Traversal Semantics in Object Graphs

Mitchell Wand and Karl Lieberherr*

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wand@ccs.neu.edu, lieber@ccs.neu.edu
http://www.ccs.neu.edu/home/{wand, lieber}

October 12, 2001

Abstract

Traversal through object graphs is needed for many programming tasks. We show how this task may be specified declaratively at a high level of abstraction, and we give a simple and intuitive semantics for such specifications. The semantics, while intuitive, requires existential quantification over all object graphs to represent the fact that during traversal we have only local knowledge of the graph. Since we cannot search explicitly over all object graphs, we reduce the infinite quantification to a graph problem over the class graph. The algorithm is implemented in a Java library called DJ. The system is aspect-oriented in that it makes a clear separation between traversal and behavior, and it is adaptive in that it relies on reflection to fill in the details of the traversal algorithm at run-time.

1 Introduction

A common task in object-oriented programming is to take an object o in an object graph and find all the objects o' that are reachable from o according to some search criterion. In [8] we introduced the idea of using an expression in a declarative language to express such a search criterion. We refer to such an expression as a

*Work supported by the National Science Foundation under grants CCR-9804115 and CCR-0097740 and by ARPA and BBN under agreement F33615-00-C-1694.

traversal specification. For example, the simplest traversal specification is of the form "`from A to B`". This means: given an object *o* of class `A`, find all the objects of class `B` that are reachable from *o*.

In order to find all the B-objects, we need to engage in search. This paper presents an algorithm for converting a declarative search criterion to a search algorithm. We give such an algorithm for two languages of traversal specification. The primary technical problem in the algorithm is a coherent treatment of inheritance.

The key idea of the search algorithm is that from an object *o*, we search only those edges from *o* that *might* lead to a `B`-object, based on the class structure of the object graph. We may explore objects that are not on the path to a `B`-object, but that is an unavoidable consequence of searching based only on meta-information.

Once we have converted the declarative criterion to a search algorithm, we can identify the subgraph explored by the search algorithm, and invoke visitor objects either just at the target nodes or at all the nodes in the subgraph.

The concept of automated traversal generation using succinct representation was introduced in [15] and is extensively treated in [8], where it is the key idea of Adaptive Programming (AP). Our most complex language of traversal specifications, called *strategy graphs*, is introduced in [10] together with an efficient implementation.

This paper provides a self-contained description of the semantics and algorithms for Adaptive Programming in a few pages. Instead of refering to class graph paths as [10] does, the basic meaning is defined directly in terms of object graphs. By dealing directly with the search algorithm in the object graph, it avoids the complications of the traversal histories of [15].

We begin in section 2 with a simple example. Section 3 discusses the relationship between DJ and aspect-oriented programming. Section 4 presents our notation, and section 5 presents our model of classes and objects. In section 6, we formulate the basic traversal problem in the terms of our model. The key to the problem is to find a set $\mathrm{FIRST}(c,c')$ such that $e \in \mathrm{FIRST}(c,c')$ iff it is possible for an object of class c to reach an object of type c' by a path beginning with an edge e. Section 7 gives semantics to the core traversal specifications of DJ and shows how to interpret them as search algorithms. Section 8 completes the story by showing how to compute the FIRST sets statically. Section 9 discusses related work. Finally, section 10 presents some conclusions and ideas for further development.

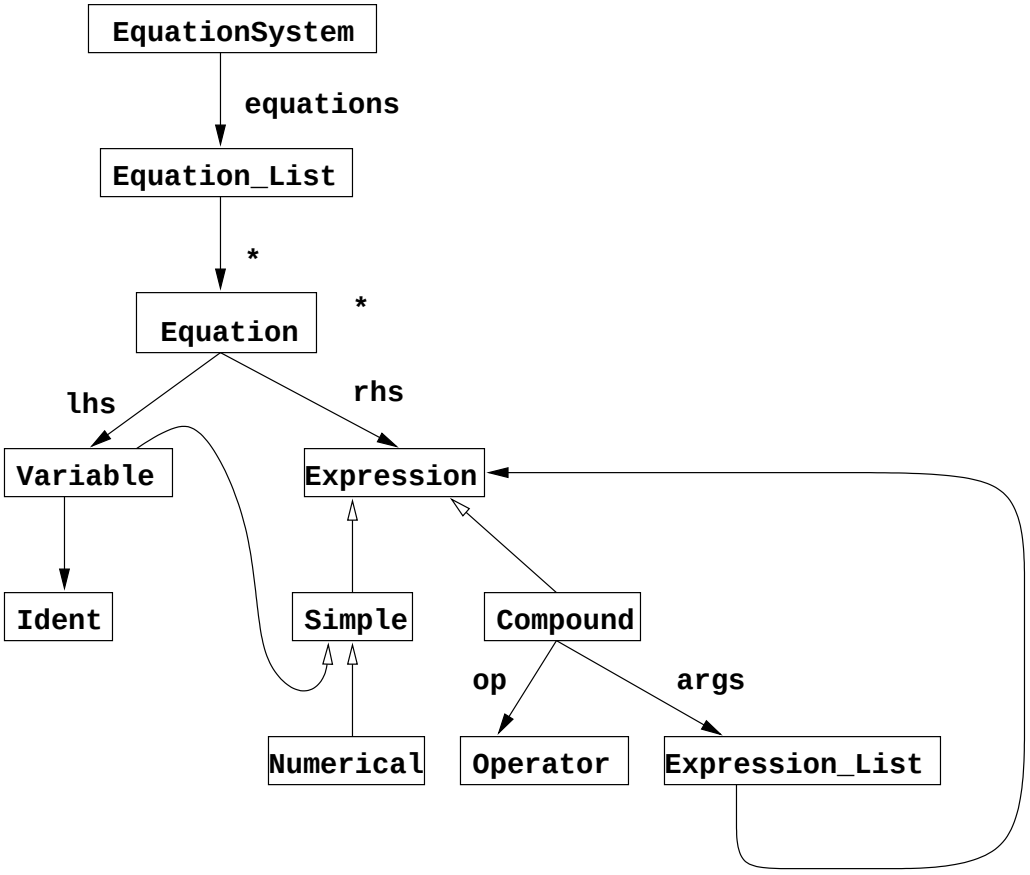

Figure 1: Class graph for `EquationSystem`.

2 A Simple Example

We illustrate automated traversal generation with a simple example, written in Java using the DJ library.

Consider an equation system that contains equations with a left-hand side and a right-hand side. We use a mathematical notation whose structure is defined by the class graph in Figure 1. A typical set of equations might be

```
X1 = (X2 + X3)
X2 = (X5 + (X3 * X1))
X3 = (X5 + (7 * 5))
```

We say a variable is *defined* if it appears on the left-hand side of an equation

and that it is *used* if it appears on the right-hand. In the example, variables `X1`, `X2` and `X3` are defined, and variables `X2`, `X3` and `X5` are used. The purpose of our Java program is to collect the defined and used variables.

To solve this problem, we identify two traversals. The first, `usedVars`, may be defined by[1]

```
from EquationSystem through Expression through rhs to Variable
```

We wish to visit all the `Variable` objects that are reachable from a given `EquationSystem` object via a path that goes through some `Expression` object and through some `rhs` edge.

The second, `definedVars`, is expressed by the traversal specification

```
from EquationSystem bypassing Expression to Variable
```

This illustrates the `bypassing` clause. This specifies that the path from the equation system to the variable should not go through an `Expression` object.[2]

The code for this task is shown in figure 2. The method `collectVars` takes as arguments the current class graph `cg` (which is constructed by reflection in `SystemClient` by calling `new ClassGraph()`), and a string `travspec` that specifies the traversal to be performed. It first creates a visitor `v` to implement the behavior that is to be performed during the traversal. In this case, the visitor methods are executed upon reaching the target (`before(Variable)`) and at completion of the search (`getReturnValue`). Once the visitor is created, `collectVars` then initiates the traversal by calling the `traverse` method of the current class graph `cg`, passing as arguments the object at which to start the traversal (`this`), the traversal to be performed (`travspec`), and the visitor to be executed along the way (`v`).

The execution of `cg.traverse(this, travspec, v)` follows the visitor pattern [4]. Every time the traversal reaches a node *n*, the traversal calls `v.before(`*n*`)`. Thus, when the node is a variable, the method `before(Variable v1)` (the "visitor method") is called, adding the node to the set.

The methods `getUsedVars` and `getDefinedVars` simply call `collectVars` with two different traversal specifications. One could easily construct more complex combinations, *e.g.* checking for undefined variables by first creating the set of

[1] We could as well have defined this set by `from EquationSystem through rhs to Variable`; the variant in the text illustrates multiple `through` clauses.

[2] We could as well have defined this set by `from Equation System through lhs to Variable`.

```
class EquationSystem{
  static String usedVarsSpec =
    "from EquationSystem through Expression through rhs to Variable";
  static String definedVarsSpec =
    "from EquationSystem bypassing Expression to Variable";
  HashSet collectVars(ClassGraph cg, String travspec){
    Visitor v = new Visitor(){
      HashSet return_val = new HashSet();
      void before(Variable v1){return_val.add(v1);}
      public Object getReturnValue(){return return_val;}
    };
    cg.traverse(this, travspec, v);
    return (HashSet)v.getReturnValue();
  }
  HashSet getUsedVars(ClassGraph cg) {
    return this.collectVars(cg, usedVarsSpec);
  }
  HashSet getDefinedVars(ClassGraph cg) {
    return this.collectVars(cg, definedVarsSpec);
    }
}

class SystemClient {
  void use() {
    EquationSystem s = new EquationSystem(...);
    ClassGraph cg = new ClassGraph();
    HashSet usedVars = s.getUsedVars(cg);
    HashSet definedVars = s.getDefinedVars(cg);
    ...
  }
}
```

Figure 2: Finding the variables in an equation system

defined variables, and then checking each used variable for undefinedness when it is visited.[3]

Unlike the case of the ordinary visitor pattern, no preparation in the underlying objects is necessary in order to use this library.

Note that we must traverse the entire right-hand side of each equation to find the used variables, even though no variable may be present in a subexpression, *e.g.* in `(7 * 5)` above, since on the basis of the class graph in figure 1, any expression might contain a variable. If we modified the class graph to add a subclass of `Expression` for, say, `CombinationOfNumericals`, which never contained a variable, then the traversal would not search such a subexpression. This distinction does not matter in this example, but it might if the visitor did non-trivial processing on the objects found during the search.

The DJ library includes a powerful language of traversal specifications and methods; these are detailed in [3]. An introduction to DJ, with emphasis on its reflective properties, may be found in [13].

3 DJ as AOP

We may regard DJ as a variety of AOP. To understand how DJ fits into AOP, we first consider conventional programming. In conventional programming, a program consists of a set of procedures that are invoked *procedurally*, that is, from other procedures. Ordinary OOP fits into this model, since methods are a kind of procedure.

A key innovation of AOP, as embodied in systems like AspectJ, is a mechanism to allow procedures to be invoked *declaratively*. Such a mechanism identifies a set of events during the computation at which procedures may be invoked (the join points), a declarative language for specifying a set of such events (the "point cut designators"), and a means of associating procedures with such declaratively-specified sets. This is a dynamic join point model.

In DJ we have:

- the computation is the traversal. In DJ, this is also specified declaratively.
- the join points are the beginning and end of the traversal and the passage of the traversal through a node or edge in the object graph.

[3] A slightly better design would have `cg` be a static variable in `SystemClient` and having `collectVars` refer to `SystemClient.cg`. Here we have made `cg` a parameter to make this dependence explicit.

- the point cut designators allow specification of the following point cuts: the beginning and end of the traversal, the passage of the traversal through a node of a specified type, and traversal of an edge from one node of a specified type to another node of a specified type.

- procedures are associated with a point cut by making them into methods of a visitor. The name of the visitor and the type of its argument (*e.g.* `before(Variable v1)`) serve as the language of point cut designators. Here, as in Aspect J, data passed through the join point is bound to a parameter such as `v1`.[4]

Because the computation is specified declaratively, the mechanism for invoking the visitors can be hidden inside the implementation of the traversal. This allows the separation of the traversal behavior from the aspect or visitor behavior.

The method `collectVars` is an aspect in the sense of [6]: each is a modular unit of crosscutting implementation. It is a modular unit because it is a Java method. It is crosscutting, because a naive implementation would require detailed knowledge of many different classes.[5] We don't care how many classes are between `EquationSystem` and `Variable`. These details are not wired into the code, because they can be reconstructed at run-time via reflection. Furthermore, this architecture passes the test of *obliviousness* because they require no preparation in the underlying object model.

We call these methods *adaptive methods* because their behavior depends on the structure of the class graph. The separation of traversal specification (where to go), behavior (what to do), and class graph structure (context to evaluate in) allows the method to be reused.

4 Notation

We will be using relations as our fundamental tool, so we will need some notation for dealing with relations.

If A and B are sets, a relation from A to B is a subset R of $A \times B$. If $a, b \in R$, we will write $R(a,b)$, $a\ R\ b$, and $(a,b) \in R$ interchangebly. A relation from A to A is often called a relation *on* A.

[4] In the current version of DJ, an edge-traversal join point is specified by a visitor method signature like `cbefore(Equation eqn, String label, Expression exp)`. this join point occurs just before the traversal of any edge from an `Equation` object to a `Expression` object. It binds `eqn` and `exp` to the source and target objects, respectively, and `label` to the label of the edge.

[5] If the method were written in good object-oriented style following the Law of Demeter [9], the implementation would require methods in each of these classes.

We denote composition of relations by concatenation *e.g.* $x\ (RS)\ z$ iff there exists a y such that $x\ R\ y$ and $y\ S\ z$. We also write $x\ R\ y\ S\ z$. R^* denotes the reflexive transitive closure of R.

We often think of directed graphs as relations (and vice versa), so we write $C(c_1, c_2)$ or $c_1\ C\ c_2$ when there is an edge from c_1 to c_2 in C. We take as given the definition of a path in a directed graph.

5 The Model

Definition 1 *A* class graph *consists of a set C (of "classes"), a set E (of field names), for each $e \in E$ a relation (also named e) on classes ("has part named e"), and a reflexive, transitive relation $\leq$ on classes ("is a subclass of"). We write $C(c_1, c_2)$ iff there exists $e \in E$ such that $e(c_1, c_2)$.*

Each relation e codes the effect of finding the e part of an object. Usually the relation e is a partial function (that is, for any c_1, there is at most one c_2 such that $e(c_1, c_2)$), but we will not need this property. When $e(c_1, c_2)$, we sometimes say that c_1 has an e-part of type c_2. (The significance of the word "type" will be explained momentarily). We use $\leq$ to denote the reflexive, transitive closure of the inheritance relation, so $c \leq c'$ means that c is either the same as c' or is one of c''s descendants.

We use C to denote the entire class graph $\langle C, E, \leq \rangle$. We write $\geq$ for the inverse of $\leq$.

An object graph is a model of the objects, represented in the heap or elsewhere, and their references to each other:

Definition 2 *If C is a class graph, then an* object graph *of C consists of:*

1. *a set O (of "objects"),*
2. *a map* class $: O \to C$*, and*
3. *for each $e \in E$, a relation (also denoted e) on O*

such that if $e(o_1, o_2)$, then

$$\text{class}(o_1)\ (\leq e \geq)\ class(o_2)$$

We say that o is of type c *when* $\text{class}(o) \leq c$.

An object is of type c when its class is some class that is a descendant of c. This corresponds to the usual expectation in a typed object-oriented language: if a variable is of type c, its value is either null or is an object whose class is either c or a descendent of c.

The traversal of an edge labeled e corresponds to retrieving the value of the e field. Condition 3 captures the notion that every edge in the object graph is an image of a has-as-part edge in the class graph: There is an edge $e(o_1, o_2)$ in O only when there exist classes c_1 and c_2 such that o_1 is of type c_1, c_1 has an e-part of type c_2, and o_2 is of type c_2, that is,

$$\text{class}(o_1) \leq c_1\, e\, c_2 \geq \text{class}(o_2)$$

See figure 3.

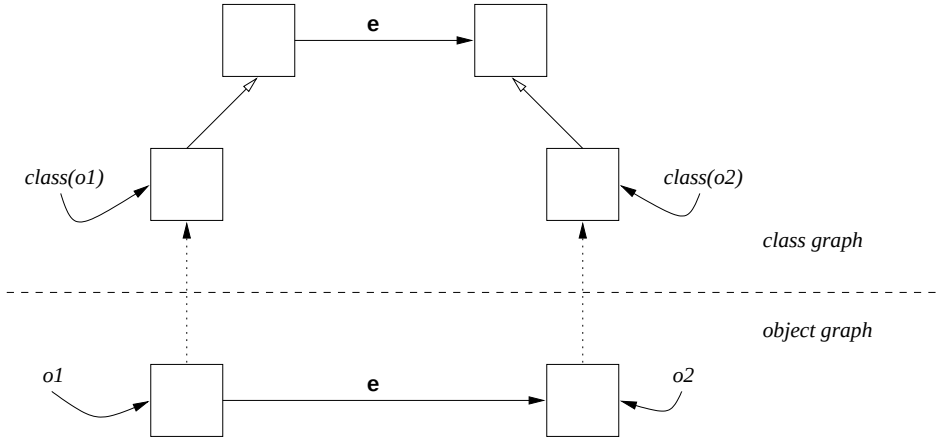

Figure 3: Typical link in an object graph. If there is an e-edge from $o1$ to $o2$, then there is a path $\text{class}(o1) \leq e \geq \text{class}(o2)$ in the class graph.

As we did for class graphs, we use O to denote the entire object graph whose set of objects is O.

All parts are optional (allowing for null values) or multi-valued (for a given object o_1, there may be many objects o_2 such that $e(o_1, o_2)$). The latter case allows us to handle collections: if class c_1 contains a field e that is a collection of objects of type c_2, we may represent this as $e(c_1, c_2)$ and use multi-valued edges in the object graph, rather than introduce the notion of collections into our model.

We might propose the additional condition that if $\text{class}(o_1) \leq c_1$ and $e(c_1, c_2)$, then there exists an o_2 such that $e(o_1, o_2)$ and $\text{class}(o_2) \leq c_2$. This means that

every edge predicted by the class graph exists in the object graph. All the theorems in the paper are true under this stronger definition, but the additional condition is undesirable because it rules out null fields.

6 The Problem

The computational problem we wish to study is the following:

We are at an object o of class c in object graph O and we wish to find all reachable objects of type c'. However, we have no information about the object graph other than that it is a legal object graph for C. Which edges must we explore in order to find all these objects?

We can formalize the problem as follows. For each pair of classes c and c', we need to find a set $\mathrm{FIRST}(c,c')$ such that $e \in \mathrm{FIRST}(c,c')$ iff it is possible for an object of class c to reach an object of type c' by a path beginning with an edge e. More precisely,

$$\begin{aligned}\mathrm{FIRST}(c,c') = \{e \in E \mid & \text{ there exists an object graph } O \text{ of } C \text{ and} \\ & \text{objects } o \text{ and } o' \text{ such that:} \\ & \quad 1.\ \mathrm{class}(o) = c, \\ & \quad 2.\ \mathrm{class}(o') \leq c' \\ & \quad 3.\ o\ eO^*\ o'\ \}\end{aligned}$$

The last condition, $o\ eO^*\ o'$ says that there is a path from o to o' in the object graph, consisting of an edge labelled e, followed by any sequence of edges in the graph.

Our lack of information about the actual object graph is represented by the existential operator. Since we cannot search explicitly over all object graphs, our goal is to find a static algorithm to compute these sets.

7 Traversal Algorithms

Before considering an algorithm for finding the FIRST sets, we consider some applications of these sets. We can use these sets to find not just reachable objects of a given type, but paths that pass through objects of a given type.

Definition 3 *Let $R = (c_1, \ldots, c_K)$ be a non-empty sequence of classes. Let $p = (o_1, \ldots, o_N)$ be a path in O. We say that p is an R-*path *iff there is a subsequence $o_{j_1}, \ldots, o_{j_K}$ of p such that for each i, o_{j_i} has type c_i, and $j_K = N$ (that is, the last element of the path must be part of the subsequence). We say an R-path is* minimal *iff it has no initial segment that is also an R-path.*

Thus an R-path is a path through the object graph that passes through objects of the types specified by R in the order specified by R. It may pass through objects of other classes along the way, but it must end at the endpoint of R. The sequence $R = (c_1, \ldots, c_n)$ corresponds to the DJ traversal directive `through` c_1 `through` c_2 `...to` c_n.[6]

Given an object o, we can visit all the endpoints of minimal R-paths starting at o as follows:

Algorithm *search*(o, R):

Let $c_1, \ldots, c_n$ be the classes such that $R = (c_1, \ldots, c_n)$.

1. If o does not have type c_1, then for each e in FIRST$(class(o), c_1)$, and each o' such that $e(o, o')$, do *search*(o', R).
2. If o has type c_1, and $R = (c_1)$, then do *visit*(o).
3. If o has type c_1, and $R = (c_1, c_2, \ldots, c_n)$, then do *search*$(o, (c_2, \ldots, c_n))$.

In case 1 we are not yet on a path, so we follow the FIRST edges to guide us to the first goal type c_1. We can find the required objects o' by retrieving the value of o's e-field. In case 2, we have reached the last goal type, so we visit the object. In case 3 we have reached the first goal type, so we recur on the rest of the goal path.

We could find all R-paths, by modifying step 2 to continue searching:

2′ If o has type c_1, and $R = (c_1)$, then do *visit*(o). Then for each e in FIRST$(class(o), c_1)$, and each o' such that $e(o, o')$, do *search*(o', R).

If the object graph is acyclic, these algorithms will always terminate, since every step either decreases the longest chain of links in the object graph or decreases the length of R. If the graph may be cyclic, then we need to mark each searched object with the state R in which it was reached, or else carry around the set of pairs (o, R) that we have already searched. This will allow us to avoid repeated visits to the same object in the same traversal state.

This algorithm is easily modified to accomodate the `bypassing` specification of DJ. An edge specification, like `through lhs` in our example, can be implemented by extending FIRST(c_1, c_2) to FIRST(c, e); this can be done by an easy modification of the algorithm in section 8.

[6] The DJ traversal `from` c_1 `through` c_2 `...` `to` c_n is identical to this traversal directive, except that it reports an error if the current object is not already of type c_1.

DJ allows the use of a graph, called a *strategy graph*, to specify a more complex traversal [10]. A strategy graph can be modeled as a non-deterministic finite-state automaton:

Definition 4 *A* strategy graph *is given by a set of states Q, a relation S on states, a map* class $: Q \to C$*, a set $QI \subseteq Q$ of initial states, and a set $QF \subseteq Q$ of final states. We denote such a strategy graph by S.*

Definition 5 *A path $p = (o_1, \ldots, o_N)$ in O is an S-*path *iff there is a subsequence $o_{j_1}, \ldots, o_{j_K}$ of p and a path $(q_1, \ldots, q_K)$ in S such that for each i, o_{j_i} has type* class(q_i)*, $j_K = N$, and $q_K \in QF$. As before, we say that an S-path is* minimal *iff it has no initial segment that is also an S-path.*

Given an object o, we can visit all the endpoints of minimal S-paths starting at o as follows, where Q' ranges over subsets of Q:

> **Algorithm** $search(o, S) : search'(o, IQ)$ where $search'(o, Q')$ for any $Q' \subseteq Q$ is defined by:
>
> > **Procedure** $search'(o, Q')$:
> >
> > 1. If $class(o) \not\leq$ class(q) for any $q \in Q'$, then for each $q \in Q'$, for each e in FIRST$(class(o), class(q))$, and each o' such that $e(o, o')$, $search'(o', Q')$.
> > 2. If $class(o) \leq$ class(q) for some $q \in Q' \cap QF$, then $visit(o)$.
> > 3. Otherwise let $Q'' = \{q \in Q' \mid o$ does not have type $class(q)\} \cup \{q' \mid \exists\, q \in Q'$ such that o has type class(q) and $S(q, q')\}$. Then $search'(o, Q'')$.

This algorithm is much like the preceding one, except that the set Q' maintains the state of a run through the non-deterministic automaton S. In step 1, we are not at a point on the subsequence, so we use the FIRST sets to search any edge that may lead us to any of the first goal classes. In step 2, we have reached a final state, so we visit the object we have reached. In step 3, we are at a set of states Q'. Some of those states represent goal classes that we have not yet reached. Other states are goal classes that we have now reached. We create the next set of goal classes by carrying along those that we have not yet reached, and taking a step in the automaton S for those that we have reached.

We typically start the algorithm with a unique start state and an object o that matches that state.

As before, when the object graph is acyclic, this always terminates. If the object graph can be cyclic, then we need to carry around a "seen" set, as above.

8 Calculating FIRST

We develop a static description of FIRST using a sequence of lemmas. This allows us to compute FIRST from the class graph. We start with a fixed class graph C.

Lemma 1 *There exists an object graph O of C and objects o_1, o_2 such that $O(o_1, o_2)$ iff* $\text{class}(o_1) \leq C \geq \text{class}(o_2)$.

Proof: The forward direction is immediate from the definition of an object graph of C. For the reverse direction, construct an object graph with two objects o_1 and o_2 and the specified link. The result is an object graph of C.

Lemma 2 *There exists an object graph O of C and objects o_1, o_2 such that $O^*(o_1, o_2)$ iff $class(o_1)\ (\leq C \geq)^*\ class(o_2)$.*

Proof: Again, the forward direction is immediate. For the reverse, induct on the definition of $(-)^*$, using the preceding lemma.

A picture of this situation is shown in figure 4.

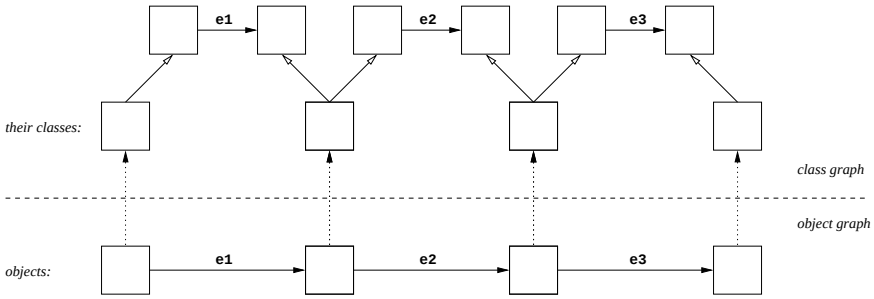

Figure 4: A path in an object graph. If there is a path in the from o_1 to o_2, then $class(o_1)\ (\leq C \geq)^*\ class(o_2)$.

Lemma 3 *Let c_1 and c_2 be classes. Then there exists an object graph O of C and objects o_1, o_2 such that* $\text{class}(o_1) \leq c_1$ *and* $\text{class}(o_2) \leq c_2$ *and $O^*(o_1, o_2)$ iff* $c_1 \geq \text{class}(o_1)\ (\leq C \geq)^*\ \text{class}(o_2) \leq c_2$.

Proof: Immediate.

Theorem 1

$$\text{FIRST}(c, c') = \{e \mid c\ (\leq e \geq)(\leq C \geq)^* \leq c'\}$$

Proof: We must show that there exists an object graph O of C and objects o and o' such that

1. $\text{class}(o) = c$
2. $\text{class}(o') \leq c'$
3. $o\ eO^*\ o'$

iff $c\ (\leq e \geq)(\leq C \geq)^* \leq c'$.

The forward direction is immediate from the definition of an object graph of C. For the reverse definition, consider a sequence of classes such that

$$c(\leq e \geq)c_1(\leq e_1 \geq)c_2(\leq e_2 \geq)\ldots(\leq e_{n+1} \geq)c_n \leq c'$$

Then construct an object graph with $n+2$ objects, of classes c, c_1, ...,c_n, c', with a link labelled e from the c-object to the c_1 object, a link labelled e_1 from the c_1 object to the c_2 object, etc. Let o be the first object in this sequence and o' be the last. This object graph satisfies the requirements. **QED**

Similarly, to find all the edges from an object of class c that might lead to an edge e, we compute

$$\text{FIRST}(c, e) = \{e' \mid (\exists c')(c\ (\leq e' \geq)(\leq C \geq)^* \leq e\ c')\}$$

9 Related Work

It is surprising to see that despite the universality of traversals in programming, very little work has been done in this direction. Until recently, the automation of traversal of object structures using succinct representations is unique to Demeter (see below); the rising popularity of markup languages in general, and XML in particular, created a new interest in traversals. In this section we list some work relevant to traversals.

In the context of *object-oriented databases*, traversals are heavily used. Some automation of traversals was suggested in [12, 16, 11, 7, 5]. Roughly speaking, the idea in these papers is to traverse to a target without specifying the full path leading to it. Cast in our terms, one can view these techniques as a variant of line-graph strategies (i.e., strategy graphs consisting of a single path) ; however, their goal is

to allow the user to abbreviate the laborious specification of a full query, and their main concern is how to complete the abbreviation when it is ambiguous, sometimes using heuristics. DJ takes advantage of reflection to complete its queries.

XML is a new standard for defining and processing markup languages for the web [1]. XML uses grammars (also called *document type definitions* or *schemas*) to define a markup language for a class of documents. To select subsets of XML document elements, the W3 Consortium recently introduced a language called XPath [2]. The way elements are selected in XPath is by navigation, including using a succinct notation similar to traversal strategies. For example, the XPath expression `A//B` refers to the set of all `B`-objects reachable from the `A`-object. It does not matter how many objects are contained between the `A`-object and the `B`-object. XPath expressions are used to describe *sets of objects*, in the sense that the value of an expression is an *unordered* collection of objects *without duplicates.* This is in contrast to traversals, whose value is a set of *paths*, so that the objects of each path are explictly ordered and may appear more than once, even on the same path. It is quite easy to implement XPath using strategies, using specialized "visitors." The converse, however, does not hold, due to the lack of structure in XPath expression values.

The *Visitor* design pattern is discussed in many software-engineering works (e.g., [4]). While this approach identifies and isolates the task of traversal, no mechanism to automate the task and make it adaptive was previously proposed, except in [8, 10].

The concept of automated traversal generation using succinct representation was introduced in [15] and is extensively treated in [8]. The traversals using the syntax of [15] essentially describe series-parallel graphs, which seem to be an important special case in practice. However, the algorithm provided by [15] cannot be applied for a significant subset of combinations of class structures and traversal specifications. A subsequent paper [14] removed this restriction for the same specification language using a finite-automata based approach to compilation. Unfortunately, the algorithm of [14] has exponential running time in the worst case, since it may generate exponentially many traversal methods [10].

Traversal strategies are introduced in [10] together with an efficient implementation. This paper is motivated by the need to have a simpler semantics for traversal strategies than the one presented in [10].

10 Conclusions and Future Work

We have presented a semantics for a declarative specification of object-graph traversals. Our organization separates the concerns of

- The search or traversal to be performed,
- The visits to objects of different classes to be executed along the way, and
- The details of the organization of the object graph, which is reconstructed by reflection.

The development of this paper itself reflects a separation of concerns. In earlier papers we expressed the semantics of a traversal strategy through a set of paths in the class graph. This required a heavier definitional approach. In this paper we separate the issues: we define the FIRST predicate at the level of object graphs, only referring to paths in object graphs and using an existential quantifier over object graphs. Only later, to compute FIRST, do we refer to paths in the class graph. This separation of basic semantics from efficiently computing the semantics leads to a concise definition of the meaning of a traversal specification basically only referring to object graphs.

Our derivation of an effective computation for FIRST illustrates that a relational formalism is the right approach to express the different path concepts that are needed; we believe that this approach will be helpful in other contexts as well.

The paper achieves another separation of concerns: In earlier works we defined the meaning of a traversal specification for an object graph to be traversal history: an ordered sequence of nodes and edges describing the visiting order. In this paper, we separate the definition of the traversal history into two parts: first we define the subgraph of the object graph selected by the traversal specification (using FIRST and search) and then we may use any traversal approach we choose, e.g., depth-first, to define a traversal history. This separation of traversal graph from traversal history leads to a more concise and understandable definition of the concept.

Now that we have a simpler technical core, we can hope to explore questions like: When the class graph changes, does the adaptive program have to change? For what kind of contexts is an adaptive (or aspect-oriented) program correct? In an aspect-oriented program, what kinds of changes in the base program require changes in the aspect code, or conversely, when do changes in aspect code require changes in the base code? We hope to explore questions of this kind first in the context of DJ and eventually in the more general setting of AOP.

References

[1] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen (eds.). Extensible Markup Language. http://www.w3.org/TR/REC-XML, February 1998.

[2] James Clark and Steve DeRose (eds.). XML Path Language (XPath), version 1.0. http://www.w3.org/TR/XPath, November 1999.

[3] DJ home page. http://www.ccs.neu.edu/research/demeter/DJ. Continuously updated.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Yannis E. Ioannidis and Yezdi Lashkari. Incomplete path expressions and their disambiguation. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 138–149. ACM Press, 1994.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mike Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.

[7] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In Michael Stonebraker, editor, *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 393–402, San Diego, CA, 1992. ACM Press.

[8] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X, entire book at www.ccs.neu.edu/research/demeter.

[9] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[10] Karl J. Lieberherr and Boaz Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997. http://www.ccs.neu.edu/research/demeter/AP-Library/.

[11] Victor M. Markowitz and Arie Shoshani. Object queries over relational databases: Language, implementation, and application. In *9th International Conference on Data Engineering*, pages 71–80. IEEE Press, 1993.

[12] Erich J. Neuhold and Michael Schrefl. Dynamic derivation of personalized views. In *Proceedings of the 14th VLDB Conference*, pages 183–194, 1988.

[13] Doug Orleans and Karl Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[14] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, 1997.

[15] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.

[16] Jan Van den Bussche and Gottfried Vossen. An extension of path expressions to simplify navigation in object-oriented queries. In Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors, *Deductive and Object-Oriented Databases*, pages 267–282, Phoenix, Arizona, 1993. Springer Verlag, Lecture Notes in CS, No. 760.