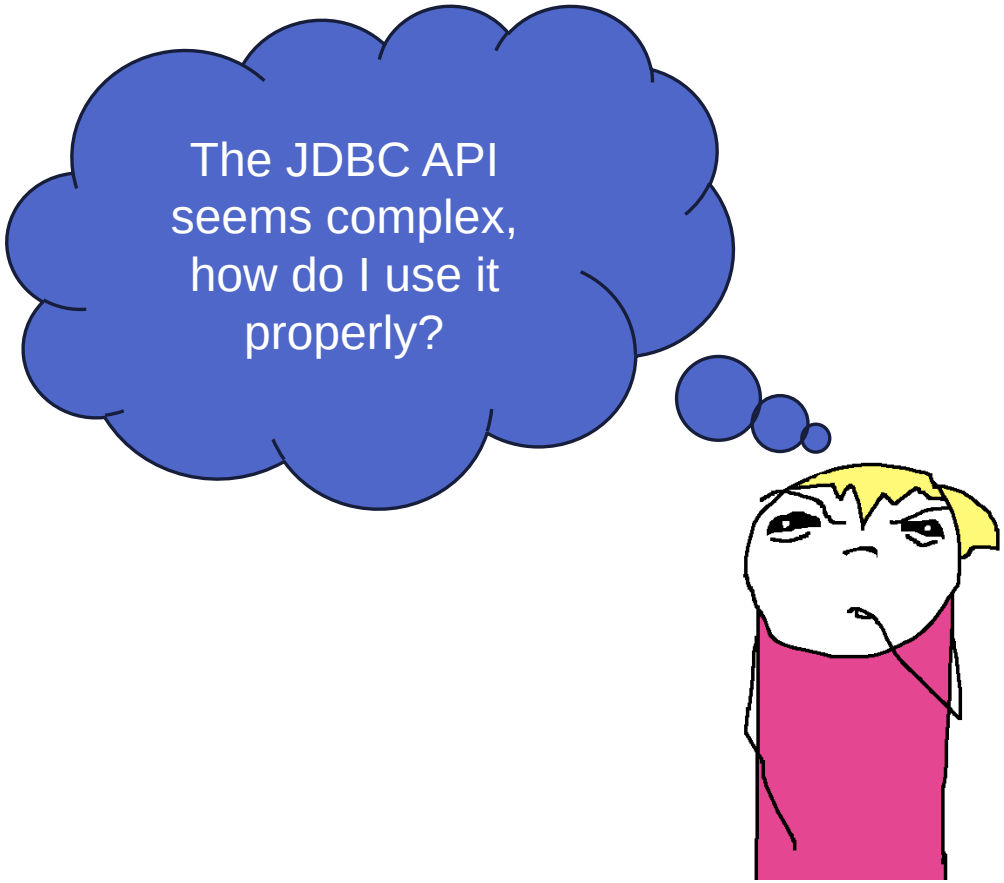
Symbolic Automata for Static Specification Mining

Alon Mishne
Hila Peleg
Sharon Shoham
Eran Yahav
Hongseok Yang

(Artwork by Allie Brosh of *Hyperbole and a Half*)

APIs can be complicated

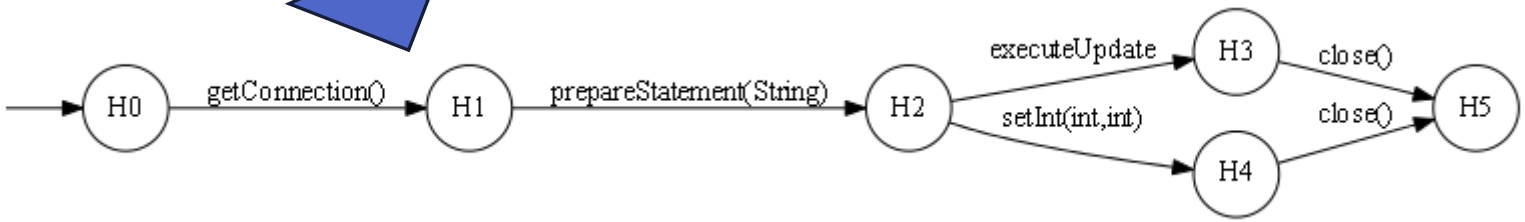



```
Class.forName("com.microsoft.jdbc.
String url = "jdbc:microsoft:sqlse
Connection conn = DriverManager.ge
PreparedStatement pstmt = null;
try {
    String query = "INSERT INTO c
    pstmt = conn.prepareStatement
    pstmt.setInt(1,5);
    pstmt.executeUpdate(); // exec
} finally {
    pstmt.close();
    conn.close();
```

We can get temporal API specifications from examples

```
Class.forName("com.microsoft.jdbc
String url = "jdbc:microsoft:sqls
Connection conn = DriverManager.g
PreparedStatement pstmt = null;
try {
    String query = "INSERT INTO c
    pstmt = conn.prepareStatement
    pstmt.setInt(1,5);
    pstmt.executeUpdate(); // exec
} finally {
    pstmt.close();
```

Translation: find out the sequence of methods programmers invoke in order to actually do stuff with the library



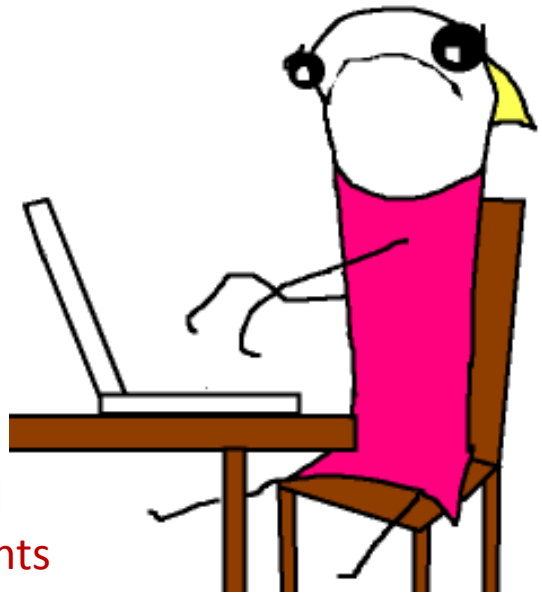
But which example should I use?

What if the one we select has a bug?

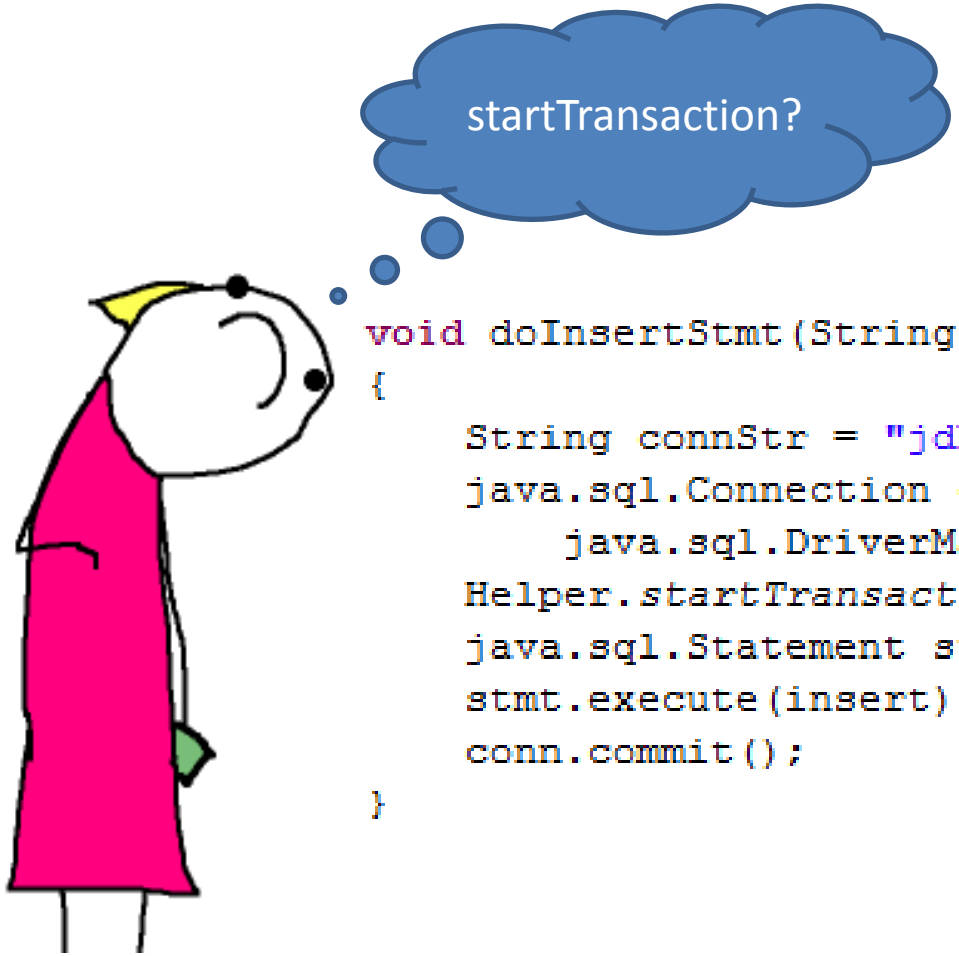
```
void doInsertStmt(String insert){
    String connStr = "jdbc:sqlserver://localhost;integr
    java.sql.Connection conn =
        java.sql.DriverManager.getConnection(connStr);
    java.sql.PreparedStatement preparedStmtInsert =
        conn.prepareStatement(insert);
    preparedStmtInsert.execute();
    conn.commit();
}
```

Connection default is auto-commit, you shouldn't be committing on it

It's also pointless to use prepared statements for run-once statements

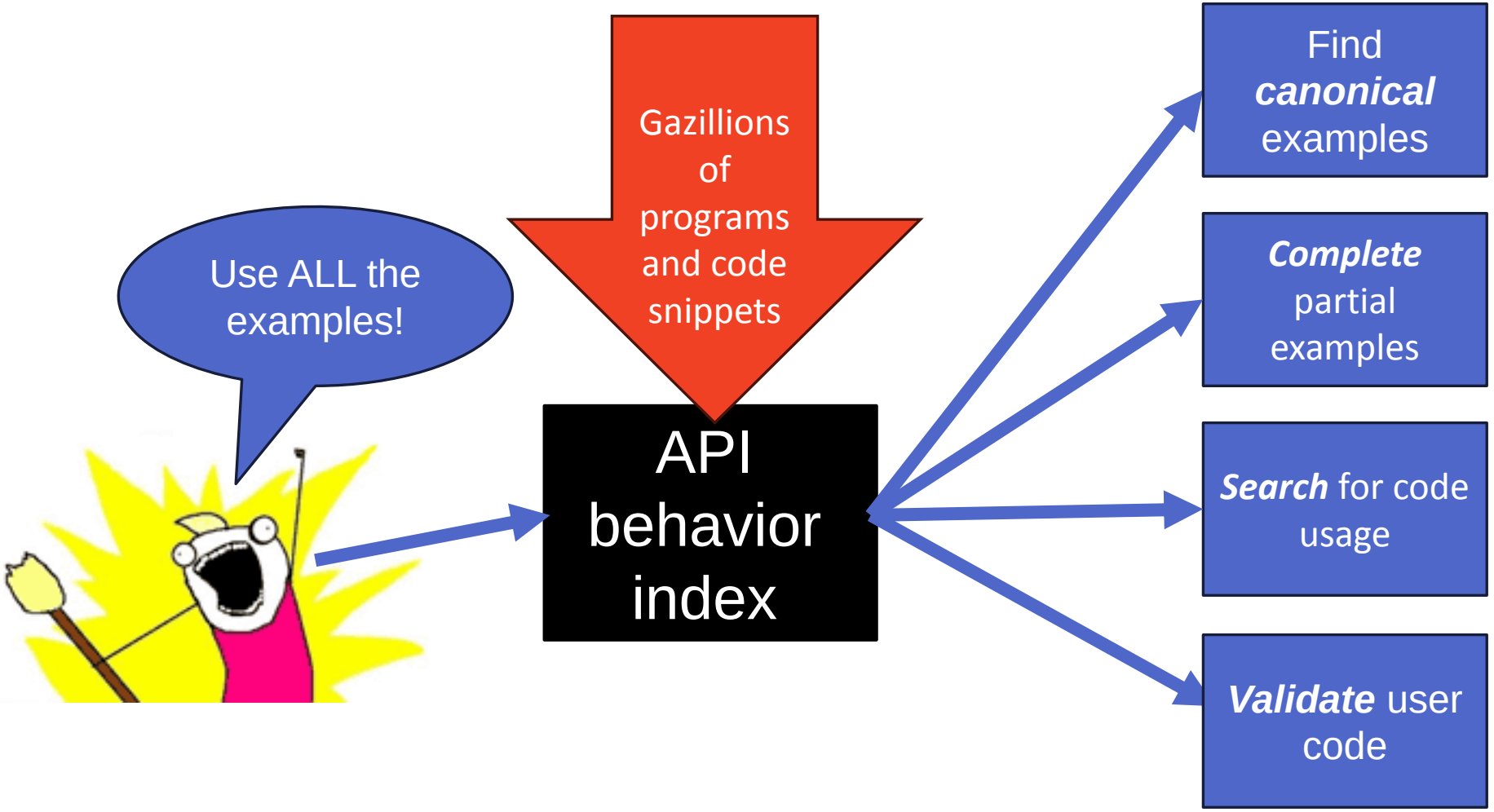


What if the one we select is missing some information?




```
void doInsertStmt(String insert)
{
    String connStr = "jdbc:sqlserver://localhost;integ
    java.sql.Connection conn =
        java.sql.DriverManager.getConnection(connStr);
    Helper.startTransaction(conn);
    java.sql.Statement stmt = conn.createStatement();
    stmt.execute(insert);
    conn.commit();
}
```

We have to learn from more examples



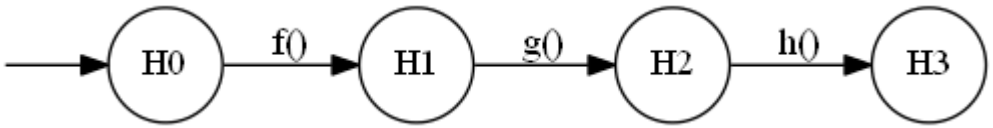
How would we do that?

1. Analyze a single code example
2. Get all the *histories* in it that use the API
3. Repeat for all other examples (a lot)
4. Create an index by consolidating all the resulting histories
5. Use the resulting index for search and verification

Concrete history

```
public void method(Something x) {
  x.f();
  x.g();
  x.h();
}
```

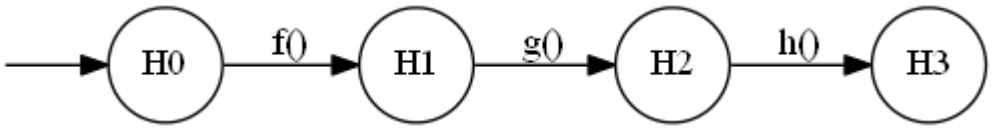
x:



Objects are sometimes related (1)

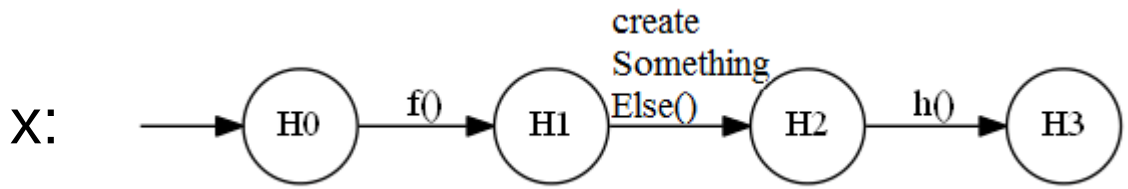
```
public void method(Something x) {
  x.f();
  BaseOfSomething y = x;
  y.g();
  y.h();
}
```

x:



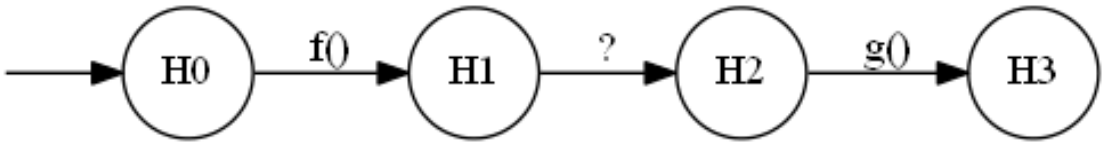
Objects are sometimes related (2)

```
public void method(Something x) {
  x.f();
  SomethingElse s = x.createSomethingElse();
  s.h();
}
```



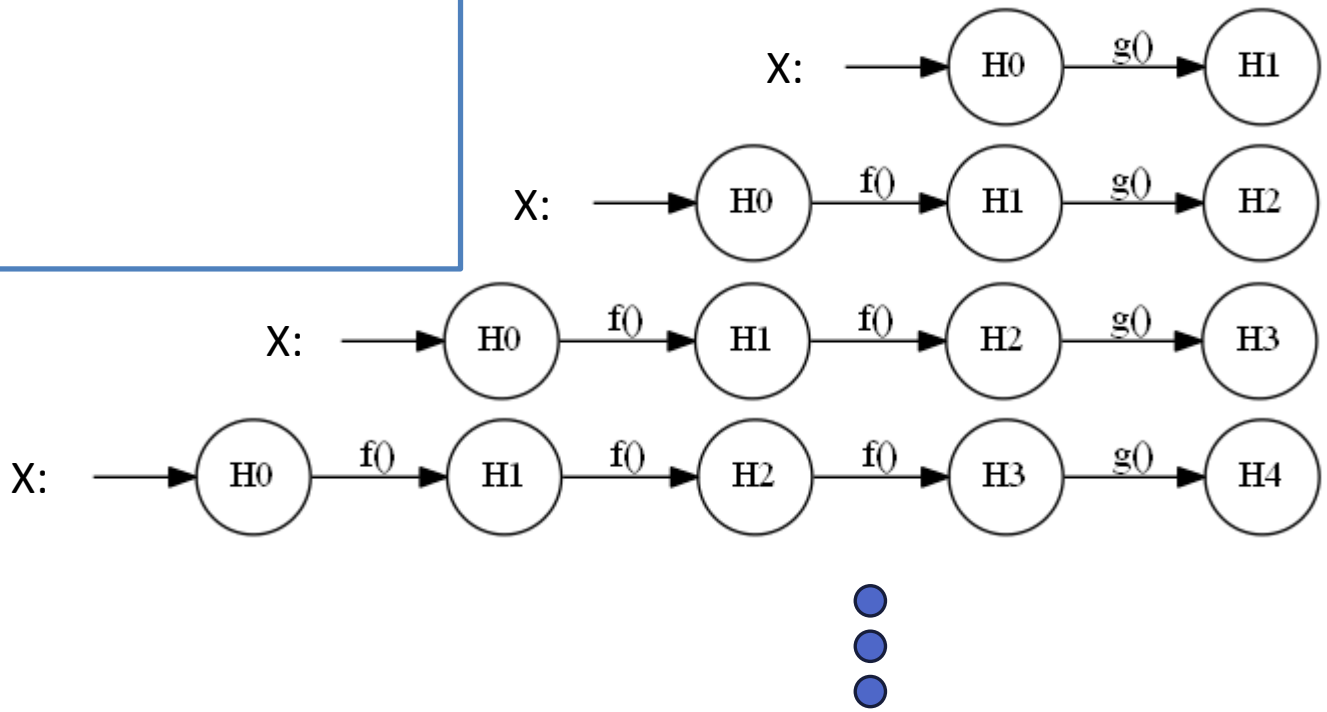
Dealing with the unknown

```
public void method1() {
  Something x = new Something();
  x.f();
  transmogrify(x);
  x.g();
}
```



An unbounded number of histories

```
public void method(Something x) {
  while(?) {
    x.f();
  }
  x.g();
}
```



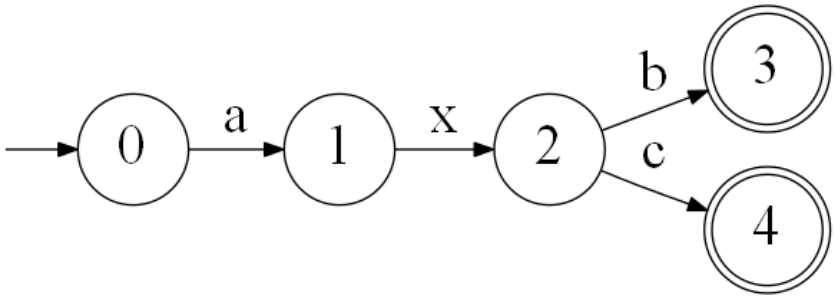
We need an abstraction

- Group all API calls for an object
  - Heap abstraction
  - Tracking creation chains
- Create an abstract history that's bounded
- Histories with unknown steps
  - Use variable for each unknown
- What abstraction? DSAs

Abstract Representation: DSA

A Deterministic **Symbolic** Automaton is a tuple (Σ;Q;δ;ι;F;*Vars*)

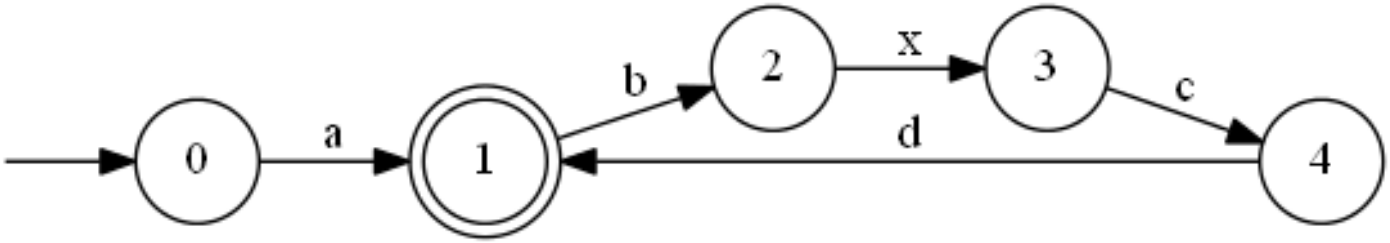
- Σ is a finite alphabet
- Q is a finite set of states
- δ is the transition relation, $Q \times (\Sigma \cup Vars) \rightarrow Q$
- $\iota \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states
- *Vars* is the finite set of variables



Semantics of DSAs: Symbolic Language

$$SL(A) = \{sw \in (\Sigma \cup Vars)^* | \delta(\iota, sw) \in F\}$$

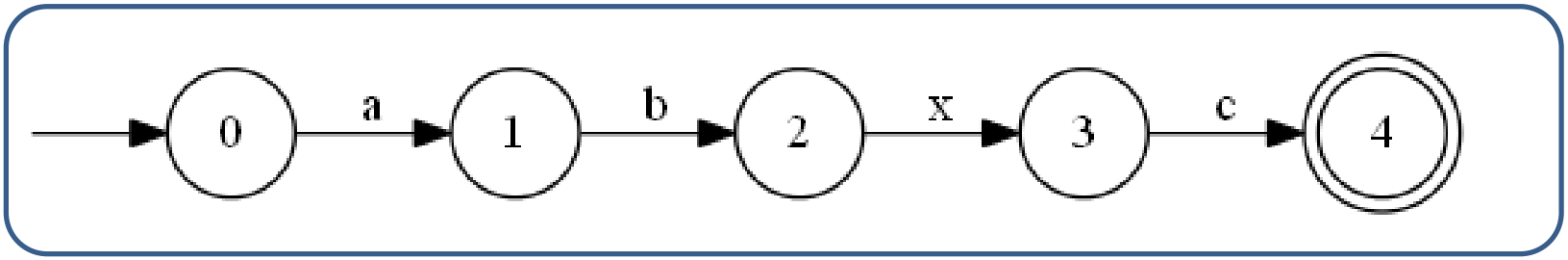
- Words over Σ are ***concrete*** *words*
- Words over $\Sigma \cup Vars$ are ***symbolic*** *words*



SL(A) = { a, abxcd, abxcdbxcd,… }

Assignment

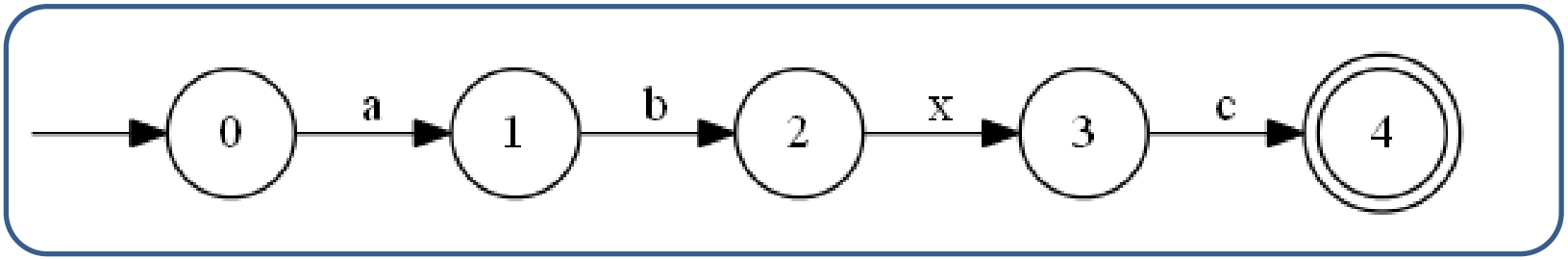
An assignment σ maps a variable x in context sw1, sw2 (sw_1, x, sw_2) to a non-empty symbolic language



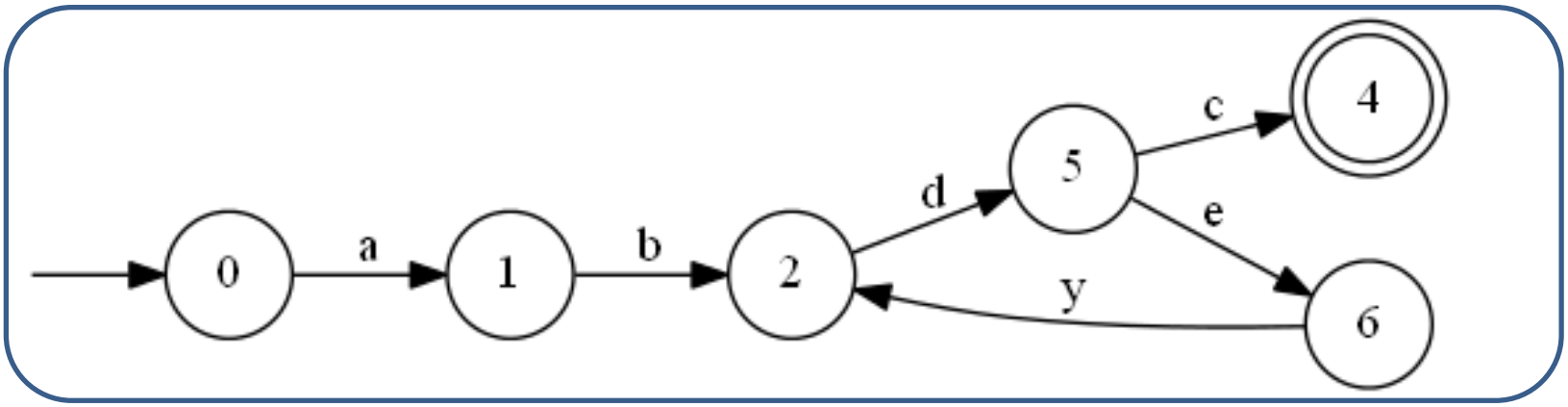
$$\sigma(\epsilon, x, \epsilon) = d(eyd)^*$$

Assignment

An assignment σ maps a variable x in context sw1, sw2 (sw_1, x, sw_2) to a non-empty symbolic language

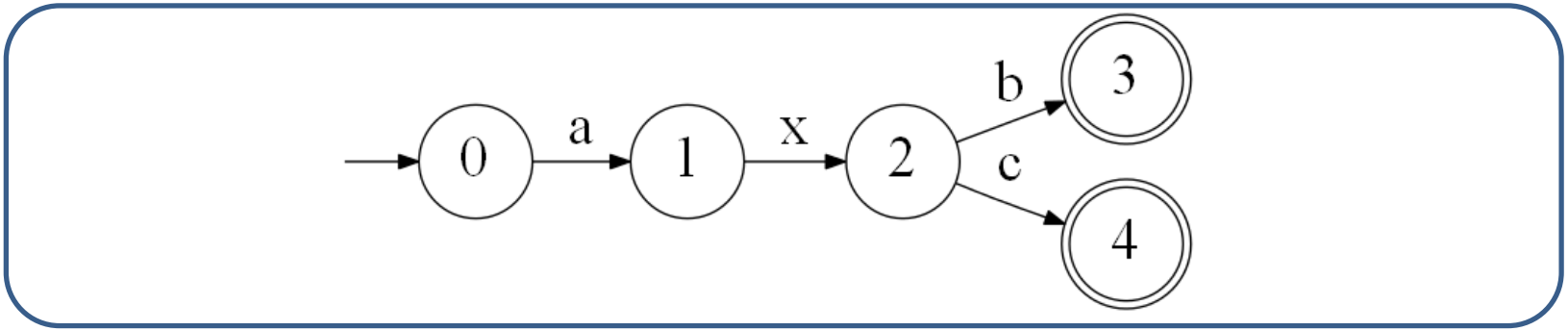


$$\sigma(\epsilon, x, \epsilon) = d(eyd)^*$$



Assignment

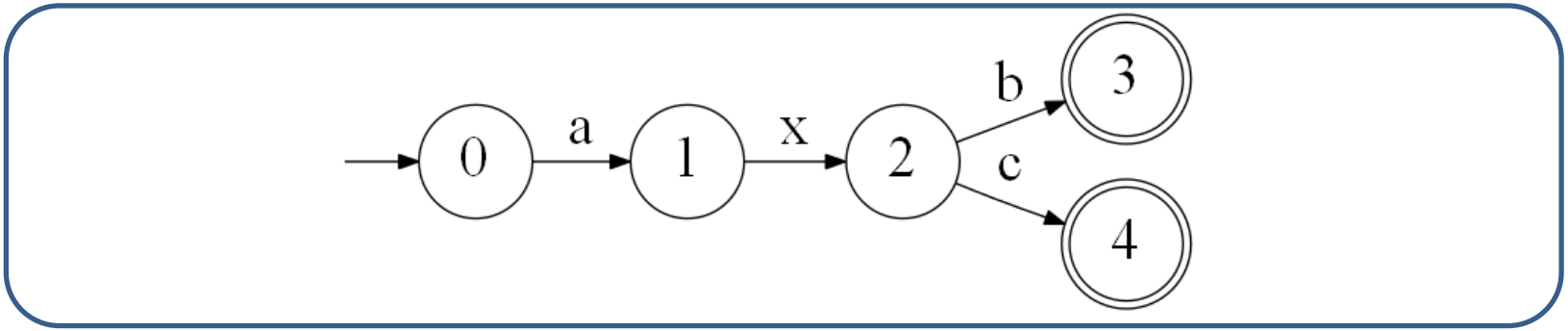
(sw_1, sw_2) is the *context* of the assignment



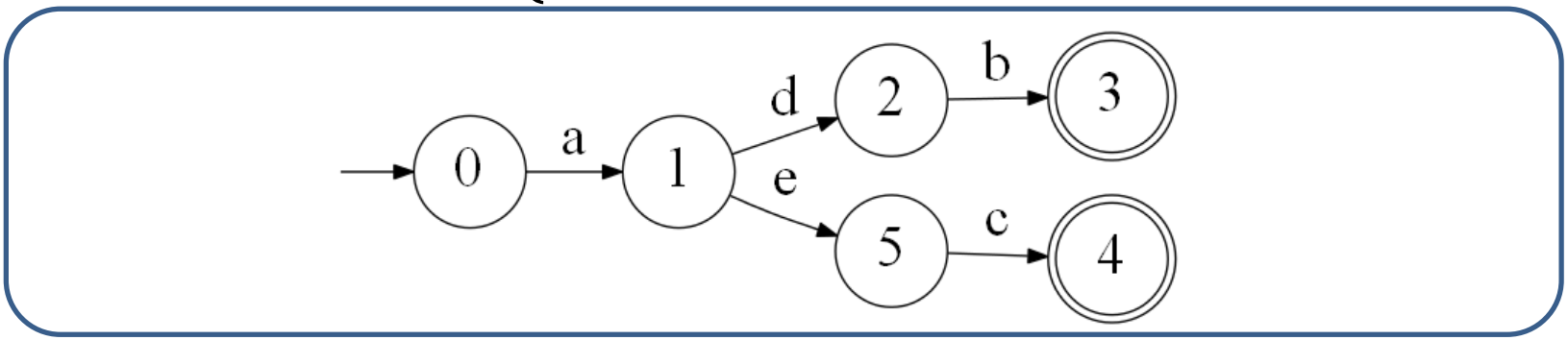
$$\begin{cases} \sigma(\epsilon, x, b) = db \\ \sigma(\epsilon, x, c) = ec \end{cases}$$

Assignment

(sw_1, sw_2) is the *context* of the assignment



$$\begin{cases} \sigma(\epsilon, x, b) = db \\ \sigma(\epsilon, x, c) = ec \end{cases}$$

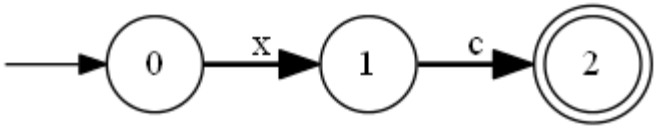


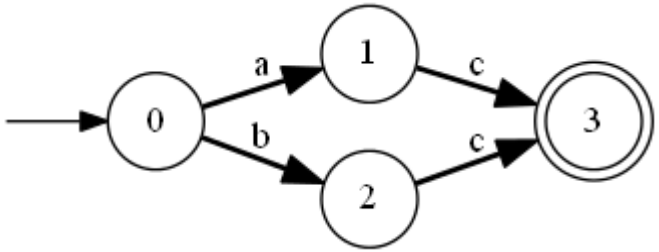
Creating an Abstract Domain

- DSA is a natural abstract representation of a (potentially unbounded) set of histories
- We need a partial order over DSAs
- We want to capture ordering along two axes
  - Precision
  - Partialness
- Other operations for applications:
  - Consolidation
  - Query matching

Order Between DSAs

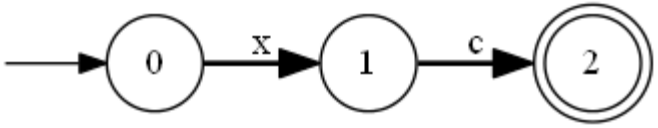
- The most natural way to define order between automata is language inclusion
- This won't work for symbolic automata:

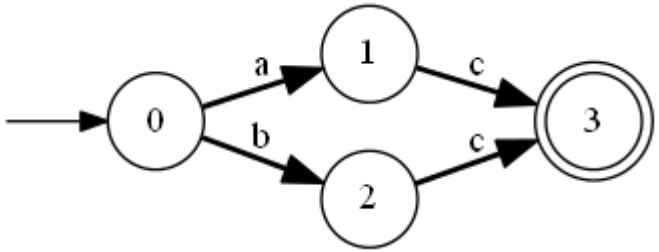




Order Between DSAs

- The most natural way to define order between automata is language inclusion
- This won't work for symbolic automata:

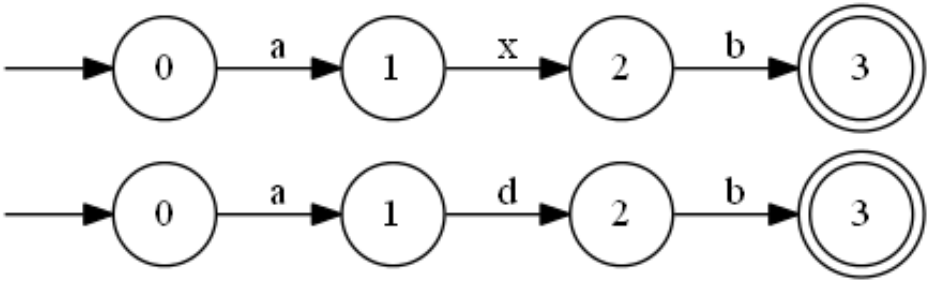




$\{xc\}$ $\not\subseteq$ $\{ac, bc\}$

Partialness

- A word is *less partial* (or *more complete*) than another if it represents a more concrete scenario

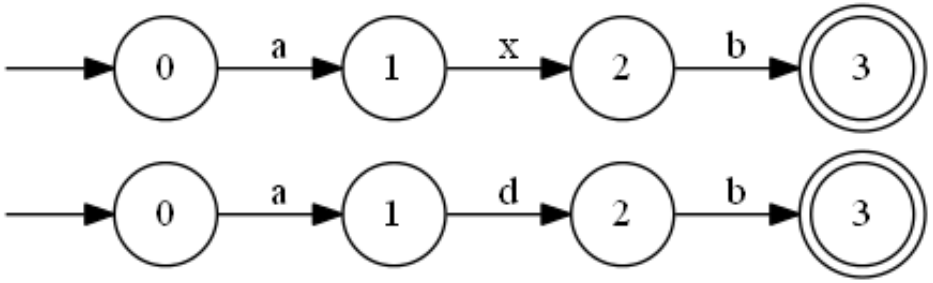


- Formally: w_1 is more partial than w_2 if for each assignment σ_2 to w_2 there is an assignment σ_1 to w_1 s.t. $\sigma_1(w_1) = \sigma_2(w_2)$

Partialness

- A word is *less partial* (or *more complete*) than another if it represents a more concrete scenario

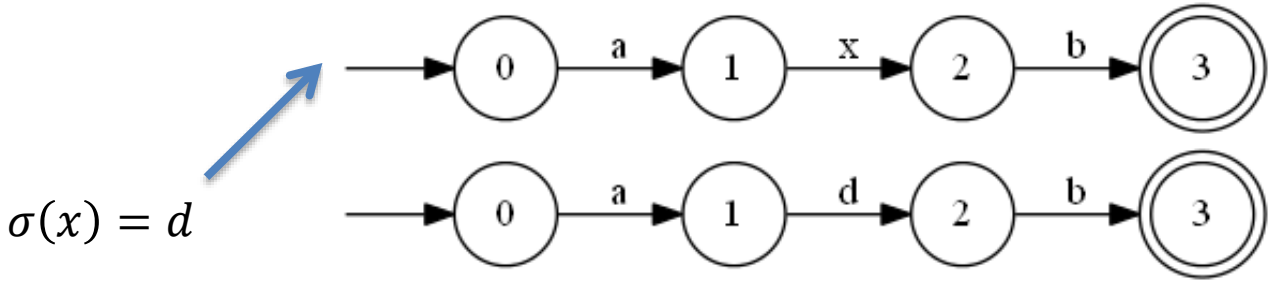
Empty assignment



- Formally: w_1 is more partial than w_2 if for each assignment σ_2 to w_2 there is an assignment σ_1 to w_1 s.t. $\sigma_1(w_1) = \sigma_2(w_2)$

Partialness

- A word is *less partial* (or *more complete*) than another if it represents a more concrete scenario

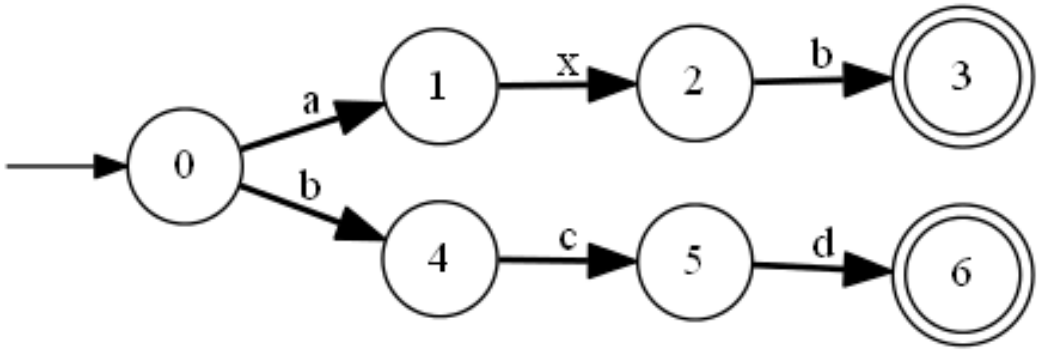


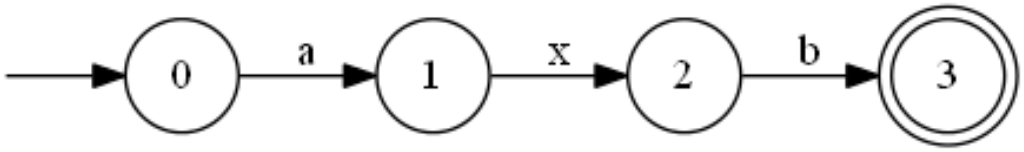
- Formally: w_1 is more partial than w_2 if for each assignment σ_2 to w_2 there is an assignment σ_1 to w_1 s.t. $\sigma_1(w_1) = \sigma_2(w_2)$

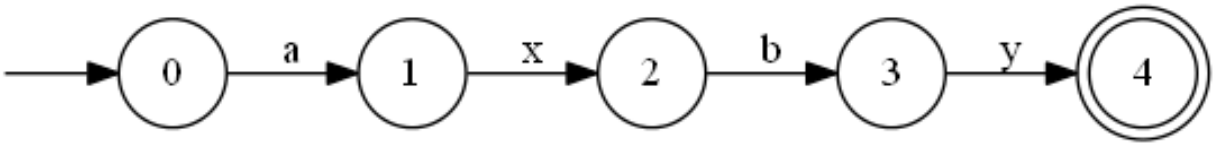
Partial Order

- We define the order over DSAs to capture both axes:
  - Precision: the natural concept of language inclusion
  - Partialness: of the individual words
- Intuitively: a DSA is smaller if it is more precise and more partial
- Precision is the “classic” upwards direction of a lattice

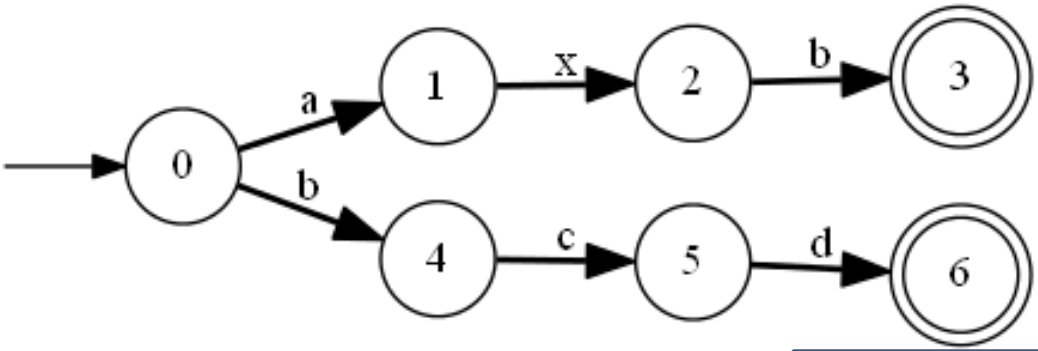
Precision vs. Partialness



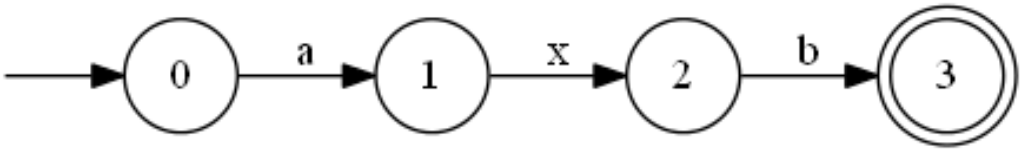


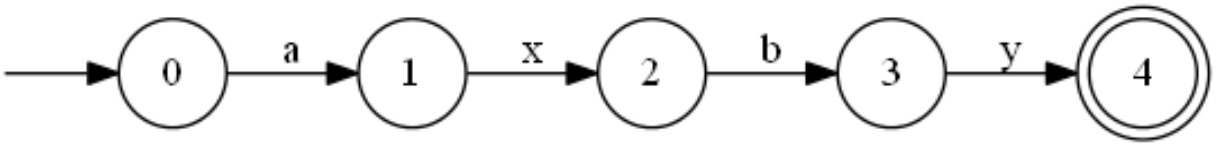


Precision vs. Partialness

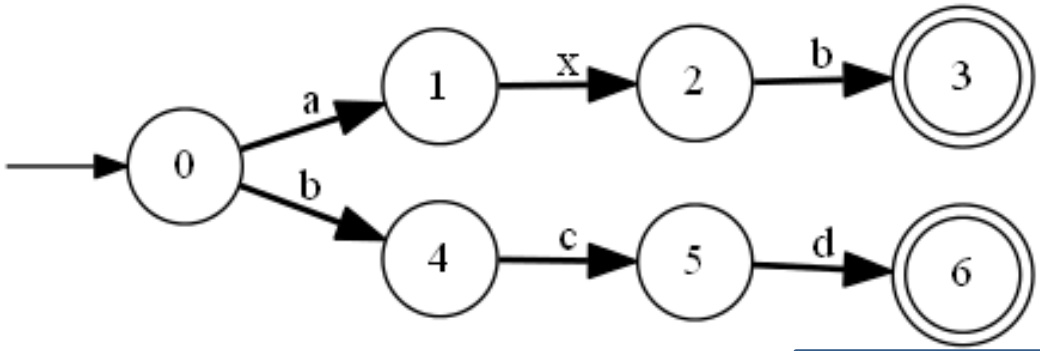


The less *precise* you are, the more behaviors you describe

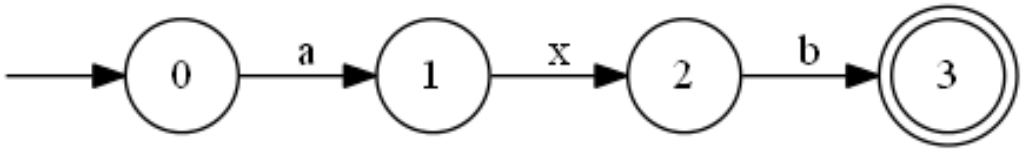




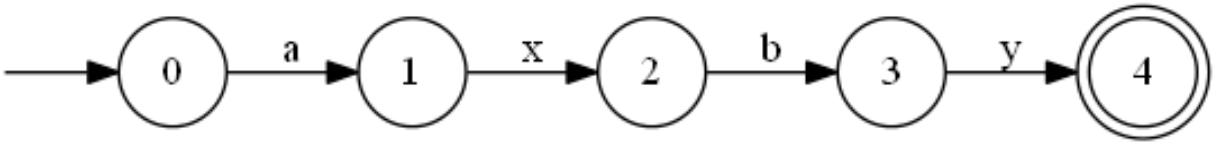
Precision vs. Partialness



The less *precise* you are, the more behaviors you describe



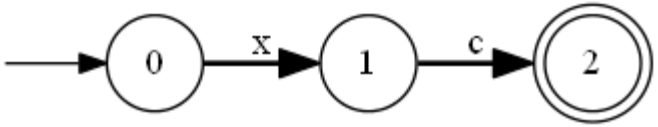
And the more *partial* you are, the more behaviors you describe…

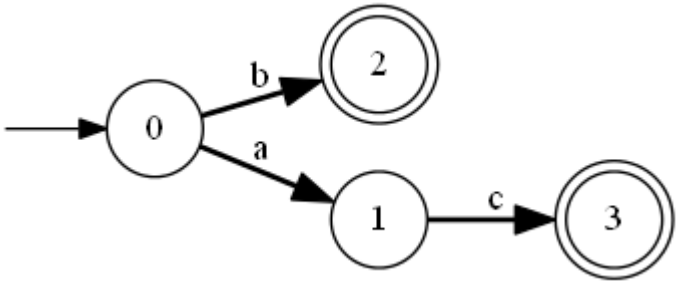


≤

The domain's ≤

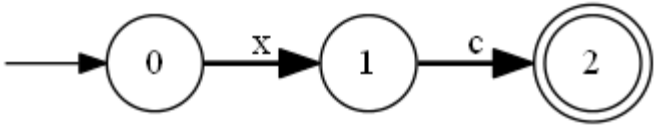
$A_1 \leq A_2$ if for every concrete assignment σ_2 of A_2 there exists a concrete assignment σ_1 of A_1 for which $\sigma_1(SL(A_1)) \subseteq \sigma_2(SL(A_2))$

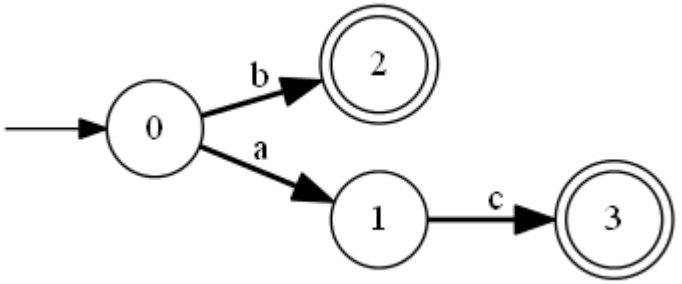




The domain's ≤

$A_1 \leq A_2$ if for every concrete assignment σ_2 of A_2 there exists a concrete assignment σ_1 of A_1 for which $\sigma_1(SL(A_1)) \subseteq \sigma_2(SL(A_2))$





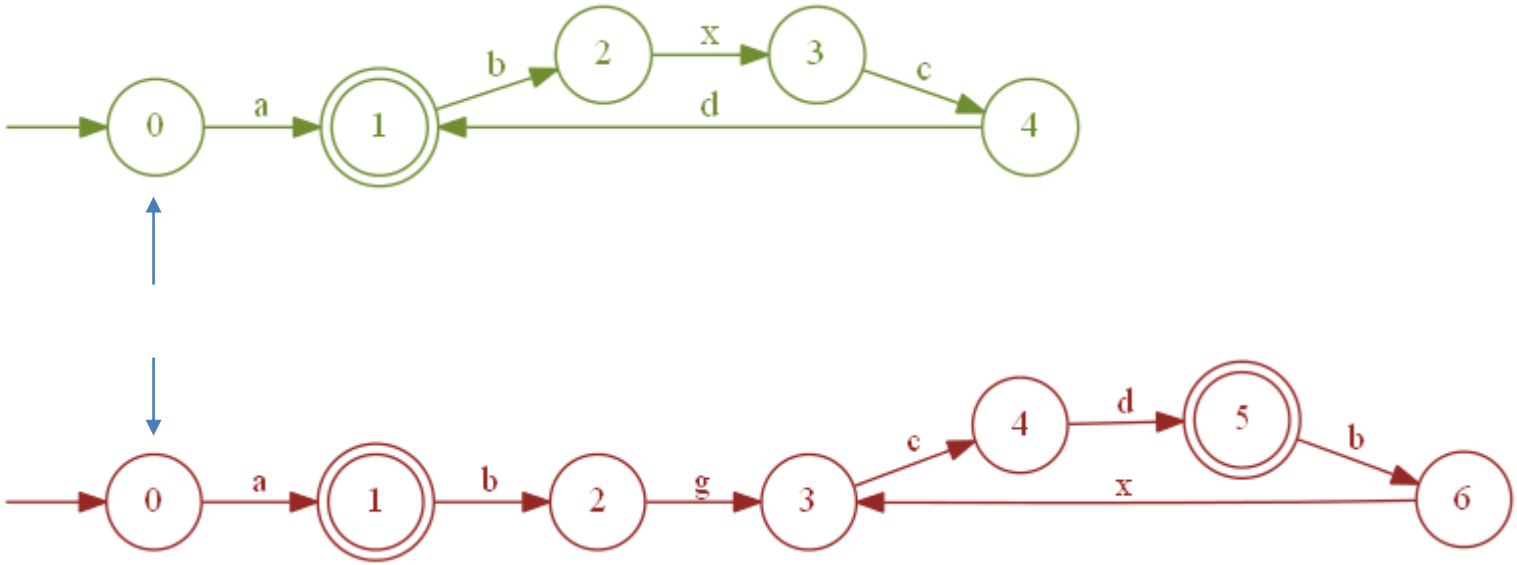
$\sigma(x) = a$, so $\{ac\} \subseteq \{ac, b\}$

Calculating Inclusion via Simulation

- Adapting the natural notion of simulation in DFAs to DSAs: symbolic simulation
  - Find pairs of one state from A_1 and a set of states from A_2 that are a witness to structural inclusion
  - Collect possible candidates using outgoing transitions
- DFA simulation already captures the notion of precision
- DSA simulation adds the notion of partialness
  - Symbols can “swallow” parts of the other DSA

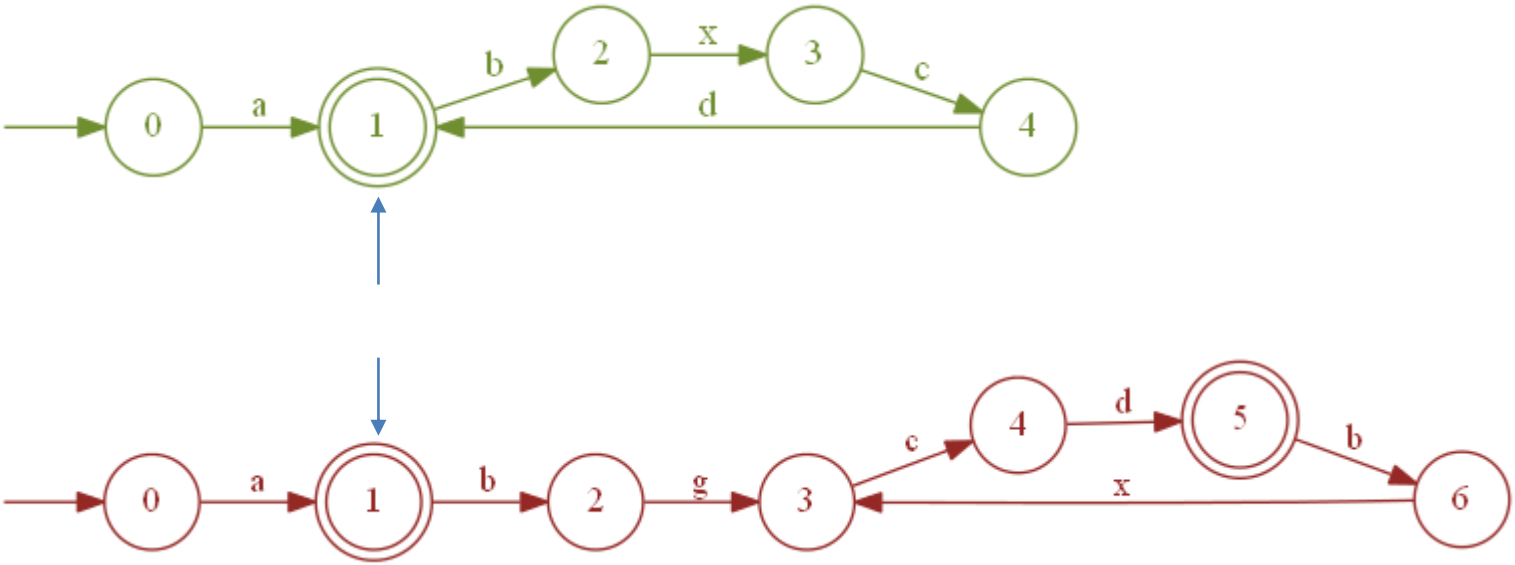
Simulation Example

Simulation: (0,{0}),



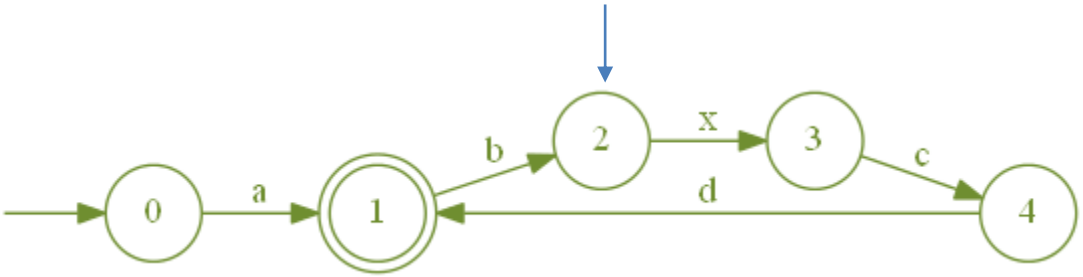
Simulation Example

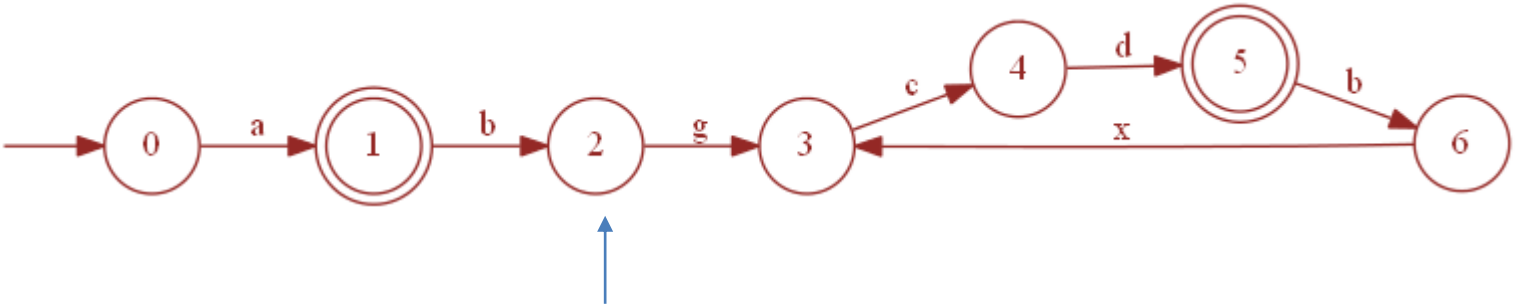
Simulation: (0,{0}),
(1,{1}),



Simulation Example

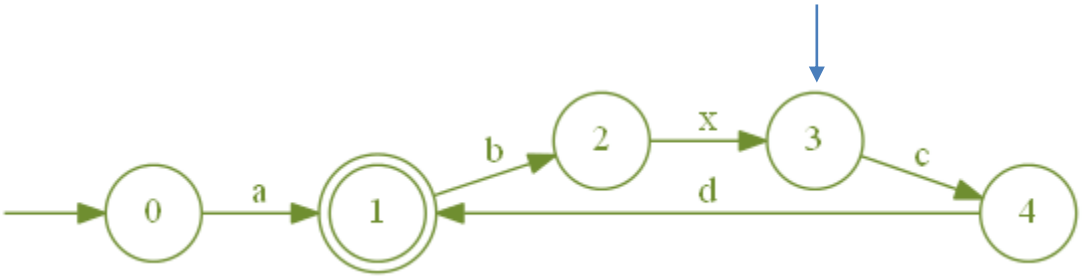
Simulation: (0,{0}),
(1,{1}),
(2,{2}),

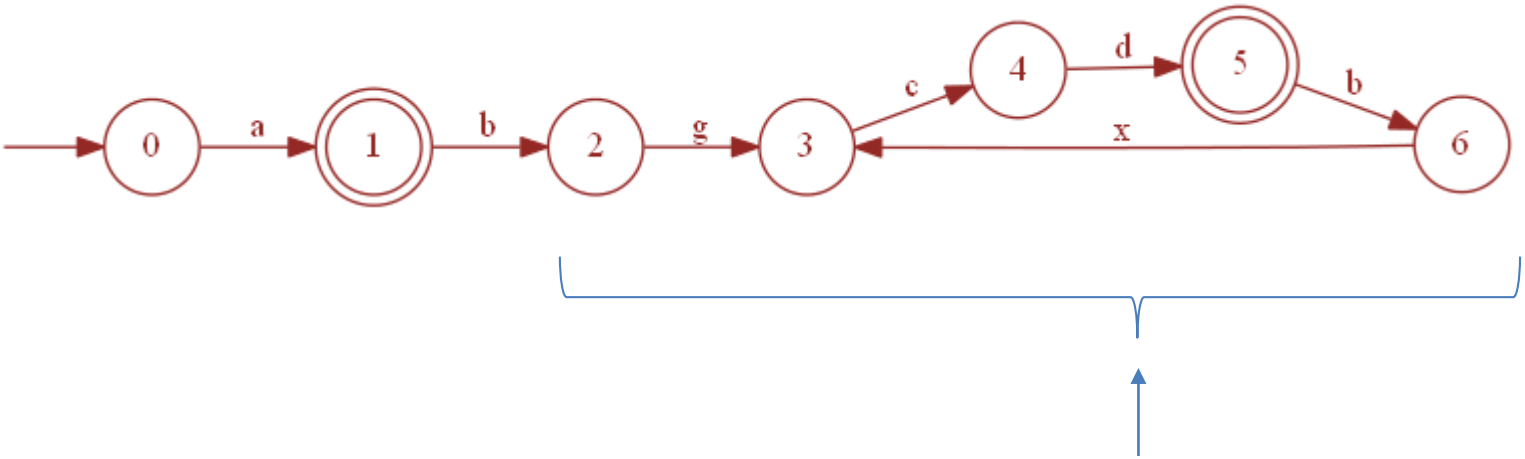




Simulation Example

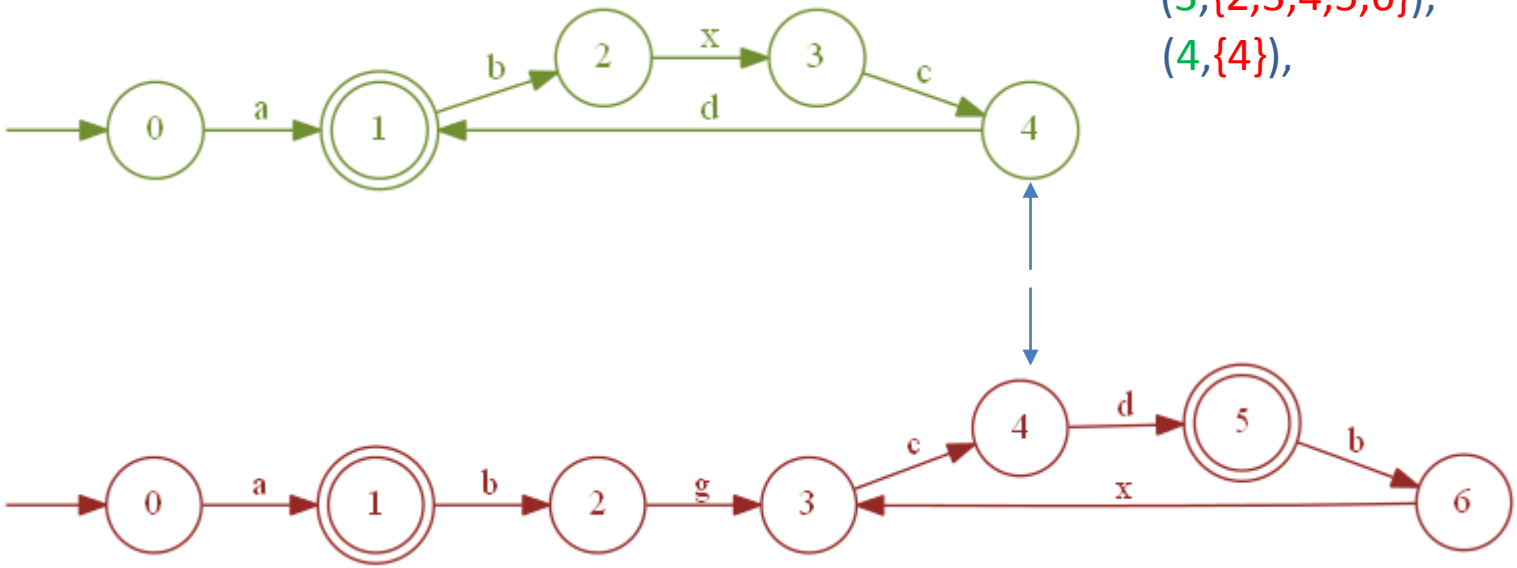
Simulation: (0,{0}),
(1,{1}),
(2,{2}),
(3,{2,3,4,5,6}),



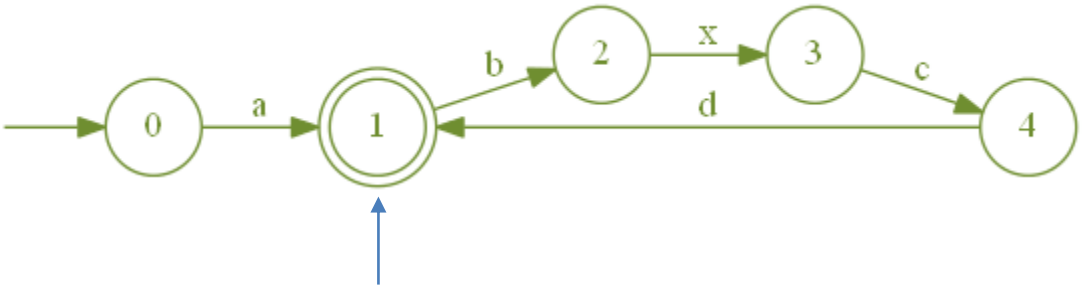


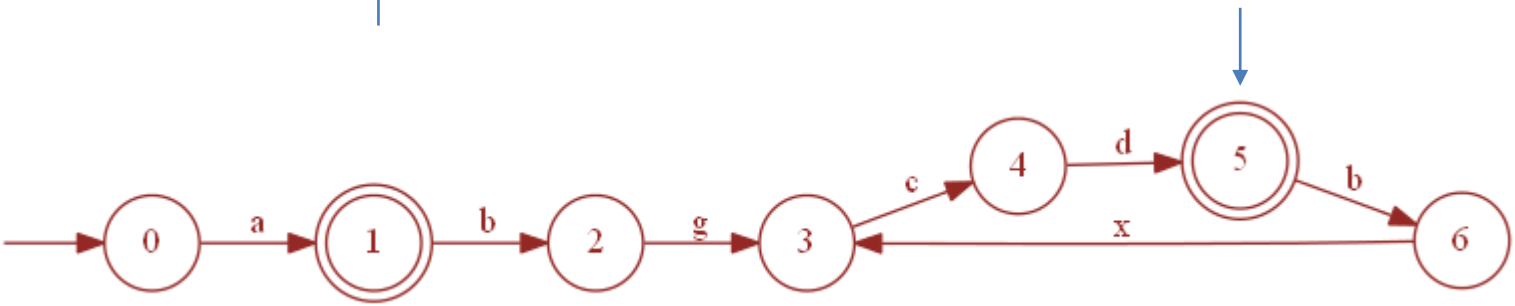
Simulation Example

Simulation: (0,{0}),
(1,{1}),
(2,{2}),
(3,{2,3,4,5,6}),
(4,{4}),



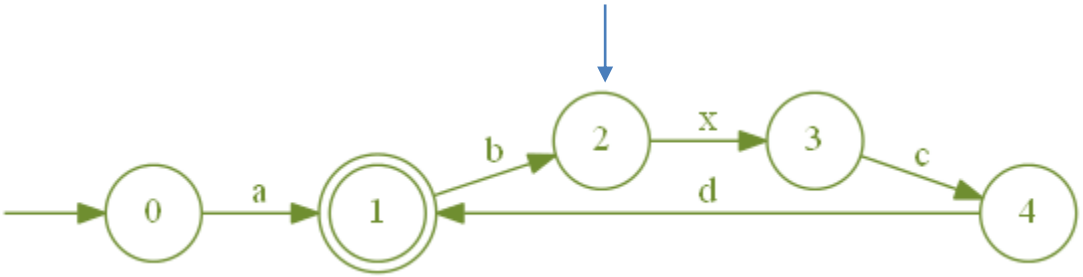
Simulation Example



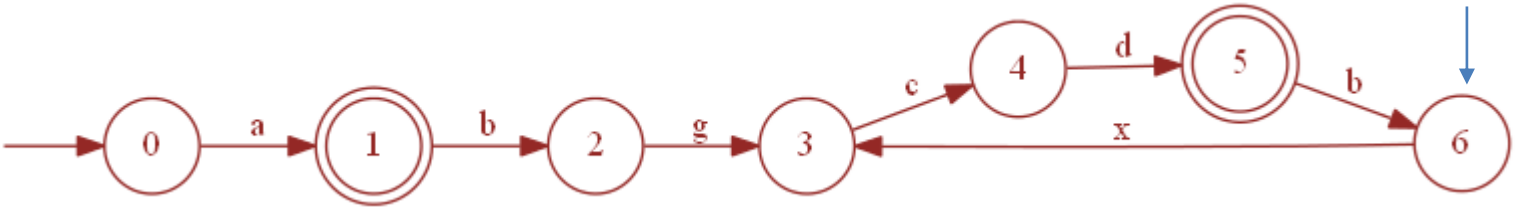


Simulation: (0,{0}),
(1,{1}),
(2,{2}),
(3,{2,3,4,5,6}),
(4,{4}),
(1,{5}),

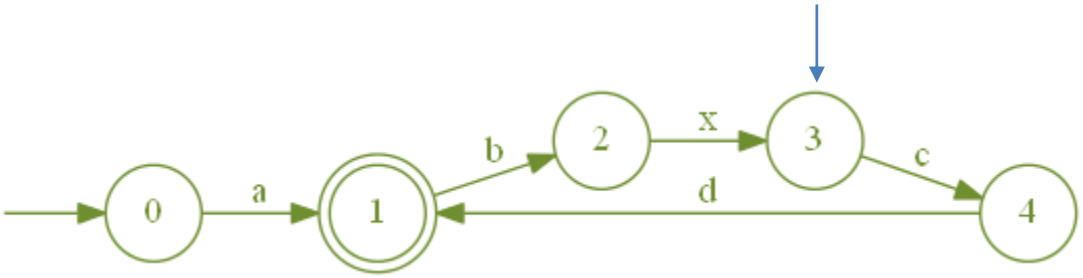
Simulation Example



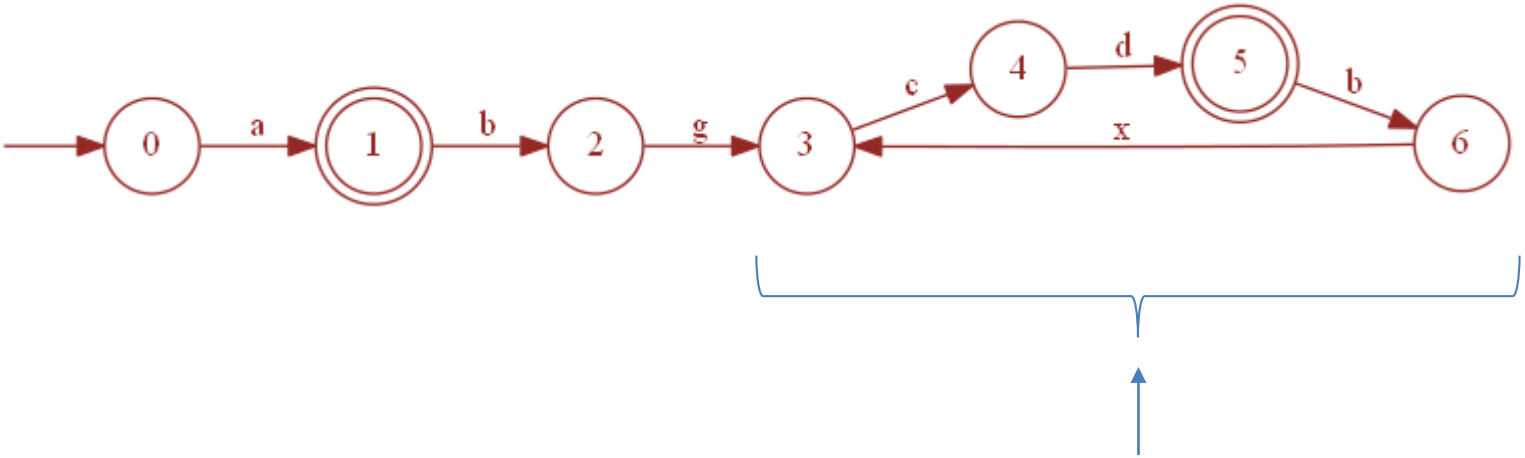
Simulation: (0,{0}),
(1,{1}),
(2,{2}),
(3,{2,3,4,5,6}),
(4,{4}),
(1,{5}),
(2,{6}),



Simulation Example

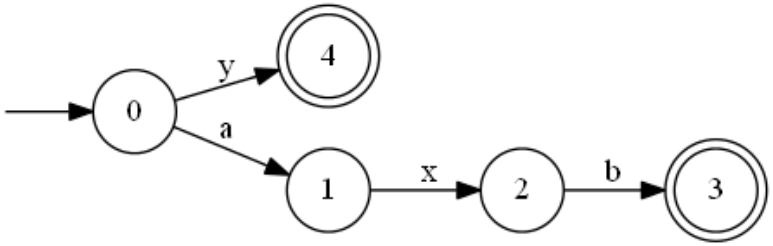


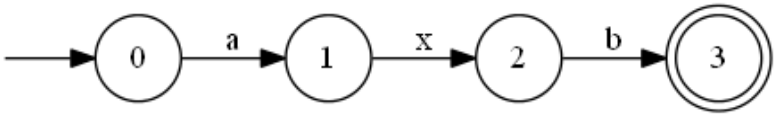
Simulation: (0,{0}),
(1,{1}),
(2,{2}),
(3,{2,3,4,5,6}),
(4,{4}),
(1,{5}),
(2,{6}),
(3,{3,4,5,6})



≤ is a Preorder

- ≤ is transitive and reflexive, but it is not antisymmetric:
- Example:





- But to have a lattice, we need a partial order

The DSA/≡ Domain

- If $A_1 \leq A_2$ and $A_2 \leq A_1$, we say $A_1 \equiv A_2$
- So, instead of looking at the automata as our domain, we look at the equivalence classes created by $\equiv$.
- For DSA/$\equiv$, $\leq$ is a partial order

Join in DSA/≡

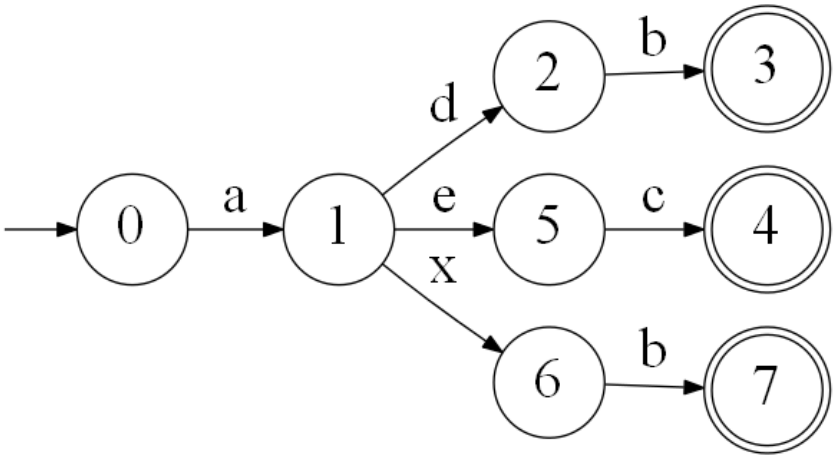
- In this domain we can now define join:
- Create the union (like DFA union) of A_1 and A_2.
- $(A_1 \sqcup A_2)$ is a representative of the equivalence class for the least upper bound of A_1 and A_2.
- Conclusion: (DSA/≡,⊑) is a join semi-lattice

Computing Join

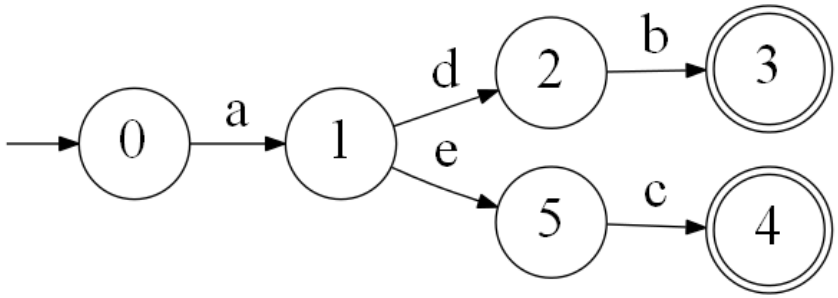
- When we compute join in DSA/$\equiv$ we start with a union operation
- But we would like to select a *most complete* representative from the resulting equivalence class
- That means we would like to throw out "duplicate" (equivalent or subsumed) words
- We call this *consolidation*

Consolidation: an example

We'd like to go from this:

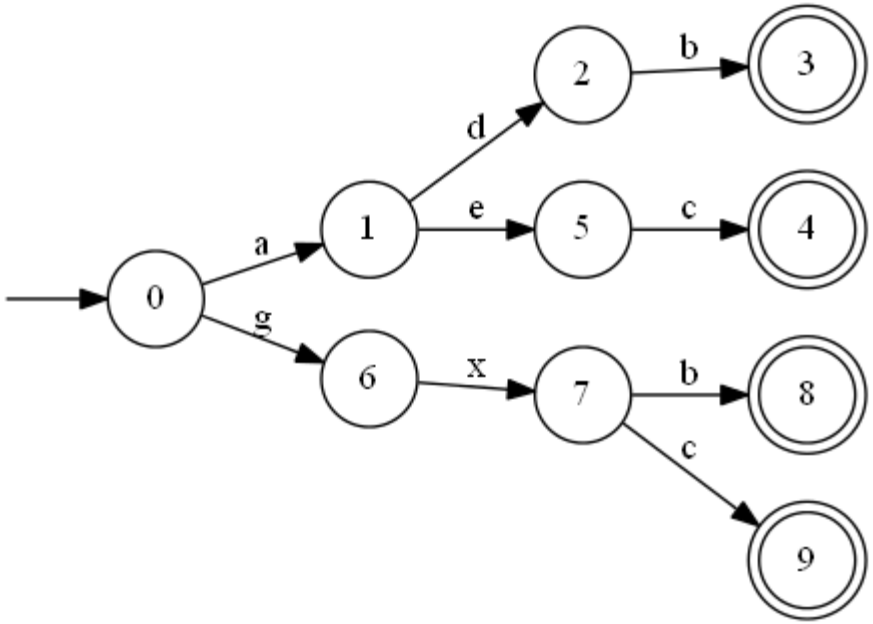


To this:



Answering Queries

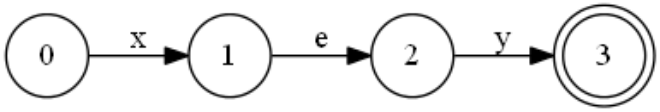
- Now we have a database representing our API



- And we would like to run queries like “what is the correct usage around *e*?”

To answer a query

- This means taking a query Q:



- And look for an assignment σ that would make $\sigma(SL(Q)) \subseteq SL(A)$
- In our case: $\sigma(x) = a$ and $\sigma(y) = c$
- This process is called *unknown elimination*

Unknown Elimination

- If we have an A_1 that is symbolically included in A_2 we can say that the concrete parts of A_1 exist in A_2.
- The partial parts of A_1 match up to some part (not necessarily concrete) of A_2.
- We already have the simulation matching up the concrete parts, we can use its result to match something up to the symbolic parts

UE with contexts

- An assignment can have context, both incoming and outgoing such as $(\epsilon, x, b) \mapsto a$
- This means that for each variable, we compute from the simulation all its incoming and outgoing contexts
- The assignment is filled for each variable with the contexts and the corresponding part of A_2

Putting It All Together: An Analysis

- Here's how we would perform an analysis of an API using everything we've got:
  - Take a bunch of programs or program snippets
  - Mine each one for the usage of the API
  - Join them to create the database
  - Use the database and unknown elimination to answer queries

A bunch of program snippets

Program A

```
void foo() {
        Socket s = new Socket();
        configure(s);
        s.connect();
        s.send(someBuffer);
        s.close();
}
```

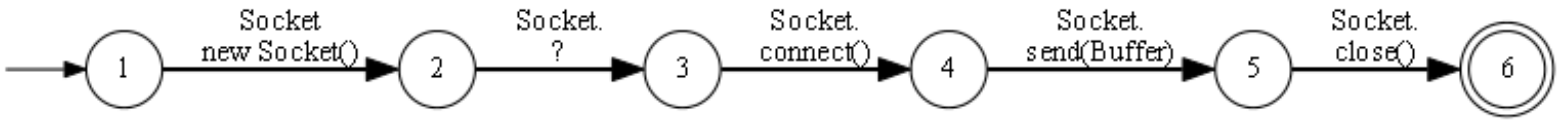
Program B

```
void bar(Socket s)
{
        while (s.canRead())
        {
                s.read();
        }
}
```

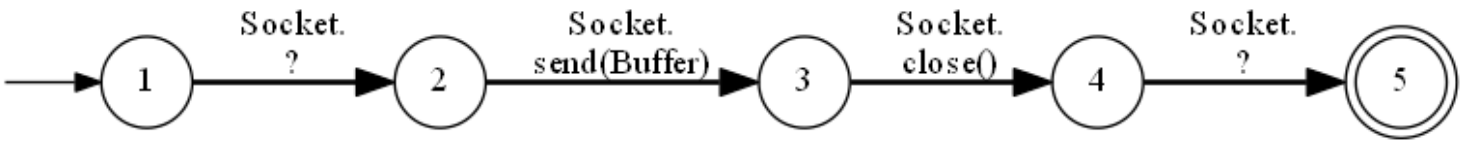
```
void zoo(Socket s, Buffer b)
{
        s.send(b);
        s.close();
}
```

Mine each one for API usage

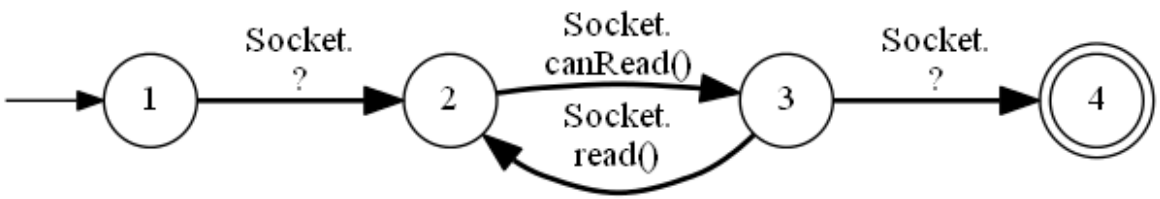
A_1



B_1

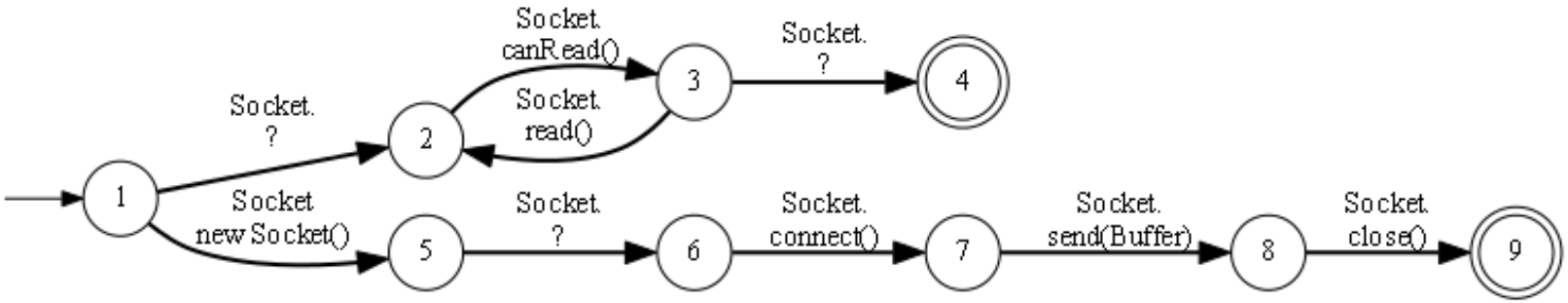


B_2

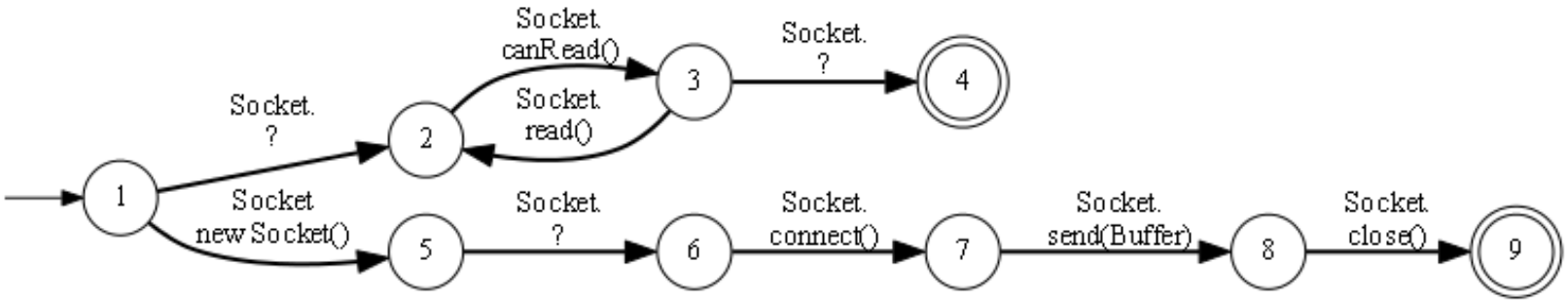


Join them to create a database

- First thing's first: $B_1 \leq A_1$
- So $A_1 \sqcup B_1 = A_1$
- Which leaves us with $A_1 \sqcup B_2$:



Use unknown elimination to answer queries

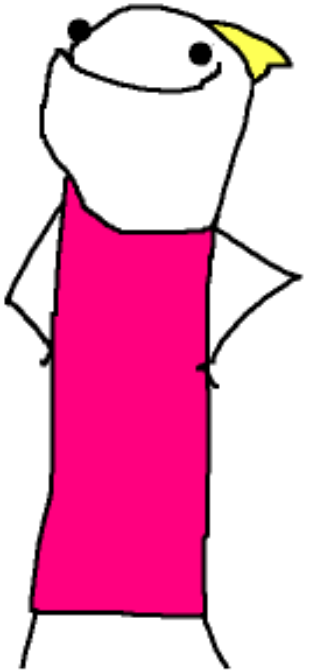


- We have our database!
  - We can give weights to transitions to weed out improbable or incorrect usage examples
- Now we can create queries, like $x \cdot read \cdot y$ which asks what to do around Socket.read
- Unknown elimination will find the assignment $\sigma(x) = z \cdot canRead$

$$\sigma(y) = canRead \cdot w$$

PRIME

- PRIME implements this analysis
- For Java, in Java
  - Uses Soot to analyze examples
  - Consolidates similar histories
  - Provides a comfy visual presentation
- Can be found at priming.sourceforge.net



Benchmarks

For presentation completion and code search:

- *Apache Commons CLI*
- *Apache Commons Net*
- *Apache Ant*
- *Eclipse JDT*
- *Eclipse GEF*
- *Eclipse UI*
- *JDBC*
- *WebDriver*

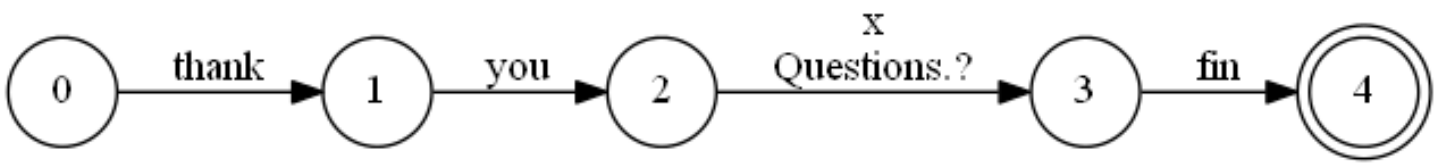
For verification, analyzed internal Google codebase snippets using *WebDriver*

Code search - simple queries

API used for the query, num of downloaded snippets	Query description	Query method	Number of textual matches	Tutorial's rank
WebDriver 9588 snippets	Selecting and clicking an element on a page	WebElement.click()	2666	3
Apache Commons CLI 8496 snippets	Parsing a getting a value from the command line	CommandLine.getValue(Option)	2640	1
Apache Commons Net 852 snippets	"connect -> login -> logout -> disconnect" sequence	FTPClient.login(String, String)	416	1
JDBC 6279 snippets	Creating and running a prepared statement	PreparedStatement.executeUpdate()	378	1
	Committing and then rolling back the commit	Connection.rollback()	177	4
Eclipse UI 17,861 mined snippets	Checking whether something is selected by the user	ISelection.isEmpty()	1110	2
Eclipse JDT 17,198 snippets	Create a project and set its nature	IProject.open(IProgressMonitor)	3924	1
Eclipse GEF 5981 snippets	Creating and setting up a ScrollingGraphicalViewer	GraphicalViewer.setEditPartFactory (EditPartFactory)	219	1

Where to next?

- Formalizing probabilistic symbolic automata PDSA
- Heuristic methods
- More explicit handling of code elements:
  - Conditional statements
  - Error handling


0
thank
1
you
2
x
Questions.?
3
fin
4