Traversal Graphs: Characterization and Efficient Implementation

Ahmed Abdelmeged Therapon Skotiniotis Panagiotis Manolios Karl Lieberherr

College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.
{mohsen,skotthe,pete,lieber}@ccs.neu.edu

Abstract

Adaptive Programming (AP) provides programmers with a graph abstraction of their OO program's structure (called a *class graph*) and encapsulates traversals and computation along traversals as adaptive methods. An adaptive method consists of a selector specification, called a *strategy*, that selects paths in the class graph, and a *behavior* definition with advice on nodes that are executed while traversing the selected paths.

Any AP implementation faces the following problem: given a strategy, S and a class graph G compute the set of paths in G that satisfy S. We represent the set of valid paths in G as a graph, dubbed a *traversal graph*.

In this paper we give the first characterization of, and a new algorithm for computing, traversal graphs. We describe our new algorithm that is more efficient – the size of the generated traversal graph is in the worst case as large as the traversal graph generated by previous approaches – and more general – our algorithm is defined over general graphs rather than on the class graph directly. We apply our algorithm in an AP setting by transforming class graphs to general graphs. We further show that our algorithm generates a traversal graph that satisfies our characterization.

1. Introduction

Adaptive Programming (AP) (**?**) has been developed as an extension to Object-Oriented (OO) programming and aims at facilitating evolutionary development by limiting dependencies between traversal code and a program's class structure. An adaptive program consists of three loosely coupled modules:

- the *class hierarchy*, represents the class structure of an OO program as a graph, called the *class graph*, including classes, inheritance edges and containment edges,[1]
- a *strategy*, a succinct navigation specification based on the program's class hierarchy, and,
- *behavior*, computation to be performed during the traversal of the program's object structure, guided by the strategy.

As a concrete example consider a Java program that contains anonymous classes and the task of refactoring these anonymous classes to nested classes.[2] Using AP we can view the abstract syntax tree (AST) as a graph and we can use the strategy,

```
from ClassDef via AnonClassDef to *.
```

where `ClassDef` and `AnonClassDef` are both AST nodes, to select the relevant subgraph for our refactoring. The strategy is a specification that describes paths in the AST. Put differently, the strategy is a selector that picks all the paths in the AST that can reach any node (denoted by a *) starting from a `ClassDef` node and going through a `AnonClassDef` node. Strategies are used along with visitor classes (**?**) (or function objects (**?**)) to define *adaptive methods*. The following adaptive method can be defined inside `ClassDef`.

```
public void refactorAnon()
  from ClassDef via AnonClassDef to *
  (AnonToInnerVis);
```

Calling the method `refactorAnon` starts traversing from the current object guided by the strategy specification and executes any applicable visitor methods in `AnonToInnerVis` along the way. Strategies allow for a high-level path specification that minimizes dependencies between the program's structure and operations on the program's structure.

An efficient implementation of adaptive methods depends on the efficiency with which we calculate all paths selected by a strategy in a given graph, i.e., all the paths in the AST that start from `ClassDef` go through an `AnonClassDef`

GPCE'08 Tennessee.

[1] A restricted form of UML's Class Diagram that does not include methods.

[2] A similar refactoring operation is supported by Eclipse (**?**).

and terminate at any AST node. In order to calculate these paths we need to take into account the notion of types and subsumption in the context of an OO language. For example, consider the situation where `ClassDef` is an abstract class, then only paths that go through fields inherited by, or defined in, `ClassDef` through `AnnonClassDef` are valid. A path to `AnnonClassDef` that goes through a field defined in a *sublcass* of `ClassDef` is *invalid*.

Previous approaches to AP (**??**) gave an algorithm for calculating valid paths from a strategy specification and a type hierarchy. The algorithm represents the set of valid paths as a graph referred to as a *traversal graph*. Given a strategy (represented as a graph), a type hierarchy, a name map between the strategy's node set and the type hierarchy's node set and a constraint map mapping nodes and edges to predicates returns a traversal graph. The predicates on nodes and edges capture the conditions under which the algorithm can use the node or edge during traversal.

In this paper we present

- a formal characterization of traversal graphs, and,
- a new algorithm for constructing traversal graphs given a selector specification and a graph that:
  1. does not restrict the form of supported strategies (**??**),
  2. is defined on general graphs; we show for the case of class hierarchies and AP how with a series of transformations the problem is reduced to a selection problem on general graphs, and
  3. is more efficient;our algorithm generates a traversal graph that is, in the worst case, as large as the traversal graph generated by previous algorithms (**?**).

We also show that our algorithm returns a traversal graph that satisfies our characterization. We believe that similar reductions to the ones we defined for AP can be defined for other selector languages, e.g., XPath queries, pointcut matching in Aspect Oriented Programming languages etc.

1.1 Paper Organization

The rest of this paper is organized as follows: in section 2 we define our notation and provide the characterization for traversal graphs. In section 3 we describe our traversal construction algorithm and in section 4 we show that our algorithm satisfies the characterization given in section 2. In section 5 we describe a transformation from class graphs to general graphs and in section 6 we conclude our paper with a discussion on related work and future directions.

2. Characterization of Traversal Graphs

In (**?**) a traversal graph refers to a graph constructed as part of the reachability algorithm that represents all paths in a graph G selected by a strategy specification s. In this section we dissassociate the notion of a traversal graph from the details of a strategy and a class graph. We provide a characterization for traversal graphs based on two general graphs as a *conforms* relationship that must hold between the two graphs and the properties that should hold between sets of paths found in these two graphs.

Before we give our characterization we introduce our notation and provide some definitions used in the remainder of our paper.

We use the standard notation for general graphs where a *graph* G is an ordered pair (N, E) where N is a set of nodes and E is a binary relation on N. We also use $G.nodes$ and $G.edges$ for the sets N and E of the graph G. An *edge* $e \in E$ is an ordered pair and we use $e.source$ and $e.target$ to denote the first and second element of e respectively. Given a node $n \in N$, we use the notation $n.in$ to denote $\{e \mid e \in E \wedge e.target = n\}$ and $n.out$ to denote $\{e \mid e \in E \wedge e.source = n\}$.

A *path* P is a nonempty sequence $v_1, \ldots, v_m$ of nodes where v_1 is called the source node (also denoted as $P.source$) and v_n is called the target node (also denoted as $P.target$). We use $P.nodes$ to denote the set of nodes in the path P and $P.edges$ to denote the set $\{(v_i, v_{i+1}) \mid i \in [1, m)\}$. A path P is *in* graph G iff $P.nodes \subseteq G.nodes$ and $P.edges \subseteq G.edges$.

Selector graphs are special kinds of graphs that have the notion of source and target nodes. Also, any node in the selector graph has to be on a path from a source node to a target node. Consider for example the strategy

```
from ClassDef via {MethodSig, FieldDef}
to TypeName.
```

in the context of Java's AST as in section 1. The strategy selects all type names reachable from a `ClassDef` but going through either a method signature or a field definition. The selector graph for this strategy has `ClassDef` as the only source and `TypeName` as the only target. On any path from source to target we either have to go through a `MethodSig` node or a `FieldDef` node. Selector graphs depend on another graph. For example names used in the strategy are names of nodes in the class graph and the "edges" implicitly defined with `via` and `to` are edges in the transitive closure of the edges in the class graph

To capture selector graphs we first define *stgraphs* and then define selector graphs as stgraphs for a given graph G where the nodes of the selector graph are a subset of the nodes of G and the edges of the stgraph are a subset of the transitive closure of the edges in G.

An stgraph $\mathcal{S}$ is a graph with the following conditions:

- A non empty set $S \subseteq \mathcal{S}.nodes$ of source nodes, denoted $\mathcal{S}.sources$.
- A non empty set $T \subseteq \mathcal{S}.nodes$ of target nodes, denoted $\mathcal{S}.targets$.

such that every node $n \in \mathcal{S}.nodes \backslash (\mathcal{S}.sources \cup \mathcal{S}.targets)$ is reachable from a node in $\mathcal{S}.sources$, and every node $\mathcal{S}.targets$ is reachable from n.

We define the predicate $expansion_G(P, Q)$ to hold iff:

- P and Q are paths. and
- $Q.nodes \subseteq P.nodes$. and
- P is obtainable from Q by inserting zero or more nodes from $G.nodes$.
- $P.source = Q.source$.
- $P.target = Q.target$.

We use the expression $EXP_G(R)$ where G is a graph and R is a set of paths over $G.nodes$[3], denotes the set of paths $\{P \mid \exists Q \in R : expansion_G(P, Q)\}$.

A path in an stgraph $\mathcal{S}$ from a source to a target is denoted $STP_{\mathcal{S}}$. Formally, $STP_{\mathcal{S}} = \{P \mid P.source \in \mathcal{S}.sources \wedge P.target \in \mathcal{S}.targets \wedge Pin\mathcal{S}\}$

The function $\mathcal{S}.stpaths$ where $\mathcal{S}$ is an stgraph, denotes the possibly infinite [4] set of all $STP_{\mathcal{S}}$ in $\mathcal{S}$.

The function $G.paths$ where G is a graph, denotes the possibly infinite set of all paths in G.

A *selector graph* $\mathcal{S}_G$ for a graph G, is an stgraph such that:

- $\mathcal{S}_G.nodes \subseteq G.nodes$
- $\mathcal{S}_G \subseteq G^+$.

An *overlay map* $\mathcal{M} : G_O \rightarrow G_B$ is a function [5] that maps nodes of the *overlayed* graph G_O to nodes in the *base* graph G_B. We say that graph G_O *conforms-to* graph G_B using the overlay map $\mathcal{M}$ iff: $\forall\ (n_1, n_2) \in G_O.edges : (\mathcal{M}(n_1), \mathcal{M}(n_2)) \in G_B.edges$. We overload overlay maps to map sets of nodes. We also overload overlay maps to map edges, paths and subgraphs of G_O to G_B only if G_O conforms-to G_B.

A *traversal graph* of a graph G and a selector graph $\mathcal{S}_G$ is denoted $T_G(\mathcal{S}_G)$. $T_G(\mathcal{S}_G)$ is an stgraph overlayed on G using an overlay map $\mathcal{M}$ such that:

(A) $T_G(\mathcal{S}_G)$ conforms-to G using $\mathcal{M}$. and

(B) $\mathcal{M}\ (T_G(\mathcal{S}_G).stpaths) = EXP_G(\mathcal{S}_G.stpaths)$.

Simply put, $\mathcal{M}\ (T_G(\mathcal{S}_G))$ are the paths of $T_G(\mathcal{S}_G)$ mapped to their underlying graphs. $\mathcal{M}\ (T_G(\mathcal{S}_G))$ contains *all* paths that are: expansion of some path in $\mathcal{S}_G$, and *in* [6] G.

3. Traversal Graph Construction

In this section we present a new algorithm for constructing traversal graphs. The entry point for our algorithm is the function $computeTG$ which takes as arguments, a graph g and a selector graph sg defined over the nodes of g. $computeTG$ starts by creating an initial traversal graph that is identical to sg, line 1. Every node in the initial traversal graph is mapped to an underlying node in g. The method $cgNode$ can be used to retrieve the underlying g node of a traversal graph node. $computeTG$ then replaces every edge e in the initial traversal graph by another traversal graph tge, resulting in the final traversal graph.

Function `computeTG`(*cg, sg*)

```
   //g is a graph.
   //sg is a selector graph over the nodes of g.
1  tg = createTG (sg.nodes, sg.edges, sg.sources,
   sg.targets);
2  foreach e ∈ tg do
3      src = e.source.cgNode ();
4      trgt = e.target.cgNode ();
5      tge = constructTG (g, src, trgt);
6      replace (tg, e, tge);
7  end
```

The function $createTG$ takes a set of nodes in g, a set of edges over these nodes, and two other subsets of these nodes defining source and target nodes. the arguments define an stgraph. $createTG$ creates a traversal graph that is identical in structure to the stgraph passed in. Every node in the newly created graph maps to an underlying node in the set of input nodes. $createTG$ is used by $computeTG$ to create the initial traversal graphs and is also used by $constructTG$ to construct traversal graphs to replace the edges of the initial graph.

Function `createTG`(*nodes, edges, sources, targets*)

```
   //nodes is a set of nodes.
   //edges is a edges over nodes.
   //sources targets are subsets of nodes.
1  tg = empty;
2  rmap = empty;
3  foreach n ∈ nodes do
4      tgNode = new TGNode (n);
5      if sources.contains (n) then
6          tg.sources.add (tgNode);
7      end
8      if targets.contains (n) then
9          tg.targets.add (tgNode);
10     end
11     tg.add (tgNode);
12     rmap.add (n, tgNode);
13 end
14 foreach e ∈ edges do
15     tgsrc = rmap.lookup (e.source);
16     tgtrgt = rmap.lookup (e.target);
17     tg.add (tgsrc, tgtrgt);
18 end
```

[3] But paths in R are not necessarily paths in G

[4] Due to loops in $\mathcal{S}$.

[5] not necessarily injective.

[6] not just over the nodes of G

The function $constructTG$ takes as arguments a graph g and two nodes src, $trgt$. $constructTG$ first finds the subgraph of g that contains all nodes reachable from src and can reach $trgt$. Then, it creates a traversal graph based on these nodes using the $constructTG$ function defined above. The functions $collectForward$ and $collectBackward$ will be described shortly. The function $filter(edges, nodes)$ takes in a set of edges and a set of nodes and returns the subset of edges whose end points are in the given set of nodes.

Function `constructTG`(*cg, src, trgt*)

```
//cg is a graph.
//src, trgt are two nodes in cg.nodes.
1  backNodes = collectBackward (g.nodes, empty, src);
2  legalNodes = collectForward (backNodes, empty, trgt);
3  legalEdges = filter (g.edges, legalNodes);
4  return createTG (legalNodes, legalEdges, empty, empty);
```

The function $collectForward$ takes in a subset of the nodes of graph g, an auxiliary argument that holds the set of nodes collected so far, and a source node src. $collectForward$ performs a depth first search, starting from src, on the subset of g defined by the given nodes. $collectForward$ collects all the node that it encounters and eventually returns them. The function $collectBackward$ is identical to $collectForward$ except that $collectBackward$ traverses g backwards.

Function `collectForward`(*nodes, collectedNodes, src*)

```
//nodes is the set of legal nodes.
//collectedNodes is the set of nodes collected so far.
1  if not (collectedNodes.contains (src)) then
2      collectedNodes.add (src);
3      foreach e ∈ src.out do
4          if nodes.contains (e.target) then
5              collectForward (nodes, collectedNodes, e.target);
6          end
7      end
8  end
9  return collectedNodes;
```

The function $replace$ takes as arguments a traversal graph tg, an edge $edge$ in tg, and another traversal graph tge that is intended to replace $edge$ in tg. Throught $replace$, we use $equals$ to compare traversal nodes. What $equals$ actually does is to compare the two underlying nodes in g rather than comparing the two traversal graph nodes directly. $replace$ removes $edge$ from tg. Then $replace$ adds all nodes from tge to tg except for two nodes in tge that correspond to $edge.source$ and $edge.target$. Finally, $replace$ adds the appropriate edges from $edge.source$, $edge.target$ to the rest of tge.

Function `replace`(*tg, edge, tge*)

```
//tg is a traversal graph.
//edge is in tg.
//tge is a traversal graph to replace edge in tg.
1  source = edge.source;
2  target = edge.target;
3  tg.removeEdge (source,target);
4  foreach n ∈ tge.nodes do
5      if not (n.equals (source)) and not (n.equals (target)) then
6          tg.add (n);
7      end
8  end
9  foreach e ∈ tge.edges do
10     if e.source.equals (source) and (e.target.equals (target)) then
11         tg.addEdge (source,target);
12     else if e.source.equals (source) then
13         tg.addEdge (source,e.target);
14     else if e.target.equals (target) then
15         tg.addEdge (e.source,target);
16     else tg.addEdge (e.source,e.target);
17 end
```

4. Analysis

In this section we prove that the algorithm described in section 3 produce traversal graphs that satisfy the characterization given in section 2. Theorem 1 states that the output of $constructTG$ satisfies condition 1 in the characterization. Theorem 2 states that the output of $constructTG$ satisfies condition 2 in the characterization.

Lemma 1. *Given a graph G, two nodes src and $trgt$ in $G.nodes$ then $constructTG(G, src, trgt)$ conforms-to G.*

Proof. $constructTG$ constructs a traversal graph from a subset of the nodes and edges in G. Edges (nodes) are added to the graph returned by $constructTG$ only if there are corresponding edges (nodes) in G. □

Theorem 1. *Given a graph G and a selector graph $\mathcal{S}_G$ defined over $G.nodes$ then $computeTG(G, \mathcal{S}_G)$ conforms-to G.*

Proof. $computeTG$ starts with the strategy graph and replace every edge in it with a traversal graph that is constructed using $constructTG$. Therefore, every edge in the

traversal graph returned by $computeTG$ has a corresponding edge in G. Hence, the theorem follows. □

Lemma 2. *Given a graph G, two nodes src and $trgt$ in $G.nodes$, let tge be $constructTG(G, src, trgt)$, let $\mathcal{M}$ be $tge.map$, then $\mathcal{M}$ (tge) contains all paths in G from src to $trgt$.*

Proof. Suppose that P is a path in G from src to $trgt$ and $\mathcal{M}$ (tge) does not contain P.

let G_1 be a subgraph of G that contains all nodes that are reachable from src in G and can reach $trgt$ in G. By construction, $\mathcal{M}$ $(tge) = G_1$.

G_1 contain all the nodes and edges that could be on any path from src to $trgt$. Therefore, G_1 must contain P. But, by assumption, G_1 does not contain P. □

Theorem 2. *Given a graph G and a selector graph $\mathcal{S}_G$ defined over $G.nodes$ let tg be $computeTG(G, \mathcal{S}_G)$, let $\mathcal{M}$ be $tg.map$, then $\mathcal{M}$ $(tg.paths) = EXP_G(\mathcal{S}_G.paths)$.*

Proof. Suppose that P is in G and in $EXP_G(\mathcal{S}_G.stpaths)$ and P is not in $\mathcal{M}$ $(tg.stpaths)$.

Since $P \in EXP_G(\mathcal{S}_G.stpaths)$, P can be broken into a sequence of subpaths $P_1, \ldots, P_k$ such that:
$\forall i \in [1, k) : P_i.target = P_{i+1}.source$.
$Q = P_1.source, \ldots, P_k.source, P_k.target$ in $\mathcal{S}_G.stpaths$.

Since Q is in $\mathcal{S}_G.stpaths$, $Q.nodes \subseteq \mathcal{S}_G.nodes$ and $Q.edges \subseteq \mathcal{S}_G.edges$. Therefore, $Q.nodes \subseteq \mathcal{M}$ $(tg.nodes)$ and every edge in $Q.edges$ is replaced with a traversal graph tge that, according to lemma 2, contains all paths from $e.source$ to $e.target$. This is done at $computeTG$ lines 4, 5. Since the final traversal graph tg with all original traversal strategy edges replaced. We have that $\mathcal{M}$ (tg) contains all P_i's. Therefore, $\mathcal{M}$ (tg) contains P. But, this contradicts our assumption that $\mathcal{M}$ (tg) does not contain P. Hence the theorem follows.

□

5. Transforming Class Graphs to General Graphs

Our algorithm works on general graphs and stgraphs, however the graph representation of an OO program in AP, called a class graph*, is not a general graph but a specialization that encodes inheritance and subtyping. In this section we present a transformation from class graphs to general graphs. Stgraphs are defined on the result of our transformation and not on the class graph directly. We discuss this issue in section 6.*

5.1 Class Graphs

A class graph in AP represents the structure of an OO program that contains two kinds of nodes, concrete and abstract, and two kinds of edges, inheritance and containment.

We formally define class graphs as a labeled graph $G = (V, E, \mathcal{L})$ where

- *$\mathcal{L}$ is a fixed, finite set of labels that correspond the an OO program's field names and includes the symbol $\diamond$, denoting an inheritance edge.*
- *V is the set of class names defined in an OO program. The node set can be further partitioned into V_c the set of concrete classes and V_a the set of abstract classes.*
- *E is the set of edges one for each field in a class, labeled with the field name, and one for each inheritance relationship between two classes labeled with the symbol $\diamond$. The edge set can be further partitioned into two $E_\diamond$ the set of inheritance edges and E_c the set of containment edges.*

The definition of edges and paths is extend to accommodate labels in the usual way, e.g., $n_1 \xrightarrow{l} n_2$ is written as (n_1, l, n_2) and a paths $n_1 \xrightarrow{l_1} n_2 \cdots n_{k-1} \xrightarrow{l_k} n_k$ is written as $n_1 l_1 n_2 \cdots n_{k-1} l_k n_k$ where $(n_i, l_i, n_{i+1}) \in G.edges$. A class n_1 that inherits from n_2 is represented as $n_1 \xrightarrow{l} n_2$. We define superclass *as a relation between any two nodes n_1 and n_2 for which there exists a path $p = p_0 \cdots p_n$ (possibly empty) such that $(p_i, \diamond, p_{i+1}) \in G.edges$.*

For the purposes of this presentation we model Java-like languages with the following properties:

- *for a node n each outgoing edge has a distinct name (except edges with the label $\diamond$),*
- *$E_\diamond$ is acyclic,*
- *for any node n and any two nodes n_a and n_b that are superclasses of n then either n_a is a superclass of n_b or n_b is a superclass of n_a*

5.2 Class Graph Transformation

The transformation takes as input a class graph with the following restrictions:

- *for each edge $n_1 \xrightarrow{l} n_2$ such that $l = \diamond$ then n_2 is abstract, and,*
- *for all edges $n_1 \xrightarrow{l} n_2$ such that $l \neq \diamond$ then n_1 is concrete.*

Given any class graph G we first use the simplification algorithm *from (**?**) to transform it into a class graph G' that satisfies the preceding restrictions. The simplification algorithm guarantees that any object graph O that is an instance of G is also an instance of G'.*

*More formally using the definitions from (**?**) an object graph $O = (V', E', L')$ is a labeled directed graph where nodes are objects and $L' \subseteq \mathcal{L}$. An object graph O is an instance of a class graph $C = (V, E, \mathcal{L})$ under a function $Class : V' \to V$, if the following conditions hold*

- *for all objects $o' \in V'$, $Class(o') \in V_c$*

- *for each object o' the set of outgoing edges is exactly the set of all outgoing containment edges (including inherited edges) of $Class(o')$*
- *for each $o_1 \xrightarrow{l} o_2 \in E'$, $Class(o_1)$ has an edge $n_1 \xrightarrow{l} n_2$ such that n_1 is superclass of o_1 and n_2 is a superclass of o_2.*

The second transformation replaces each labeled edge with a new node holding the same name as the label on the replaced edge and connected to the source and target nodes of the replaced edge. In the case of subclass edges the new node name is a concatenation of the superclass name the symbol $\diamond$ and the subclass name, i.e., $u \xrightarrow{\diamond} v$ creates a the new node $\diamond_u^v$. Given $C = (V, E, \mathcal{L})$ we create a new class graph $C' = (V', E')$ as follows

- *initialize V' to V and E' to $\emptyset$*
- *for each $n_i \xrightarrow{l_i} n_{i+1}$*
  - *if $l_i \neq \diamond$, then $V' = V' \cup \{n_i, n_{i+1}, l_i\}$ and $E' = E' \cup \{(n_i, l_i), (l_i, n_{i+1})\}$*
  - *else $V' = V' \cup \{n_i, n_{i+1}, \diamond_{n_i}^{n_{i+1}}\}$ and $E' = E' \cup \{(n_i, \diamond_{n_i}^{n_{i+1}}), (\diamond_{n_i}^{n_{i+1}}, n_{i+1})\}$*

We can partition V' into the set of original nodes in C, V and the set of newly created nodes that represent the original edge labels $V_{\mathcal{L}}$. The resulting graph is a bipartite graph on which we can use two colors to distinguish the original graph nodes (white) and the nodes that represent labels in the original class graph (black). A path p' in C' that starts and ends with a white node, consists of alternating colored nodes and the path can be mapped to a path p in C by considering pairs of three consecutive nodes starting from the p''s start node, i.e., white,black,white. Each triple represents an edge in p.

6. Conclusions and Future Work

In this paper we presented the first characterization for traversal graphs. We also presented a new algorithm for computing travresal graphs as well as two transformations of a restricted yet useful subset of class graphs to and from general graphs.

Acknowledgments

This work is supported in part by Grantham Mayo Van Otterloo, LLC.