Navigating through Object Graphs Using Local Meta-Information ⋆

Karl Lieberherr, Mitchell Wand

College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115, USA

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 algorithm is implemented in a Java library called DJ.

Key words: Traversal specifications, Object-Oriented programming, Adaptive programming, Aspect-Oriented programming, Visitor pattern, Adaptive Visitor pattern, Formal semantics

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. For example, given an object o of class `A`, find all the objects of class `C` that are reachable from o along a path that goes through some object of class `B`. In order to make the search tractable, we typically search based on *local meta-information*. In such a search, we search only those edges from o that *might* lead to an object that satisfies the search criterion, based on the class structure of the object graph. We may explore objects that

⋆ Work supported by the National Science Foundation under grants CCR-9804115, CCR-0097740, and CCR-0098643, and by ARPA and BBN under agreement F33615-00-C-1694.

Email addresses: `lieber@ccs.neu.edu` (Karl Lieberherr), `wand@ccs.neu.edu` (Mitchell Wand).

URLs: `http://www.ccs.neu.edu/home/lieber` (Karl Lieberherr), `http://www.ccs.neu.edu/home/wand` (Mitchell Wand).

are not on the path to a desired target object, but that is an unavoidable consequence of searching based only on meta-information.

In this paper we present a simple semantics for such a search, and introduce an algorithm for performing the search, given a search criterion like the one above. The semantics and algorithm address two technical problems: the meaning of the modal operator "might", and the treatment of inheritance. We give an example of this algorithm as embodied in the DJ library for Java [3].

Section 2 presents our notation, and section 3 presents our model of classes and objects. In section 4, we formulate the basic traversal problem in the terms of our model. The key to the problem is to find a set $\mathrm{FIRST}(c, c')$ of edges 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 5 gives semantics to the core traversal specifications of DJ and shows how to interpret them as search algorithms. Section 6 completes the story by showing how to compute the FIRST sets statically. Section 7 presents an example showing how these concepts are employed in the DJ library. Section 8 discusses related work. Finally, section 9 presents some conclusions and ideas for further development.

2 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.

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. We allow paths to be empty, so that every node in a graph has a path of length 0 to itself.

3 The Model

Definition 1 *A* class graph *(sometimes called a* class diagram*) 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 *for C consists of:*

(1) a set O (of "objects"),
(2) a map $\text{class} : O \to C$*, 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 1.

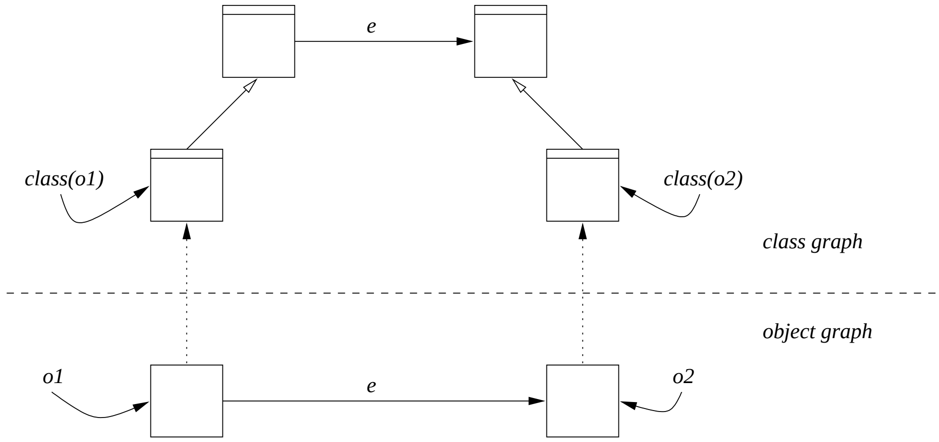

Fig. 1. Typical link in an object graph. If there is an e-edge from $o1$ to $o2$, then there is a path class($o1$) $\leq e \geq$ class($o2$) in the class graph.

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 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 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.

4 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.

5 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 $o_{j_1}, \ldots, o_{j_K}$ $(K \geq 1)$ 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.

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

Algorithm 1 *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* $\mathrm{FIRST}(class(o), c_1)$*, 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, instead of just the minimal ones, 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* $\mathrm{FIRST}(class(o), c_1)$*, 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.

DJ allows the use of a graph, called a *strategy graph*, to specify a more complex traversal [9]. 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* $\mathrm{class} : Q \to C$*, 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* $o_{j_1}, \ldots, o_{j_K}$ $(K \geq 1)$ *of* p *and a path* $(q_1, \ldots, q_K)$ *in* S *such that for each* i*,* o_{j_i} *has type* $\mathrm{class}(q_i)$*,* $j_K = N$*, 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 2 $search(o, S) : search'(o, QI)$*, where for any* $Q' \subseteq Q$*,* $search'(o, Q')$ *is defined by:*

Procedure *search*$'(o, Q')$:

(1) *If* $class(o) \not\leq \text{class}(q)$ *for any* $q \in Q'$*, then for each* $q \in Q'$*, for each* e *in* $\text{FIRST}(class(o), class(q))$*, and each* o' *such that* $e(o, o')$*,* $search'(o', Q')$.

(2) *If* $class(o) \leq \text{class}(q)$ *for some* $q \in Q' \cap QF$*, then* $visit(o)$.

(3) *Otherwise let* $Q'' = \{q \in Q' \mid o \text{ does not have type } class(q)\} \cup \{q' | \exists\, q \in Q' \text{ such that } o \text{ has type } \text{class}(q) \text{ and } S(q, q')\}$*. 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.

6 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 6 *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. $\square$

Lemma 7 *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, apply induction on the standard definition of $(-)^*$ (reflexive transitive closure), using the preceding lemma. $\square$

A picture of this situation is shown in figure 2.

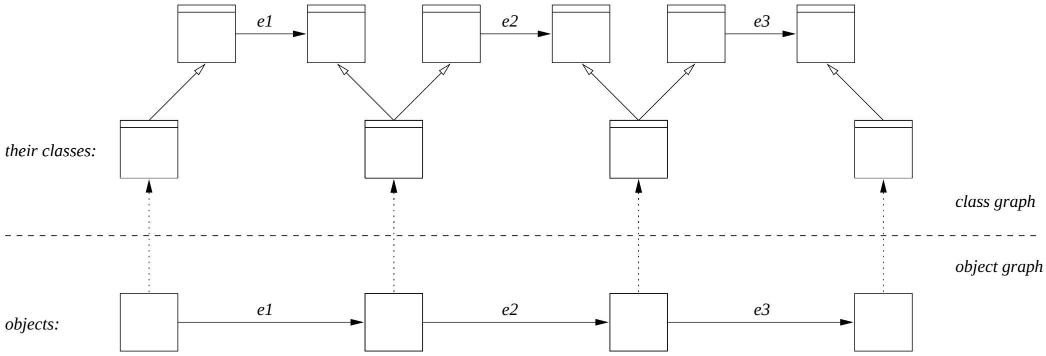

Fig. 2. A path in an object graph. If there is a path in the graph from o_1 to o_2, then $class(o_1)\ (\leq C \geq)^*\ class(o_2)$.

Lemma 8 *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 9 $\text{FIRST}(c, c') = \{e \mid c\ (\leq e \geq)(\leq C \geq)^* \leq c'\}$

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, $\ldots$,c_n, c', with a link labelled e from the c-object to the c_1 object, a link labelled e_1 from

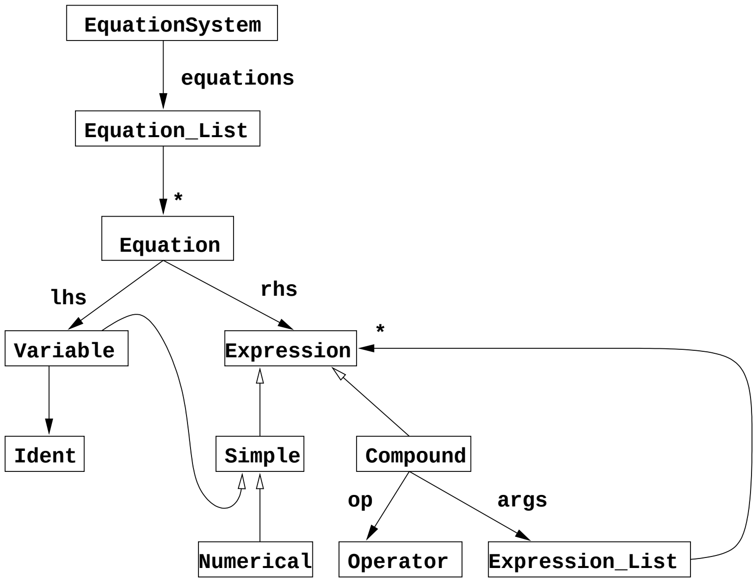

Fig. 3. Class graph for `EquationSystem`.

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. $\square$

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

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

7 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. A typical set of equations might be

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

A possible class diagram for these structures is shown in Figure 3, using notation similar to that of our previous figures. The asterisks indicate one-many relations; they are included for UML compatibility but are not part of our model; in our model has-as-part relations are always possibly one-many.

In DJ, we search for all minimal $(c_1, \ldots, c_K)$-paths using the search criterion

`through` c_1 `through` c_2 `...` `to` c_K

DJ traversal specifications allow several extensions of this notation. The criterion

`from` c_0 `through` c_1 `through` c_2 `...` `to` c_K

has the same behavior as the search above, but fails if its starting point is not of type c_0. DJ also allows edge specifications of the form `through` e. This can be implemented by extending $\mathrm{FIRST}(c_1, c_2)$ to $\mathrm{FIRST}(c, e)$ as defined in section 6 and modifying the algorithm in section 5 correspondingly.

A a DJ traversal specification may include a clause `bypassing` c_1 specifying that the portion of the path in which this clause occurs may not pass through an object of class c_1. The DJ library also includes ways to specify strategy graphs [9].

In our example, let us say that 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 traversal may be written as `from EquationSystem through rhs to Variable`. The second may be written as `from Equation System through lhs to Variable`.

There are of course many ways of specifying the same traversals. For example the first could have been written as `from EquationSystem through rhs through Expression to Variable`, and the second could have been written as `from EquationSystem bypassing Expression to Variable`.

The DJ library uses reflection to compute the relevant FIRST sets. This has two advantages: first, it allows the same code to be reused even if the class structure changes. Second, it allows the system to be implemented as a pure Java library rather than as a preprocessor. This is an example of *adaptive* behavior [8].

The code for this task is shown in figure 4. The static member `cg` is bound to a `ClassGraph` object that contains a representation of the current class diagram[1]. The method `collectVars` takes a string `travspec` that specifies the traversal to be performed. It first creates a visitor `v` to implement the

[1] In a real application, this would probably done globally rather than on a per-class basis.

```
class EquationSystem{
  static ClassGraph cg = new ClassGraph();
  // constructors omitted ...
  HashSet collectVars(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() {
    return this.collectVars
     ("from EquationSystem through Expression"
       + "through rhs to Variable");
  }
  HashSet getDefinedVars() {
    return this.collectVars
     ("from EquationSystem bypassing Expression"
      + "to Variable");
    }
}
```

Fig. 4. Finding the variables in an equation system

behavior that is to be performed during the traversal. 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], augmented by reflection. Every time the traversal reaches a node n for which a `before` method has been defined, it calls `v.before(`n`)`. The traversal algorithm uses reflection to call the `before` method only on objects for which a `before` method has been defined. Thus, when the node is a variable, the method `before(Variable v1)` (the "visitor method") is called, adding the node to the set. Similarly, on completion of the search, the traversal calls `v.getReturnValue()`, which returns the collected hash set.

The DJ library includes a powerful language of traversal specifications and methods, including the possibility of precomputing the FIRST sets, and of attaching methods to edge traversals; see [3]. An introduction to DJ, with emphasis on its reflective properties, may be found in [13].

8 Related Work

The *Visitor* design pattern is discussed in many software-engineering works (e.g., [4]).

The standard visitor pattern proposes a code organization that allows an operation to be defined with o.`accept`(*what_and_when_to_do*), where *what_and_when_to_do* is a visitor object, and `accept` is a method on the visited object that sends information from the visited object to the visitor.

In contrast with the ordinary visitor pattern, no preparation in the underlying objects, such as the `accept` method above, is necessary in order to use the DJ library.

The version of the visitor pattern used in DJ proposes a code organization that allows an operation to be defined with

cg.`traverse`(*where_to_start*, *where_to_go*, *what_and_when_to_do*)

where *cg* is the class graph containing the meta-information for the current object graph.

This version of the visitor pattern has been called Adaptive Visitor ([7], pages 426-427) where an operation is defined by the pair (*where_to_go*, *what_and_when_to_do*).

The concept of automated traversal generation using succinct representation was introduced in [7]. It was formalized in [14] and is extensively treated in [8], where it is the key idea of Adaptive Programming (AP), and is used to implement the Adaptive Visitor pattern [8, pages 426–427].

Our most complex language of traversal specifications, called *strategy graphs*, is introduced in [9] 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 paths in the class diagram as [9] 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 [14].

Our use of the visitor pattern may be regarded as an example of Aspect-Oriented Programming [2]. Aspect-oriented programming systems typically consist of a framework that 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 descriptors"), and a means of associating procedures with such declaratively-specified sets. In DJ, computation is the traversal, and a join point is the passage of the traversal through a node of the object graph. Each visitor method has a signature that plays the role of a point cut descriptor, a body that plays the role of the associated procedure.

In the context of *object-oriented databases*, traversals are heavily used. Some automation of traversals was suggested in [12,15,10,6,5]. Roughly speaking, the idea in these papers is to traverse to a target without specifying the full path leading to it. Most of this work concerns what we call minimal R-paths; however the primary concern is how to complete the abbreviation when it is ambiguous, sometimes using heuristics. DJ takes advantage of reflection to complete its queries.

Mendelzon and Wood [11] show the computational intractability of finding actual paths in an object graph, even without the presence of inheritance. We avoid this difficulty because we are searching only for potential paths based on local meta-information, and we therefore accept the possibility of traversing nodes that are not actually on a path to the target class.

The XPath [1] language is used to specify sets of elements in XML documents. XPath uses a succinct notation somewhat like ours; for example, the XPath expression `A//B` refers to the set of all `B`-objects reachable from the `A`-object. The semantics of an XPath expression is an unordered set of objects. This closely matches the algorithms as presented in section 5, although our implementation in DJ also captures the paths by allowing visitors to be called on all objects in those paths.

9 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.

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.

This paper was motivated by the need to have a simpler semantics for traversal strategies than the one presented in [9]. 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] J. Clark, S. D. (eds.), XML Path Language (XPath), version 1.0 (Nov. 1999). URL `http://www.w3.org/TR/XPath`

[2] Communications of the ACM, Vol. 44:10, ACM, 2001, special issue on Aspect-Oriented Programming.

[3] Demeter research group, DJ home page (continuously updated). URL `http://www.ccs.neu.edu/research/demeter/DJ`

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[5] Y. E. Ioannidis, Y. Lashkari, Incomplete path expressions and their disambiguation, in: Proceedings of ACM/SIGMOD Annual Conference on Management of Data, ACM Press, 1994, pp. 138–149.

[6] M. Kifer, W. Kim, Y. Sagiv, Querying object-oriented databases, in: M. Stonebraker (Ed.), Proceedings of ACM/SIGMOD Annual Conference on Management of Data, ACM Press, San Diego, CA, 1992, pp. 393–402.

[7] K. J. Lieberherr, Component enhancement: An adaptive reusability mechanism for groups of collaborating classes, in: J. van Leeuwen (Ed.), Information Processing '92, 12th World Computer Congress, Elsevier, Madrid, Spain, 1992, pp. 179–185.

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

[9] K. J. Lieberherr, B. Patt-Shamir, Traversals of Object Structures: Specification and Efficient Implementation, Tech. Rep. NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA (Sep. 1997). URL `http://www.ccs.neu.edu/research/demeter/AP-Library/`

[10] V. M. Markowitz, A. Shoshani, Object queries over relational databases: Language, implementation, and application, in: 9th International Conference on Data Engineering, IEEE Press, 1993, pp. 71–80.

[11] A. O. Mendelzon, P. T. Wood, Finding regular simple paths in graph databases, in: Proceedings of the 15th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Amsterdam, 1989, pp. 185–194.
URL `citeseer.nj.nec.com/mendelzon89finding.html`

[12] E. J. Neuhold, M. Schrefl, Dynamic derivation of personalized views, in: Proceedings of the 14th VLDB Conference, 1988, pp. 183–194.

[13] D. Orleans, K. Lieberherr, DJ: Dynamic Adaptive Programming in Java, in: Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns , Springer Verlag, Kyoto, Japan, 2001, pp. 73–80.

[14] J. Palsberg, C. Xiao, K. Lieberherr, Efficient implementation of adaptive software, ACM Transactions on Programming Languages and Systems 17 (2) (1995) 264–292.

[15] J. Van den Bussche, G. Vossen, An extension of path expressions to simplify navigation in object-oriented queries, in: S. Ceri, K. Tanaka, S. Tsur (Eds.), Deductive and Object-Oriented Databases, Springer Verlag, Lecture Notes in CS, No. 760, Phoenix, Arizona, 1993, pp. 267–282.