Efficient Implementation of Adaptive Software

JENS PALSBERG, CUN XIAO, and KARL LIEBERHERR
Northeastern University

Adaptive programs compute with objects, just like object-oriented programs. Each task to be accomplished is specified by a so-called propagation pattern which traverses the receiver object. The object traversal is a recursive descent via the instance variables where information is collected or propagated along the way. A propagation pattern consists of (1) a name for the task, (2) a succinct specification of the parts of the receiver object that should be traversed, and (3) code fragments to be executed when specific object types are encountered. The propagation patterns need to be complemented by a class graph which defines the detailed object structure. The separation of structure and behavior yields a degree of flexibility and understandability not present in traditional object-oriented languages. For example, the class graph can be changed without changing the adaptive program at all. We present an efficient implementation of adaptive programs. Given an adaptive program and a class graph, we generate an efficient object-oriented program, for example, in C++. Moreover, we prove the correctness of the core of this translation. A key assumption in the theorem is that the traversal specifications are consistent with the class graph. We prove the soundness of a proof system for conservatively checking consistency, and we show how to implement it efficiently.

Categories and Subject Descriptors: D.2.m [**Software Engineering**]: Miscellaneous—*reusable software*; D.3.2 [**Programming Languages**]: Language Classifications—*object-oriented languages*; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Languages

Additional Key Words and Phrases: Adaptive software, correctness proof, object traversal

1. INTRODUCTION

1.1 Background

One goal of object-oriented programming is to obtain flexible software, through such mechanisms as inheritance and late binding. For example, flexibility was one of the goals in the project led by Booch [1990] where an Ada package was converted to a C++ component library. He used templates to parameterize certain components so that local substitutions are possible. But the degree of variability of such components is limited. Later he stated [Booch 1994]: “Building frameworks is hard. In crafting general class libraries, you must balance the needs for functionality, flexi-

ACM Transactions on Programming Languages and Systems, 17(2):264–292, 1995.
The authors were partially supported by the National Science Foundation under grant numbers CCR-9102578 (Software Engineering), CDA-9015692 (Research Instrumentation), and CCR-9402486 (Software Engineering).
Authors' address: Northeastern University, College of Computer Science, 161 Cullinane Hall, Boston, MA 02115-9959; email: {palsberg;cunxiao;lieberherr}@ccs.neu.edu.

bility, and simplicity. Strive to build flexible libraries, because you can never know exactly how programmers will use your abstractions. Furthermore, it is wise to build libraries that make as few assumptions about their environments as possible so that programmers can easily combine them with other class libraries."

A key feature of most popular approaches to object-oriented programming is to attach every method of a program explicitly to a specific class. As a result, when the class structure changes, the methods often need modifications as well. In Gibbs et al. [1990], we read "... the class hierarchy may become a rigid constraining structure that hampers innovation and evolution."

The idea of *adaptive* programs has been presented in papers by the second and third author and their colleagues (see Keszenheimer [1993], Lieberherr [1992; 1995], Lieberherr and Xiao [1993a; 1993b], and Lieberherr et al. [1992; 1994]). The basic idea is to *separate* the program text and the class structure. The result is called an adaptive program. It is a collection of *propagation patterns*, and it computes with objects, just like object-oriented programs. Each propagation pattern accomplishes a specific task by traversing the receiver object. In a corresponding object-oriented program, the same task may require a family of methods specified in several classes. The object traversal is a recursive descent via the instance variables where information is collected or propagated along the way. A propagation pattern consists of

(1) name for the task,
(2) succinct specification of the parts of the receiver object that should be traversed, and
(3) code fragments to be executed when specific object types are encountered.

The separation of structure and behavior yields a degree of flexibility and understandability not present in traditional object-oriented languages. For example, the class graph can be changed without changing the adaptive program at all. Moreover, with adaptive software, it is possible to make a first guess on a class graph, and later with minimal effort change to a new class graph. In contrast, if we write, for example, a C++ program, then it usually needs significant updates to work on another class graph.

1.2 Our Results

In this article, we present an efficient implementation of adaptive programs. Given an adaptive program and a class graph, we generate an object-oriented program, for example, in C++. Moreover, we prove the correctness of the core of this translation. A key assumption in the theorem is that the traversal specifications are consistent with the class graph. We prove the soundness of a proof system for conservatively checking consistency, and we show how to implement it efficiently.

The translation of an adaptive program and a class graph into a C++ program is implemented in the Demeter system. The Demeter system itself is an adaptive program, compiled by itself to C++.

1.3 Example

We now give an example of adaptive programming. Along the way, we will informally introduce the concepts that will be defined and used in Sections 2, 3, and

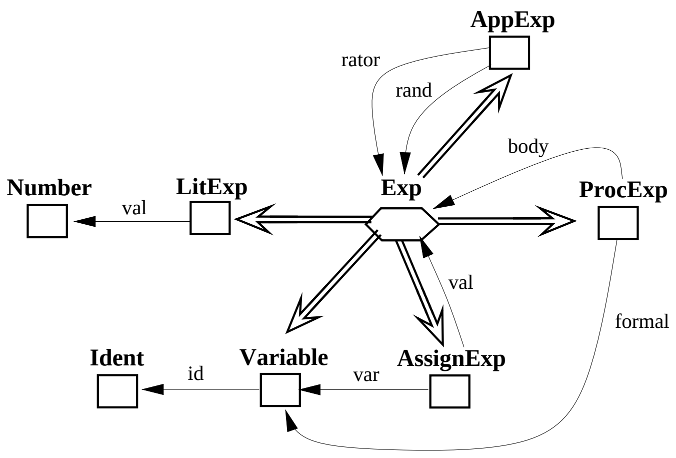
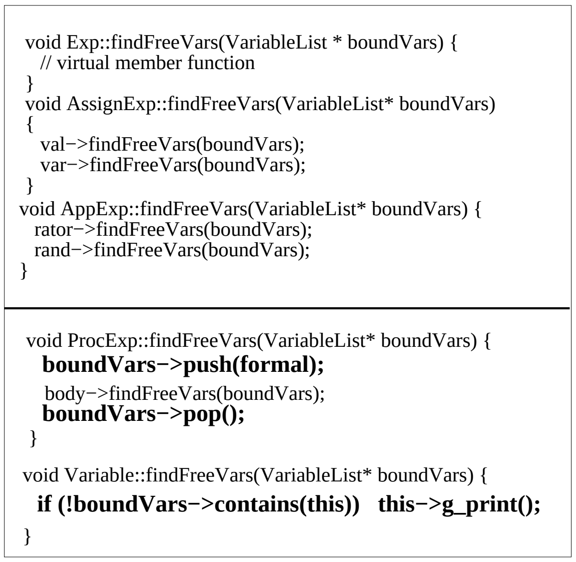


Fig. 1. Class graph.



```
void Exp::findFreeVars(VariableList * boundVars) {
   // virtual member function
}
void AssignExp::findFreeVars(VariableList* boundVars)
{
   val−>findFreeVars(boundVars);
   var−>findFreeVars(boundVars);
}
void AppExp::findFreeVars(VariableList* boundVars) {
   rator−>findFreeVars(boundVars);
   rand−>findFreeVars(boundVars);
}
```

```
void ProcExp::findFreeVars(VariableList* boundVars) {
   boundVars−>push(formal);
   body−>findFreeVars(boundVars);
   boundVars−>pop();
}
void Variable::findFreeVars(VariableList* boundVars) {
   if (!boundVars−>contains(this))   this−>g_print();
}
```

Fig. 2. C++ program.

```
operation void findFreeVars(VariableList* boundVars)
  traverse
    [Exp, Variable]
  wrapper ProcExp
    prefix
      { boundVars->push(formal); }
    suffix
      { boundVars->pop(); }
  wrapper Variable
    prefix
      { if (!boundVars->contains(this)) this->g_print(); }
```

Fig. 3. Adaptive program.

4. Suppose we want to write a C++ program to print out all free variables in a Scheme expression. We will do that by first writing an adaptive program and then generating the C++ program.

While analyzing the problem, we identify several classes and relationships, yielding the class graph shown in Figure 1. (For simplicity, the example does not cover all of Scheme.) We take this graph as our first guess on a class graph for solving the problem. The figure uses two kinds of classes: *concrete* classes (drawn as □), which are used to instantiate objects, and *abstract* classes (drawn as ⬡), which are not instantiable. The figure uses two kinds of edges: *subclass* edges (drawn as $\Longrightarrow$) representing kind-of relations, and *construction* edges (drawn as $\longrightarrow$ and with labels) representing has-a relations. For example, the subclass edge Exp$\Longrightarrow$ LitExp means that class Exp is a superclass of class LitExp; the construction edge LitExp$\xrightarrow{\text{val}}$ Number means that class LitExp has a part called val of type Number.

Should we write this program directly in C++ (see Figure 2), a natural solution is to write methods first, all called `findFreeVars`, for the following classes: Exp, Variable, AssignExp, ProcExp, and AppExp. (An explanation of C++ terminology and syntax is given in the Appendix.) The C++ program has two ingredients: traversal and processing. The traversal is specified by the C++ code in Figure 2 which is not in bold font. It finds all Variable objects in an Exp object. The processing is the code in bold font which maintains a stack of bound variables and checks whether a variable found is a free variable by using the stack.

We use the adaptive program in Figure 3 to specify an equivalent C++ program. Compared with the C++ program, the adaptive program is shorter than the one in Figure 1. (Later, we will demonstrate how it can be combined with class structures other than the one in Figure 1.)

The adaptive program in Figure 3 contains just one propagation pattern (because the problem to be solved is simple). The propagation pattern consists of a signature, a traversal specification, and some code wrappers. The propagation pattern specifies a collection of collaborating methods, as described in the following. The code fragments to be executed during traversals are called wrappers; they are written in C++.

Informally, the traversal specification, `[Exp, Variable]`, describes a traversal of Exp objects: traverse an Exp object; locate all Variable objects nested inside. We call this specification fragment a *traversal specification.*

$$
\begin{aligned}
&((\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rator}) + \\
&\ (\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rand}) + \\
&\ (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{body}) + \\
&\ (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{val}))^* \, ((\texttt{Exp},\diamond,\texttt{Variable}) + \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{var},\texttt{Variable}) + \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{formal},\texttt{Variable}))
\end{aligned}
$$

Fig. 4. Regular expressions describe traversals.

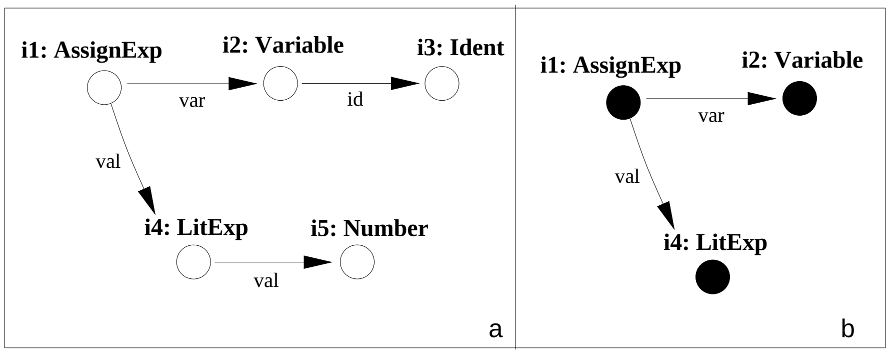


Fig. 5. `Exp` object.

We interpret this traversal specification as specifying the set of paths from Exp to Variable. A path is described by an alternating sequence of nodes and labels. The set of paths can be described by the regular expression shown in Figure 4.

The regular expression is a concatenation of two subexpressions. The first half is a Kleene-closure of a union of four expressions. The second half is a union of three expressions. A sentence of the regular language is an alternating sequence of nodes and labels (construction edge labels or ◇). A diamond between two vertices means that the second vertex is a subclass of the first vertex. For example, the sentence

$$\texttt{Exp}, \diamond, \texttt{Variable}$$

corresponds to the path

$$\texttt{Exp} \Longrightarrow \texttt{Variable}\ ;$$

and the sentence

$$\texttt{Exp}, \diamond, \texttt{AssignExp}, \texttt{val}, \texttt{Exp}, \diamond, \texttt{AssignExp}, \texttt{var}, \texttt{Variable}$$

corresponds to the path

$$\texttt{Exp} \Longrightarrow \texttt{AssignExp} \xrightarrow{\texttt{val}} \texttt{Exp} \Longrightarrow \texttt{AssignExp} \xrightarrow{\texttt{var}} \texttt{Variable}\ .$$

We use the set of paths to guide the traversal of an `Exp` object. Consider the graph in Figure 5(a) representing a particular object instance of the class graph in Figure 1.

There are five object nodes in the graph; `i1`, `i2`, `i3`, `i4`, and `i5` are object identifiers. The names after colons are the classes of the objects. The edges with labels are part-of relationships between the objects. We want to traverse the object graph starting from node i1, being guided by the set of paths above. Since the class of node i1 is AssignExp, we first select all the paths from the set that begin with Exp,◇,AssignExp. These paths are described by the following regular expression.

$$\begin{array}{l}
(\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{val})((\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rator}) + \\
\qquad\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rand}) + \\
\qquad\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{body}) + \\
\qquad\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{val}))^* \\
\qquad\qquad\qquad\qquad\qquad ((\texttt{Exp},\diamond,\texttt{Variable}) + \\
\qquad\qquad\qquad\qquad\qquad\; (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{var},\texttt{Variable}) + \\
\qquad\qquad\qquad\qquad\qquad\; (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{formal},\texttt{Variable})) + \\
(\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{var},\texttt{Variable})
\end{array}$$

Moreover, we remove the prefix Exp,◇ from the paths, since this prefix only gives the insignificant information that AssignExp is a subclass of Exp. We are then left with a set of paths described by the following regular expression which we denote E.

$$\begin{array}{l}
(\texttt{AssignExp},\texttt{val})((\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rator}) + \\
\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rand}) + \\
\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{body}) + \\
\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{val}))^* \\
\qquad\qquad\qquad\qquad ((\texttt{Exp},\diamond,\texttt{Variable}) + \\
\qquad\qquad\qquad\qquad\; (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{var},\texttt{Variable}) + \\
\qquad\qquad\qquad\qquad\; (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{formal},\texttt{Variable})) + \\
(\texttt{AssignExp},\texttt{var},\texttt{Variable})
\end{array}$$

Having visited the object i1, we continue with visiting part objects of i1. There are two such parts, called val and var. To visit the val part, which is the object i4, we first select all those paths that begin with AssignExp,val,Exp, and we remove the prefix AssignExp,val, yielding:

$$\begin{array}{l}
((\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rator}) + \\
\;(\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rand}) + \\
\;(\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{body}) + \\
\;(\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{val}))^*((\texttt{Exp},\diamond,\texttt{Variable}) + \\
\qquad\qquad\qquad\qquad\qquad\; (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{var},\texttt{Variable}) + \\
\qquad\qquad\qquad\qquad\qquad\; (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{formal},\texttt{Variable}))
\end{array}$$

Since the class of the object i4 is LitExp, we will select paths from the set described by the above regular expression that begins with (Exp,◇,LitExp). However, there is none. Therefore, the traversal stops at node i4. Notice that we let the set of paths guide the traversal as long as possible. This is the reason why the traversal visits node i4; only from the run-time information about the contents of the val-part can it be determined if we should continue the traversal or not. For example, we could have a ProcExp object in the val part. In that case we would have to traverse further. When the traversal meets i4, it simply abandons that path.

To visit the var part of i1, which is the object i2, we first select those paths described by the regular expression E that begin with Exp,◇,Variable, and we

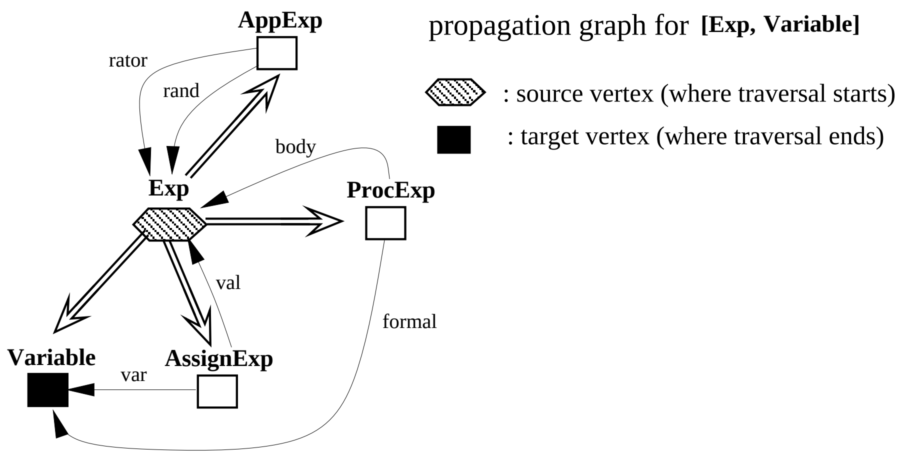

propagation graph for [Exp, Variable]
: source vertex (where traversal starts)
: target vertex (where traversal ends)
AppExp
rator
rand
body
Exp
ProcExp
val
formal
Variable
AssignExp
var

Fig. 6. Propagation graph.

```
void
Exp::findFreeVars(VariableList * boundVars) {
  // virtual member function
}
void
AssignExp::findFreeVars(VariableList* boundVars)
{
  val−>findFreeVars(boundVars);
  var−>findFreeVars(boundVars);
}
void
AppExp::findFreeVars(VariableList* boundVars) {
  rator−>findFreeVars(boundVars);
  rand−>findFreeVars(boundVars);
}
```

```
void
ProcExp::findFreeVars(VariableList* boundVars) {

  formal−>findFreeVars(boundVars);
  body−>findFreeVars(boundVars);

}

void
Variable::findFreeVars(VariableList* boundVars) {

}
```

Fig. 7. Traversal skeleton.

```
void
Exp::findFreeVars(VariableList * boundVars)
{
  // virtual member function
}

void
AssignExp::findFreeVars(VariableList* boundVars)
{
  val−>findFreeVars(boundVars);
  var−>findFreeVars(boundVars);
}

void
AppExp::findFreeVars(VariableList* boundVars)
{
  rator−>findFreeVars(boundVars);
  rand−>findFreeVars(boundVars);
}
```

```
void
ProcExp::findFreeVars(VariableList* boundVars)
{
  boundVars−>push(formal);
  formal−>findFreeVars(boundVars);
  body−>findFreeVars(boundVars);
  boundVars−>pop();
}

void
Variable::findFreeVars(VariableList* boundVars)
{
   if (!boundVars−>contains(this))
       this−>g_print();
}
```

Fig. 8. Generated C++ program.

remove the prefix `Exp`,◇, yielding just `Variable`. After visiting the object `i2`, we check how many part objects that need to be visited further. Since there is no outgoing edge from `Variable` on the path, the traversal stops at node `i2`.

The nodes which are visited are marked black in Figure 5(b).

Cycles in the object graph may lead to a nonterminating traversal. In the Demeter system, one can handle such situations by inserting appropriate code into the code wrappers. This possibility will not be discussed further in this article.

We now indicate how to implement the above traversal efficiently. The set of paths described by the traversal specification forms a graph called the *propagation graph*; see Figure 6. This graph is a subgraph of the class graph in Figure 1. The propagation pattern in Figure 3 is then translated into C++ as follows.

—Each class in the propagation graph gets a method with the interface specified on line 1 in Figure 3. Methods of abstract classes are virtual (for an explanation of the term virtual, see the Appendix).

—If a class has an outgoing construction edge in the propagation graph, then the method will contain a method invocation through the corresponding part.

Notice that the subclass edges in the propagation graph do not cause generation of code; the late binding of C++ gives the right behavior.

Based on these rules, the propagation graph in Figure 6 is translated into the program skeleton in Figure 7.

The code wrappers in lines 4–11 of Figure 3 enhance the traversal specification to print out free variables. The **wrapper** clause attached to class ProcExp adds one statement at the beginning and the end of the method of class ProcExp in Figure 7. The **wrapper** clause attached to class Variable adds one statement at the beginning of the method of class Variable in Figure 7. The resulting enhanced program is the one in Figure 8, where the statements in bold font are from the wrappers.

The automatically generated program in Figure 8 differs from the handwritten one in Figure 2 in just one way, as follows. The ProcExp method in Figure 8 contains an *extra* method invocation. The reason is simply that *every* outgoing construction edge causes the generation of a method invocation. For this particular example, the extra method invocation has no effect, so it does no harm. It does make the program less efficient, of course. In general, we may be interested in writing in the traversal specification that certain edges should be bypassed. For example, if we write that the edge from ProcExp to Variable should be bypassed, then the generated code should be exactly that of Figure 2. This is possible in our Demeter system, but it will not be further discussed in this article.

Suppose we change the class graph by adding two new classes IfExp and VarExp as subclasses of Exp, letting the class Variable be a part class of VarExp, and renaming the labels var and val to rvalue and lvalue respectively. The resulting class graph is in Figure 9. Had we written the C++ program by hand, it would need considerable change. In contrast, the adaptive program needs no change at all. The C++ program generated from the adaptive program for the new class graph is in Figure 10. The example indicates that compared to object-oriented software, adaptive software can be shorter and more flexible, and therefore easier to understand and maintain.

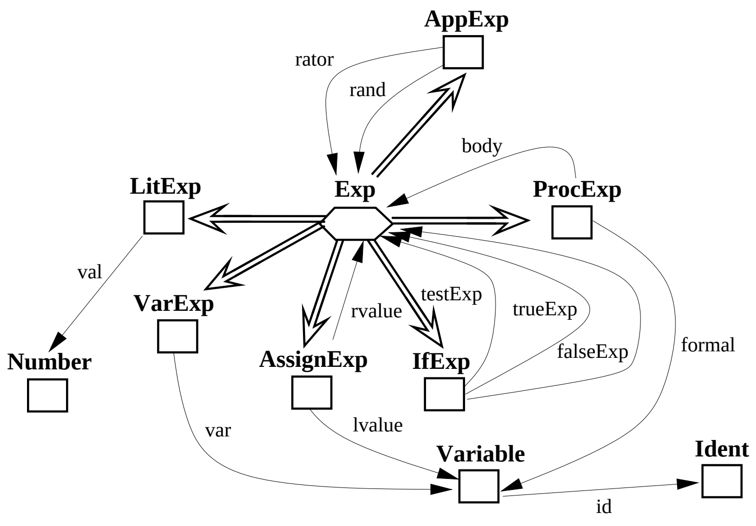


Fig. 9. Another class graph.

```
void
Exp::findFreeVars(VariableList * boundVars)
{
  // virtual member function
}

void
VarExp::findFreeVars(VariableList* boundVars)
{
 var−>findFreeVars(boundVars);
}

void
AssignExp::findFreeVars(VariableList* boundVars)
{
  lvalue−>findFreeVars(boundVars);
  rvalue−>findFreeVars(boundVars);
}

void
IfExp::findFreeVars(VariableList* boundVars)
{
  testExp−>findFreeVars(boundVars);
  trueExp−>findFreeVars(boundVars);
  falseExp−>findFreeVars(boundVars);
}
```

```
void
AppExp::findFreeVars(VariableList* boundVars)
{
  rator−>findFreeVars(boundVars);
  rand−>findFreeVars(boundVars);
}


void
ProcExp::findFreeVars(VariableList* boundVars)
{
 boundVars−>push(formal);
  formal−>findFreeVars(boundVars);
  body−>findFreeVars(boundVars);
 boundVars−>pop();
}

void
Variable::findFreeVars(VariableList* boundVars)
{
  if (!boundVars−>contains(this))
      this−>g_print();
}
```

Fig. 10. Adapted C++ program.

1.4 Compatibility, Consistency, and Subclass Invariance

When generating an object-oriented program from an adaptive program and a class graph, we require the traversal specifications to be *compatible* and *consistent* with the class graph, and we require the propagation graph determined by the traversal specification to be a *subclass-invariant* subgraph of the class graph. This section gives an informal motivation for these concepts.

The notions of compatibility, consistency, and subclass invariance are tied to the concept of a propagation graph which was briefly mentioned in Section 1.3. The propagation graph is the starting point when generating code from a traversal specification. Thus, when given a propagation pattern and a class graph, the first task is to compute the propagation graph. The propagation graph represents the paths to be traversed. This set of paths may be infinite, yet the propagation graph represents it compactly. Intuitively, compatibility, consistency, and subclass invariance can be understood as follows.

—*Compatibility.* The propagation graph represents at least some paths.

—*Consistency.* The propagation graph represents at most the specified paths.

—*Subclass invariance.* Any two nodes in the propagation graph have a subclass path between them if they do in the class graph.

These three conditions ensure the correctness of the rules from the previous section for generating efficient traversal code. If the specification is not compatible with the class graph, then the traversals may not reach the specified subobjects. If the specification is not consistent with the class graph, or the propagation graph is not a subclass-invariant subgraph of the class graph, then the traversals would "go wrong" as illustrated below.

We might attempt to compile adaptive programs without the above three conditions. This would require another representation of the paths. Currently, we do not know how to do that efficiently, however; so we prefer to outlaw class graphs that lead to violation of the above conditions. Our experience with the Demeter system indicates that the three conditions are met by typical programs. Moreover, in cases where the conditions are violated, it is usually straightforward to decompose the traversal specification such that each of the components meets the conditions.

Checking compatibility is straightforward: compute the propagation graph and check if it represents some paths. Checking subclass invariance is also straightforward: compute the propagation graph, and for each node pair in the propagation graph, check that if they are connected by subclass edges in the original graph, then they are also connected by subclass edges in the propagation graph. Checking consistency, however, is nontrivial, and Section 4 is devoted to this problem.

Traversal specifications can be combined in several ways, for example, by "concatenation of paths" and "union of sets of paths." By analogy with type checking, we want *compositional* consistency checking. Thus, when we combine two specifications which both are consistent with a graph, we want to check that also their combination is consistent with that graph. In Section 4 we present a compositional proof system for checking that, and we prove it to be sound. We also give an efficient algorithm for checking compositional consistency. Here, we give an informal outline of the system.

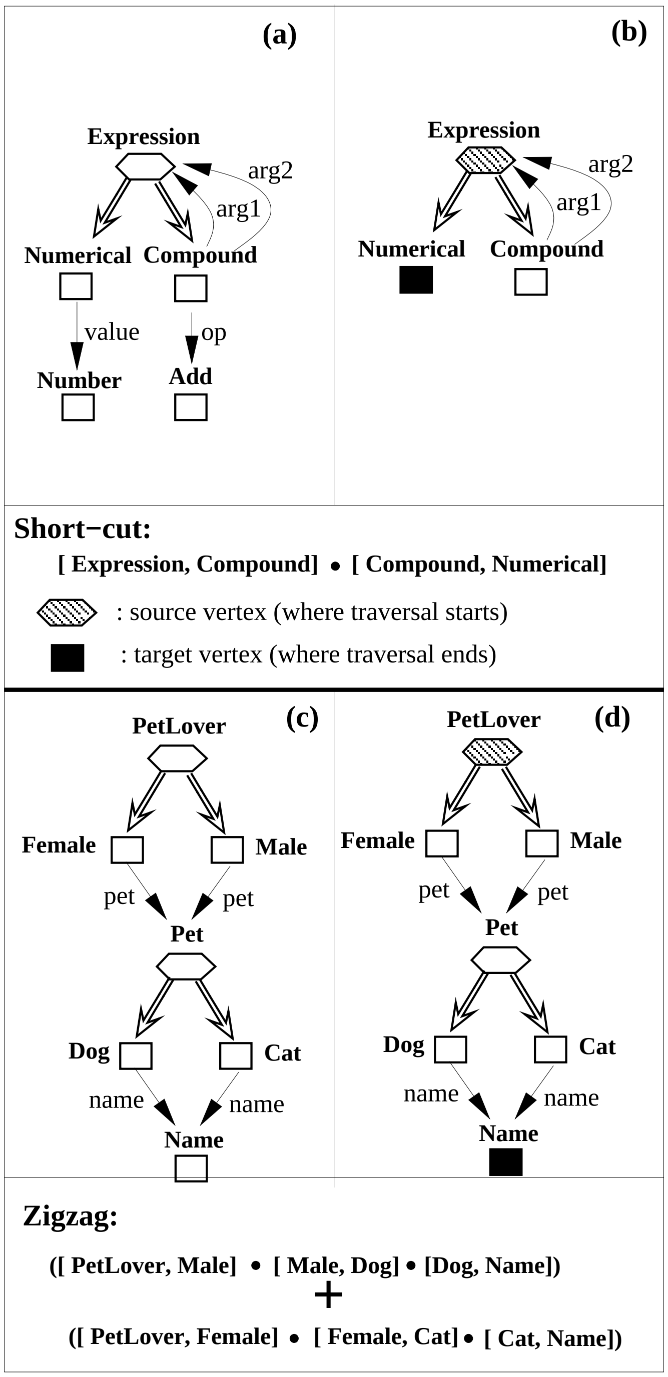
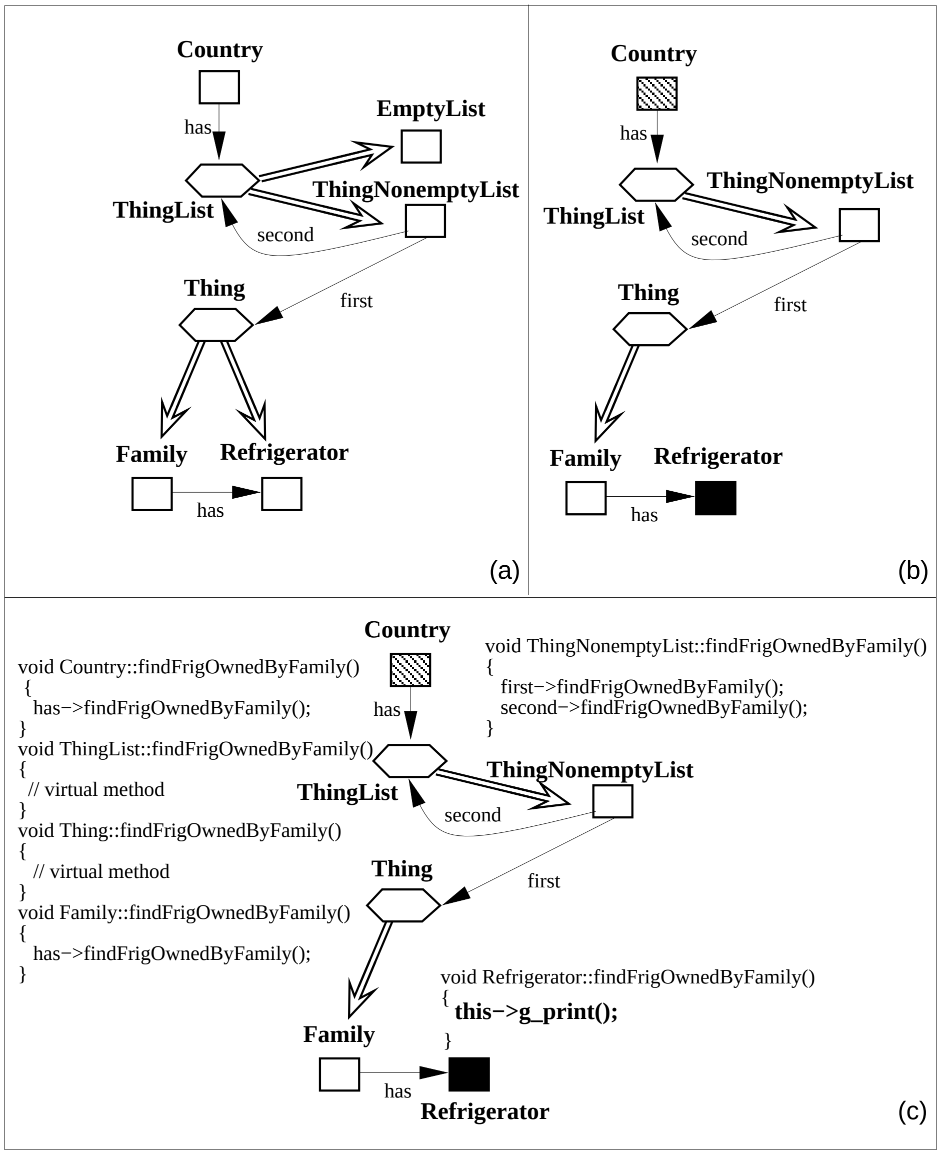

(a)
Expression
arg2
arg1
Numerical
Compound
value
op
Number
Add
(b)
Expression
arg2
arg1
Numerical
Compound
Short−cut:
[Expression, Compound] • [Compound, Numerical]
: source vertex (where traversal starts)
: target vertex (where traversal ends)
PetLover
(c)
Female
Male
pet
pet
Pet
Dog
Cat
name
name
Name
PetLover
(d)
Female
Male
pet
pet
Pet
Dog
Cat
name
name
Name
Zigzag:
([PetLover, Male] • [Male, Dog] • [Dog, Name])
+
([PetLover, Female] • [Female, Cat] • [Cat, Name])

Fig. 11. Inconsistency.



```
void Country::findFrigOwnedByFamily()
 {
  has−>findFrigOwnedByFamily();
}
void ThingList::findFrigOwnedByFamily()
{
  // virtual method
}
void Thing::findFrigOwnedByFamily()
{
  // virtual method
}
void Family::findFrigOwnedByFamily()
{
  has−>findFrigOwnedByFamily();
}
```

```
void ThingNonemptyList::findFrigOwnedByFamily()
{
  first−>findFrigOwnedByFamily();
  second−>findFrigOwnedByFamily();
}
```

```
void Refrigerator::findFrigOwnedByFamily()
{ this−>g_print();
}
```

Fig. 12. Violation of subclass invariance.

```
operation void findFrigOwnedByFamily()
  traverse
    [Country, Family] · [Family, Refrigerator]
  wrapper Refrigerator
    prefix
      { this->g_print(); }
```

Fig. 13. Find refrigerators owned by families.

Consider the class graph in Figure 11(a). The traversal specification on the left bottom describes a traversal of an Expression object. This traversal visits all the Numerical objects nested in Compound objects in an Expression object. In Figure 11(b) is the corresponding propagation graph. Unfortunately, this propagation graph has a path from Expression to Numerical that does not go through Compound. We call such a path a *short cut*. The efficient traversal code generated from the propagation graph will visit a Numerical object even if it is not nested in a Compound object.

Consider then the class graph in Figure 11(c). The propagation specification on the right bottom describes a traversal of a PetLover object. This specification says that we want to visit a Name object which is nested in a Cat object owned by a female pet lover, or is nested in a Dog object owned by a male pet lover. In Figure 11(d) is the corresponding propagation graph. Unfortunately, this propagation graph has two paths which do not satisfy the specification: $\mathsf{PetLover} \Longrightarrow \mathsf{Male} \xrightarrow{pet} \mathsf{Pet} \Longrightarrow \mathsf{Cat} \xrightarrow{name} \mathsf{Name}$ and $\mathsf{PetLover} \Longrightarrow \mathsf{Female} \xrightarrow{pet} \mathsf{Pet} \Longrightarrow \mathsf{Dog} \xrightarrow{name} \mathsf{Name}$. We call such paths *zigzag paths*. The efficient traversal code generated from the propagation graph will visit all Name objects no matter what.

In these cases, the traversal specifications are not *consistent* with the class graphs. Our soundness theorem states that for conservatively checking for consistency, it is sufficient to be able to rule out short cuts and zigzag paths.

Finally we show an example of the significance of the subclass invariance condition. The program in Figure 13 prints out all refrigerators owned by families. Figure 12(b) illustrates the propagation graph when the program is applied to the class graph in Figure 12(a). Notice that `Refrigerator` is a subclass of `Thing` in the class graph of Figure 12(a), but not in the propagation graph of Figure 12(b). Figure 12(c) shows the C++ code generated from the propagation graph. In the C++ code, the method attached to `Thing` is a virtual method. Because of late binding, any `Refrigerator` object, no matter whether it is a part object of a `Family` object or not, will be printed out. Subclass invariance rules out this program. Notice that we can decompose the traversal specification into [`Country`, `Family`] and [`Family`, `Refrigerator`], and it is easy to see that the propagation graphs for both of these satisfy the subclass invariance condition. Hence, the programmer can rewrite the program using *two* propagation patterns to solve the problem once it is detected.

We will now turn to the formal presentation of our results. In the following section we present the syntax and semantics of adaptive programs. In Section 3 we show the core part of the efficient implementation of adaptive programs, and we prove the corresponding correctness theorem. In Section 4 we prove the soundness of a proof system for conservatively checking consistency, and we show how to implement it efficiently. Finally, in Section 5 we compare our approach to previous work.

2. THE SEMANTICS OF ADAPTIVE PROGRAMS

In the following we first define the concepts of graphs, paths, class graphs, object graphs, traversal specifications, and wrappers, and we then present the semantics of adaptive programs and the semantics of an object-oriented target language.

2.1 Graphs

We will use graphs for three purposes: to define (1) classes (class graphs), (2) objects (object graphs), and (3) propagation graphs (subgraphs of class graphs).

A directed graph is a pair (N, E) where N is a set of nodes, and E is a set of edges where $E \subseteq N \times N$. If $(v_1, v_2) \in E$, then v_1 is the source, and v_2 is the target of (v_1, v_2).

We will use the operation $\cup$ on graphs, defined as follows. If $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$, then $G_1 \cup G_2 = (N_1 \cup N_2, E_1 \cup E_2)$.

Let $(L, \leq)$ be a totally ordered set of labels, such that $\diamond \notin L$. Define $\mathcal{L} = L \cup \{ \diamond \}$.

We will consider only graphs where each edge has a label from $\mathcal{L}$. An edge (u, v) with label l will be written $u \xrightarrow{l} v$.

If G is a graph and u is a node of G, $\mathsf{Edges}_G(u)$ denotes the set of edges from u.

2.2 Paths

A *path* in a graph is a sequence $v_1 l_1 v_2 l_2 \ldots v_n$ where $v_1, \ldots, v_n$ are nodes of the graph; $l_1, \ldots, l_{n-1}$ are labels; and $v_i \xrightarrow{l_i} v_{i+1}$ is an edge of the graph for all $i \in 1..n-1$. We call v_1 and v_n the source and the target of the path, respectively. If $p_1 = v_1 \ldots v_i$ and $p_2 = v_i \ldots v_n$, then we define the concatenation $p_1 p_2 = v_1 \ldots v_i \ldots v_n$.

Suppose P_1 and P_2 are sets of paths where all paths in P_1 have the target v and where all paths of P_2 have the source v. Then we define

$$P_1 \cdot P_2 = \{p \mid p = p_1 p_2 \text{ where } p_1 \in P_1 \text{ and } p_2 \in P_2\} .$$

Next we introduce the Reduce function which is used in the definition of several other functions. Reduce is an operator which removes zero or more leading subclass edges from a set of paths. If R is a path set, then

$$\mathsf{Reduce}(R) = \{v_n \ldots v_{n+m} \mid v_1 l_1 v_2 \ldots v_n \ldots v_{n+m} \in R, l_i = \diamond, i \in 1..n-1, m \geq 0\}$$

$$\mathsf{Head}(R) = \{v_1 \mid v_1 \ldots v_n \in \mathsf{Reduce}(R)\} .$$

Intuitively, each path in $\mathsf{Reduce}(R)$ can be obtained from a path in R by removing a prefix where all labels are $\diamond$. Note that the prefix does not have to be maximal: we can remove zero or more subclass edges. Moreover, $\mathsf{Head}(R)$ is the set of classes one can get to in R when following zero or more $\diamond$-labels.

For example, consider the graph in Figure 1, and denote the set of paths from `Exp` to `Variable` as R'; see Figure 4.

$$\mathsf{Head}(R') = \{ \texttt{Exp}, \texttt{Variable}, \texttt{AssignExp}, \texttt{ProcExp}, \texttt{AppExp} \}$$

If R is a path set, u is a node, and l is a label, then

$$\begin{aligned}
\mathsf{Select}(R,u) &= \{v_1 \dots v_n \mid v_1 \dots v_n \in \mathsf{Reduce}(R), v_1 = u\}\\
\mathsf{Car}(R,u) &= \{v_1 \overset{l_1}{\rightarrow} v_2 \mid v_1 l_1 v_2 \dots v_n \in \mathsf{Select}(R,u)\}\\
\mathsf{Cdr}(l,R,u) &= \{v_2 \dots v_n \mid v_1 l_1 v_2 \dots v_n \in \mathsf{Select}(R,u), l_1 = l\}\ .
\end{aligned}$$

Intuitively, $\mathsf{Select}(R,u)$ is the set of postfixes of paths in R where each postfix begins with u and where $u \in \mathsf{Head}(R)$. Moreover, $\mathsf{Car}(R,u)$ is the set of the first edges on such postfixes. Finally, $\mathsf{Cdr}(l,R,u)$ is the set of tails of postfixes where the head has label l.

For the same example,

$$\begin{aligned}
&\mathsf{Select}(R', \texttt{AssignExp}) =\\
&\quad \mathsf{Language}(\\
&\quad (\texttt{AssignExp},\texttt{val})((\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rator}) +\\
&\quad\qquad\qquad (\texttt{Exp},\diamond,\texttt{AppExp},\texttt{rand}) +\\
&\quad\qquad\qquad (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{body}) +\\
&\quad\qquad\qquad (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{val}))^*\\
&\quad\qquad\qquad\qquad ((\texttt{Exp},\diamond,\texttt{Variable}) +\\
&\quad\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{AssignExp},\texttt{var},\texttt{Variable}) +\\
&\quad\qquad\qquad\qquad (\texttt{Exp},\diamond,\texttt{ProcExp},\texttt{formal},\texttt{Variable})) +\\
&\quad (\texttt{AssignExp},\texttt{var},\texttt{Variable}))
\end{aligned}$$

$$\mathsf{Car}(R', \texttt{AssignExp}) = \{\texttt{AssignExp} \overset{\texttt{val}}{\rightarrow} \texttt{Exp}, \texttt{AssignExp} \overset{\texttt{var}}{\rightarrow} \texttt{Variable}\}$$

$$\mathsf{Cdr}(\texttt{var}, R', \texttt{AssignExp}) = \{\texttt{Variable}\}$$

where $\mathsf{Language}(E)$ denotes the language generated by the regular expression E.

The operator Graph maps a set of paths to the smallest graph that contains all the paths.

The set $\mathsf{Paths}_G(A,B)$ consists of all paths from A to B in the graph G.

A set R of paths is *convex* over a graph G if R is nonempty and of the form $\mathsf{Paths}_G(A,B)$. We write $\mathsf{Root}(R) = A$ and $\mathsf{Leaf}(R) = B$.

LEMMA 2.2.1. *If R is a convex path set over G, $u \in \mathsf{Head}(R)$, and $\mathsf{Car}(R,u) = \{u \overset{l_i}{\rightarrow} v_i \mid i \in 1..n\}$, then $\mathsf{Cdr}(l_i,R,u) = \mathsf{Paths}_G(v_i, \mathsf{Leaf}(R))$ for all $i \in 1..n$.*

PROOF. Immediate. □

2.3 Class Graphs

The following notion of class graph is akin to those presented in Lieberherr and Xiao [1993b] and Palsberg and Schwartzbach [1994]. A *class graph* is a finite directed graph. Each node represents either an abstract or a concrete class. The predicate $\mathsf{Abstract}$ is true of nodes that represent abstract classes, and it is false otherwise. Each edge is labeled by an element of $\mathcal{L}$. If $l \in L$, then the edge $u \overset{l}{\rightarrow} v$ indicates that the class represented by u has an instance variable with name l and with a type represented by v. Such an edge is called a *construction* edge. If $l = \diamond$, then the edge $u \overset{l}{\rightarrow} v$ indicates that the class represented by u has a subclass represented by v. Such an edge is called a *subclass* edge. If not $\mathsf{Abstract}(u)$, then there are only construction edges from u. Moreover, for each $l \in L$, there is at most one outgoing construction edge from u with label l.

The binary equality predicate on classes will be written $=_{\text{nodes}}$.

If Φ is a class graph, and u, v are nodes of Φ, then $\mathsf{Subclass}_\Phi(u, v)$ is true if $v \in \mathsf{Head}(\mathsf{Paths}_\Phi(u, v))$ and false otherwise. Intuitively, there is at least one path in Φ from u to v which consists of only subclass edges.

If Φ and Φ' are class graphs, then Φ' is a *subclass-invariant* subgraph of Φ, if Φ' is a subgraph of Φ, and for $u, v \in \Phi'$, if $\mathsf{Subclass}_\Phi(u, v)$ then $\mathsf{Subclass}'_\Phi(u, v)$.

A node v is a *Rome-node*[1] of a class graph Φ if for every node u in Φ,

$$\mathsf{Paths}_\Phi(u, v) \neq \emptyset \ .$$

Clearly, if u is a node and v is a Rome-node, then for every $u' \in \mathsf{Head}(\mathsf{Paths}_\Phi(u, v))$,

$$\mathsf{Car}(\mathsf{Paths}_\Phi(u, v), u') = \mathsf{Edges}_\Phi(u') \ .$$

The notion of Rome-node will be central in the proof of correctness of the implementation of adaptive programs.

A class graph is *flat* if for every node u where $\mathsf{Abstract}(u)$, all outgoing edges are subclass edges. We are only interested class graphs for which there exists an equivalent flat one. Two class graphs are equivalent if they define the same set of objects. Object-preserving class transformations has been studied by Bergstein [1991]. We will henceforth assume that all class graphs are flat.

2.4 Object Graphs

An *object graph* is a finite directed graph. Each node represents an object, and the function Class maps each node to "its class," that is, a concrete class in some class graph. Each edge is labeled by an element of L. The edge $u \xrightarrow{l} v$ indicates that the object represented by u has a part object represented by v. For each node u and each label $l \in L$, there is at most one outgoing edge from u with label l.

Given a class graph Φ and an object graph Ω, Ω *conforms* to Φ if for every node o of Ω, $\mathsf{Class}(o)$ is a node of Φ, and moreover

> If $\mathsf{Class}(o) \xrightarrow{l} v$ is in Φ, then there exists $o \xrightarrow{l} o'$ in Ω such that $\mathsf{Subclass}_\Phi(v, \mathsf{Class}(o'))$.

2.5 Traversal Specifications

A *traversal specification* is generated from the grammar

$$D \ ::= \ [A, B] \ \mid \ D \cdot D \ \mid \ D + D$$

where A and B are nodes of a class graph.

Our slogan is: "This language is the λ-calculus of traversal specifications." The idea is that although this language can be extended in many ways to ease programming, it does contain the essential constructs. Possible extensions include the empty specification, notation for including or excluding certain edges, and boolean connectives. In the Demeter system, we use those extensions.

A traversal specification denotes a set of paths in a given class graph Φ, intuitively as shown in Table I.

[1]All paths lead to Rome.

Table I. Traversal Specification

Directive	Set of paths
$[A, B]$	The set of paths from A to B in Φ
$D_1 \cdot D_2$	Concatenation of sets of paths
$D_1 + D_2$	Union of sets of paths

For a traversal specification to be meaningful, it has to be well formed. A traversal specification is well formed if it determines a *source* node and a *target* node, if each concatenation has a "meeting point," and if each union of a set of paths preserves the source and the target. Formally, the predicate WF is defined in terms of two functions Source and Target which both map a specification to a node.

$$\begin{aligned}
\mathsf{WF}([A,B]) &= \text{true} \\
\mathsf{WF}(D_1 \cdot D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\
&\ \mathsf{Target}(D_1) =_{\text{nodes}} \mathsf{Source}(D_2) \\
\mathsf{WF}(D_1 + D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\
&\ \mathsf{Source}(D_1) =_{\text{nodes}} \mathsf{Source}(D_2) \wedge \mathsf{Target}(D_1) =_{\text{nodes}} \mathsf{Target}(D_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Source}([A,B]) &= A & \mathsf{Target}([A,B]) &= B \\
\mathsf{Source}(D_1 \cdot D_2) &= \mathsf{Source}(D_1) & \mathsf{Target}(D_1 \cdot D_2) &= \mathsf{Target}(D_2) \\
\mathsf{Source}(D_1 + D_2) &= \mathsf{Source}(D_1) & \mathsf{Target}(D_1 + D_2) &= \mathsf{Target}(D_1)
\end{aligned}$$

$\mathsf{Source}(D)$ is the source node determined by D, and $\mathsf{Target}(D)$ is the target node determined by D.

If D is well formed, then $\mathsf{PathSet}_\Phi(D)$ is a set of paths in Φ from the source of D to the target of D, defined as follows:

$$\begin{aligned}
\mathsf{PathSet}_\Phi([A,B]) &= \mathsf{Paths}_\Phi(A,B) \\
\mathsf{PathSet}_\Phi(D_1 \cdot D_2) &= \mathsf{PathSet}_\Phi(D_1) \cdot \mathsf{PathSet}_\Phi(D_2) \\
\mathsf{PathSet}_\Phi(D_1 + D_2) &= \mathsf{PathSet}_\Phi(D_1) \cup \mathsf{PathSet}_\Phi(D_2)
\end{aligned}$$

LEMMA 2.5.1. *If* $\mathsf{WF}(D)$*, then* (i) $\mathsf{PathSet}_\Phi(D)$ *is well defined and* (ii) *each path in* $\mathsf{PathSet}_\Phi(D)$ *starts in* $\mathsf{Source}(D)$ *and ends in* $\mathsf{Target}(D)$.

PROOF. By induction on the structure of D. □

2.6 Wrappers

A *wrapper map* is a mapping from concrete classes in some class graph to *code wrappers*, that is, statements in some language, for example, C++. The idea is that when an object is processed by an adaptive program, the code wrapper for the class of that object will be executed. To ease programming, it is convenient to have both prefix and suffix wrappers, as indicated by the example in Section 1. The Demeter system supports both vertex wrappers and construction edge wrappers while we here consider only vertex wrappers. For simplicity, we will in the following consider just one wrapper for each concrete class.

2.7 Adaptive Programs

In general, an adaptive program is a collection of propagation patterns. For simplicity, we will here consider the case where there is just one propagation pattern. Such an adaptive program (D, W) consists of a well-formed traversal specification

D, and a wrapper map W. Given class graph Φ, an object graph Ω, and a node o in Ω, the semantics of (D, W) is given by the function Run:

$$\begin{aligned}\mathsf{Run}(D,W)(\Phi,\Omega,o) &= \mathsf{Execute}_W(\mathsf{Traverse}(\mathsf{PathSet}_\Phi(D),\Omega,o))\\ \mathsf{Traverse}(R,\Omega,o) &= \begin{cases} H & \text{if } \Omega \vdash_s o : R \triangleright H, \text{ for some } H\\ \bot & \text{otherwise}\end{cases}\end{aligned}$$

If Ω is an object graph, o a node in Ω, R a path set over Φ, and H a sequence of objects, then the judgment

$$\Omega \vdash_s o : R \triangleright H$$

means that when traversing the object graph Ω starting in o, and guided by the path set R, then H is the *traversal history*, that is, the sequence of objects that are traversed. Formally, this holds when the judgment is derivable using the following rule:

$$\frac{\Omega \vdash_s o_i : \mathsf{Cdr}(l_i, R, \mathsf{Class}(o)) \triangleright H_i \qquad \forall i \in 1..n}{\Omega \vdash_s o : R \triangleright o \cdot H_1 \cdot \ldots \cdot H_n} \quad \begin{array}{l}\text{if } \mathsf{Car}(R, \mathsf{Class}(o)) = \\ \qquad \{\mathsf{Class}(o) \xrightarrow{l_i} w_i \mid i \in 1..n\},\\ o \xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n, \text{ and}\\ l_j < l_k \text{ for } 1 \le j < k \le n.\end{array}$$

The label s of the turnstile indicates "semantics." The functions Car and Cdr perform the operations on sets of paths that were informally described in the example in Section 1. Notice that for $n = 0$, the rule is an axiom; it is then simply

$$\frac{}{\Omega \vdash_s o : R \triangleright o} \quad \text{if } \mathsf{Car}(R, \mathsf{Class}(o)) = \emptyset.$$

Notice that $\mathsf{Traverse}$ is well defined: if both $\Omega \vdash_s o : R \triangleright H_1$ and $\Omega \vdash_s o : R \triangleright H_2$, then $H_1 = H_2$. This can be proved by induction on the structure of the derivation of $\Omega \vdash_s o : R \triangleright H_1$.

The call $\mathsf{Execute}_W(H)$ executes in sequence the wrappers for the class of each of the objects in H. We leave $\mathsf{Execute}_W$ unspecified, since its definition depends on the language in which the code wrappers are written.

2.8 The Target Language

We will compile adaptive programs into an object-oriented target language. Given that the source language contains only adaptive programs consisting of one propagation pattern, we make the target language correspondingly simple. A program in the target language is a partial function from nodes in a class graph to methods. All of those methods have the *same* name. In the semantics below, that name is not made explicit, but for clarity we will call it M in the following discussion. A method is a tuple of the form $\langle l_1 \ldots l_n \rangle$, where $l_1 \ldots l_n \in L$. When invoked, such a method executes by sending the message M to each of the subobjects labeled $l_1 \ldots l_n$.

If Ω is an object graph, o a node in Ω, P a program in the target language, and H a sequence of objects, then the judgment

$$\Omega \vdash_t o : \mathsf{P} \triangleright H$$

means that when sending the message M to o, we get a traversal of the object graph Ω starting in o so that H is the traversal history. Formally, this holds when the judgment is derivable using the following rule:

$$\frac{\Omega \vdash_t o_i : \mathsf{P} \rhd H_i \qquad \forall i \in 1..n}{\Omega \vdash_t o : \mathsf{P} \rhd o \cdot H_1 \cdot \ldots \cdot H_n} \quad \begin{array}{l}\text{if } \mathsf{P}(\mathsf{Class}(o)) = \langle l_1 \ldots l_n \rangle, \\ o \xrightarrow{l_i} o_i \text{ is in } \Omega, i \in 1..n, \text{ and} \\ l_j < l_k \text{ for } 1 \leq j < k \leq n.\end{array}$$

The label t of the turnstile indicates "target". Intuitively, the rule says that when sending the message M to o, we check if o understands the message, and if so, then we invoke the method. Notice that for $n = 0$, the rule is an axiom; it is then simply

$$\frac{}{\Omega \vdash_t o : \mathsf{P} \rhd o} \quad \text{if } \mathsf{P}(\mathsf{Class}(o)) = \langle\rangle.$$

Given a program in the target language, it is straightforward to generate, for example, a C++ program.

3. EFFICIENT IMPLEMENTATION OF ADAPTIVE PROGRAMS

We will implement adaptive programs efficiently by representing $\mathsf{PathSet}_\Phi(D)$ as $\mathsf{Graph}(\mathsf{PathSet}_\Phi(D))$, the *propagation graph.* The advantage of this representation is that $\mathsf{Graph}(\mathsf{PathSet}_\Phi(D))$ can be efficiently computed by the function PG_Φ, defined as follows:

$$\begin{aligned} PG_\Phi([A, B]) &= \mathsf{Graph}(\mathsf{Paths}_\Phi(A, B)) \\ PG_\Phi(D_1 \cdot D_2) &= PG_\Phi(D_1) \cup PG_\Phi(D_2) \\ PG_\Phi(D_1 + D_2) &= PG_\Phi(D_1) \cup PG_\Phi(D_2) \end{aligned}$$

Lemma 3.1. *If* $\mathsf{WF}(D)$*, then* $PG_\Phi(D) = \mathsf{Graph}(\mathsf{PathSet}_\Phi(D))$ *and* $\mathsf{PathSet}_\Phi(D) \subseteq \mathsf{Paths}_{PG_\Phi(D)}(\mathsf{Source}(D), \mathsf{Target}(D))$.

Proof. By induction on the structure of D. $\square$

Intuitively, we may view

$$\mathsf{PathSet}_\Phi(D)$$

as a high-level interpretation of the traversal specification D. It describes the intent of the programmer. In contrast,

$$\mathsf{Paths}_{PG_\Phi(D)}(\mathsf{Source}(D), \mathsf{Target}(D))$$

is a low-level interpretation of D. It describes those paths the implementation will consider.

The drawback of the low-level interpretation is that $\mathsf{Graph}(\mathsf{PathSet}_\Phi(D))$ may contain paths from $\mathsf{Source}(D)$ to $\mathsf{Target}(D)$ that are not in $\mathsf{PathSet}_\Phi(D)$. Given a well-formed specification D and a class graph Φ, a well-formed specification D is *consistent* with a class graph Φ, written $\Phi \models D$, if

$$\mathsf{PathSet}_\Phi(D) = \mathsf{Paths}_{PG_\Phi(D)}(\mathsf{Source}(D), \mathsf{Target}(D)) .$$

Intuitively, a well-formed specification is consistent with a class graph, if the high-level interpretation and the low-level interpretation coincide.

Moreover, D is *compatible* with Φ, if for any subspecification D' of D, there is a path in Φ from $\mathsf{Source}(D')$ to $\mathsf{Target}(D')$.

The following translation of adaptive programs into the target language requires compatibility, consistency, and subclass invariance.

Given a class graph Φ and a traversal specification D, we define the target program $\mathsf{Comp}(D, \Phi)$ by $\mathsf{Comp}(D, \Phi) = \mathsf{P}_{PG_\Phi(D),\Phi}$. For any two class graphs Φ, Φ' where Φ' is a subgraph of Φ, $\mathsf{P}_{\Phi',\Phi}$ is the partial function from nodes in Φ to methods, such that:

—For a concrete class $v \in \Phi'$, $\mathsf{P}_{\Phi',\Phi}(v) = \langle l_1 \dots l_n \rangle$, where $\mathsf{Edges}_{\Phi'}(v) = \{v \xrightarrow{l_i} w_i \mid i \in 1..n\}$ and $l_j < l_k$ for $1 \le j < k \le n$.

—For a concrete class $v \in (\Phi \backslash \Phi')$ where $\mathsf{Subclass}_\Phi(u, v)$ for some $u \in \Phi'$, $\mathsf{P}_{\Phi',\Phi}(v) = \langle\rangle$.

—For all other classes $v \in \Phi$, $\mathsf{P}_{\Phi',\Phi}(v)$ is undefined.

In the Demeter system, the compiler generates empty *virtual* C++ methods for the abstract classes of Φ'. Here, we use a target language *without* inheritance, so to model empty virtual methods, we generate empty methods for all concrete classes outside Φ' that are subclasses of some class in Φ'.

The correctness of the above translation is proved as follows.

LEMMA 3.2. *If* $\mathsf{WF}(D)$, $\Phi \models D$, *and* D *is compatible with* Φ, *then* (i) $\mathsf{PathSet}_\Phi(D)$ *is convex over* $PG_\Phi(D)$, (ii) $\mathsf{Source}(D) = \mathsf{Root}(\mathsf{PathSet}_\Phi(D))$, *and* (iii) $\mathsf{Target}(D) = \mathsf{Leaf}(\mathsf{PathSet}_\Phi(D))$.

PROOF. Immediate, using Lemma 2.5.1. □

LEMMA 3.3. *If* $\mathsf{WF}(D)$, *and* D *is compatible with* Φ, *then* $\mathsf{Target}(D)$ *is a Rome-node of* $PG_\Phi(D)$.

PROOF. By induction on the structure of D. □

LEMMA 3.4. *For two class graphs* Φ, Φ' *such that* Φ' *is a subclass-invariant subgraph of* Φ, *an object graph* Ω *conforming to* Φ, *a convex path set* R *over* Φ', *a node* o *in* Ω *such that* $\mathsf{Subclass}_\Phi(\mathsf{Root}(R), \mathsf{Class}(o))$, *where* $\mathsf{Leaf}(R)$ *is a Rome-node of* Φ', *and a traversal history* H, *we have*

$$\Omega \vdash_s o : R \rhd H \quad \textit{iff} \quad \Omega \vdash_t o : \mathsf{P}_{\Phi',\Phi} \rhd H \ .$$

PROOF. Suppose first that $\Omega \vdash_s o : R \rhd H$ is derivable. We proceed by induction on the structure of the derivation of $\Omega \vdash_s o : R \rhd H$. Since $\Omega \vdash_s o : R \rhd H$ is derivable, we have that $\mathsf{Car}(R, \mathsf{Class}(o)) = \{\mathsf{Class}(o) \xrightarrow{l_i} w_i \mid i \in 1..n\}$; $o \xrightarrow{l_i} o_i$ is in Ω, $i \in 1..n$, $l_j < l_k$ for $1 \le j < k \le n$; and that $\Omega \vdash_s o_i : \mathsf{Cdr}(l_i, R, \mathsf{Class}(o)) \rhd H_i$ is derivable for all $i \in 1..n$. There are two cases.

First, if $\mathsf{Class}(o) \notin \Phi'$, then $\mathsf{Car}(R, \mathsf{Class}(o)) = \emptyset$. Moreover, since we have that $\mathsf{Subclass}_\Phi(\mathsf{Root}(R), \mathsf{Class}(o))$, we have $\mathsf{P}_{\Phi',\Phi}(\mathsf{Class}(o)) = \langle\rangle$; so $H = o$, and $\Omega \vdash_t o : \mathsf{P}_{\Phi',\Phi} \rhd H$ is derivable.

Second, if $\mathsf{Class}(o) \in \Phi'$, then since $\mathsf{Leaf}(R)$ is a Rome-node of Φ', there is a path p in Φ' from $\mathsf{Class}(o)$ to $\mathsf{Leaf}(R)$. Moreover, since $\mathsf{Subclass}_\Phi(\mathsf{Root}(R), \mathsf{Class}(o))$, and Φ' is a subclass-invariant subgraph of Φ, we have $\mathsf{Subclass}_{\Phi'}(\mathsf{Root}(R), \mathsf{Class}(o))$ and thus a path p' in Φ' from $\mathsf{Root}(R)$ to $\mathsf{Class}(o)$ consisting of only subclass edges. Since R is convex over Φ', we get $p'p \in R$, and hence $\mathsf{Class}(o) \in \mathsf{Head}(R)$. Since $\mathsf{Leaf}(R)$ is a Rome-node of Φ', we then have $\mathsf{Car}(R, \mathsf{Class}(o)) = \mathsf{Edges}_{\Phi'}(\mathsf{Class}(o))$. Thus,

$\mathsf{P}_{\Phi',\Phi}(\mathsf{Class}(o)) = \langle l_1 \dots l_n \rangle$. Using Lemma 2.2.1 we obtain that $\mathsf{Cdr}(l_i, R, \mathsf{Class}(o))$ is convex and that $\mathsf{Leaf}(\mathsf{Cdr}(l_i, R, \mathsf{Class}(o)))$ is a Rome-node of Φ'. Since Ω conforms to Φ, we also have $\mathsf{Subclass}_\Phi(\mathsf{Root}(\mathsf{Cdr}(l_i, R, \mathsf{Class}(o))), \mathsf{Class}(o_i))$ for all $i \in 1..n$. By the induction hypothesis, $\Omega \vdash_t o_i : \mathsf{P}_{\Phi',\Phi} \rhd H_i$ is derivable for all $i \in 1..n$. Hence, $\Omega \vdash_t o : \mathsf{P}_{\Phi',\Phi} \rhd H$ is derivable.

The converse is proved similarly. $\square$

Theorem 3.5 (Correctness). *For a class graph Φ, a well-formed specification D, an object graph Ω conforming to Φ, a node o in Ω such that*

$$\mathsf{Subclass}_\Phi(\mathsf{Source}(D), \mathsf{Class}(o)) \ ,$$

and a traversal history H, if $\Phi \models D$, D is compatible with Φ, and $PG_\Phi(D)$ is a subclass-invariant subgraph of Φ, then

$$\Omega \vdash_s o : \mathsf{PathSet}_\Phi(D) \rhd H \quad \textit{iff} \quad \Omega \vdash_t o : \mathsf{Comp}(D, \Phi) \rhd H \ .$$

Proof. By Lemma 3.2, (1) $\mathsf{PathSet}_\Phi(D)$ is convex over $PG_\Phi(D)$, (2) $\mathsf{Source}(D) = \mathsf{Root}(\mathsf{PathSet}_\Phi(D))$, and (3) $\mathsf{Target}(D) = \mathsf{Leaf}(\mathsf{PathSet}_\Phi(D))$. From Lemma 3.3 we obtain that $\mathsf{Target}(D)$ is a Rome-node of $PG_\Phi(D)$. Finally, $\mathsf{Comp}(D, \Phi) = \mathsf{P}_{PG_\Phi(D),\Phi}$. The conclusion then follows from Lemma 3.4. $\square$

4. COMPOSITIONAL CONSISTENCY

We now present an algorithm that does compositional consistency checking. First we present a specification of the algorithm, in the form of three inference rules.

Given class graphs Φ_1 and Φ_2 and nodes A, B, and C, we write

$$\mathsf{NoShortcut}(\Phi_1, \Phi_2, A, B, C)$$

if it is the case that $\mathsf{Paths}_{\Phi_1 \cup \Phi_2}(A, C) \subseteq \mathsf{Paths}_{\Phi_1}(A, B) \cdot \mathsf{Paths}_{\Phi_2}(B, C)$.

Given Φ_1 and Φ_2 and nodes A and B, we write

$$\mathsf{NoZigzag}(\Phi_1, \Phi_2, A, B)$$

if $\mathsf{Paths}_{\Phi_1 \cup \Phi_2}(A, B) \subseteq \mathsf{Paths}_{\Phi_1}(A, B) \cup \mathsf{Paths}_{\Phi_2}(A, B)$.

The judgment $\Phi \vdash D$ means that D is compositionally consistent with Φ. The judgment is conservative in the sense that for well-formed specifications, $\Phi \vdash D$ implies $\Phi \models D$, but not necessarily vice versa. There are three rules:

$$\Phi \vdash [A, B] \tag{1}$$

$$\frac{\Phi \vdash D_1 \qquad \Phi \vdash D_2}{\Phi \vdash D_1 \cdot D_2} \quad \text{if } \mathsf{NoShortcut}(PG_\Phi(D_1), PG_\Phi(D_2), \mathsf{Source}(D_1), \mathsf{Target}(D_1), \mathsf{Target}(D_2)) \tag{2}$$

$$\frac{\Phi \vdash D_1 \qquad \Phi \vdash D_2}{\Phi \vdash D_1 + D_2} \quad \text{if } \mathsf{NoZigzag}(PG_\Phi(D_1), PG_\Phi(D_2), \mathsf{Source}(D_1), \mathsf{Target}(D_1)) \tag{3}$$

Theorem 4.1 (Soundness). *If $\mathsf{WF}(D)$, then $\Phi \vdash D$ implies $\Phi \models D$.*

Proof. We proceed by induction on the structure of the derivation of $\Phi \vdash D$. In the base case, consider $\Phi \vdash [A, B]$. We must prove $\Phi \models [A, B]$ which amounts to proving $\mathsf{PathSet}_\Phi([A, B]) = \mathsf{Paths}_{PG_\Phi([A,B])}(A, B)$ which is immediate.

In the induction step, consider first $\Phi \vdash D_1 \cdot D_2$. We must prove $\Phi \models D_1 \cdot D_2$ which amounts to proving $\mathsf{PathSet}_\Phi(D_1 \cdot D_2) = \mathsf{Paths}_{PG_\Phi(D_1 \cdot D_2)}(\mathsf{Source}(D_1), \mathsf{Target}(D_2))$ which in turn amounts to proving

$$\mathsf{PathSet}_\Phi(D_1) \cdot \mathsf{PathSet}_\Phi(D_2) = \mathsf{Paths}_{PG_\Phi(D_1) \cup PG_\Phi(D_2)}(\mathsf{Source}(D_1), \mathsf{Target}(D_2)) \ .$$

By the induction hypothesis we have $\Phi \models D_1$ and $\Phi \models D_2$ so we need to prove

$$\mathsf{Paths}_{PG_\Phi(D_1)}(\mathsf{Source}(D_1), \mathsf{Target}(D_1)) \cdot \mathsf{Paths}_{PG_\Phi(D_2)}(\mathsf{Source}(D_2), \mathsf{Target}(D_2)) = \mathsf{Paths}_{PG_\Phi(D_1) \cup PG_\Phi(D_2)}(\mathsf{Source}(D_1), \mathsf{Target}(D_2)).$$

From $\mathsf{WF}(D_1 \cdot D_2)$ we obtain that $\mathsf{Target}(D_1) = \mathsf{Source}(D_2)$ and clearly

$$\mathsf{Paths}_{PG_\Phi(D_1)}(\mathsf{Source}(D_1), \mathsf{Target}(D_1)) \cdot \mathsf{Paths}_{PG_\Phi(D_2)}(\mathsf{Source}(D_2), \mathsf{Target}(D_2)) \subseteq \mathsf{Paths}_{PG_\Phi(D_1) \cup PG_\Phi(D_2)}(\mathsf{Source}(D_1), \mathsf{Target}(D_2)).$$

The reverse inclusion follows from

$$\mathsf{NoShortcut}(PG_\Phi(D_1), PG_\Phi(D_2), \mathsf{Source}(D_1), \mathsf{Target}(D_1), \mathsf{Target}(D_2)).$$

Consider then $\Phi \vdash D_1 + D_2$. We must prove $\Phi \models D_1 + D_2$ which amounts to proving $\mathsf{PathSet}_\Phi(D_1 + D_2) = \mathsf{Paths}_{PG_\Phi(D_1 + D_2)}(\mathsf{Source}(D_1), \mathsf{Target}(D_1))$ which in turn amounts to proving

$$\mathsf{PathSet}_\Phi(D_1) \cup \mathsf{PathSet}_\Phi(D_2) = \mathsf{Paths}_{PG_\Phi(D_1) \cup PG_\Phi(D_2)}(\mathsf{Source}(D_1), \mathsf{Target}(D_1)) \ .$$

By the induction hypothesis we have $\Phi \models D_1$ and $\Phi \models D_2$, so we need to prove

$$\mathsf{Paths}_{PG_\Phi(D_1)}(\mathsf{Source}(D_1), \mathsf{Target}(D_1)) \cup \mathsf{Paths}_{PG_\Phi(D_2)}(\mathsf{Source}(D_2), \mathsf{Target}(D_2)) = \mathsf{Paths}_{PG_\Phi(D_1) \cup PG_\Phi(D_2)}(\mathsf{Source}(D_1), \mathsf{Target}(D_1)).$$

From $\mathsf{WF}(D_1 \cdot D_2)$ we get that $\mathsf{Source}(D_1) = \mathsf{Source}(D_2)$ and $\mathsf{Target}(D_1) = \mathsf{Target}(D_2)$, and clearly

$$\mathsf{Paths}_{PG_\Phi(D_1)}(\mathsf{Source}(D_1), \mathsf{Target}(D_1)) \cup \mathsf{Paths}_{PG_\Phi(D_2)}(\mathsf{Source}(D_2), \mathsf{Target}(D_2)) \subseteq \mathsf{Paths}_{PG_\Phi(D_1) \cup PG_\Phi(D_2)}(\mathsf{Source}(D_1), \mathsf{Target}(D_1)).$$

The reverse inclusion follows from

$$\mathsf{NoZigzag}(PG_\Phi(D_1), PG_\Phi(D_2), \mathsf{Source}(D_1), \mathsf{Target}(D_1)) \ .$$

This completes the proof. $\square$

The converse of Theorem 4.1 is in general false. For example, consider the specification $D = ([A, B] \cdot [B, C]) + [A, C]$ and the graph $\Phi = (\{A, B, C\}, \{A \xrightarrow{l} B, B \xrightarrow{m} A, A \xrightarrow{m} C\})$. Clearly, $\mathsf{WF}(D)$ and $\Phi \models D$, but $\Phi \not\vdash D$ because $\Phi \not\vdash ([A, B] \cdot [B, C])$. To see $\Phi \not\vdash ([A, B] \cdot [B, C])$, notice that $AmC \in \mathsf{Paths}_{PG_\Phi([A,B] \cdot [B,C])}(A, C)$, but $AmC \notin \mathsf{PathSet}_\Phi([A, B] \cdot [B, C])$.

Given D and Φ, we can decide if $\Phi \vdash D$ by the following algorithm:

```
Input:  A specification $D$ and a graph $\Phi$.
(1)     Check $\mathsf{WF}(D)$.
(2)     Check $\Phi \vdash D$ by
          — building $PG_\Phi(D)$ recursively, and along the way
          — computing the appropriate instances of NoShortcut and NoZigzag.
```

We can compute $\mathsf{WF}(D)$ in $O(|D|)$ time; we can build $PG_\Phi(D)$ in $O(|D|\ |\Phi|)$ time; and we can check each instance of NoShortcut and NoZigzag in $O(|\Phi|)$ time. Hence, the total running time is $O(|D|\ |\Phi|)$.

5. RELATED WORK

There are many approaches to making software flexible. In comparison, adaptive programming has a unique feature: succinct traversal specifications. In the following we briefly assess some of the approaches that are most closely related to the idea of adaptive programs.

Metaprogramming systems tend to spend considerable time doing recursive descents across program structures represented as data structures. In such systems, graph traversals are usually expressed either with attribute grammars [Waite and Goos 1984] or other syntax-directed facilities such as code walkers [Goldman 1989] (see also Wile [1983; 1986]). With attribute grammars, detailing the traversal is necessary and laborious, and it is subject to the same maintenance problems as raw object-oriented methods containing explicit traversal code. With code walkers, the traversal is specified separately from the functionality, like in adaptive programs. Specifying the traversal is either more laborious than with our traversal specifications, or it uses defaults that are similar to what adaptive programming provides.

Object-oriented databases have been introduced to ease the development of database applications. Object navigation is a common activity of processing in hierarchical or object-oriented databases [Coburn and Weddell 1991; Dayal 1989]. Queries can be specified in terms of navigating property value paths. However, as observed by Abiteboul and Bonner [1991], the current object-oriented databases applications still demonstrate lack of flexibility. For example, restructuring object schemas often triggers a good amount of work in restructuring database applications accordingly. Markowitz and Shoshani [1989; 1993] also observed the need to write adaptive database queries. They state: "In order to express database queries, users are often required to remember and understand large, complex database structures. It is important to relax this requirement by allowing users to express concise (*abbreviated*) queries, so that they can manage with partial or even no knowledge of the database structure" [Markowitz and Shoshani 1989]. Kifer et al. [1992] allow for similar abbreviated queries where a path expression can be bound to a sequence of attributes. Bussche and Vossen [1993] use weights to help determine the meaning of certain abbreviated path expressions. Our use of succinct traversal specifications is intended to achieve such conciseness.

Rumbaugh [1988] proposed an operation propagation mechanism to specify object-oriented software. The motivation of his work was to increase the clarity of program specifications and to reduce the amount of code to be written. He found that lots of operations such as *copy*, *print*, and *save* always propagate to some objects in a collection. He proceeded by separating the propagation part out of an operation, and specified the propagation by attaching propagation attributes to classes involved in the operation. This is similar to the code walker approach. By doing so, the rules for propagating were clearly declared, easier to understand and modify, and the amount of code to be written is reduced. Rumbaugh's mechanism is run-time based, however, and appears to be less flexible than the succinct traversal specifications. Rumbaugh's mechanism requires explicitly attaching propagation attributes to each individual class involved in an operation. When the class structure evolves, programmers have to update propagation attributes. With an adaptive program, there may be no need to update the program even if the underlying class structure changes.

Harrison and Ossher [1991] also found the need to separate the navigation responsibility from the processing responsibility, which simplifies system implementations and eliminates a good amount of explicit navigation code. They proposed a means of propagating messages between objects that are widely separated in a network based on routing specifications. A single, central navigator propagates messages according to routing specifications. They used default routing specifications to define how messages pass uninteresting objects. Their mechanism seems better than Rumbaugh's mechanism because routing specifications can be described relatively independent of object structures. The primary difference between their mechanism and ours is that theirs is run-time based.

Lamping and Abadi [1994] discuss the view of "methods as assertions." This view generalizes object-oriented programming and helps the programmer to express flexibly when a certain piece of code will correctly implement an operation. The methods-as-assertions view is consistent with the adaptive view, and moreover the two views are complementary and might be combined.

Wile and Balzer [1994] discuss decontextualized components. In a decontextualized component, an architecture description language provides the usage context. Compilation decisions are delayed until the context information is available. Decontextualized components make fewer commitments to data and control decisions than ordinary components. They do not use succinct traversal specifications, however.

Kiczales and Lamping [1992] wrote: "The problem then is how to say enough about the internal workings of the [class] library that the user can write replacement modules, without saying so much that the implementor has no room to work." While the metaobject protocol community addresses the problem with metaobject protocol programs, we address it with succinct subgraph specifications which exploit regularities in object-oriented software.

Object-oriented programs, especially those which follow such programming styles as the Law of Demeter [Lieberherr and Holland 1989], have the "small-methods" problem (see Wilde and Huitt [1991; 1992] and Wilde et al. [1993]). The small-methods problem results in dispersed program structure, hindering high-level and detailed understanding. To maintain object-oriented software, software developers have to trace how an operation is propagated along an object hierarchy and where the processing job is getting done. Experience shows that such tracing is time consuming and error prone [Wilde et al. 1993]. We could avoid the small-methods by creating larger methods. This, however, would be at the price of a significant maintenance problem because in every method we would then encode more details of the class structure. Adaptive software solves the small-methods problem without introducing large methods and the associated maintenance problem.

Adaptive programs may be used as a succinct way to document object-oriented software. A large group of small cooperative methods can be summarized by a propagation pattern. As a result, the specification of the operation becomes localized and possibly shorter and easier to understand.

In conventional object-oriented programming, object traversal may be specified using patterns. Gamma et al. [1995] introduce the structural design pattern Composite and the behavioral design pattern Visitor. The Composite pattern describes how to compose objects into tree structures to represent part-whole hierarchies, and the Visitor pattern serves to traverse a part-whole hierarchy. With respect to the traversal operations, we read in Gamma et al. [1995]: "The problem here is that distributing all these operations across the various node classes leads to a system that is hard to understand, maintain and change." The idea of the Visitor pattern is to code a traversal once and then to use it several times. The consequences of using the Visitor pattern are:

—Adding new operations which use the same traversal is easy. There is no need to change each class in the traversal domain.

—Related behavior is localized in a visitor and not spread over the classes defining the object structure.

—It is hard to add a new class as a participant in the traversal. In Gamma et al. [1995] we read: "When new classes which participate in the traversal are added frequently, it is probably easier just to define the operations on the classes that make up the structure."

With adaptive software we achieve the goals of the Visitor pattern more effectively. We can use a propagation pattern which gathers together in one place the code that describes the traversal. This makes it easy to add a new class as a participant in the traversal.

In deductive databases, searching is guided by logical rules. Current work [Ceri et al. 1993] addresses the combining of deductive databases and object technology. We believe that our succinct traversal specifications can help eliminate the need for at least some of the rules.

6. CONCLUSION

Our implementation of adaptive programs has two main advantages. First, there is no loss of efficiency compared to conventional object-oriented programming. The generated object-oriented code is as efficient as equivalent hand-written traversal code. In the examples of this article, we use C^{++} as the target language. It would be possible instead to use any typed language with classes, multiple inheritance, instance variables, methods, and late binding, e.g., Eiffel [Meyer 1988].

Second, it scales well. Intuitively, the more classes a program contains, the longer are the paths in the corresponding class graphs. Thus, larger programs often mean more traversal code. With our implementation of adaptive programs, the traversal code is automatically generated.

The usefulness of adaptive software hinges on two questions:

Table II. Terminology

C++	**Smalltalk**	**CLOS**
data member	instance variable	named slot
member function	method	function
virtual function	method	generic function
member function call	message send	function call

(1) How much traversal happens in object-oriented programs?

(2) If there is traversal, can it be specified succinctly?

Regarding (1), statistics of object-oriented systems show that they contain many small methods. Those small methods tend to contain traversal code so their presence documents that traversal is common. The reason why traversal is common is that, for each task we implement, there are often only a few "worker" classes which do interesting work but many other "bystanders" which participate in the traversals only. Moreover, as we go from task to task, a class which was a worker may become a bystander and vice versa.

Regarding (2), our experience with the Demeter system indicates that the forms of traversal which often appear in object-oriented programs can nicely be captured in our language of traversal specifications.

If there happens to be no traversal going on, or if there is no succinct specification for the traversal we want, we may simply use the empty traversal specification in each propagation pattern. This leads to the generation of an object-oriented program with just one method for each propagation pattern. We believe that both situations ("no traversal" and "no succinct traversal specification") are rare in practice.

Notice that any object-oriented program can be reengineered into an adaptive program. The idea is to specify each method as a propagation pattern with the empty traversal specification. This demonstrates that an adaptive program need be at most as long as an object-oriented program for the same task.

APPENDIX. C++

This appendix is for readers who are not familiar with C++ [Ellis and Stroustrup 1990].

C++ is an extension of C in that classes in C++ are a generalization of structures in C. Members of a class can be not only data (called data members) but also functions (member functions). Table II shows the different terminology used in C++, Smalltalk [Goldberg and Robson 1983], and CLOS [Steele 1990].

In C++ terminology, when a class A is inherited by a class B, class A is called a base class or superclass, and class B is called a derived class or subclass. Moreover, class A may be a *supertype* of class B. (This need not be the case in C++, e.g., when the inheritance is so-called private.) When class A is a supertype of class B, class B supports all member function interfaces that A supports. Furthermore, for a member function defined in A, the class B can have a member function with the same interface but with different implementation which overrides the implementation in A. Late binding of function calls is made possible by declaring the member function *virtual*, outlined as follows.

```
    class A
```

```
{
  public:
    virtual void f();
};
```

The following code fragment gives a member function definition.

```
void A::f()
{
   // A's implementation goes here
}
```

The keyword void means that the method does not return any value. The syntax "::" resolves the function as a member function of class A. The fragment enclosed by braces is the implementation of the member function, which contains a line of C++ comment starting with // (double slash).

In C++, a variable v, which holds objects, can be of at least the following two kinds:

—Holding an object directly. Defined as

```
A v;
```

—Holding an address of an object. Defined as

```
A* v;
```

When class A is a supertype of B, the variable v with the second definition above can not only hold addresses of A objects but also addresses of B objects. To invoke the member function f() on the object pointed to by variable v, C++ uses the following syntax.

```
v->f();
```

ACKNOWLEDGMENTS

The authors thank Mitchell Wand for numerous discussions and a wealth of suggestions for how to improve the ideas presented in this article. The authors also thank Jan Van den Bussche, William Clinger, Walter Hursch, Linda Keszenheimer, and the anonymous referees for helpful comments on a draft of the article.

REFERENCES

Abiteboul, S. and Bonner, A. 1991. Objects and views. *In Proceedings of ACM SIGMOD International Conference on management of Data*, J. Clifford and R. King, Eds. ACM Press, New York, 238–247.

Bergstein, P. 1991. Object-preserving class transformations. In *Proceedings OOPSLA'91, ACM SIGPLAN 6th Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, New York, 299–313.

Booch, G. 1994. Design an application framework. *Dr. Dobbs J. 19,* 2 (Feb.), 24–32.

Booch, G. 1990. The design of the C++ Booch components. In *Object-Oriented Programming Systems, Languages and Applications Conference.* ACM Press, New York, 1–11.

Ceri, S., Tanaka, K., and Tsur, S., (Eds.) 1993. *Deductive and Object-Oriented Databases.* Lecture Notes in Computer Science, vol. 760. Springer-Verlag, Berlin.

Coburn, N. and Weddell, G. E. 1991. Path constraints for graph-based data models: Towards a unified theory of typing constraints, equations, and functional dependencies. In *2nd International Conference, DOOD'91*, C. Delobel, M. Kifer, and Y. Masunaga, Eds. Springer-Verlag, Berlin, 313–331.

Dayal, U. 1989. Queries and views in an object-oriented data model. In *Proceedings of the 2nd International Workshop on Database Programming Languages*, R. Hull, R. Morrison, and D. Stemple, Eds. Morgan Kaufmann, San Mateo, Calif., 80–102.

den Bussche, J. V. and Vossen, G. 1993. An extension of path expressions to simplify navigation in object-oriented queries. In *Proceedings Deductive and Object-Oriented Databases*. Lecture Notes in Computer Science, vol. 760. Springer-Verlag, Berlin, 267–282.

Ellis, M. A. and Stroustrup, B. 1990. *The Annotated* C++ *Reference Manual*. Addison-Wesley, Reading, Mass.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass.

Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O., and Pintado, X. 1990. Class management for software communities. *Commun. ACM 33,* 9 (Sept.), 90–103.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80—The Language and its Implementation*. Addison-Wesley, Reading, Mass.

Goldman, N. M. 1989. Code walking and recursive descent: A generic approach. In *Proceedings 2nd CLOS Users and Implementors Workshop*. ACM Press, New York.

Harrison, W. and Ossher, H. 1991. Structure-bound messages: Separating navigation from processing. Manuscript. IBM T. J. Watson Research Center.

Keszenheimer, L. 1993. Specifying and adapting object behavior during system evolution. In *Conference on Software Maintenance*. IEEE Press, New York, 254–261.

Kiczales, G. and Lamping, J. 1992. Issues in the design and documentation of class libraries. In *Proceedings OOPSLA'92, ACM SIGPLAN 7th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, New York, 435–451.

Kifer, M., Kim, W., and Sagiv, Y. 1992. Querying object-oriented databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, M. Stonebraker, Ed. ACM, New York, 393–402.

Lamping, J. and Abadi, M. 1994. Methods as assertions. In *European Conference on Object-Oriented Programming*, R. Pareschi and M. Tokoro, Eds. Lecture Notes in Computer Science, vol. 821. Springer-Verlag, Berlin, 60-80.

Lieberherr, K. J. 1995. *The Art of Growing Adaptive Object-Oriented Software*. PWS Publishing Company, Boston.

Lieberherr, K. J. 1992. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In *Information Processing '92, 12th World Computer Congress*, J. van Leeuwen, Ed. Elsevier, New York, 179–185.

Lieberherr, K. J. and Holland, I. 1989. Assuring good style for object-oriented programs. *IEEE Softw.*, 38–48.

Lieberherr, K. J. and Xiao, C. 1993a. Minimizing dependency on class structures with adaptive programs. In *International Symposium on Object Technologies for Advanced Software*, S. Nishio and A. Yonezawa, Eds. Lecture Notes in Computer Science, vol. 742. Springer-Verlag, Berlin, 424–441.

Lieberherr, K. J. and Xiao, C. 1993b. Object-oriented software evolution. *IEEE Trans. Softw. Eng. 19,* 4 (Apr.), 313–343.

Lieberherr, K. J., Hürsch, W., Silva-Lepe, I., and Xiao, C. 1992. Experience with a graph-based propagation pattern programming tool. In *International Workshop on CASE*, Forte, G. et al., Eds. IEEE Computer Society, Washington, D.C., 114–119.

Lieberherr, K. J., Silva-Lepe, I., and Xiao, C. 1994. Adaptive object-oriented programming using graph-based customization. *Commun. ACM 37,* 5 (May), 94–101.

Markowitz, V. M. and Shoshani, A. 1993. Object queries over relational databases: Language, implementation, and application. In *Proceedings 9th International Conference on Data Engineering*. IEEE Press, New York, 71–80.

Markowitz, V. M. and Shoshani, A. 1989. Abbreviated query interpretation in entity-relationship oriented databases. Lawrence Berkeley Lab., Berkeley, Calif..

Meyer, B. 1988. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, N.J.

Palsberg, J. and Schwartzbach, M. I. 1994. *Object-Oriented Type Systems*. John Wiley & Sons, New York.

Rumbaugh, J. 1988. Controlling propagation of operations using attributes on relations. In *Object-Oriented Programming Systems, Languages and Applications Conference*. ACM Press, New York, 285–297.

Steele, G. L. 1990. *Common Lisp: The Language*, 2nd ed. Digital Press, Burlington, Mass.

Waite, W. and Goos, G. 1984. *Compiler Construction*. Springer-Verlag, Berlin.

Wilde, N. and Huitt, R. 1992. Maintenance support for object-oriented programs. *IEEE Trans. on Softw. Eng. 18*, 12 (Dec.), 1038–1044.

Wilde, N. and Huitt, R. 1991. Maintenance support for object-oriented programs. In *Conference on Software Maintenance*. IEEE Press, New York, 162–170.

Wilde, N., Matthews, P., and Huitt, R. 1993. Maintaining object-oriented software. *IEEE Softw.*, 75–80.

Wile, D. S. 1986. Organizing programming knowledge into syntax-directed experts. In *Proceedings International Workshop on Advanced Programming Environments*. Lecture Notes in Computer Science, vol. 244. Springer-Verlag, Berlin, 551–565.

Wile, D. S. 1983. Program developments: Formal explanations of implementations. *Commun. ACM 26*, 11 (Nov.), 902–911.

Wile, D. S. and Balzer, R. M. 1994. Architecture-based compilation. Manuscript, sponsored by ARPA.

Received June 1994; revised November 1994 and January 1995; accepted January 1995