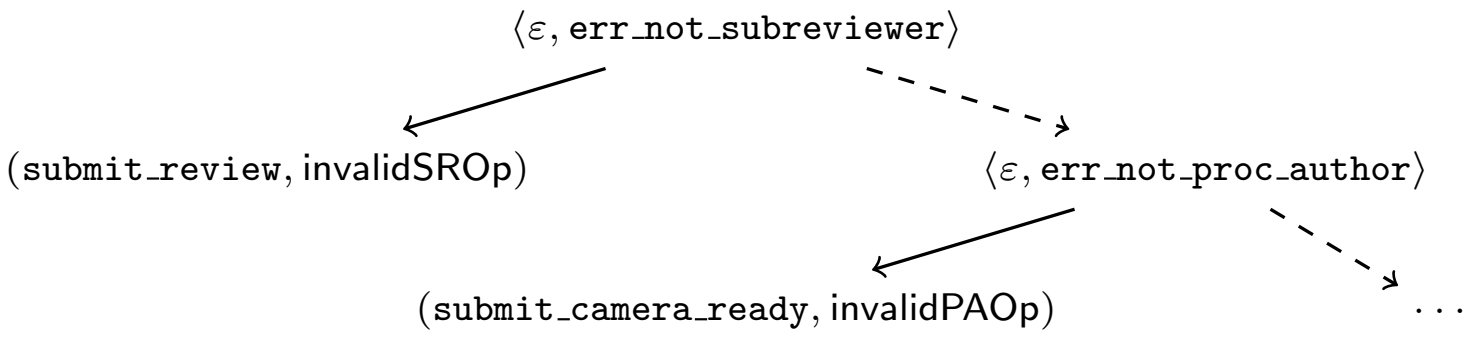
Organizing Abstractions – Abstraction Trees

- (Strict) black-box setting: SUL stimulus/reaction only source of information (also for alphabet abstraction)
- Abstraction trees: decision trees based on membership query outcomes (one for each state)
- $\mathcal{T}_{\texttt{login role_author}}$:



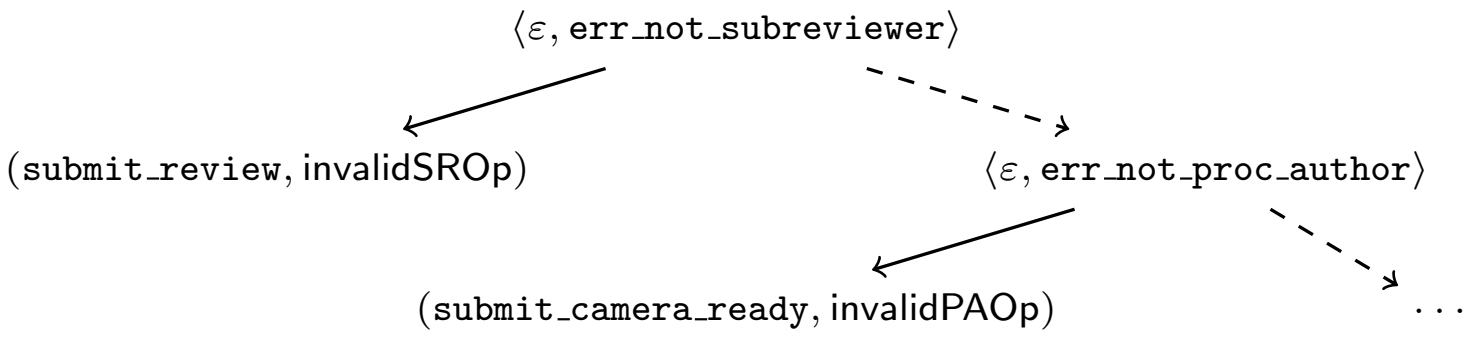
- Inner nodes (decision nodes): classifiers; suffix v, output o
- Leaves (symbol nodes): (abstract) alphabet symbols; abstract symbol, concrete representative

Abstraction Trees



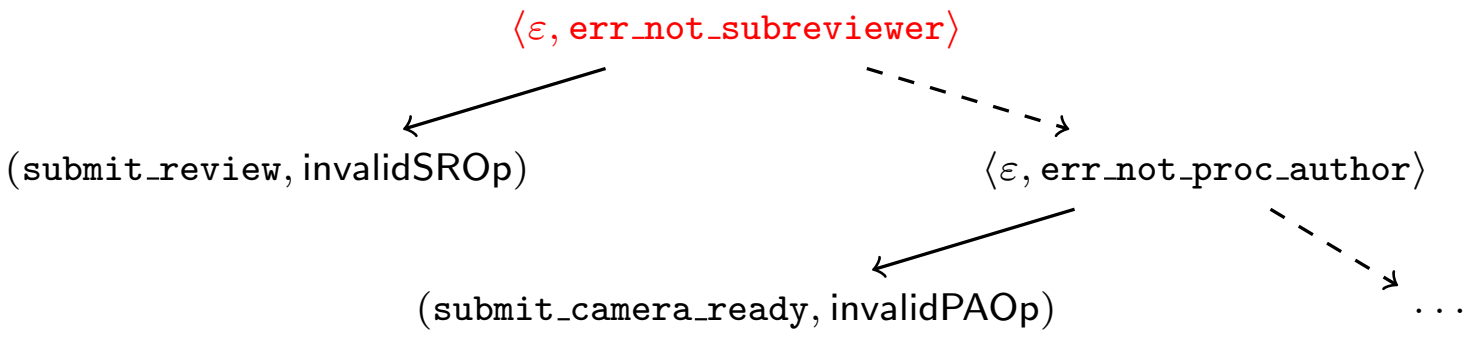
- Concrete symbol a_C, short prefix u: check $MQ(u \cdot a_C \cdot v)$ at decision nodes

Abstraction Trees



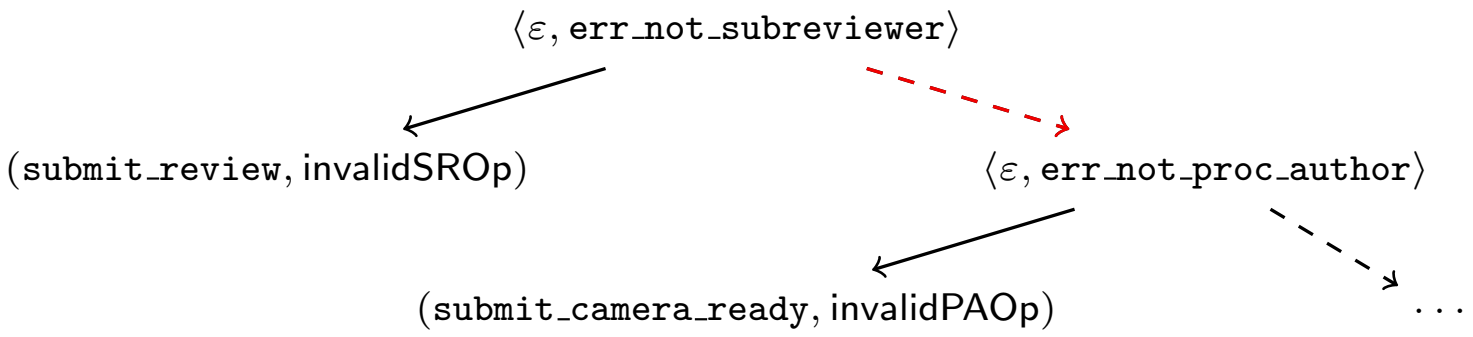
- Concrete symbol a_C, short prefix u: check $MQ(u \cdot a_C \cdot v)$ at decision nodes
- $a_C = \texttt{submit_copyright_form}, u = \texttt{login role_author}$:

Abstraction Trees



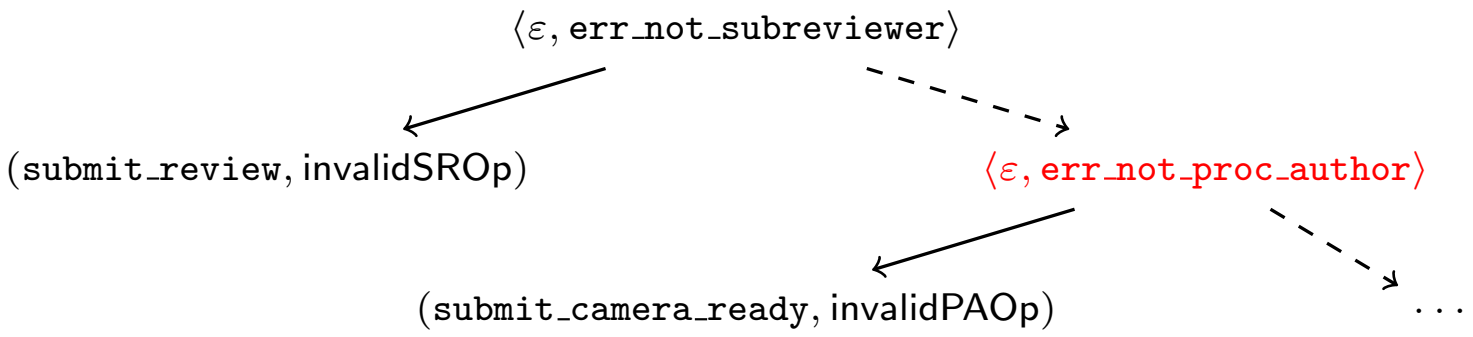
- Concrete symbol a_C, short prefix u: check $MQ(u \cdot a_C \cdot v)$ at decision nodes
- $a_C = \texttt{submit_copyright_form}, u = \texttt{login role_author}$:
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$

Abstraction Trees



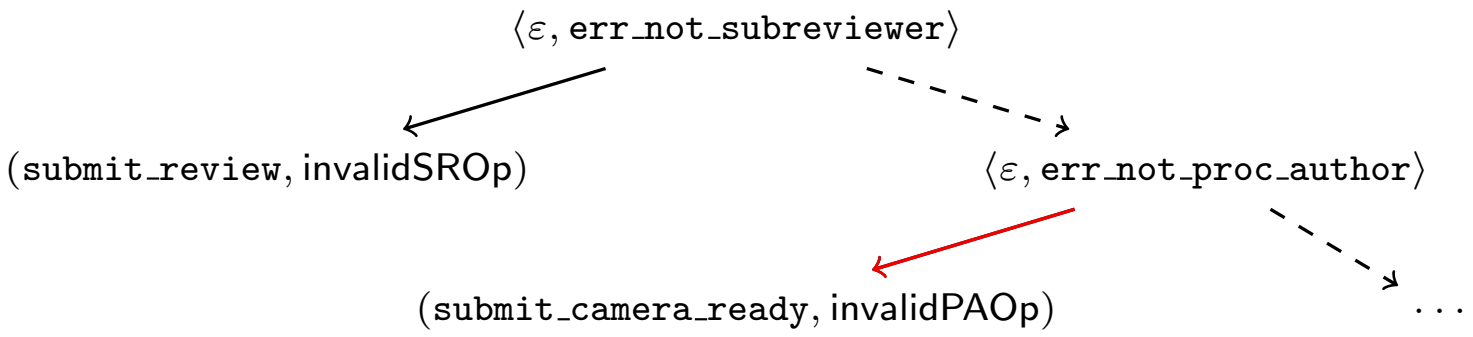
- Concrete symbol a_C, short prefix u: check $MQ(u \cdot a_C \cdot v)$ at decision nodes
- $a_C = \texttt{submit_copyright_form}, u = \texttt{login role_author}$:
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$
    $= \texttt{err_not_proc_author} \neq \texttt{err_not_subreviewer}$

Abstraction Trees



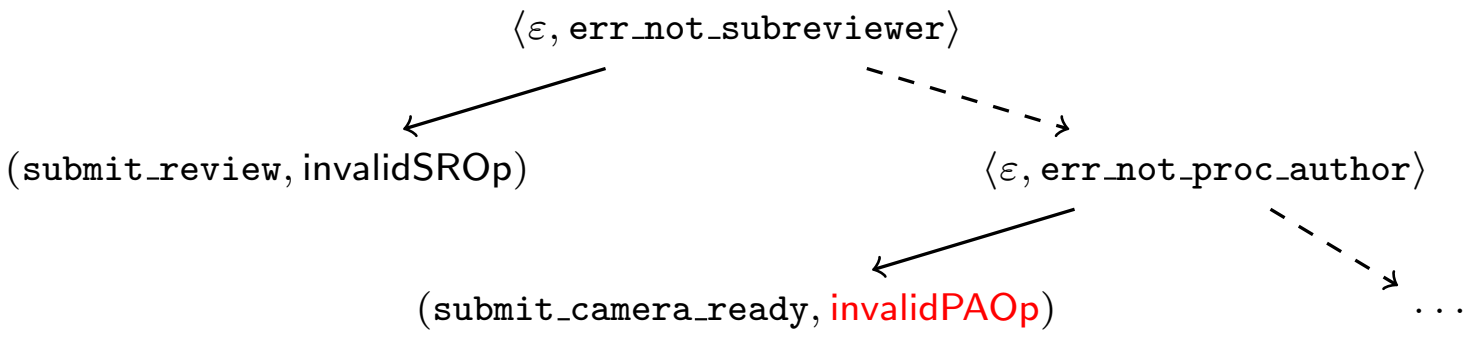
- Concrete symbol a_C, short prefix u: check $MQ(u \cdot a_C \cdot v)$ at decision nodes
- $a_C = \texttt{submit_copyright_form}, u = \texttt{login role_author}$:
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$
    $= \texttt{err_not_proc_author} \neq \texttt{err_not_subreviewer}$
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$

Abstraction Trees



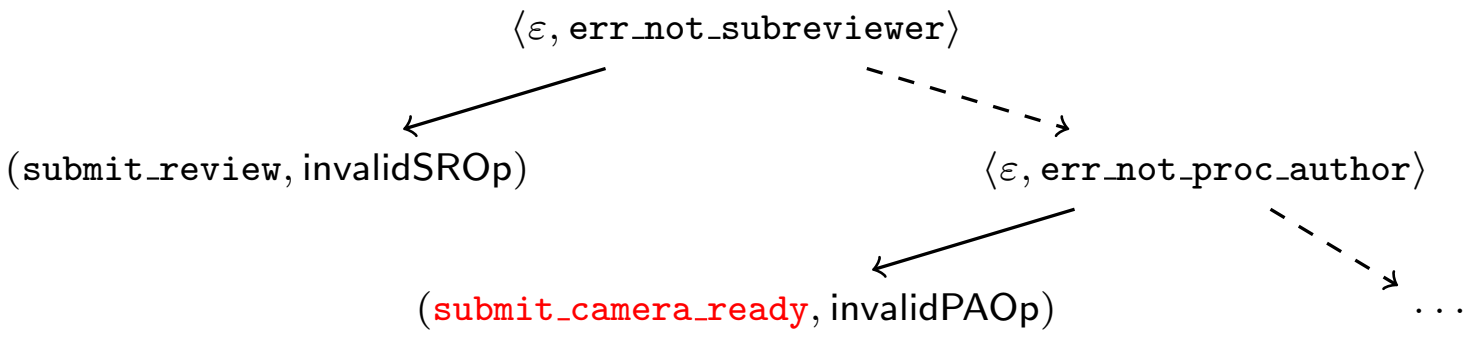
- Concrete symbol a_C, short prefix u: check $MQ(u \cdot a_C \cdot v)$ at decision nodes
- $a_C = \texttt{submit_copyright_form}, u = \texttt{login role_author}$:
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$ $= \texttt{err_not_proc_author} \neq \texttt{err_not_subreviewer}$
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$ $= \texttt{err_not_proc_author}$

Abstraction Trees



- Concrete symbol a_C, short prefix u: check $MQ(u \cdot a_C \cdot v)$ at decision nodes
- $a_C = \texttt{submit_copyright_form}, u = \texttt{login role_author}$:
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$ $= \texttt{err_not_proc_author} \neq \texttt{err_not_subreviewer}$
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$ $= \texttt{err_not_proc_author}$
  - Abstract symbol invalidPAOp, representative `submit_camera_ready`

Abstraction Trees



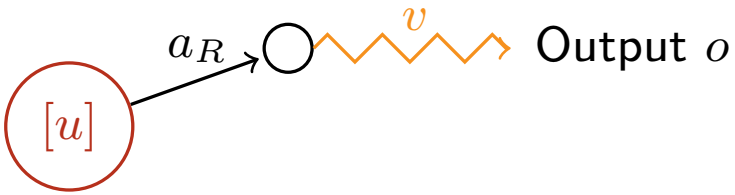
- Concrete symbol a_C, short prefix u: check $MQ(u \cdot a_C \cdot v)$ at decision nodes
- $a_C = \texttt{submit_copyright_form}, u = \texttt{login role_author}$:
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$ $= \texttt{err_not_proc_author} \neq \texttt{err_not_subreviewer}$
  - $MQ(\texttt{login role_author} \cdot \texttt{submit_copyright_form} \cdot \varepsilon)$ $= \texttt{err_not_proc_author}$
  - Abstract symbol invalidPAOp, representative submit_camera_ready

Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior

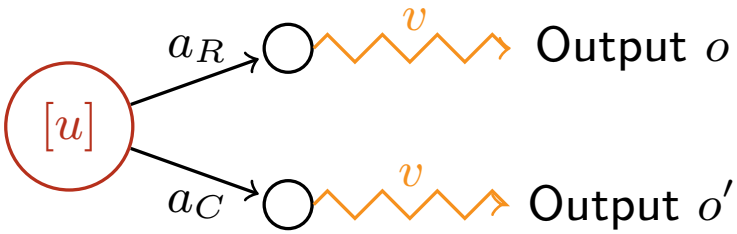
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



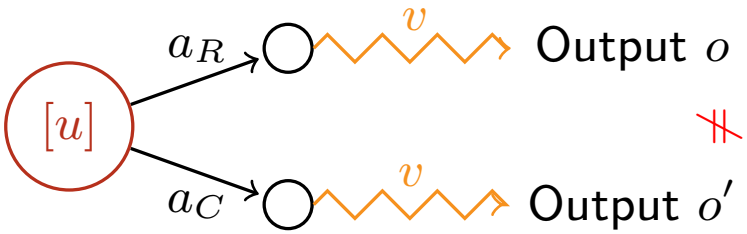
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



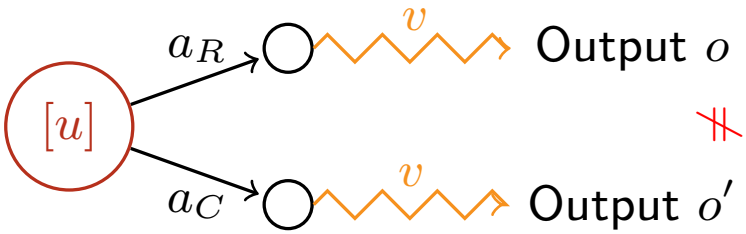
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



Abstraction Refinement

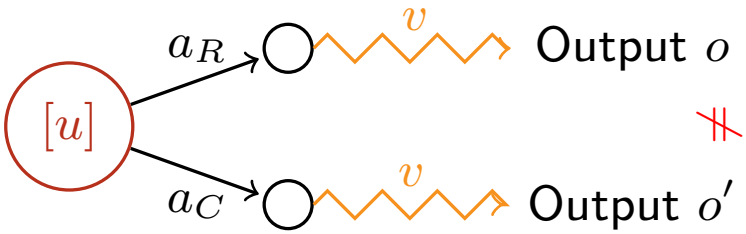
- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



$\not\parallel \Rightarrow$ split a_R in $\mathcal{T}_u$ using $\langle v, o \rangle$

Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v

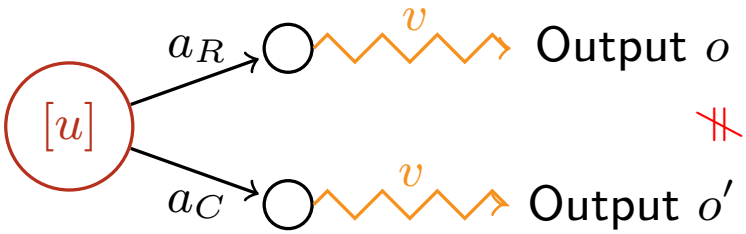


$\not\parallel \Rightarrow$ split a_R in $\mathcal{T}_u$ using $\langle v, o \rangle$

  - Test for $\boxed{MQ(u \cdot a_C \cdot v) \neq MQ(u \cdot a_R \cdot v)}$

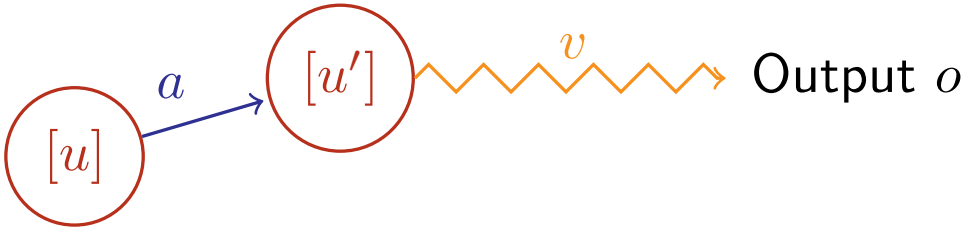
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



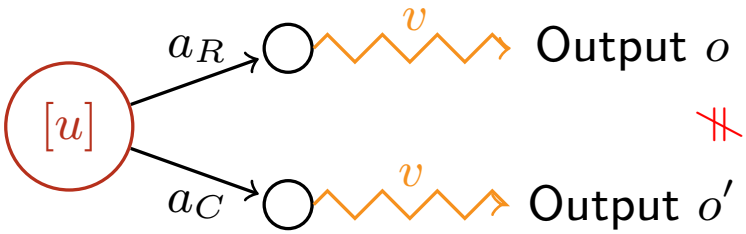
$\not\parallel \Rightarrow$ split a_R in $\mathcal{T}_u$ using $\langle v, o \rangle$

  - Test for $\boxed{MQ(u \cdot a_C \cdot v) \neq MQ(u \cdot a_R \cdot v)}$
- Cf. State-space refinement:



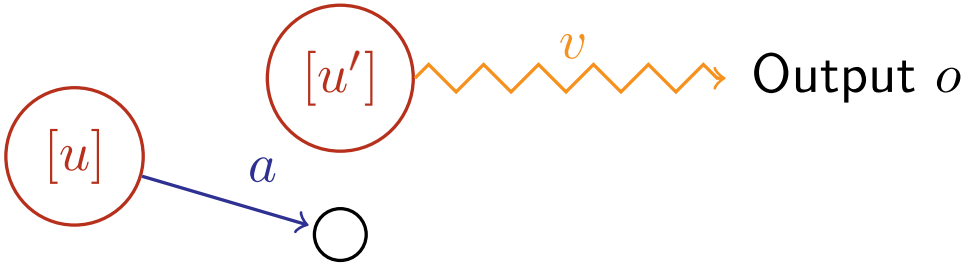
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



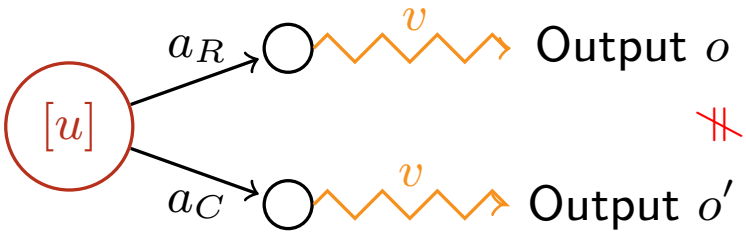
$\not\parallel \Rightarrow$ split a_R in $\mathcal{T}_u$ using $\langle v, o \rangle$

  - Test for $\boxed{MQ(u \cdot a_C \cdot v) \neq MQ(u \cdot a_R \cdot v)}$
- Cf. State-space refinement:



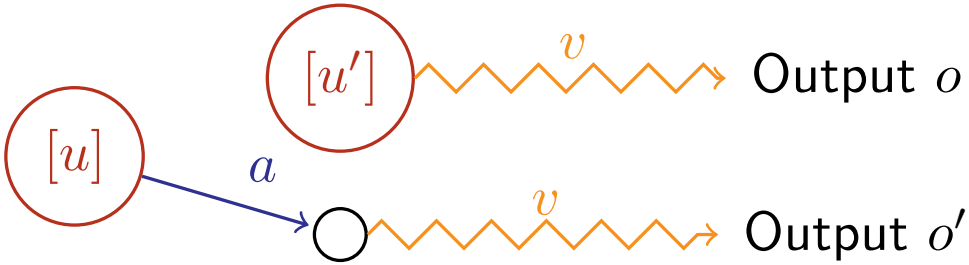
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



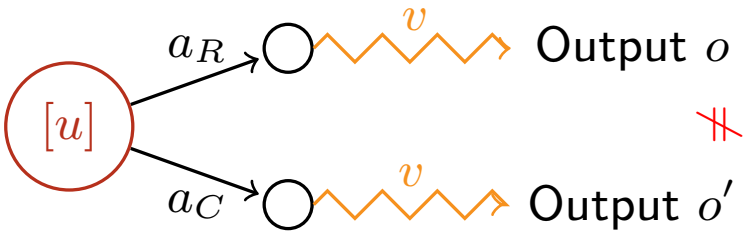
$\not\parallel \Rightarrow$ split a_R in $\mathcal{T}_u$ using $\langle v, o \rangle$

  - Test for $\boxed{MQ(u \cdot a_C \cdot v) \neq MQ(u \cdot a_R \cdot v)}$
- Cf. State-space refinement:



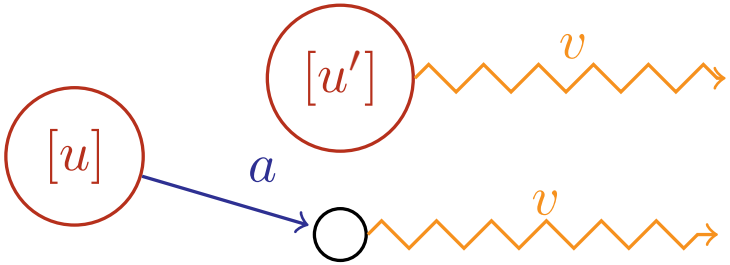
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



$\not\parallel \Rightarrow$ split a_R in $\mathcal{T}_u$ using $\langle v, o \rangle$

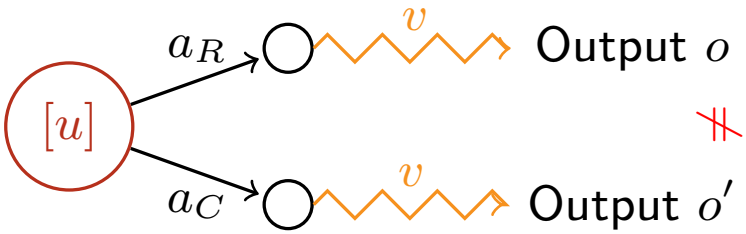
  - Test for $\boxed{MQ(u \cdot a_C \cdot v) \neq MQ(u \cdot a_R \cdot v)}$
- Cf. State-space refinement:



$\not\parallel \Rightarrow$ add v as suffix

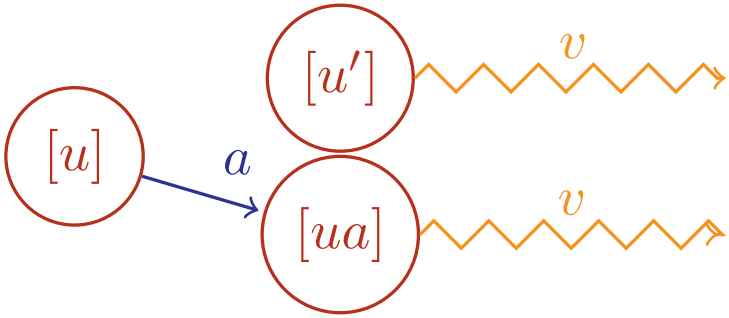
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



$\not\parallel \Rightarrow$ split a_R in $\mathcal{T}_u$ using $\langle v, o \rangle$

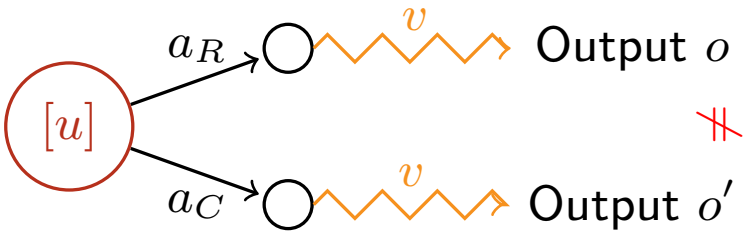
  - Test for $\boxed{MQ(u \cdot a_C \cdot v) \neq MQ(u \cdot a_R \cdot v)}$
- Cf. State-space refinement:



$\not\parallel \Rightarrow$ add v as suffix

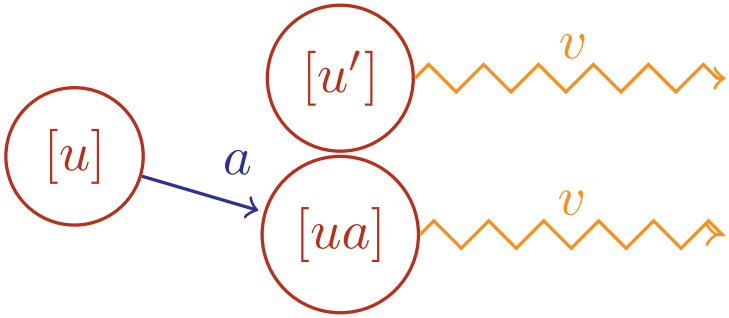
Abstraction Refinement

- Refine alphabet when current representative $a_R = \gamma(\alpha(a_C))$ has deviating future behavior
  - Future behavior defined in terms of suffix v



$\not\parallel \Rightarrow$ split a_R in $\mathcal{T}_u$ using $\langle v, o \rangle$

  - Test for $\boxed{MQ(u \cdot a_C \cdot v) \neq MQ(u \cdot a_R \cdot v)}$
- Cf. State-space refinement:



$\not\parallel \Rightarrow$ add v as suffix

  - Test for $\boxed{MQ(u \cdot a \cdot v) \neq MQ(u' \cdot v)}$

Experiments and Evaluation

- Comparison: Classical L_M^*, Global AAR, Local AAR

Experiments and Evaluation

- Comparison: Classical L^*_M, Global AAR, Local AAR
- Hierarchical systems:

Example	L^*_M			Global AAR			Local AAR		
	# MQ	# EQ	Time	# MQ	# EQ	Time	# MQ	# EQ	Time
CCC	320,200	1	51s	10,953	**201**	5m55s	2,600	204	22s
FHN1	190,122	3	28s	39,456	**93**	39s	3,798	99	4s
FHN2	7,391,946	9	58m0s	1,766,688	**354**	6h50m16s	50,371	417	7m30s

Experiments and Evaluation

- Comparison: Classical L^*_M, Global AAR, Local AAR
- Hierarchical systems:

Example	L^*_M			Global AAR			Local AAR		
	# MQ	# EQ	Time	# MQ	# EQ	Time	# MQ	# EQ	Time
CCC	320,200	1	51s	10,953	**201**	5m55s	2,600	204	22s
FHN1	190,122	3	28s	39,456	**93**	39s	3,798	99	4s
FHN2	7,391,946	9	58m0s	1,766,688	**354**	6h50m16s	50,371	417	7m30s

- Global abstraction 2 to 6 times bigger than biggest local

Experiments and Evaluation

- Comparison: Classical L_M^*, Global AAR, Local AAR
- Hierarchical systems:

Example	L_M^*			Global AAR			Local AAR		
	# MQ	# EQ	Time	# MQ	# EQ	Time	# MQ	# EQ	Time
CCC	320,200	1	51s	10,953	**201**	5m55s	2,600	204	22s
FHN1	190,122	3	28s	39,456	**93**	39s	3,798	99	4s
FHN2	7,391,946	9	58m0s	1,766,688	**354**	6h50m16s	50,371	417	7m30s

- Global abstraction 2 to 6 times bigger than biggest local
- Bitvector projection systems:

Example	L_M^*			Global AAR			Local AAR		
	# MQ	# EQ	Time	# MQ	# EQ	Time	# MQ	# EQ	Time
BV7	1,269,564	14	4m27s	175,458	146	2m00s	10,926	91	26s
BV8	5,994,280	24	25m51s	677,176	287	16m49s	25,168	116	1m43s
BV9	19,370,515	15	2h00m09s	890,475	535	53m48s	39,507	99	2m54s
BV10	80,473,959	26	13h19m02s	2,795,183	1060	8h57m32s	80,455	111	10m13s

Experiments and Evaluation

- Comparison: Classical L^*_M, Global AAR, Local AAR
- Hierarchical systems:

Example	L^*_M			Global AAR			Local AAR		
	# MQ	# EQ	Time	# MQ	# EQ	Time	# MQ	# EQ	Time
CCC	320,200	1	51s	10,953	**201**	5m55s	2,600	204	22s
FHN1	190,122	3	28s	39,456	**93**	39s	3,798	99	4s
FHN2	7,391,946	9	58m0s	1,766,688	**354**	6h50m16s	50,371	417	7m30s

- Global abstraction 2 to 6 times bigger than biggest local
- Bitvector projection systems:

Example	L^*_M			Global AAR			Local AAR		
	# MQ	# EQ	Time	# MQ	# EQ	Time	# MQ	# EQ	Time
BV7	1,269,564	14	4m27s	175,458	146	2m00s	10,926	91	26s
BV8	5,994,280	24	25m51s	677,176	287	16m49s	25,168	116	1m43s
BV9	19,370,515	15	2h00m09s	890,475	535	53m48s	39,507	99	2m54s
BV10	80,473,959	26	13h19m02s	2,795,183	1060	8h57m32s	80,455	111	10m13s

- "Model" application for local alphabet abstraction, global abstraction performs poorly

Conclusions

- Algorithm for simultaneously inferring
  - a model (Mealy machine)
  - a locally coarsest determinism-preserving alphabet abstractions
- Very natural extension of "classical" automata learning
- Pure black-box technique
  - also applicable to white-box scenarios (e.g., JPF Psyco)
- considerably reduces the number of membership queries
  - more equivalence queries, but more room for clever search heuristics

Questions?

Q & A

L^* – Correctness & Convergence

- The algorithm only terminates if an equivalence query signals "yes"
  - Trivial correctness argument

L^* – Correctness & Convergence

- The algorithm only terminates if an equivalence query signals "yes"
  - Trivial correctness argument
- Every counterexample leads to (at least) one new state

L^* – Correctness & Convergence

- The algorithm only terminates if an equivalence query signals "yes"
  - Trivial correctness argument
- Every counterexample leads to (at least) one new state
- The number of states will never exceed the size of the (unique) minimal canonical automaton
  - Convergence guaranteed ...

L^* – Correctness & Convergence

- The algorithm only terminates if an equivalence query signals "yes"
  - Trivial correctness argument
- Every counterexample leads to (at least) one new state
- The number of states will never exceed the size of the (unique) minimal canonical automaton
  - Convergence guaranteed ...
- ... for perfect equivalence oracles
  - Equivalence queries in general undecidable
  - Various heuristics, very much setting-dependent

Handling Counterexamples

- Cleverly exploiting counterexamples key to advanced learning techniques
- Counterexample triggers model refinement (new state) or alphabet refinement
- Idea: Transformation to representatives ([Rivest&Schapire, 1989] for model refinement)
  - $\lfloor \cdot \rfloor : \Sigma^* \to \Sigma^*$ maps words to short prefix of their target state (hypothesis)
  - Counterexample $w = u \cdot a \cdot v$
  - $MQ(\lfloor u \rfloor a \cdot v) \neq MQ(\lfloor ua \rfloor \cdot v)$: transition target (long prefix) $\lfloor u \rfloor a$ differs from hypothesis state (short prefix) $\lfloor ua \rfloor$ (suffix v exposes difference)
  - Provides state $\lfloor u \rfloor$ with wrong transition (not used in L^*)

Local Alphabet Refinement

- Transformation to representative symbol: $\gamma(\alpha(\cdot))$
- Perform check: $MQ(\lfloor u \rfloor \cdot a \cdot v) \neq MQ(\lfloor u \rfloor \cdot \gamma(\alpha(a)) \cdot v)$?
  - Split leaf for $\alpha(a)$ in abstraction tree for $\lfloor u \rfloor$
  - Witness: $\langle v, MQ(\lfloor u \rfloor \cdot a \cdot v) \rangle$

Alphabet vs. Model Refinement

Alphabet vs. Model Refinement

- Discriminate using futures v (different rows in observation table)
- Discriminate using futures v (different paths in abstraction tree)

Alphabet vs. Model Refinement

- Discriminate using futures v (different rows in observation table)
- Refine by adding futures (separating rows in OT)

- Discriminate using futures v (different paths in abstraction tree)
- Refine by splitting abstraction tree nodes (using futures)

Alphabet vs. Model Refinement

- Discriminate using futures v (different rows in observation table)
- Refine by adding futures (separating rows in OT)
- Analyze by representative transformation $\lfloor \cdot \rfloor$

- Discriminate using futures v (different paths in abstraction tree)
- Refine by splitting abstraction tree nodes (using futures)
- Analyze by representative transformation $\gamma(\alpha(\cdot))$

Alphabet vs. Model Refinement

- Discriminate using futures v (different rows in observation table)
- Refine by adding futures (separating rows in OT)
- Analyze by representative transformation $\lfloor \cdot \rfloor$

- Discriminate using futures v (different paths in abstraction tree)
- Refine by splitting abstraction tree nodes (using futures)
- Analyze by representative transformation $\gamma(\alpha(\cdot))$

Combination of both methods/analyses: interleaving (ce. $w = w_1 \dots w_n$)

$$MQ(\lfloor \varepsilon \rfloor w_1 \dots w_n)$$

Alphabet vs. Model Refinement

- Discriminate using futures v (different rows in observation table)
- Refine by adding futures (separating rows in OT)
- Analyze by representative transformation $\lfloor \cdot \rfloor$

- Discriminate using futures v (different paths in abstraction tree)
- Refine by splitting abstraction tree nodes (using futures)
- Analyze by representative transformation $\gamma(\alpha(\cdot))$

Combination of both methods/analyses: interleaving (ce. $w = w_1 \dots w_n$)

$$MQ(\lfloor \varepsilon \rfloor w_1 \dots w_n) \stackrel{?}{=} MQ(\gamma(\alpha(w_1)) \cdot w_2 \dots w_n)$$

Alphabet vs. Model Refinement

- Discriminate using futures v (different rows in observation table)
- Refine by adding futures (separating rows in OT)
- Analyze by representative transformation $\lfloor \cdot \rfloor$

- Discriminate using futures v (different paths in abstraction tree)
- Refine by splitting abstraction tree nodes (using futures)
- Analyze by representative transformation $\gamma(\alpha(\cdot))$

Combination of both methods/analyses: interleaving (ce. $w = w_1 \dots w_n$)

$$\begin{aligned} & MQ(\lfloor \varepsilon \rfloor w_1 \dots w_n) \\ \stackrel{?}{=} \quad & MQ(\lfloor \gamma(\alpha(w_1)) \rfloor \cdot w_2 \dots w_n) \end{aligned}$$

$$\stackrel{?}{=} \quad MQ(\gamma(\alpha(w_1)) \cdot w_2 \dots w_n)$$

Alphabet vs. Model Refinement

- Discriminate using futures v (different rows in observation table)
- Refine by adding futures (separating rows in OT)
- Analyze by representative transformation $\lfloor \cdot \rfloor$

- Discriminate using futures v (different paths in abstraction tree)
- Refine by splitting abstraction tree nodes (using futures)
- Analyze by representative transformation $\gamma(\alpha(\cdot))$

Combination of both methods/analyses: interleaving (ce. $w = w_1 \dots w_n$)

$$\begin{aligned} & MQ(\lfloor \varepsilon \rfloor w_1 \dots w_n) \\ \stackrel{?}{=} \quad & MQ(\lfloor \gamma(\alpha(w_1)) \rfloor \cdot w_2 \dots w_n) \end{aligned}$$

$$\begin{aligned} \stackrel{?}{=} \quad & MQ(\gamma(\alpha(w_1)) \cdot w_2 \dots w_n) \\ \stackrel{?}{=} \quad & MQ(\lfloor \gamma(\alpha(w_1)) \rfloor \gamma(\alpha(w_2)) \cdot w_3 \dots w_n) \end{aligned}$$

Alphabet vs. Model Refinement

- Discriminate using futures v (different rows in observation table)
- Refine by adding futures (separating rows in OT)
- Analyze by representative transformation $\lfloor \cdot \rfloor$

- Discriminate using futures v (different paths in abstraction tree)
- Refine by splitting abstraction tree nodes (using futures)
- Analyze by representative transformation $\gamma(\alpha(\cdot))$

Combination of both methods/analyses: interleaving (ce. $w = w_1 \dots w_n$)

$$
\begin{aligned}
& MQ(\lfloor \varepsilon \rfloor w_1 \dots w_n) \\
\stackrel{?}{=} \; & MQ(\lfloor \gamma(\alpha(w_1)) \rfloor \cdot w_2 \dots w_n) \\
\stackrel{?}{=} \; & MQ(\lfloor \gamma(\alpha(w_1))\gamma(\alpha(w_2)) \rfloor \cdot w_3 \dots w_n)
\end{aligned}
$$

$$
\begin{aligned}
\stackrel{?}{=} \; & MQ(\gamma(\alpha(w_1)) \cdot w_2 \dots w_n) \\
\stackrel{?}{=} \; & MQ(\lfloor \gamma(\alpha(w_1)) \rfloor \gamma(\alpha(w_2)) \cdot w_3 \dots w_n) \\
\stackrel{?}{=} \; & \dots
\end{aligned}
$$