AP/S++: Case-study of a MOP for purposes of software evolution

Cristina Videira Lopes and Karl Lieberherr

Published in Proceedings of Reflection'96, San Francisco, USA, April 1996.

© Copyright 1996 Xerox Corporation. All rights reserved.

AP/S++: Case-study of a MOP for Purposes of Software Evolution *

Cristina Videira Lopes †

Xerox PARC and Northeastern University
lopes@parc.xerox.com

Karl J. Lieberherr

Northeastern University
lieber@ccs.neu.edu

Copyright ©1995 by Xerox PARC and Karl Lieberherr.
All rights reserved.

Abstract

We study a recent programming paradigm known as Adaptive Programming (AP) as an ideal candidate for a metaobject protocol (MOP) for object-oriented programming languages; we call it the AP^{MOP}. The major benefit of the AP^{MOP} is to provide a mechanism for writing base-level programs in a structure-shy manner. Doing so, the programs are more robust to changes in the structural aspects of the applications. We describe AP/S++, an implementation of the AP^{MOP} using the Scheme-based, object-oriented language S++. AP/S++ is a compile-time MOP and has no negative effects on the run-time performance of programs.

The contributions of this paper are: (i) to show a new application for reflection; (ii) to clearly identify the abstraction boundaries of AP; and (iii) to propose an implementation of the AP^{MOP} that can easily be reproduced in many object-oriented programming languages.

1 Introduction

Maintaining and evolving large software applications is hard. It is widely accepted that the object-oriented paradigm made the task less brutal by providing certain abstractions that elegantly capture the objects in the application's domain into program objects. Nevertheless, maintaining and evolving large applications written in object-oriented programming languages (OOPLs) is still hard and time-consuming. If the program results from a careful analysis and design, and models all the important abstractions of the problem, evolving it may be relatively easy, but in practice it still takes a lot of time to do: a simple change in the modelling of the problem may lead to modifications in many different modules of the program; most of those modifications are straightforward and don't need a lot of thinking, but, nevertheless, they consume a significant portion of the evolution effort.

*Partially supported by the National Science Foundation under grant numbers CDA-9015692 (Research Instrumentation), and CCR-9402486 (Software Engineering).

†Partially supported by the Portuguese Foundation for Research, Science and Technology (JNICT).

One of the reasons this happens is because programs written in OOPLs end up suffering from what we call "structural anomaly." Structural anomaly results from the fact that the implementation of the operations usually involves a group of objects, rather than only one object. In OOPLs, objects interact by sending messages to each other; the collection of message sends throughout the code defines an implicit *structural knowledge path* on the programs. We see these paths as an *anomaly* because they weaken the valuable concept of encapsulation that OOPLs provide: it's true that each object executes over its own data, but each object also defines the links of the network of objects that collaborate for the implementation of the operations.

As consequence of the structural anomaly, when the application structure evolves, the structural knowledge paths that exist in the implementation of the methods may be broken, and the programmers must fix them manually. This task is more tedious than creative, consumes a significant part of the evolution effort (editing, debugging and testing) and its straightforwardness suggests that it can be automated.

This paper views the anomaly as a failure of the base-meta separation and presents a mechanism to minimize the structural anomalies of object-oriented programs using reflection. Most of the ideas presented here come from the Demeter System/C++ [SLHS94, LSX94, Lie96] developed at Northeastern University. Demeter is one implementation of the theoretical foundations of Adaptive Programming (AP) [PXL95]. AP has been presented in the literature as a new programming paradigm for writing highly reusable OO software. Demeter includes its own programming language that encapsulates C++.

In this paper we reformulate AP under the perspective of reflection [Mae87, dS84], and present a system that clearly separates the meta-issues from the base-issues of AP. The result is AP/S++, an ordinary OOPL that contains a metaobject-protocol (MOP) for minimizing the structural anomalies of programs. Although the idea for the basic engine of AP/S++ comes from previous work on AP, there are significant differences between Demeter/C++ and AP/S++; those differences start at the conceptual organization of both systems and end up in two totally different programming styles.

Since we have used S++ [Chi96] as the OOPL for AP/S++, throughout this paper the examples will also be presented in S++. The particular language is not important, and we could have used any other OOPL for our prototype. Since the reader is probably not familiar with the syntax of S++, we present a very short reference guide of S++ in Fig. 1.

The remainder of this paper is organized as follows. Section 2 explains the structural anomaly, setting up the motivation for AP, and identifies the source of the anomaly under a reflective approach. Section 3 describes the idea of AP to minimize the anomalies and section 4 presents AP/S++, the implementation of the AP^{MOP}. Section 5 compares this work other systems, with special attention to the Demeter system, and finally section 6 concludes the paper.

2 Structural Anomaly

A study by Wilde et al. [WMH93] of three large object-oriented systems shows that more than half of the member functions and methods have fewer than two C++ statements or four Smalltalk lines of code. Most of the small methods are pure traversal methods, that

S++ can be seen as an extension of Scheme with a C++ intent (see example on the side which contrasts S++ with C++). It provides mechanisms for defining a class, instantiating and accessing objects in a similar way to C++. Being a prototype language, S++ ignores most of the complexity of C++; for example, there's no multiple inheritance and all member functions are virtual. As in C++, the type of an object is defined by its class. Also, the object's reference to itself is `this`. S++ programs are compiled into Scheme; then the resulting Scheme images are executed.

```
class Document   /* : nosuper */ {
public:
  Document();
  void search (String *);
  ParList *paragraphs;
};
```

S++

```
(class document ()
  ((initialize) (search))
  ((paragraphs list<paragraph>)))
```

```
void Document::search (String *aword) {
  // code here
  Paragraph *p;
  p->search (aword);
}
```

S++

```
(member document ((aword string))
  void
  ;; code here
  (typedecl p paragraph)
  (-> p search aword))
```

Figure 1: *Quick Reference Guide for S++.*

is, methods whose sole purpose is to traverse the object graph and pass a message along to another object.

There are reasons for these results. Suppose, for example, that on an editor application written in S++, we want to search for all occurrences of a given word in a document; the structure of the classes involved is shown graphically in Fig. 2 (i.e, a document consists of a list of paragraphs, etc.). In implementing the `search` operation, most experienced programmers will be tempted to write the following code:

```
(member document search ((aword string)) void
  (typedecl p paragraph) (typedecl l line) (typedecl w word)
  (let ploop ((plist (-> this paragraphs))) ;; get paragraphs list
    (unless (null? plist)                   ;; end of paragraphs list?
      (let ((p (car plist)))                ;; no - get a paragraph p
        (let lloop ((llist (-> p lines)))   ;; get lines list
          (unless (null? llist)             ;; end of lines list?
            (let ((l (car llist)))          ;; no - get a line l
              (let wloop ((wlist (-> l words)));; get words list
                (unless (null? wlist)       ;; end of words list?
                  (let ((w (car wlist)))    ;; no - get a word w
                    (if (eq? (-> w val) aword) ;; same word?
                       (-> w highlight)))   ;; highlight this word
                  (wloop (cdr wlist)))))    ;; loop on rest of words
            (lloop (cdr llist)))))          ;; loop on rest of lines
      (ploop (cdr plist)))))             ;; loop on rest of paragraphs
```

Although the above code is straightforward, this style of programming is usually not

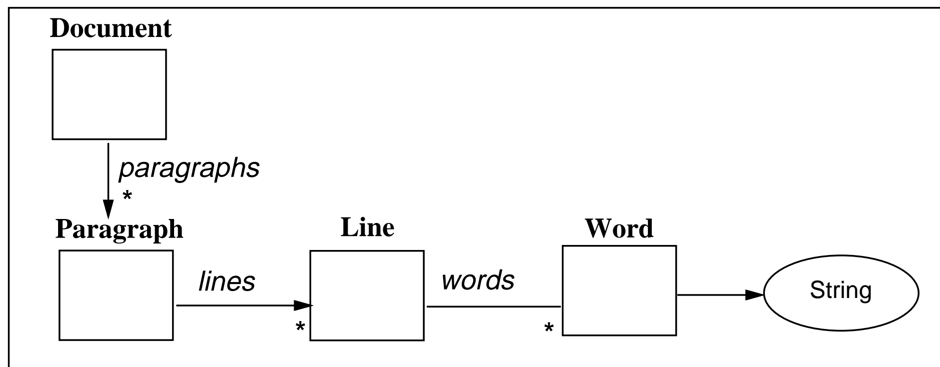


Figure 2: *Simple class structure for an editor application. Square boxes represent concrete classes; ellipse boxes represent primitive classes; edges represent references and may have a cardinality (e.g. ∗ means 0 or more).*

encouraged, for it increases the level of coupling of classes.[1] In this case, class document depends on all the other classes of the application. Strong coupling is seen as bad quality code for purposes of reuse and evolution [Mar95], because it slows down the evolution process – more dependencies have to be considered and more testing has to be done. Instead, the code in Fig. 3 is considered to be of better quality. In this new code, each class depends only on one other class.

The presence of traversal methods assures a low level of coupling but results in the proliferation of the small, unimportant methods found in Wilde's study. However, although the overall level of coupling decreased[2], there is an implicit commitment in the implementation of the operation `search` to the particular class structure in Fig. 2.

This is what we call *structural anomaly.* While each class has minimal coupling and works only on its private data, the implementation of the operation has been spread across many classes and implicitly assumes one particular class organization that hard-wires *structural knowledge paths* into the program.

From an evolutionary perspective, those structural knowledge paths are an obstacle. As the organization of the classes evolves to accommodate new application requirements, the existing code must be updated. For example, suppose that we want to evolve the editor so that (1) documents, besides paragraphs, may also contain tables; (2) paragraphs and tables are both concrete implementations of the abstract entity DocObject; and (3) tables contain a list of entries which, in turn, contain a list of words. The new class structure is shown graphically in Fig. 4.

In evolving the class structure from Fig. 2 to Fig. 4, we want to add new operations to the editor, but we also want to maintain all the previous functionality. In particular, we want to maintain the operation `search` so that it *adapts* to the new class structure. Because of the structural anomaly, the structural knowledge paths that exist in the code of Fig. 3

[1] The *coupling of a class* is defined to be the number of classes of different categories that the given class depends upon.

[2] Note that in the first case, class `document` depended on 3 other classes and the other 3 classes depended on 1 class each; in the second case, each of the 4 classes depends on only one class.

```
(member document search ((aword string)) void
  (let ploop ((plist (-> this paragraphs))) ;; get paragraphs list
    (unless (null? plist)                   ;; end of paragraphs list?
      (typedecl p paragraph)                ;; no
      (let ((p (car plist)))                ;; get a paragraph p
        (-> p search aword))                ;; invoke paragraph p and
      (ploop (cdr plist)))))                ;; loop on rest of paragraphs
(member paragraph search ((aword string)) void
  (let lloop ((llist (-> this lines)))      ;; get lines list
    (unless (null? llist)                   ;; end of lines list?
      (typedecl l line)                     ;; no
      (let ((l (car llist)))                ;; get a line l
        (-> l search aword))                ;; invoke line l and
      (lloop (cdr llist)))))                ;; loop on rest of lines
(member line search ((aword string)) void
  (let wloop ((wlist (-> this words)))      ;; get words list
    (unless (null? wlist)                   ;; end of words list?
      (typedecl w word)                     ;; no
      (let ((w (car wlist)))                ;; get a word w
        (-> w search aword))                ;; invoke word w and
      (wloop (cdr wlist)))))                ;; loop on rest of words
(member word search ((aword string)) void
  (if (eq? (-> this val) aword)             ;; same word?
      (-> this highlight)))                 ;; highlight this word
```

Figure 3: *S++ code for searching all occurrences of a given word in a document.*

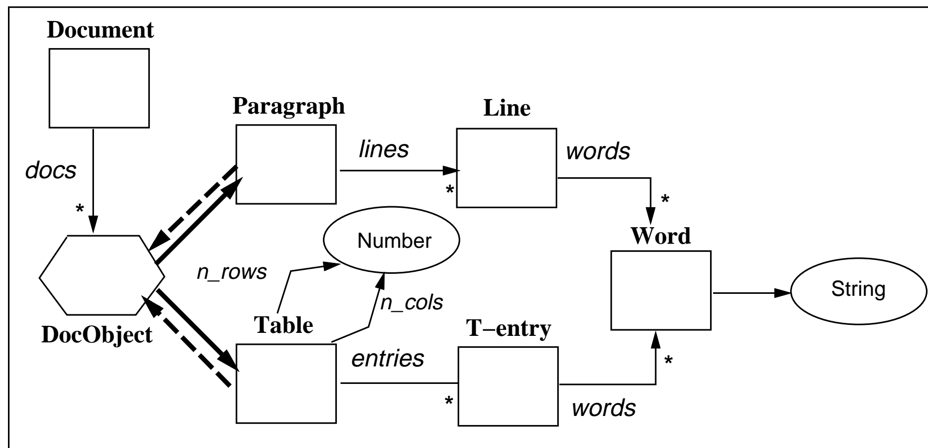


Figure 4: *Evolved class structure for the editor application. The hexagon box represents an abstract class; the thick, solid edges represent* `is-superclass` *relations and the reverse dashed edges represent the* `is-subclass` *relations.*

are broken for the new class structure: a document is no longer just a list of paragraphs; instead it now contains a list of objects, named `docs`, that can either be paragraphs or tables, and words can be reached from both paragraphs and tables.

In order to evolve the implementation of `search`, we need to edit the methods, make some modifications at class `document`, and write methods for the new classes. That is, we need to rearrange the structural knowledge paths, because the class structure changed, even though the "intention" of `search` is the same: searching for occurrences of a given word in a document.

Although this kind of maintenance seems natural, in many cases it can be avoided, speeding up the process of software evolution. In fact, the structural anomalies occur because the OO code intertwines base and meta information, as we will discuss next.

The Reflective Perspective:
By analyzing the structural anomalies under the light of reflective systems, we can describe those anomalies as failures that occur in the base-meta separation of OO programs. In the simpliest reflective approach, we can draw a line between the base- and the meta-level: at the base-level we have classes that know very little about the global class structure; at the meta-level we have a graph of class metaobjects, and this graph defines the structure of the base-level program.

Although for good quality programs the meta-information is invisible for each of the base-level class implementations, that meta-information becomes part of the base-level program under the form of explicit message sends and traversal code. Explicit message sends and traversal code implicitly define one particular graph of class metaobjects.

When the class structure evolves – defining a new graph of class metaobjects – the pre-existing base-level traversal code will most likely be broken. Tolerance to changes in the class structure is increasingly important as the number of classes in the application increases. If the programming language includes some mechanism to split the base-level code from the structural meta-level information, some of these *adaptation changes* will be tolerated without modifying the existing code, while some others will require localized modifications that are faster to do than its equivalents at the OO level.

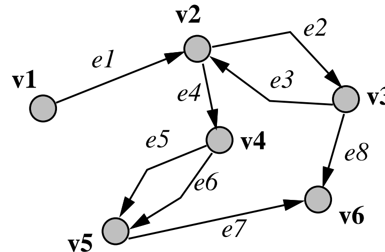
3 Adaptive Programming

In order to increase the tolerance to changes, AP uses the concept of *traversal*, as first proposed by Lieberherr in [Lie92]. The basic idea is to define and use a traversal language construct that *succinctly* identifies structural knowledge paths, so that (1) programs become shorter; and (2) when the class structure changes, the traversals may still be correct with respect to the new structure. In this section we define what traversals are, and how traversals defined in a succinct form can improve the adaptability of the programs.

Intentionally, we present AP in very general terms, without relating it to reflection. The reason for this is that AP can be implemented in several different ways; in fact, AP/S++ and Demeter/C++ are two different instances of the same AP idea. Also, traversals may have different semantics (see [PXL95] and [HS96]). In section 4 we present how AP can be seen as a MOP, and how we have implemented it as such.

3.1 Traversals

Given a graph G, a traversal directive is a tuple $(source, Target, Constraint)$ that identifies the set of paths, $PathSet_G(D)$, from the source vertex to the target vertices, satisfying the given path constraints. The constraints are given in terms of *exclusion* of vertices and edges. Consider, for example, the graph on the right. The traversal directive `(v4, (v6), NoConstraints)` identifies the two paths from vertex $v4$ to vertex $v6$, namely $\{v4\,(e5\,+\,e6)\,v5\,e7\,v6\}$.
The traversal directive `(v2, (v6), excluding(v2`$\stackrel{e4}{\rightarrow}$`v4))` identifies the infinite set of paths $\{v2\,e2\,v3\,(e3\,v2\,e2\,v3)^{*}\,e8\,v6\}$.



The names involved in the constraints can be given in terms of wildcards, $*$, meaning "all". The wildcard can be used with an optional extension, $*\backslash(e1...en)$, meaning "all except *(e1...en)*", where e represents a graph element of the same type of the one that is being excluded by the $*$. So `excluding (v4` $\stackrel{*\backslash\{e5\}}{\rightarrow}$ `v5)` excludes all edges from vertex $v4$ to vertex $v5$ except the edge labeled $e5$.

The primitive traversal directives $(source, Target, Constraint)$ are only powerful enough to define simple traversals in the graph: there are paths that cannot be expressed by a simple traversal. For example, the path $\{v2\,e2\,v3\,e3\,v2\,e4\,v4\}$ cannot be expressed in terms of a simple exclusion of vertices and edges. Therefore, we define the composition of directives by the *join* operator '.'.

$$PathSet_G(D1\,.\,D2)\;=\;PathSet_G(D1)\,.\,PathSet_G(D2)$$

The result of the composition of two traversal directives is the concatenation of the corresponding sets of paths. For a composition to be meaningful it must observe the condition $source(D1) =_{vertex} Target(D2)$. $\{v2\,e2\,v3\,e3\,v2\,e4\,v4\}$ results from `(v2, (v3), excluding(v3` $\stackrel{e3}{\rightarrow}$ `v2)).(v3, (v4), excluding(v2` $\stackrel{e2}{\rightarrow}$ `v3))`. $\diamond$

AP defines traversal directives with respect to class graphs. Formally, a class graph, denoted $G = (V, E, L)$, is a finite, labeled, directed graph, where each vertex v in V represents a class, each edge e in E represents either a reference, inheritance or subclass edge, and each label l in L represents either a class name, a reference name, the special inheritance label $\triangleleft$ or the special sub-class label $\triangleright$. There are several conditions to observe for a class graph G to be meaningful, but in this paper we will not go into details (for the complete formal description of class graphs, please see either [PXL95] or [HS96]).

The structural knowledge paths that exist in OO programs correspond to paths in the class graph. Traversal directives succinctly capture that structural information. We apply the directive $(source, Target, Constraint)$ to class graphs exactly as explained for the generic graph; the exclusion constraints may include any class and any edge of the class graph. For example, the structural knowledge path in the code of Fig. 3 can be expressed by the directive `(document, (word), NoConstraints)`.

When the OO program executes, the paths in the class graph are transformed into paths in the object graph. At run-time the meaning of $PathSet_G(D)$ can be seen as "all reachable objects of the involved classes". So if we have `(document, (word), NoConstraints)`, it means "all word objects reachable from a document object".

```
(traverse
  ((from document to (word) ())       ; the traversal directive
     (at (word)                       ; and its wrappers of code
       (if (eq? (-> this val) aword)
           (-> this highlight)))))
```

Figure 5: *Piece of code for searching all occurrences of a given word in a document.*

3.2 How Traversals Are Used

Traversals directives *per se* simply define path sets, and don't do anything else. However, they can be used in many different ways to operate on the objects. One of those ways is to execute some actions on the objects along the object traversal. The actions are captured by code fragments that are given as parameters of the traversal directives:

$$traverse(D_1(W_1).D_2(W_2)...D_n(W_n))$$

As a concrete example, in Fig. 5 we present the directive for the operation `search`, along with its wrappers. The piece of code in this figure replaces all the explicit message sends and traversal methods in the code of Fig. 3.

Two remarks must be made with respect to the piece of code in Fig. 5. First we see the special form `traverse`, which gets a list of traversal directives with wrappers of code; in this case, there is only one traversal directive, `(from document to (word) ())`, but there may be more, which will be composed as explained in section 3.1. `traverse` is the engine for code execution, and will be explained in more detail in Section 4.2.

Second we have the wrappers of code (`(at (word) (...))`, in this case only for class `word`. The wrappers may be attached to classes or to reference edges, and contain code that will be executed every time the traversal reaches the corresponding run-time objects or references. In this case, the given body `(if ...)` will be executed for all word objects that are reached from a certain document.

There are three kinds of wrappers: `at`, `prefix` and `suffix`. `at` should only be used for end point classes of traversals (such as `word`, in this case), for it breaks any further traversal that may be left to do. `prefix` wrappers are executed as soon as the traversal reaches the correspondent object or reference, and just before any further traversal occurs. `suffix` wrappers are executed at the correspondent object or reference after the traversal from it has been done.

The body of the wrapper is ordinary OOPL code, in this case S++. This code may include recursive calls to the special form `traverse`. The variable `this` in the wrappers is bound to the current object; in this example, when the wrapper for class `word` is executed, `this` refers to a word object.

Another usage of traversal directives is to copy particular graphs of objects from one address space to another in remote invocations; instead of `traverse` we use `copy`. That work has been presented in [Lop96], and falls out of the scope of this paper.

3.3 Adaptability: Brief Evaluation

The piece of code in Fig. 5 is noticeably shorter than any of the implementations given in Section 2; it includes only the necessary information that is relevant for reaching words from a document. Traversal directives are given by succinct specifications, namely the source, the targets and the constraints; they don't over specify all the nodes and edges in the paths. $PathSet_G(D)$ may be very different depending on the particular class graph G. In the example, $PathSet_G$(from document to (word) ()) adapts from the class structure in Fig. 2 to the class structure in Fig. 4; the piece of code in Fig. 5 can be applied without modification to both class structures.

Although traversal directives help in writing short, adaptable code, they don't always adapt to the changes in the class graph. In evolving the application from G_1 to G_2, it may happen that the traversal directives that were correct for G_1 will be incorrect for G_2 because they expand into new but unwanted paths. For example, although the editor evolves from Fig. 2 to Fig. 4, for some reason, we may not want to search words in tables. In that case the traversal `(from document to (word) ())` expands into one wanted path in the second graph, namely the one that goes through class Table, so we would have to rewrite it; for example `(from document to (word) (excluding (table)))`. (See [HS96] for a detailed analysis of automatic evolution of AP programs.)

The maintenance of AP programs consists in redefining the traversal directives by adding or removing new target classes and/or exclusion constraints; it is localized on the traversals and it's easy to do. Automatic adaptation and localized modifications contribute to speed up the proccess of evolution.

Although we have no significant data that allows us to evaluate AP/S++, the Demeter System/C++ has been intensively used in medium sized projects (20 – 50 classes), most of them part of graduate courses at Northeastern University. Depending on the project, as well as the specific designs, the statistics vary, but the benefits of using traversals instead of C++ traversal methods are visible in two ways: first, from the students' testimonies, who claim that writing traversals and thinking in terms of graphs helps them to write their projects faster and to make less mistakes as the project evolves; second, from the number of lines of code – for each project a statistics is made comparing the number of lines of "Demeter code" with the generated C++ code, and for a medium sized project the "Demeter code" is at least 30% shorter than the correspondent C++ code.

4 AP/S++: Design and Implementation

We take a new approach to the AP idea of structure-shy programming, and model it as a metaobject protocol which we call AP^{MOP}. The AP^{MOP} is AP seen to the light of base-meta separation, and using only introspection. It uses the information on how the classes relate in order to navigate through them. The base-level programs may then use the introspection capabilities of the AP^{MOP} for defining traversals on the class graph.

With the AP^{MOP}, the abstraction boundaries of AP become very clear: (1) the base-level programs are ordinary object-oriented programs, with classes and member functions; (2) AP is implemented at the meta-level, based on a collection of class and edge metaobjects that defines a graph; (3) traversal directives $(source, Target, Constraint)$ refer, by name or

```
(member document search ((aword string)) void
  (traverse                        ; call the meta-level
    ((from this to (word) ())      ; the traversal directive
       (at (word)                  ; and its wrappers of code
         (if (eq? (-> this val) aword)
             (-> this highlight))))))
```

Figure 6: *Implementation of* `search` *in AP/S++.*

indirectly, to class and edge metaobjects; (4) paths in the class graph are paths (meta-paths) in the graph of class and edge metaobjects; (5) meta-paths identify paths that may occur at run-time for the base-level objects; and (6) the only base-meta interface is the special form `traverse`, and it may occur several times and to several objects in the implementation of the member functions.

Programming in AP/S++ is exactly like programming in plain S++; the only difference is that the base program may call the special form `traverse`. Also, AP/S++ provides an extended type system that includes parameterized lists and arrays. Fig. 6 presents the implementation of the operation `search` written in AP/S++ that applies to both class structures in Fig. 2 and Fig. 4. Although not shown by this example, the implementation of the member functions may include several sequential traversal calls starting at any object that has a valid binding in the current scope. For simplicity reasons, the current version of AP/S++ does not support recursive calls to `traverse` (the code wrappers are not parsed further).

4.1 The AP Metaobjects

At the AP meta-level, there are three important kinds of objects: `ap-class`, `ap-edge` and `class-graph`. The first is similar to the class metaobjects defined in other MOPs (e.g. CLOS [KdRB91]); the second corresponds to the slot metaobjects also found in CLOS; the third is specific for AP/S++. Below is the detailed explanation of each of these kinds of metaobjects.

4.1.1 ap-class

A `ap-class` metaobject determines the structure and the interface of its instances, that is, the S++ classes. `ap-class` contains the following information:

- The name.
- The direct superclasses and direct subclasses, given as a list of metaobjects.
- The direct references (slots and their types), given as a list of `ap-edge` metaobjects. The list of all references, direct and inherited, is also available.
- The inverse references, that is, the list of all references in other classes that are of this class type; given as alist of metaobjects.
- The list of method names of the class.

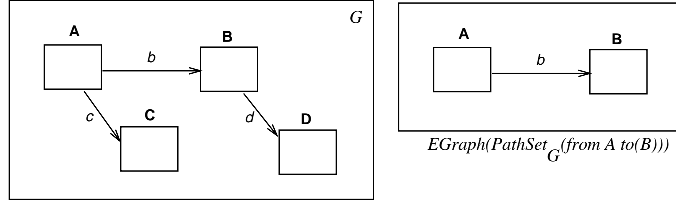


Figure 7: *A set of application classes; the traversal* `from A to B` *results in a graph of class metaobjects which are incomplete copies of the metaobjects for classes A and B.*

4.1.2 ap-edge

We choose not to model inheritance and sub-class edges by `ap-edges`, and instead include that information directly in the class metaobjects. Therefore, an `ap-edge` metaobject contains information about the definition of a reference. `ap-edges` contain the base-level reference name and the reference to the class metaobject of that base-level reference. That is to say, `ap-edges` link `ap-classes`. In relation to the extended type system, `ap-edges` contain cardinality information that can be *only-one*, *zero-or-more* or N, where N is a number greater than 0.

4.1.3 class-graph

A `class-graph` metaobject is simply a list of class metaobjects which are connected through edge metaobjects, that is they form a referential/inheritance closure. Class graphs are constructed and used by the expansion algorithm that computes the path sets. The expansion of a traversal directive D in a class graph G results in the path set $PathSet_G(D)$ that may be infinite. However, $PathSet_G(D)$ has a finite representation which we call *expansion graph*, $EGraph(PathSet_G(D))$, and is effectively computed by a simple algorithm which works in two steps: first, do a forward depth first traversal from the *source* vertex in order to identify the subgraph reachable from *source*; then do a backwards depth first traversal from the *targets* in order to identify the subgraph of classes that reach those targets; the expansion graph is the intersection of the two subgraphs[3]. The metaobjects in those graphs are clones of the existing class metaobjects, that exclude the references not used by the expansion result. (see Fig. 7.)

4.2 The AP Base-meta Protocol

As we said before, the interface to the meta-level is done through the special form `traverse` that takes as parameter a list of traversal directives with wrappers. Each of the traversal directives identifies a source base-level object or a source class name, a set of target class names and a set of path exclusion constraints.

[3]There are, of course, some other important details of the algorithm regarding inheritance and subclassing, that can be summarized as follows: 1) when some class is reached, the depth first traversal continues not only on its reference edges but also on its superclasses; 2) when a class is reached through a reference edge, all its subclasses are also marked as reached and are further traversed; 3) when a class is reached through a inheritance (from one of its subclasses), its other subclasses are not marked as reached.

```
(member document count-words () integer
  (let ((count 0))                ;; reset the counter
    (begin
      (traverse
       ((from this to (word) ())  ;; go to all words
          (at (word)              ;; at each word
            (+ count 1))))        ;; increment the counter
      count)))                    ;; return the total
```

Figure 8: *Using variables along the traversal; the variable* `count` *has a binding within the code wrapper for class* `word`*.*

If the *source* is given as a base-level object, as `this` in Fig. 6, the *source* class metaobject is the class of the given object – in this case, `document`. Giving a base-level object as the source serves both for identifying the source class metaobject and source base-level object from which the base-level traversal starts. The source class can also be given as a class name; that happens only for composition of traversals that were explained previously. In any case, for a traversal call to be meaningful, the first traversal directive must specify the first base-level object from which the traversal will start.

In broad terms, the meta-level works as follows. When the meta-level gets a traversal call, it identifies the source class metaobject and the target class metaobjects for each of the directives, and computes the expansion graphs $EGraph(PathSet(D_i))$. Then the traversal translation algorithm uses those graphs to translate the traverse call into nested functions that will implement object traversals surrounded by the prefix and suffix wrappers (or ended by the `at` wrappers).

The expansion graph that results from `(expand (from this to (word) ()))` depends on the specific class metaobjects of the application. For example, if we have the classes in Fig. 2 the expansion of the given directive results in one path that corresponds to the four application classes along with their references; if we have the classes in Fig. 4 the result includes the seven application classes, its references (except the rows and columns), and there are two distinct paths from Document to Word.

Traversals are defined within a certain scope in the member function and the wrappers of code have access to the variable bindings of that scope. The variables are used to carry or accumulate information along the traversals. Fig. 8 presents an example of this situation.

4.3 Translation

AP/S++ is a compile-time MOP. We have a translation algorithm that implements the intended semantics for `traverse` at compile-time, so there is no run-time overhead. As an example of how the algorithm works, Fig. 9 presents the partial translation of the code shown in Fig. 6. (Comments were inserted in order to explain the translation)

The translation of AP/S++ programs into plain S++ programs involves other issues that are not shown here, such as extending the interface of the application classes and implementing member functions that interact with the meta-level information at run-time

```
(member document search ((aword string)) void
  (letrec ((document--handler ;; handler for document objects
             (lambda (this) (typedecl this document)
               (let loop ((objs (-> this objs))) ;; get doc-objects list
                 (if (not (null? objs)) ;; end of doc-objects list?
                   (begin               ;; no
                     (let ((_aref (car objs)))  ;; get a doc-object,
                       (typedecl _aref doc-obj) ;; which may be of
                       (case (-> _aref _class)  ;; different types: dispatch
                         ((paragraph) (paragraph--handler _aref))
                         ((table) (table--handler _aref))
                         ((doc-obj) #t)))
                       (loop (cdr objs))))))) ;; loop over rest of objs
           (paragraph--handler ;; handler for paragraph objects
             (lambda (this) (typedecl this paragraph)
               (let loop ((lines (-> this lines))) ;; get lines list
                 (if (not (null? lines)) ;; end of lines list?
                   (begin ;; no
                     (let ((_aref (car lines))) ;; get a line
                       (line--handler _aref))   ;; call line handler
                     (loop (cdr lines)))))))    ;; loop over rest of lines
           (table--handler ;; handler for table objects
             (lambda (this) (typedecl this table)
               (let loop ((entries (-> this entries))) ;; get entries list
                 (if (not (null? entries)) ;; end of entries list?
                   (begin ;; no
                     (let ((_aref (car entries))) ;; get a t-entry
                       (t-entry--handler _aref))  ;; call t-entry handler
                     (loop (cdr entries)))))))    ;; loop over entries list
           ;;
           ;; more handlers here
           ;;
           (word--handler ;; handler for word objects
             (lambda (this) (typedecl this word)
               (if (eq? (-> this val) aword) ;; the wrapper
                 (-> this highlight)))))

     (document--handler this))) ;; initial call on 'this' document object
```

Figure 9: *The result of the compile-time* AP^{MOP}*; the AP/S++ code in Fig. 6 is translated into this code.*

(for example, the member function `_class` that returns the class name and allows the implementation of dynamic dispatching).

4.4 The AP Evolutionary Protocol

From an evolutionary perspective, this kind of metaobjects can also serve to evolve the application at run-time. By having both the base- and the meta-level objects executing at the same time, we can implement an evolutionary metaobject protocol that deals with requests such as *add-class*, *remove-class*, *add-reference*, *remove-reference*, etc. The CLOS MOP [KdRB91], for example, provides such capabilities. Those capabilities were further studied for AP by Hürsch (chapter 15 in [Hür95]), but they have not been implemented in AP/S++. The idea is very interesting, and poses some challenging problems that have been partially solved in the study done by Hürsch.

5 Related Work

AP/S++ is closely related to Demeter/C++ [Lie96]. Traversals in class graphs were first proposed by Lieberherr in [Lie92], and since then evolved to a new programming paradigm and a new programming language implemented by the Demeter system. We show what the editor example used throughout this paper would look like in Demeter/C++:

```
// Definition of the class graph
Document = <objs> List(DocObject).       // concrete class
DocObject : Paragraph | Table *common* . // abstract class
Paragraph = <lines> List(Lines).
Line = <words> List(Word).
Table = <n_cols> DemNumber <n_rows> DemNumber <entries> List(T_entry).
T_entry = <words> List(Word).
Word = <val> DemString.
List(C) ~ {C}.                           // repetition class

// implementation of search
*operation* void search(DemString *aword)
  *traverse*
    *from* Document *to* Word
    *wrapper* Word *suffix*
    (@ if (*this->val == *aword) this->highlight(); @)
```

The Demeter system includes its own language - which is, in fact, a meta-language for object-oriented programs. The implementation of Demeter – not the conceptual adaptive programming model – imposes a specific data model that doesn't cover all the possibilities of C++: for example, arrays are not supported and there is the requirement for the Abstract Superclass Rule [Hür94]. Furthermore, the operations are not defined as member functions of classes; instead they are functions – called *propagation patterns* – that don't belong to any particular class, include only one traversal and use transportation of objects as the base for sharing variables along the traversal.

The work proposed by Seiter et al. in [SHL95] is another instantiation of adaptive programming. This work proposes a programming language construct called *adaptive behavioral component* (ABC) that is similar to Demeter's propagation patterns but enhances the mechanism of composition by defining local variables which are scoped within the ABC, instead of the transportation of variables. It also allows for parameterization of the ABCs, that can receive traversal directives and classes as parameters. ABCs and AP/S++ share the notion of scoping in which the traversals take place. AP/S++, unlike ABCs, doesn't support parameterization of member functions. ABCs, just like Demeter, define a new programming (meta-)language.

This is where AP/S++ diverges from both Demeter/C++ and ABCs. AP/S++ is implemented as a MOP for an ordinary object-oriented language. There are no significant extensions to the base OOPL, other than the possibility of using the meta-interface `traverse` and the extended type system (i.e., parameterized lists and arrays). There may be several traversal calls in the implementation of one member function, and the source of the traversals can be any object.

While the Demeter system and language are very simple to learn, the idea of programming with class graphs instead of classes and propagation patterns instead of member functions is sometimes viewed with suspicion by experienced OO programmers. One of the reasons is that the abstraction boundaries of AP have been somehow, fuzzy. Analyzing AP under the perspective of reflective systems is one important step for clearing out the goals and the range of application for AP. AP/S++ helps to identify the abstraction boundaries of AP, and proposes one implementation that is integrated with the OOPL without imposing a new programming paradigm.

AP/S++ is strongly influenced by the work in reflection and MOPs, but we couldn't find any application of reflection for the same purposes as AP. So far most of the applications of MOPs have been for reifying or altering the implementation or semantics of some base-level construct (EuLisp [Pad92], CLOS [KdRB91], OpenC++ [Chi95] and many others). The AP^{MOP} is a very simple MOP for the specific purpose of writing object-oriented code in a structure-shy manner, and it relates to the introspection capabilities of those other MOPs.

Also, most of the MOPs in the literature are run-time MOPs and strongly differ from AP/S++ both in purpose and in implementation. However, a few compile-time MOPs have been presented in the literature, and AP/S++ shows some similarities with them. Anibus [Rod92] and Intrigue [LKRR92] are compile-time MOPs for Scheme programs. CRML [HS93] is a meta-language for Standard ML; although different in purpose, CRML has strong similarities with the Demeter System, in the sense that it is implemented as a meta-language that manipulates object programs. OpenC++ [Chi95] is a compile-time MOP for C++ that provides a generic mechanism for language extensions. Anibus, OpenC++ and AP/S++ are all implemented as preprocessors.

6 Conclusions and Future Work

Maintaining and evolving large applications written in object-oriented programming languages is hard and time-consuming. We have identified one of the obstacles to evolution which we called "structural anomaly" of OO programs. As a consequence of this anomaly, when the application structure evolves, the structural knowledge paths that exist in the

implementation of the methods may be broken, and the programmers must fix them manually; this task is more tedious than creative and consumes a significant part of the evolution effort.

We have presented AP/S++, the implementation of a metaobject protocol that allows writing object-oriented programs in a structure-shy manner. AP/S++ uses the AP concept of *traversal directive* which encapsulates the structural knowledge paths that exist in OO programs. Traversal directives are succinct specifications of sets of paths in the class graph, and, being succinct, they are tolerant to some changes in the structure of the classes; the specific set of paths of a traversal directive depends on the specific class graph for which the directives will be expanded. The base-level programs may use the meta-level information by calling the special form `traverse` which takes as parameter a list of traversal directives with wrappers of base-level code to be executed along the traversal. AP/S++ is a compile-time MOP that translates traversal calls into pieces of S++ code, and has no run-time overhead.

AP/S++ has been prototyped at Xerox PARC with the specific goal of studying how Adaptive Programming relates to reflection and MOPs. The conclusion is that AP/S++ is a win for both worlds: on the one hand, it shows a new application for reflection, namely using the structural meta-level information to write base-level programs in a structure-shy manner, resulting in programs that are easier to evolve; on the other hand, it helps to identify the abstraction boundaries of AP. At the same time, it proposes an implementation that can easily be reproduced in many different OOPLs without imposing a new programming paradigm. We are currently studying how the AP^{MOP} can be implemented for C++.

Acknolwdgements:
To the Open Implementation group at Xerox PARC, that made this work possible. Special thanks to Shigeru Chiba, Anurag Mendheka, John Lamping and Jean-Marc Loingtier for their feedback on earlier versions of this paper. Thanks also to Linda Seiter for the fruitful discussions on AP.

References

[Chi95] Shigeru Chiba. A metaobject protocol for C++. In *10th Annual Conference on Object-oriented Programming Systems, Languages and Applications*, volume 30 of *ACM SIGPLAN Notices*, pages 285–299, Austin, Texas, October 1995. ACM Press.

[Chi96] Shigeru Chiba. *A Study on a Compile-time Metaobject Protocol.* PhD thesis, Graduate School of Science, The University of Tokyo, Japan, October 1996. (To Appear).

[dS84] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *ACM Conference on LISP and Functional Programming*, pages 331–347, Austin, TX, 1984. How 3-lisp is implemented.

[HS93] J. Hook and T. Sheard. A semantics of compile-time reflection. Technical Report 93-019, Dept. of Computer Science and Engineering, Oregon Graduate Institute, Portland, Oregon, 1993.

[HS96] Walter L. Hürsch and Linda M. Seiter. Automating the Evolution of Object-Oriented Systems. In *2nd International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, March 1996. Springer-Verlag.

[Hür94] Walter L. Hürsch. Should superclasses be abstract? In Mario Tokoro, editor, *Lecture Notes in Computer Science*, pages 12–31, Bologna, Italy, July 1994. ECOOP'94, Springer-Verlag.

[Hür95] Walter Hürsch. *Maintaining Consistency and Behavior of Object-Oriented Systems during Evolution.* PhD thesis, Northeastern University, Boston, MA, September 1995.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Lie92] Karl J. Lieberherr. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In J. van Leeuwen, editor, *Information Processing '92, 12th World Computer Congress*, pages 179–185, Madrid, Spain, 1992. Elsevier.

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.

[LKRR92] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An architecture for an open compiler. In A. Yonezawa and B. C. Smith, editors, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture*, pages 95–106, 1992.

[Lop96] Cristina Videira Lopes. Adaptive parameter passing. In *2nd International Symposium on Object Technologies for Advanced Software*, Kanazawa, Japan, March 1996. Springer-Verlag.

[LSX94] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Commun. of the ACM*, 37(5):94–101, May 1994.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155, 1987.

[Mar95] R. Martin. *Designing Object-oriented C++ applications using the Booch method.* Prentice Hall, 1995.

[Pad92] *The EuLisp Definition*, April 1992. Draft.

[PXL95] 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.

[Rod92] L. Rodriguez. A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler. In A. Yonezawa and B. C. Smith, editors, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architecture*, pages 107–112, 1992.

[SHL95] Linda M. Seiter, Walter L. Hürsch, and Karl J. Lieberherr. Composing Collaborations Using Adaptive Behavioral Components. Technical Report NU-CCS-95-21, College of Computer Science, Northeastern University, Boston, MA, December 1995.

[SLHS94] Ignacio Silva-Lepe, Walter Hürsch, and Greg Sullivan. A Demeter/C++ Report. *C++ Report, SIGS Publication*, February 1994.

[WMH93] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, pages 75–80, January 1993.