MILCOM 2016

SECURE COMMUNICATIONS AT THE SPEED OF CYBER

Introduction to Graphical Program Analysis Using Atlas

Suresh C. Kothari
Richardson Professor
Department of Electrical and Computer Engineering

Ben Holland, Iowa State University

Acknowledgement: Team members at Iowa State University and EnSoft, DARPA contracts FA8750-12-2-0126 & FA8750-15-2-0080

BALTIMORE, MD • NOVEMBER 1–3, 2016

Module Outline

- Successfully write graph traversals
  - Example: Forward, Reverse, Set Operations
- Understand basics of Atlas program graphs
  - Structure, Control Flow, Data Flow
  - Attributes, Tags
- Explore query composability/reusability
- Locate additional resources
- Build some analysis utilities

Interactive reasoning with graph models

- eXtensible Common Software Graph (XCSG) schema
  - Heterogeneous, attributed, directed graph data structure as an abstraction to represent the essential aspects of the program's syntax and semantics (structure, control flow, and data flow), which are required to reason about software.
  - Expressive query language for users to write composable analyzers
  - Results computed in the form of subgraphs defined by the query, which can be visualized or used as input in to other queries

Thought Experiment - Given the following program what graph(s) could we produce?

```
public class MyClass {
   public static void A() {
      B();
   }
   public static void B() {
      C();
   }
   public static void C() {
      B();
      D();
   }
   public static void D() {
      G();
      E();
   }
   public static void E() {}
   public static void F() {}
   public static void G() {}
}
```

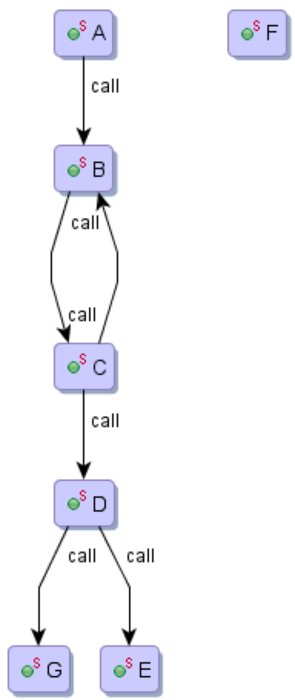
Control Flow (summary)

A *calls* B
B *calls* C
C *calls* B
C *calls* D
D *calls* G
D *calls* E

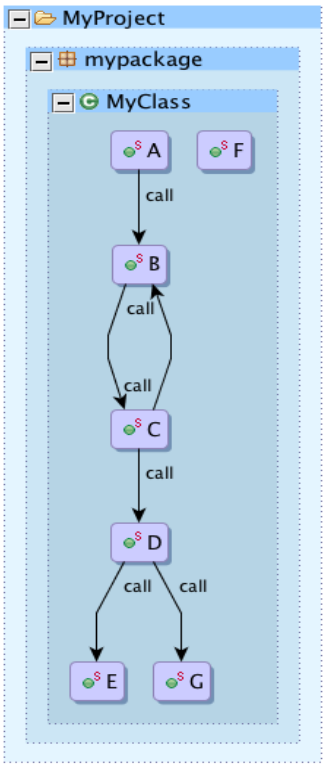
Structure

MyProject *contains* mypackage
mypackage *contains* MyClass
MyClass *contains* methods: A, B, C, D, E, F, G

Call Summaries



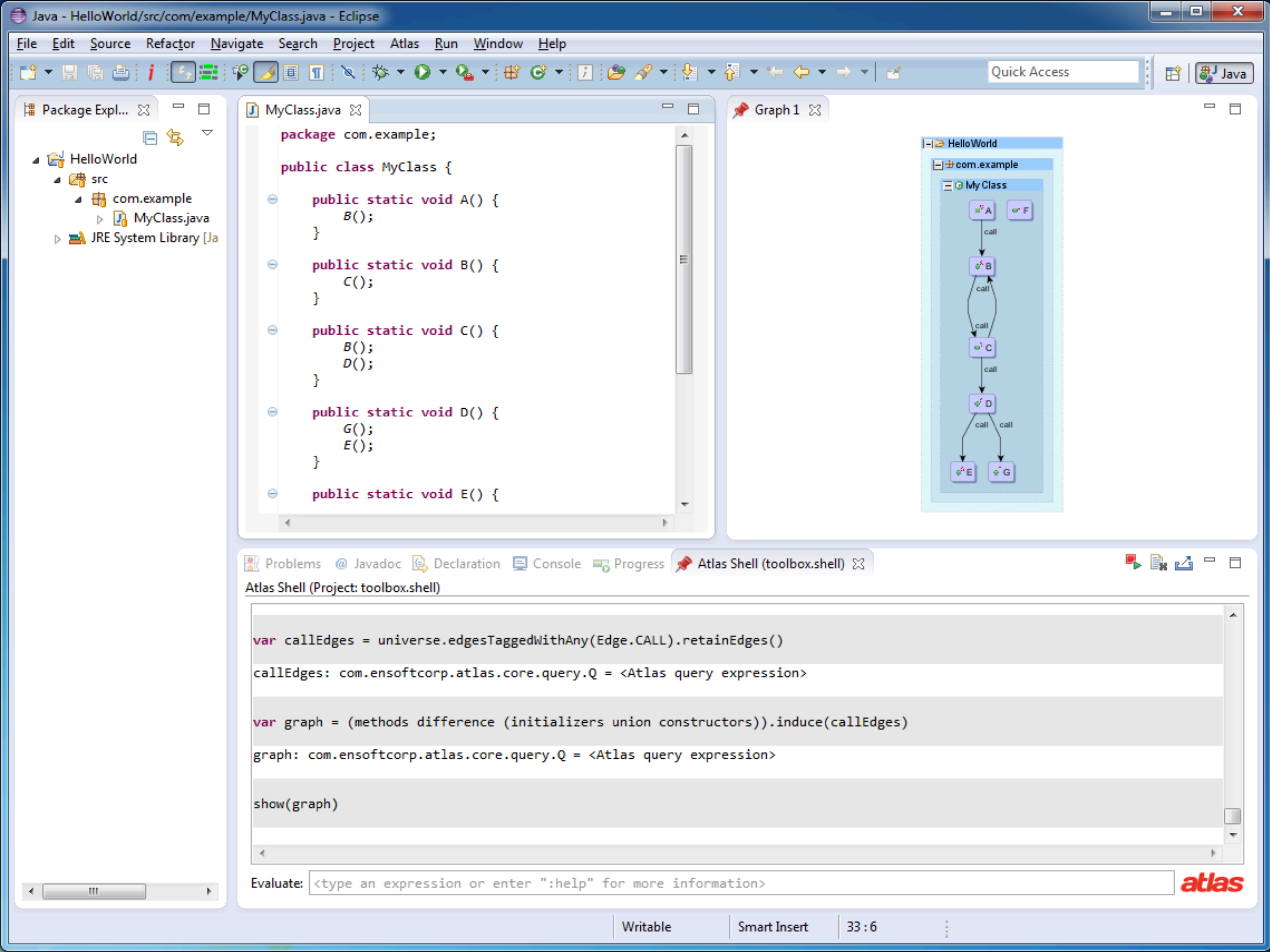
Call and Structure



Basic Queries

- Map the Workspace project "MyProject"
- Execute the following queries on the Atlas Shell (we will discuss what they mean later)

```
var containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)
var app = containsEdges.forward(universe.project("MyProject"))
var appMethods = app.nodesTaggedWithAny(XCSG.Method)
var initializers = app.methods("<init>") union app.methods("<clinit>")
var constructors = app.nodesTaggedWithAny(XCSG.Constructor)
var callEdges = universe.edgesTaggedWithAny(XCSG.Call)
var q = (appMethods.difference(initializers.union(constructors))).induce(callEdges)
show(q)
```


Java - HelloWorld/src/com/example/MyClass.java - Eclipse
File Edit Source Refactor Navigate Search Project Atlas Run Window Help
Quick Access
Java
Package Expl...
HelloWorld
src
com.example
MyClass.java
JRE System Library [Ja
MyClass.java
package com.example;
public class MyClass {
public static void A() {
B();
}
public static void B() {
C();
}
public static void C() {
B();
D();
}
public static void D() {
G();
E();
}
public static void E() {
Graph 1
HelloWorld
com.example
MyClass
A
F
B
C
D
E
G
call
Problems
Javadoc
Declaration
Console
Progress
Atlas Shell (toolbox.shell)
Atlas Shell (Project: toolbox.shell)
var callEdges = universe.edgesTaggedWithAny(Edge.CALL).retainEdges()
callEdges: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>
var graph = (methods difference (initializers union constructors)).induce(callEdges)
graph: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>
show(graph)
Evaluate:
<type an expression or enter ":help" for more information>
atlas
Writable
Smart Insert
33 : 6

- Declare a few variables to represent different methods in our example graph.
- Note that you can also use the “selected” variable after clicking on the Atlas graph or corresponding source file element.

var A = app.methods("A")
var B = app.methods("B")
var C = app.methods("C")
var D = app.methods("D")
var E = app.methods("E")
var F = app.methods("F")
var G = app.methods("G")

…alternatively…

var A = selected
var B = selected
…

Note: In the following examples you will need to pass the result to the “show” method on the Atlas Shell to view the results.

Example: show(q.forward(D))

Forward Traversals

q.forward(origin)

Selects the graph reachable from the given nodes using the forward transitive traversal. Includes the origin in the resulting graph query.

q.forward(D) outputs the graph D→E and D→G.

q.forward(C) outputs the graph C→B→C, C→D→E and C→D→G.

q.forwardStep(origin)

Selects the graph reachable from the given nodes along forward paths of length one. Includes the origin in the resulting graph query.

q.forwardStep(D) outputs the graph D→E and D→G.

q.forwardStep(C) outputs the graph C→B and C→D.

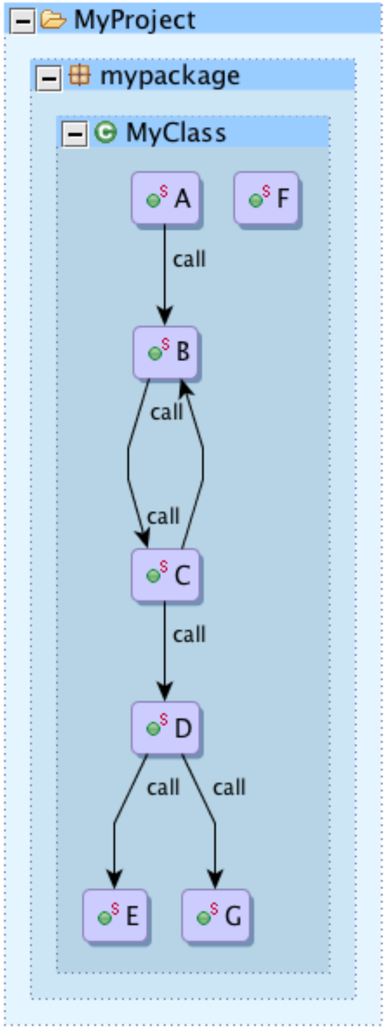
q.forwardStep(F) outputs the graph only F.

q.successors(origin)

Selects the immediate successors reachable from the given nodes Does not include the origin unless it succeeds itself. The result does not includes edges.

q.successors(C) outputs: {D, B}

q.successors(F) outputs: Empty graph

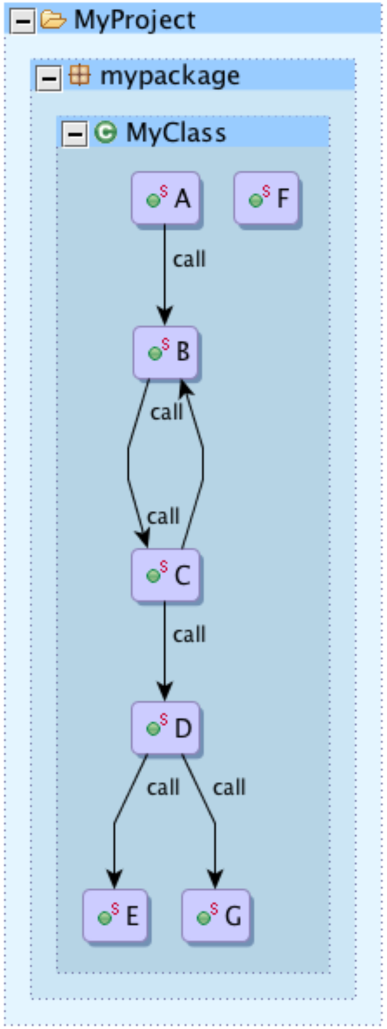


Reverse Traversals

q.reverse(origin)
Selects the graph reachable from the given nodes using the reverse transitive traversal. Includes the origin in the resulting graph query.
q.reverse(D) outputs the graph D←C←B←A and D←C←B←C.
q.reverse(C) outputs the graph C←B←A and C←B←C.

q.reverseStep(origin)
Selects the graph reachable from the given nodes along reverse paths of length one. Includes the origin in the resulting graph query.
q.reverseStep(D) outputs the graph D←C.
q.reverseStep(C) outputs the graph C←B.

q.predecessors(origin)
Selects the immediate predecessors reachable from the given nodes. Does not include the origin unless it precedes itself. The result does not include edge.
q.predecessors(C) outputs: {B}
q.predecessors(F) output: Empty graph.



Set Operations

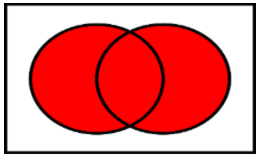
q.union(q2...)

Yields the union of nodes and edges of *this* graph (*q*) and the *other* graphs (*q2*).

B.union(C) outputs a graph with nodes B and C.

A.union(B, C) outputs a graph with nodes A, B, and C.

q.union(C) outputs the entire graph.

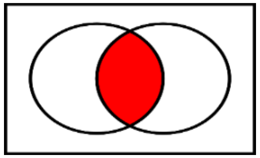


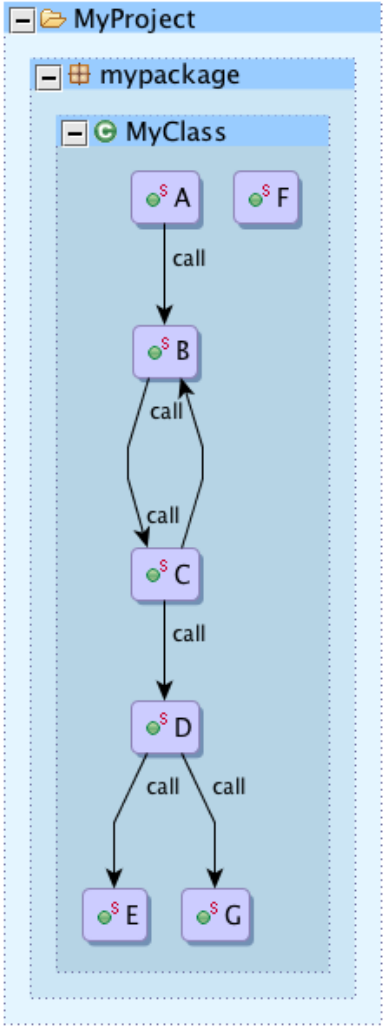
q.intersection(q2...)

Yields the intersection of nodes and edges of *this* graph (*q*) and the *other* graphs (*q2*).

A.intersection(B) outputs an empty graph.

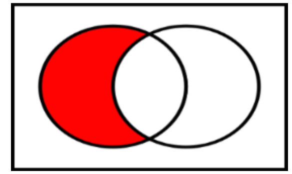
q.intersection(C) outputs a graph with only the node C.





Set Operations

q.difference(q2...)

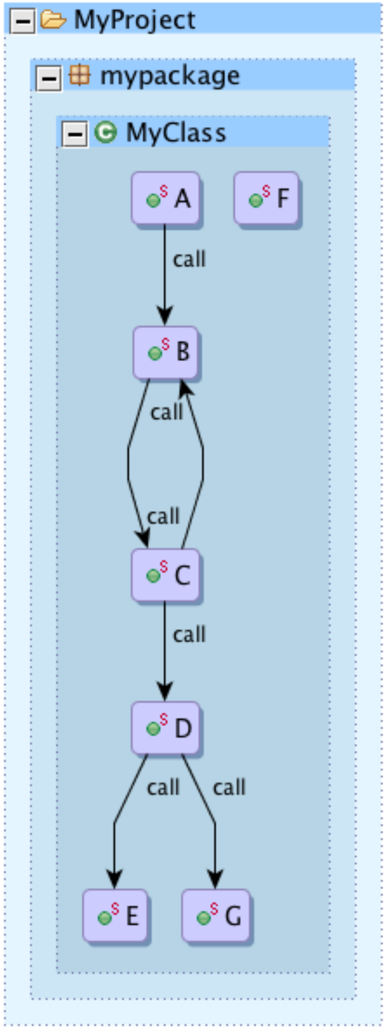


Selects *q*, excluding nodes and edges in q2. Removing an edge necessarily removes the nodes it connects. Removing a node removes the connecting edge as well.

B.difference(C) outputs a graph with only the node B.

B.difference(A, B) outputs an empty graph.

q.difference(C) outputs the shown graph without the node C and any edges entering or leaving node C.



q.differenceEdges(q2...)

Selects q, excluding the edges from q2.

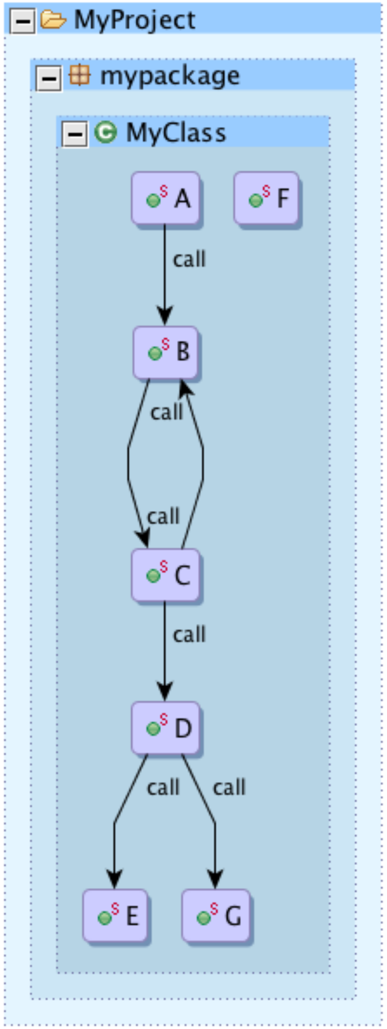
q.differenceEdges(q) outputs only the nodes A,B,C,D,E,F,G.

q.differenceEdges(q.forwardStep(B)) outputs the graph A→B, C→B, C→D, D→E, D→G, and F (the edge B→C is removed from the original graph).

Between Traversals

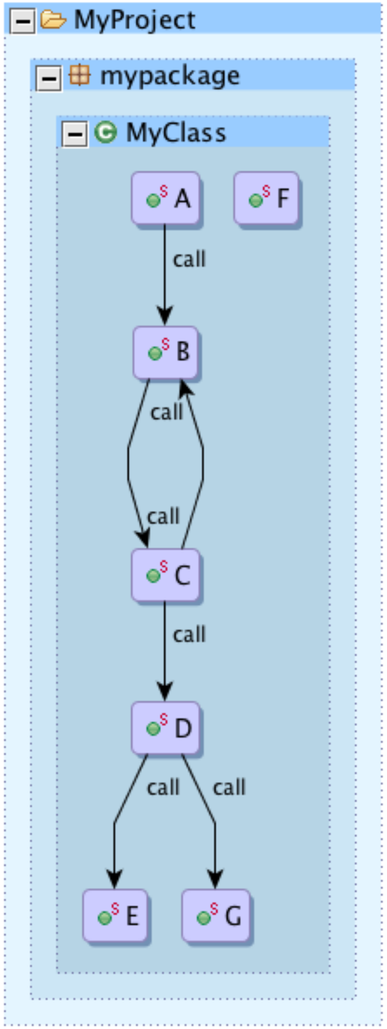
q.between(fromX, toY)
Selects the subgraph containing all paths starting from a set X to a set Y.
q.between(C, A) outputs Empty graph.
q.between(C, E) outputs the graph C→D→E, C→B→C.

q.betweenStep(fromX, toY)
Selects the subgraph containing all paths of length one starting from a set X to a set Y.
q.betweenStep(C, D) outputs the graph C→D.
q.betweenStep(D, C) outputs Empty graph.
q.betweenStep(C, E) outputs Empty graph.



Note: A possible implementation of betweenStep could be:
q.forwardStep(fromX).intersection(q.reverseStep(toY))

Graph Operations



q.leaves()

Selects the nodes from the given graph with no successors.

q.leaves() outputs {E, F, G}.

q.roots()

Selects the nodes from the given graph with no predecessors.

q.roots() outputs {A, F}.

q.retainNodes()

Selects all nodes from the graph, ignoring edges.

q.retainNodes() outputs {A,B,C,D,E,F,G}.

q.retainEdges()

Retain only edges and nodes connected to edges.

q.retainEdges() outputs the shown graph without F

q2.induce(q)

Adds edges from the given graph query *q2* to *q*.

var q2 = B.union(C)

q2.induce(q) outputs the graph B→C→B.

Graph Elements

- In Atlas a Q object can be thought of as a recipe to a *constraint satisfaction problem* (CSP). Building and chaining together Q's costs you almost nothing, but when you ask to see what is in the Q (by showing or evaluating the Q) Atlas must evaluate the query and execute the graph traversals.
- The evaluated result is a Graph. A Graph is a set of GraphElement objects. In Atlas both a node and an edge are GraphElement objects.

```
Graph graph = q.eval();
AtlasSet<Node> graphNodes = graph.nodes();
AtlasSet<Edge> graphEdges = graph.edges();
```

GraphElement Attributes

- GraphElement objects (nodes/edges) can have attributes
- An attribute is a key that corresponds to a value in the GraphElement attribute map.
  - An attribute on almost all nodes and edges is *XCSG.name*.

```
for(Node graphNode : graphNodes){
  String name = (String) graphNode.attr().get(XCSG.name);
}
```

  - Another common attribute is the source correspondence that stores the file and character offset of the source code corresponding to the node or edge. Double clicking on a node or edge takes us to the corresponding source code!

Selecting Graph Elements by Attributes

- Attributes can be used to select GraphElements (nodes/edges) out of a graph.
  - For example from the graph we can select all method nodes with the attribute key *XCSG.name* that have the value "main".

    Q mainMethods = q.selectNode(XCSG.name, "main");

  - We could also select all array's with 3 dimensions.

    Q 3DimArrays = q.selectNode(XCSG.arrayDimension, 3);

Tags: A Special Kind of Attribute

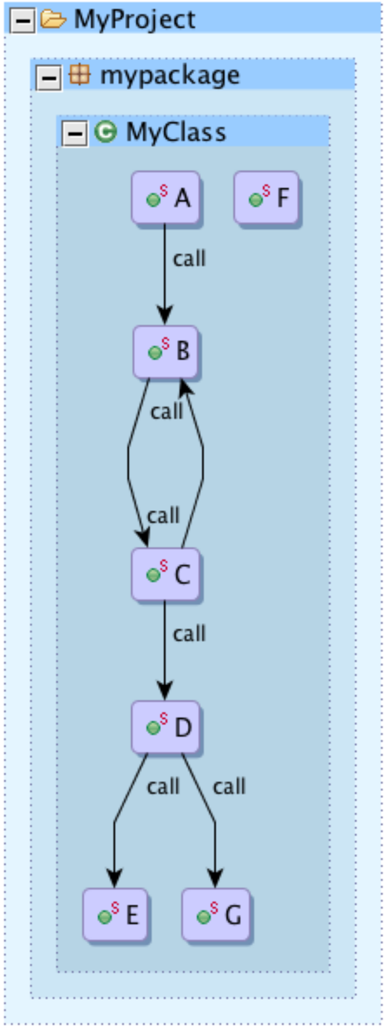
- A Tag is an attribute whose value is TRUE (T)
- The presence of a tag denotes that a node or edge is a member of a set.
  - For example, all method nodes are tagged with *XCSG.Method*.
- Atlas provides several default tags such as *XCSG.Method* that should be used to make code cleaner (and safer from possible Schema changes in the future!).

Selecting by Tags

q.nodesTaggedWithAny(...)
Selects the nodes tagged with at least one of the given tags.
q.nodesTaggedWithAny(XCSG.Method) outputs: {A,B,C,D,E,F,G}.
q.nodesTaggedWithAny(XCSG.Class) outputs: empty graph.
q.nodesTaggedWithAny(XCSG.Method, XCSG.Class) outputs: {A,B,C,D,E,F,G}.

q.nodesTaggedWithAll(...)
Selects the nodes tagged with all of the given tags.
q.nodesTaggedWithAll(XCSG.Method) outputs: {A,B,C,D,E,F,G}.
q.nodesTaggedWithAll(XCSG.Class) outputs: empty graph.
q.nodesTaggedWithAll(XCSG.Method, XCSG.Class) outputs: empty graph.

Note: The output contains only the nodes.



Selecting by Tags

q.edgesTaggedWithAny(...)

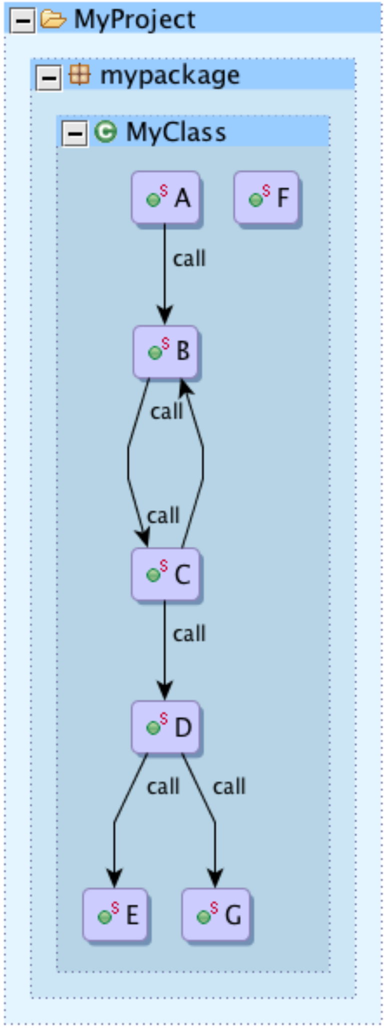
Selects edges tagged with at least one of tags. Includes all nodes.

q.edgesTaggedWithAny(XCSG.Call) outputs: the shown graph (*q*).

q.edgesTaggedWithAll(...)

Selects edges tagged with all of the given tags. Includes all nodes.

q.edgesTaggedWithAll(XCSG.Call) outputs: the shown graph (*q*).



Note: The output contains nodes connected by edges.

Chaining Queries

- We can chain queries to form more complex queries
- Q objects may contain multiple nodes and edges (so an origin can include multiple starting points).
- A second look at the queries we started the example with:

1. First create a subgraph (called containsEdges) of the universe that only contains nodes and edges connected by a *contains* relationship. The Atlas map is heterogeneous, meaning there are many edge and node types. Here we are specifying that we want edges that represent a *contains* relationship from a parent node to a child node.

var containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)

2. We then define yet another subgraph (called *app*) which contains nodes and edges declared under the MyProject project.

var app = containsEdges.forward(universe.project("MyProject"))

3. From the app subgraph we select all nodes that are methods.

var appMethods = app.nodesTaggedWithAny(XCSG.Method)

4. From the subgraph *app* we select all method nodes named "<init>" (instance initializer methods) or "<clinit>" (static initializer methods). We are using a query method called methods(String methodName) that selects methods that have a name that matches the given string. We will explore more query methods later.

var initializers = app.methods("<init>") union app.methods("<clinit>")

Note that since this is Scala the ".", "(", ")", and ";" characters are implied. So the above could also be written as the following:

var initializers = app.methods("<init>").union(app.methods("<clinit>"));

5. From the app subgraph we select all nodes that are constructors.

var constructors = app.nodesTaggedWithAny(XCSG.Constructor)

6. From the universe create a subgraph (called callEdges) that only contains nodes and edges connected by a call relationship.

var callEdges = universe.edgesTaggedWithAny(XCSG.Call)

7. Define graph to be the methods in the app ignoring initializers and constructors with call edges added in where they exist.

var q = (appMethods difference (initializers union constructors)).induce(callEdges)

8. Evaluate and display the graph query.

show(q)

Atlas Schema

- To become proficient in wielding Atlas, you should have:
  - Firm understanding of Extensible Common Software Graph (XCSG) schema
  - Firm understanding of the language you are analyzing (Java source, Jimple, C)
- Examples:
  - How do we detect an inner class with XCSG?
  - No tag for inner class, inner class is defined by a contains relationship.
    - *containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)*
    - *topLevelClasses = containsEdges.successors(universe.nodesTaggedWithAny(XCSG.Package))*
    - *innerClasses = containsEdges.forward(topLevelClasses).difference(topLevelClasses)*
  - What about Java vs. Jimple?
    - No concept of inner classes in bytecode

Atlas Schema Resources

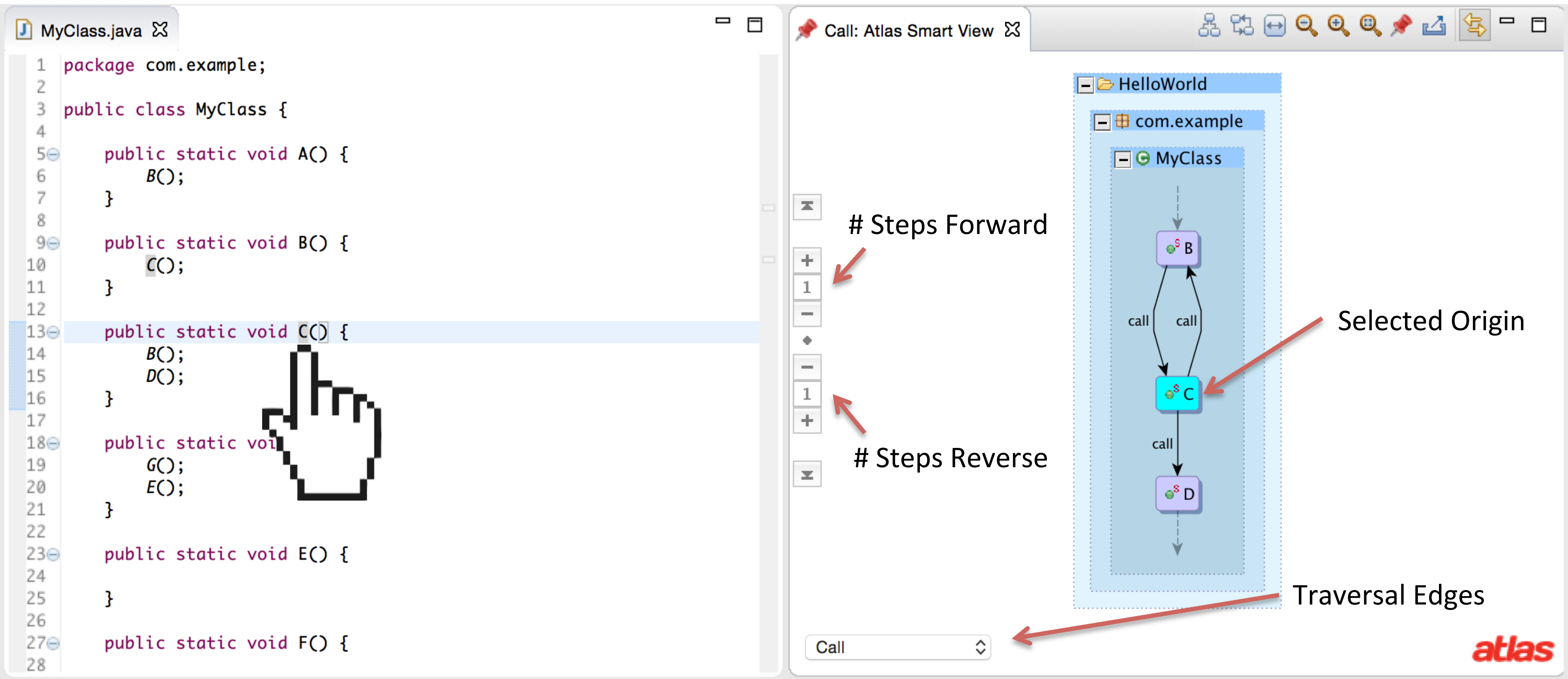
- http://ensoftatlas.com/wiki/Extensible_Common_Software_Graph
- Eclipse → Show Views → Other… → Atlas → Element Detail View
- Atlas Shell (test out queries on the fly!)
- Atlas Smart Views (interactive graphs)

SECURE COMMUNICATIONS AT THE SPEED OF CYBER

BALTIMORE, MD • NOVEMBER 1–3, 2016

Atlas Smart Views

- Smart Views automate queries for common tasks.



Lab Challenges

- Data structures are a very important part of how we write and understand programs. Java programs tend to make heavy use of collections (java.util.Collection).
- Trivia: Is java.util.Map a collection?
  - How could we check with Atlas?
- Can we discover all "uses of custom collections" in a Java program?
  - Define a custom collection to be a collection type defined in the application and a use to be a new allocation of that type.
  - Could we write a program to automatically answer this?