Microsoft Research

Each year Microsoft Research hosts hundreds of influential speakers from around the world including leading scientists, renowned experts in technology, book authors, and leading academics, and makes videos of these lectures freely available.

2016 © Microsoft Corporation. All rights reserved.
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

The JDBC API seems complex, how do I use it properly?

```java
Class.forName("com.microsoft.jdbc
String url = "jdbc:microsoft:sqlserv"
Connection conn = DriverManager.get
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

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

```
java.sql.Connection
```

Results 1-25 of about 59,182
What if the one we select has a bug?

```java
void doInsertStmt(String insert) {
    String connStr = "jdbc:sqlserver://localhost:integ;
    java.sql.Connection conn =
        java.sql.DriverManager.getConnection(connStr);
    java.sql.PreparedStatement preparedStmtInsert =
        conn.prepareStatement(insert);
    preparedStmtInsert.executeUpdate();
    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?

```java
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.executeUpdate(insert);
conn.commit();
}
```
We have to learn from more examples

Gazillions of programs and code snippets

API behavior index

- Find *canonical* examples
- Complete partial examples
- Search for code usage
- Validate user code

Use ALL the 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

```java
public void method(Something x) {
    x.f();
    x.g();
    x.h();
}
```
Objects are sometimes related (1)

```java
public void method(Something x) {
    x.f();
    BaseOfSomething y = x;
    y.g();
    y.h();
}
```
Objects are sometimes related (2)

```java
public void method(Something x) {
    x.f();
    SomethingElse s = x.createSomethingElse();
    s.h();
}
```
Dealing with the unknown

```java
public void method1() {
    Something x = new Something();
    x.f();
    transmogrify(x);
    x.g();
}
```
An unbounded number of histories

```java
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 \((\Sigma; Q; \delta; i; F; Vars)\)

- \(\Sigma\) is a finite alphabet
- \(Q\) is a finite set of states
- \(\delta\) is the transition relation, \(Q \times (\Sigma \cup Vars) \rightarrow Q\)
- \(i \in Q\) 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

\[ SL(A) = \{ sw \in (\Sigma \cup Vars)^* | \delta(i, sw) \in F \} \]

- Words over \( \Sigma \) are **concrete** words
- Words over \( \Sigma \cup Vars \) are **symbolic** words

\[ SL(A) = \{ a, abxcdbxcd, ... \} \]
Assignment

An assignment $\sigma$ maps a variable $x$ in context $sw_1$, $sw_2$ ($sw_1, x, sw_2$) to a non-empty symbolic language.

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

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

$$\{\sigma(\epsilon, x, b) = db \}
\{\sigma(\epsilon, x, c) = ec \}$$
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:
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 \( \sigma_2 \) to \( w_2 \) there is an assignment \( \sigma_1 \) 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.

\[
\sigma(x) = d
\]

- Formally: \( w_1 \) is more partial than \( w_2 \) if for each assignment \( \sigma_2 \) to \( w_2 \) there is an assignment \( \sigma_1 \) 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 $\leq$

$A_1 \leq A_2$ if for every concrete assignment $\sigma_2$ of $A_2$ there exists a concrete assignment $\sigma_1$ of $A_1$ for which $\sigma_1(SL(A_1)) \subseteq \sigma_2(SL(A_2))$
The domain’s $\leq$

$A_1 \leq A_2$ if for every concrete assignment $\sigma_2$ of $A_2$ there exists a concrete assignment $\sigma_1$ 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 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}), (1, {5}), (4, {4}), (5, {5}), (6, {6})
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\})
$\leq$ is a Preorder

- $\leq$ is transitive and reflexive, but it is not antisymmetric:
- Example:

![Diagram showing a partial order](image)

- But to have a lattice, we need a partial order
The DSA/$\equiv$ 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 $\text{DSA}/\equiv$

- 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: $(\text{DSA}/\equiv,\sqsubseteq)$ 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:

```
0 → a, e → 1
1 → d → 2, e → 5
2 → b → 3, c → 4
3 →       
4 →       
5 → b, c → 6
6 → b, e → 7
7 →       
```

```
0 → a → 1, e → 5
1 → d → 2, e → 5
2 → b, c → 3
3 →       
4 →       
5 → b, c → 7
6 → b, e → 7
7 →       
```
Answering Queries

- Now we have a database representing our API
Consolidation: an example

We’d like to go from this:

```
0 → 1
  ↓ e
  ↓ x
  ↓ b
  ↓ c
  ↓ a

1 ← 2 ← 3
  ↑ b
  ↑ c
  ↑ d
  ↑ e

5 ← 6 ← 7
  ↑ b
  ↑ c
  ↑ d
```

To this:

```
0 → 1
  ↓ d
  ↓ e

1 ← 2 ← 3
  ↑ b

5 ← 6 ← 7
  ↑ b
```
Dealing with the unknown

```java
public void method1() {
    Something x = new Something();
    x.f();
    transmogrify(x);
    x.g();
}
```
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 $\sigma$ that would make $\sigma(\text{SL}(Q)) \subseteq \text{SL}(A)$
- 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 \((\epsilon, x, b) \rightarrow a\)
• 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

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

Program B

```java
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

A₁
 Socket new Socket() → 1
 Socket ? → 2
 Socket connect() → 3
 Socket send(Buffer) → 4
 Socket close() → 5
 Socket close() → 6

B₁
 Socket ? → 1
 Socket send(Buffer) → 2
 Socket close() → 3
 Socket close() → 4
 Socket ? → 5

B₂
 Socket ? → 1
 Socket canRead() → 2
 Socket read() → 3
 Socket ? → 4
Join them to create a database

- First thing’s first: $B_1 \leq A_1$
Mine each one for API usage

A1

1. Socket
2. newSocket()
3. Socket
4. connect()
5. Socket
6. send(Buffer)
7. close()

B1

1. Socket
2. send(Buffer)
3. close()
4. Socket
5. ?

B2

1. Socket
2. ?
3. Socket
4. canRead()
5. Socket
6. read()
7. ?
8. 4
Join them to create a database

- First thing’s first: \( B_1 \leq A_1 \)
- So \( A_1 \cup B_1 = A_1 \)
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
Mine each one for API usage

A₁
1. Socket
2. newSocket
3. Socket
?  connect
4. Socket
send(Buffer)
5. Socket
close
6.

B₁
1. Socket
?  send(Buffer)
2. Socket
close
3. Socket
close
4. Socket
?  send(Buffer)
5.

B₂
1. Socket
?  send(Buffer)
2. Socket
canRead
3. Socket
read
4.
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 $x \cdot \text{read} \cdot y$ which asks what to do around Socket.read
- Unknown elimination will find the assignment
  $$
  \sigma(x) = z \cdot \text{canRead}
  $$
  $$
  \sigma(y) = \text{canRead} \cdot w
  $$
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

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

```java
public void method(Something x) {
    x.f();
    x.g();
    x.h();
}
```
An unbounded number of histories

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