Centrum voor Wiskunde en Informatica

Visitor combination and traversal control

J.M.W. Visser

Software Engineering (SEN)

SEN-R0109 May 31, 2001

Report SEN-R0109
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Visitor Combination and Traversal Control

Joost Visser
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
The *Visitor* design pattern allows the encapsulation of polymorphic behavior outside the class hierarchy on which it operates. A common application of *Visitor* is the encapsulation of tree traversals. Unfortunately, visitors resist composition and allow little traversal control.

To remove these limitations, we introduce visitor *combinators*. These are implementations of the visitor interface that can be used to *compose* new visitors from given ones. The set of combinators we propose includes *traversal* combinators that can be used to obtain full traversal control.

A clean separation can be made between the generic parts of the combinator set and the parts that are specific to a particular class hierarchy. The generic parts form a reusable framework. The specific parts can be generated from a (tree) grammar. Due to this separation, programming with visitor combinators becomes a form of *generic programming* with significant reuse of (visitor) code.

1998 ACM Computing Classification System: D.1.2, D.1.5, D.2.3, D.2.13, D.3.4, E.2
Keywords and Phrases: Generalized LR parsing, tree traversal, program transformation, program analysis, program generation, object-orientation.
Note: Work carried out under projects SEN 1.1, *Software Renovation*, and SEN 1.5, *Domain-Specific Languages*, sponsored by the *Telematica Instituut*.

1. Introduction

Language processing involves tree traversal. For instance, program analysis and transformation require traversal of syntax trees. In the object-oriented paradigm, tree traversal can be performed in accordance with a particular variant of the Visitor design pattern [8]. In this approach, the tree to be traversed is represented according to the Composite pattern, the actions to be performed at tree nodes are encapsulated in a visitor class, and iteration over the tree is performed either by the visitor or by the accept methods in the tree classes. Several parser and visitor generation tools exist that support tree traversal with visitors (e.g. Java Tree Builder, SableCC [7], JJForester [14]).

Unfortunately, tree traversal with visitors suffers from two main limitations. The first limitation is lack of traversal control. The tree traversal strategy is either hard-wired into the accept methods, or entangled in the visitor code. In the first case, traversal control is absent, or limited to selecting one out of a few predefined strategies. In the second case, traversal control is limited to overriding the iteration behavior of particular visit methods. For instance, downward traversal can be cut off by omitting the call to the accept method of one or more subtrees of a particular node. In neither case can full traversal control be exerted.

The second limitation of tree traversal with visitors is that visitors resist combination. Visitors can only be specialized. In a language that supports multiple implementation inheritance, visitors may even inherit behavior from different parent visitors. But the only flavor of combination obtained with multiple inheritance is static exclusive disjunction (each visit method is either inherited from the one or from the other parent, and this is decided at compile time), and may require an overwhelming ambiguity resolution effort from the programmer. Less restricted combination of visitors would allow better visitor code reuse.

Combinator	*Description*
Identity	Do nothing (non-iterating default visitor).
Sequence(v1,v2)	Sequentially perform visitor v2 after v1.
Fail	Raise exception.
Choice(v1,v2)	Try visitor v1. If v1 fails, try v2 (left-biased choice).
All(v)	Apply visitor v sequentially to every immediate subtree.
One(v)	Apply visitor v sequentially to the immediate subtrees until it succeeds.

Table 1: The set of basic visitor combinators.

```
Node ::= 'Fork' '(' Node ',' Node ')'
       |  'Leaf' '(' Integer ')'
```

Figure 1: BNF definition of the running tree example.

In this paper, we propose a solution to both limitations. We introduce a small set of *visitor combinators* that can be used to construct new visitors from given ones. As will become clear, by *combinators* we mean reusable classes capturing basic functionality that can be composed in different constellations to obtain new functionality. The basic visitor combinators to be introduced are summarized in Table 1. This set of combinators is inspired by the *strategy* primitives of the term rewriting language *Stratego* [23]. The combinators `All` and `One` are traversal combinators that can be used to obtain full traversal control.

In our explanation of the visitor combinators we will use a tiny tree syntax as running example. Figure 1 shows its description in BNF. As can be gleaned from this BNF definition, there are two kinds of nodes in our example tree syntax. An internal node, or *fork*, has two subtrees as children, and *leaf* nodes have an Integer value as child. These two kinds of nodes suffice to capture all relevant variability to be found in non-trivial syntaxes, which generally contain large numbers of sorts (non-terminals) and syntax rules.

The use of the Visitor and Composite patterns for object-oriented tree traversal is illustrated for our example syntax in Figure 2. The ***Visitor*** interface declares a visit method for each syntax rule in the grammar. The ***Visitable*** interface declares the `accept` method. This method applies the appropriate visit method from its argument visitor to the current top node. Iteration over a tree can either be implemented in the accept methods, or in default implementations of the visit methods. For the `Fork` node, the Java implementation of the (non-iterating) accept method is shown in a note. Throughout the paper, we will use Java as implementation language.

The paper is structured as follows. Sections 2 through 4 introduce and explain each of the basic visitor combinators of Figure 1. In Section 5 these combinators are refactored into a generic framework, to make them independent of any specific class hierarchy. In Section 6 we discuss the support of visitor combinators by our visitor generator JJForester, and an application in the domain of legacy system redocumentation. Finally, Section 7 summarizes our contributions, and discusses related work.

2. Sequential composition

The first (nullary) combinator in our set of visitor combinators is the traditional non-iterating default visitor, which we call `Identity`. It satisfies the ***Visitor*** interface. The name `Identity` is justified by

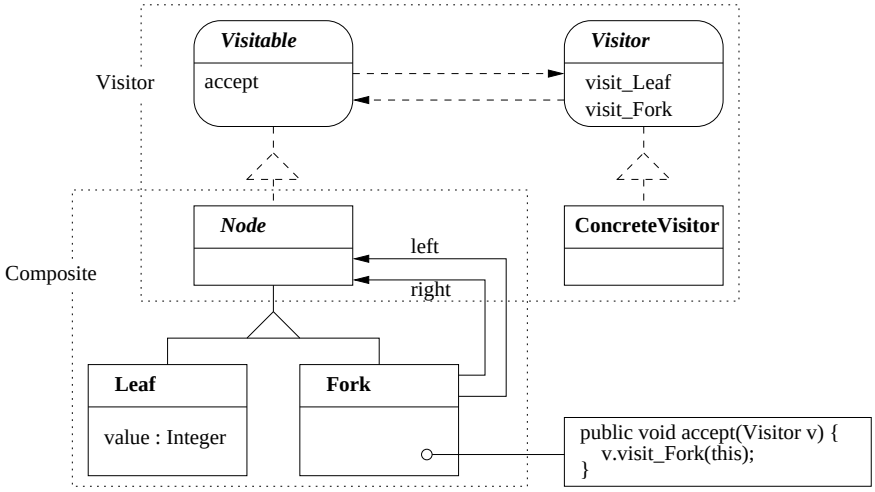

Figure 2: Using the Visitor pattern and the Composite pattern for object-oriented tree traversal.

```
interface Visitor {
  public void visit_Leaf(Leaf leaf);
  public void visit_Fork(Fork fork);
}
```

```
class Identity implements Visitor {
  public void visit_Leaf(Leaf leaf) {}
  public void visit_Fork(Fork fork) {}
}
```

Figure 3: The *Visitor* interface and the `Identity` combinator.

the behavior of its constituent visit methods: these methods have empty bodies, and therefore preserve the nodes to which they are applied. Also, this default visitor will fulfill the role of an identity element in our set of combinators in an algebraic sense. This will be explained later.

For our example grammar, the *Visitor* interface and the `Identity` combinator are shown in Figure 3. The *Visitor* interface declares a visit method for each kind of node. The visit method for a node takes this node as its argument. The `Identity` visitor implements this interface by providing an empty method body for every visit method.

The `Identity` visitor may seem to be useless, because it does nothing, literally. However, in the case of larger tree grammars its usefulness becomes clear. By creating specific visitors as specializations of `Identity`, instead of writing them from scratch, only those methods need to be refined which correspond to nodes at which "special" action needs to be taken. For all other nodes, the default identity behavior is reused. As we will see, its usefulness increases further in the presence of other visitor combinators. Figure 4 shows how `Identity` can be refined to a visitor that increments all values in leaf nodes with 1.

Having defined `Identity`, we can proceed to our first "real" combinator: `Sequence`. This is a

```
class AddOne extends Identity {
  public void visit_Leaf(Leaf leaf) {
    leaf.value = leaf.value + 1;
  }
}
```

Figure 4: Refinement of the default visitor combinator `Identity`.

```
class Sequence implements Visitor {
  Visitor first;
  Visitor then;
  public Sequence(Visitor first, Visitor then) {
    this.first = first;
    this.then  = then;
  }
  public void visit_Fork(Fork fork) {
    fork.accept(first);
    fork.accept(then);
  }
  public void visit_Leaf(Leaf leaf) {
    leaf.accept(first);
    leaf.accept(then);
  }
}
```

Figure 5: The binary visitor combinator `Sequence` applies its argument visitors `first` and `then` one after the other.

binary combinator that takes two visitors as arguments, which it sequentially applies to a node. For our example tree syntax, the sequential visitor combinator is show in Figure 5. The `Sequence` combinator is again modeled as a class that implements the *`Visitor`* interface. The arguments of the combinator are modeled as fields `first` and `then` of type *`Visitor`*. The constructor method of `Sequence` initializes these arguments. The visit method for each kind of node is implemented by `Sequence` in the following way. First, the visitor argument stored in field `first` is applied to the node, by calling its `accept` method with this visitor. Then, the same is done with the visitor stored in `then`.

How is the `Sequence` combinator used to create new visitors from given ones? For instance, Figure 6 demonstrates how to create a visitor `AddTwo` that applies the `AddOne` visitor twice. First the `Sequence` combinator is refined to the auxiliary combinator `Twice`, which applies its argument visitor twice,

```
class Twice extends Sequence {
  public Twice(Visitor v) {
    super(v,v);
  }
}
```

```
class AddTwo extends Twice {
  public AddTwo() {
    super(new AddOne());
  }
}
```

Figure 6: Definition of `AddTwo` in terms of `Sequence`, using an auxiliary combinator `Twice`.

```
public class VisitFailure extends Exception { }
```

Figure 7: The notion of success and failure of visitors is modeled using a refinement `VisitFailure` of the `Exception` class.

```
class Fail implements Visitor {
  public void visit_Leaf(Leaf leaf) throws VF {
    throw new VisitFailure();
  }
  public void visit_Fork(Fork fork) throws VF {
    throw new VisitFailure();
  }
}
```

Figure 8: The `Failure` combinator. The `throws VisitFailure` clause is abbreviated to `throws VF` for reasons of space. We will do so in all figures throughout the paper.

sequentially. Then, `Twice` is further refined to use the `AddOne` visitor as argument. In a more concise, mathematical notation these definitions would be written as follows:

$$\begin{array}{lll} \mathit{Twice(v)} & =_{def} & \mathit{Sequence(v,v)} \\ \mathit{AddTwo} & =_{def} & \mathit{Twice(AddOne)} \end{array}$$

The benefit of creating additional combinators (classes), instead of inlining their definitions, is that this makes them reusable.

In an algebraic sense, the combinator `Identity` is an identity element for the combinator `Sequence`. This means that the following equalities hold:

$$\begin{array}{lcl} \mathit{Sequence(Identity,v)} & = & v \\ \mathit{Sequence(v,Identity)} & = & v \end{array}$$

These equations do not necessarily hold for subclasses of `Sequence` and `Identity`, as they might introduce side-effects. Note also that `Twice` applies the *same* argument visitor twice, which means that the same state can be accessed at both applications.

3. Alternative composition

We will now introduce a binary visitor combinator `Choice`, which alternatively applies its first or its second argument visitor. More precisely, it will first try one argument visitor, and when this visitor fails, it will try the other. This notion of left-biased alternative composition presupposes a notion of success and failure of visitors.

Success and failure of visitors can be modeled with exceptions. At failure, a `VisitFailure` exception is thrown. Figure 7 shows the corresponding refinement of the `Exception` class. The `try` and `catch` constructs are used at choice points.

Given our modeling of visitor failure with exceptions, a nullary visitor combinator `Fail` can be defined, which raises a `VisitFailure` exception for every node. For our example tree syntax, this combinator is defined in Figure 8. The clause `throws VisitFailure` also needs to be added to the headers of the previously defined visit and accept methods.

Finally, we can proceed to the `Choice` combinator we set out to define. Figure 9 shows its definition for our example tree syntax. Like `Sequence`, the `Choice` combinator has two visitor arguments, which are modeled by fields `first` and `then`. It implements the visit method for each kind of node by a try statement that attempts to apply the `first` visitor. If this visitor raises a `VisitFailure` exception, the subsequent catch statement defaults to the `then` visitor.

```
class Choice implements Visitor {
  Visitor first;
  Visitor then;
  public Choice(Visitor first, Visitor then) {
    this.first = first;
    this.then  = then;
  }
  public void visit_Leaf(Leaf leaf) throws VF {
    try { leaf.accept(first); }
    catch (VisitFailure f) { leaf.accept(then); }
  }
  public void visit_Fork(Fork fork) throws VF {
    try { fork.accept(first); }
    catch (VisitFailure f) { fork.accept(then); }
  }
}
```

Figure 9: The `Choice` combinator.

`Fail` is a zero element for `Sequence`, and an identity for `Choice`. Because of its left-bias, `Choice` has `Identity` as left-zero, but not as right-zero.

$$
\begin{array}{lcll}
\mathit{Sequence(Fail,v)} & = & \mathit{Fail} & \\
\mathit{Sequence(v,Fail)} & = & \mathit{Fail} & \text{(if } v \text{ side-effect free)} \\
\mathit{Choice(Fail,v)} & = & v & \\
\mathit{Choice(v,Fail)} & = & v & \\
\mathit{Choice(Identity,v)} & = & \mathit{Identity} &
\end{array}
$$

The `Fail` and `Choice` combinators can be used to create visitors that *conditionally* fire at certain nodes. For instance, Figure 10 demonstrates how to construct a visitor `IfZeroAddOne` that applies the `AddOne` visitor only to leaf nodes that contain the value *0*. A visitor `IsZero` that tests the value of a leaf node is defined by extending `Fail`. The auxiliary unary visitor combinator `Try` is defined as the alternative composition of its visitor argument and the `Identity` visitor. The sequential composition of `IsZero` and `AddOne` is supplied as visitor argument to `Try`. In mathematical notation, these definitions would be written down as follows:

$$
\begin{array}{lcl}
\mathit{Try(v)} & =_{def} & \mathit{Choice(v,Identity)} \\
\mathit{IfZeroAddOne} & =_{def} & \mathit{Try(Sequence(IsZero,AddOne))}
\end{array}
$$

4. Traversal combinators

The visitor combinators introduced above can be used to construct new visitors from given ones. A visitor thus created can be applied to a tree by passing it to an `accept` method. Depending on whether this method performs iteration, the visitor is applied to the top node of the tree, or according to a fixed traversal strategy to all nodes in the tree. To obtain more control over traversal behavior, we additionally introduce two *traversal combinators*.

Our first traversal combinator, called `All`, takes one visitor as argument, and applies it to every immediate subtree of the current top node. For our example tree syntax, Figure 11 provides the definition of `All`. Since leaf nodes have no subtrees, their visit method does nothing. Fork nodes have two subtrees, to which the argument visitor is applied one after the other.

The `All` combinator suffices to reconstruct the top-down (pre-order) and bottom-up (post-order) traversal strategies. In mathematical terms, their definitions are as follows:

$$
\begin{array}{lcl}
\mathit{TopDown(v)} & =_{def} & \mathit{Sequence(v,All(TopDown(v)))} \\
\mathit{BottomUp(v)} & =_{def} & \mathit{Sequence(All(BottomUp(v)),v)}
\end{array}
$$

```
class IsZero extends Fail {
  public void visit_Leaf(Leaf leaf) throws VF {
    if (leaf.value != 0) {
      throw new VisitFailure();
    }
  }
}
```

```
class Try extends Choice {
  public Try(Visitor v) {
    super(v,new Identity());
  }
}
```

```
class IfZeroAddOne extends Try {
  public IfZeroAddOne() {
    super(new Sequence(new IsZero(),new AddOne()));
  }
}
```

Figure 10: Conditional application of visitors, using `Choice` and `Fail`.

```
class All implements Visitor {
  Visitor v;
  public All(Visitor v) {
    this.v = v;
  }
  public void visit_Leaf(Leaf leaf) throws VF { }
  public void visit_Fork(Fork fork) throws VF {
    fork.left.accept(v);
    fork.right.accept(v);
  }
}
```

Figure 11: The traversal combinator `All` applies its argument visitor to each immediate subtree.

```
class TopDown extends Sequence {
  public TopDown(Visitor v) {
    super(v,null);
    then = new All(this);
  }
}
```

```
class BottomUp extends Sequence {
  public BottomUp(Visitor v) {
    super(null,v);
    first = new All(this);
  }
}
```

Figure 12: Reconstruction of the top-down and bottom-up traversal strategies in terms of `Sequence` and `All`.

```
class One implements Visitor {
  Visitor v;
  public void One(Visitor v) {
    this.v = v;
  }
  public void visit_Leaf(Leaf leaf) throws VF {
    throw new VisitFailure(); // Leaf has no kids.
  }
  public void visit_Fork(Fork fork) throws VF {
    try { fork.left.accept(v); }
    catch (VisitFailure f) {
      fork.right.accept(v);
    }
  }
}
```

Figure 13: The traversal combinator `One` applies its argument visitor to one of its immediate subtrees.

Note that these definitions are recursive: the combinator being defined occurs in its own definition. For our example tree syntax, the definitions of `TopDown` and `BottomUp` are given in Figure 12. The `TopDown` and `BottomUp` visitor combinators are both defined as specializations of the `Sequence` combinator. To model the recursive call of the combinator in Java is somewhat tricky, since it is not allowed to reference `this` before the superclass constructor has been called. This is solved by first setting the corresponding visitor argument to `null`. Subsequently this argument is set to its proper value `All(this)`. The combinators `TopDown` and `BottomUp` demonstrate that our set of basic visitor combinators obviates the need for implementing the pre- or post-order traversal strategies in accept methods or in a default visitor implementation. In fact, visitor combinators make virtually any traversal strategy programmable.

A visitor combinator similar to `All` is the `One` traversal combinator. Whereas `All` applies its argument visitor to all its subtrees, `One` applies it to exactly one. More precisely, it tries to apply it to each subtree in turn, until application succeeds. Figure 13 gives the definition of `One` for our example tree grammar. As leaf nodes have no subtrees, the corresponding visit method of `One` immediately fails. In the case of fork nodes, a visit to the left subtree is attempted first. If it fails, a visit to the next subtree is attempted.

As an indication of the level of traversal control that can be obtained with the combinators `All` and `One`, Figure 14 lists a number of different traversal strategy combinators. These combinators are object-oriented reconstructions of a few out of many strategy combinators that can be found in the

```
class OnceBottomUp extends Choice {
  public OnceBottomUp(Visitor v) {
    super(null,v);
    first = new One(this);
  }
}
```

```
class SpineBottomUp extends Sequence {
  public SpineBottomUp(Visitor v) {
    super(null,v);
    first = new Choice(new One(this),
                       new All(new Fail()));
  }
}
```

```
class DownUp extends Sequence {
  public DownUp(Visitor down, stop, up) {
    super(null, up);
    first = new Sequence(
                down,
                new Choice(stop,new All(this)));
  }
}
```

Figure 14: With the combinators `All` and `One`, arbitrary traversal strategies can be defined. `OnceBottomUp` applies `v` exactly once at the first location found during bottom-up traversal. `SpineBottomUp` applies `v` bottom-up along a path which reaches from the root to one of the leaves. `DownUp` applies `down` going down the tree, and applies `up` when coming back up. It cuts off the traversal below nodes where `stop` succeeds.

standard library of the strategic term rewriting language Stratego [22].

5. Syntax-independence

All combinators presented above were defined relative to our example tree syntax of Figure 1. So the question arises to what extent they are specific to this particular syntax, and to what extent they are generic[1]and reusable for any syntax. In this section we will explain that our combinators are generic in nature, and we will show how we can modify their encoding to isolate them from the specifics of any particular syntax.

5.1 Lack of genericity

The combinators presented above lack genericity in two respects.

Firstly, when defining a visitor, whether from scratch or by specialization of a basic visitor combinator, the visit methods of all constructors need to be (re)defined separately, even if the *same* behavior is required for each of them. At the risk of sounding paradoxical, one might say that a more generic way of specialization is needed. For instance, specialization should be possible of all visit methods at once, or, in case of a many-sorted syntax, of all visit methods for a particular sort at once.

Genericity is lacking in a second respect. In natural language we would have no trouble defining the behavior of all our basic combinators and most defined ones without reference to the example syntax. In fact, we gave just such a generic explanation in the running text of this paper. Still, they

[1]We use the term 'generic' in the general sense that a generic program is not restricted to one particular type, but can work on entities of many different types. In a more restricted sense, 'generic' has been used for programs that *uniformly* work for entities of *every* type (parametric polymorphism), for instance in the context of Ada, Eiffel, and GenericJava. In the context of PolyP [11], 'generic' has been used for programs that perform induction on a type parameter (polytypism).

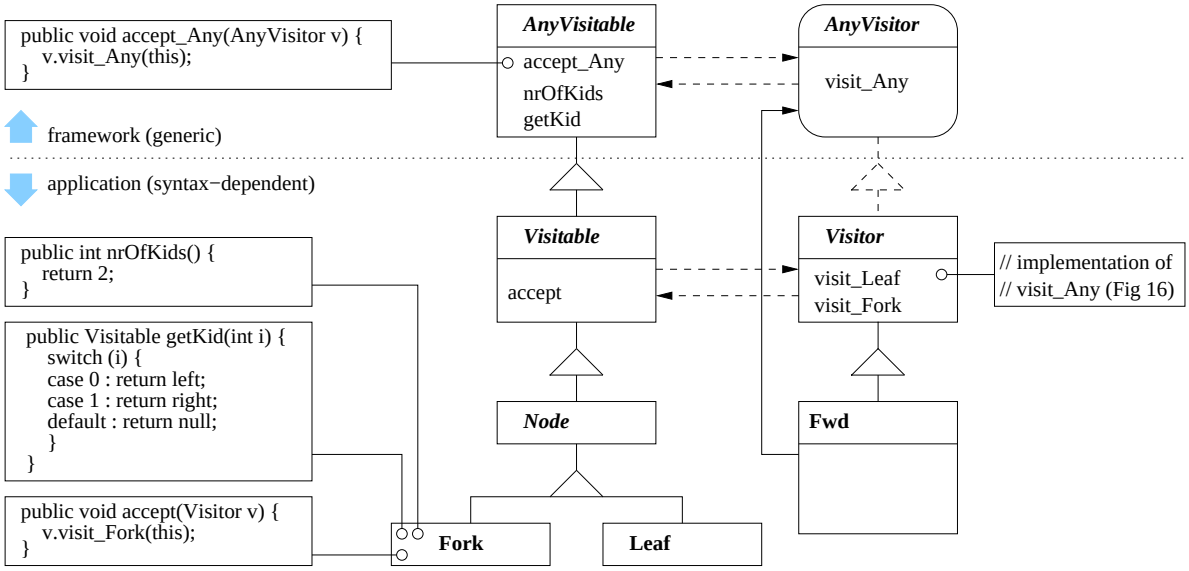

Figure 15: Separating out generics from specifics.

```
public void visit_Any(AnyVisitable x) throws VF {
  if (x instanceof Visitable) {
      ((Visitable) x).accept(this);
  } else {
      throw new VisitFailure();
  }
}
```

Figure 16: The default implementation of `visit_Any` of the syntax-specific abstract *`Visitor`* class.

can not be reused 'as is' for other syntaxes, because they refer (directly or indirectly) to the specific constructor names of the example syntax.

5.2 Visitor combinators in frameworks

To remove these limitations on the genericity of our visitor combinators, we need to refactor the design pattern of Figure 2 such that syntax-specific functionality is separated from generic functionality. Our solution is a variation on the *staggered* Visitor pattern [24], which introduces generic counterparts *`AnyVisitor`* and *`AnyVistable`* for the syntax-specific interfaces. This is illustrated in Figure 15. Note that *`AnyVisitable`* and *`Visitable`* are now abstract classes instead of interfaces, because *`AnyVisitable`* implements the `accept_Any` method. Likewise, *`Visitable`* is now an abstract class, because it implements the `visit_Any` method. For our example tree syntax, this implementation is shown in Figure 16. It uses runtime type identification (RTTI) to cast its generic *`AnyVisitable`* argument to a syntax-specific *`Visitable`*. If the cast succeeds, it applies itself, using the syntax-specific `accept` method. Thus, syntax-specific visitors have forwarding (delegation) built-in from generic to syntax-specific visit methods.

In the original staggered visitor pattern, the *`Visitor`* class also has the converse forwarding built-in, from syntax-specific visit methods to generic visit methods. In Figure 15, this converse forwarding has been factored out into a new combinator `Fwd`. Its definition for our example syntax is shown in Figure 17. `Fwd` is a unary combinator that takes a generic visitor as argument, and implements all

```
public class Fwd implements Visitor {
  AnyVisitor v;
  public Fwd(AnyVisitor v) {
    this.v = v;
  }
  public void visit_Leaf(Leaf leaf) throws VF {
    v.visit_Any(leaf);
  }
  public void visit_Fork(Fork fork) throws VF {
    v.visit_Any(fork);
  }
}
```

Figure 17: The visitor combinator `Fwd` creates a syntax-specific visitor from a generic visitor.

```
Identity:  ; // Skip

Sequence:  x.accept_Any(this.first);
           x.accept_Any(this.then);

    Fail:  throw new VisitFailure();

  Choice:  try { x.accept_Any(first); }
           catch (VisitFailure f) {
             x.accept_Any(then);
           }

     All:  for (int i = 0; i < x.nrOfKids(); i++) {
             x.getKid(i).accept_Any(this.v);
           }

     One:  for (int i = 0; i < x.nrOfKids(); i++) {
             try {
               x.getKid(i).accept_Any(this.v);
               return;
             }
             catch(VisitFailure f) { ; }
           }
           throw new VisitFailure();
```

Figure 18: Generic reformulations of the basic visitor combinators. Only the body of `visit_Any(AnyVisitable x)` for the various combinators is shown.

syntax-specific visit methods by forwarding to that generic visitor. The benefit of this new combinator over built-in converse forwarding is that the forwarding behavior is reusable for several generic visitors. This is essential in the setting of visitor *combinators*, because these are typically used to continuously construct new visitors.

Apart from the generic visit method `visit_Any`, the abstract ***`AnyVisitor`*** class declares two additional methods. `nrOfKids` returns the number of children of a visitable node, and `getKid(i)` returns its ith child. Below, we will use these methods for the generic definition of the traversal combinators `All`, and `One`. For *fork* nodes, the definitions of these methods are shown in Figure 15.

5.3 Generic combinators

With this refactoring in place, 'generic specialization' of visitors is possible. In particular, the basic visitor combinators can now be given syntax-independent definitions. The required implementations of the corresponding `visit_Any` methods are shown in Figure 18. When these generic combinators are passed to `Fwd`, the original, syntax-specific combinators are obtained. The penalty for the additional genericity is the method call from `Fwd` to the generic visitor, and, in the case of combinators with

arguments, the explicit cast from *AnyVisitable* to *Visitable* in the visit_Any method.

Note that the generic formulation of the traversal combinators All and One make use of the methods nrOfKids and getKid of the generic *AnyVisitor* interface. Thus, the syntax-specific knowledge about children is hidden behind these two methods.

To assess the performance penalty of the additional genericity, we compared three implementations of a topdown traversal.

Iterating visitor Node actions and traversal code are entangled in a single syntax-specific visitor, which is passed to a syntax-specific accept method.

Syntax-specific combinator The syntax-specific TopDown combinator (see Figure 12) takes a syntax specific visitor with node actions as argument, and is passed to a syntax-specific accept method.

Generic combinator The generic TopDown combinator takes a syntax-specific visitor with node actions as argument, and is passed to a generic accept method.

Benchmarks on balanced trees of various sizes indicated that the syntax-specific combinator is a constant factor 3 slower than the iterating visitor. The generic combinator is yet another constant factor 2 slower.

5.4 Towards libraries of generic algorithms

Given these generic definitions of our visitor combinators where generics are cleanly separated from syntax-specifics, all (traversal) combinators that are generic 'in nature' can indeed be implemented as classes that are reusable across syntaxes. This opens the door to the construction of libraries of reusable generic visitor combinators, such as those in Figures 12 and 14. Programming with visitor combinators then becomes a matter of specializing (in generic or syntax-specific manner) and composing predefined combinators, and feeding them to the accept methods of the particular tree structures that need to be visited. The required syntax-specific code is limited to the specific *Visitor* and *Visitable* interfaces, the accept method implementations, and the Fwd combinator. Such syntax-specific code can be generated from a grammar (see Section 6.1).

To demonstrate the development of generic combinators and their instantiation for specific syntaxes we discuss a small example. First, we will define a generic combinator for simple def-use analysis. This combinator abstracts from which syntactic constructs count as definitions, and which as uses. Secondly, we will instantiate the generic combinator for the syntax of a specific language: GraphXML [10]. GraphXML is an XML-based graph description and exchange language. It allows graphs to be described in terms of nodes and edges, where each edge is defined by a source and a target attribute. We will use the generic def-use analysis to determine the roots and sinks of a graph.

Generic def-use analysis The simple algorithm we wish to implement collects *use* and *definition* occurrences from an input tree while performing a single topdown traversal. After the traversal, two sets should be obtained: the set of entities that are used but not defined, and the set of entities that are defined but not used. Our implementation is shown in Figure 19. It consists of the interface *Collector*, which extends the generic visitor interface *AnyVistor* with a getSet method, and the class DefUse which implements the actual algorithm. Informally, DefUse is the following combinator:

$$\mathit{DefUse(def,use)} \quad =_{def} \quad \mathit{TopDown(Sequence(def,use))}$$

Here, *def* and *use* are visitor parameters that have collecting defined and used entities as side effect. Because *TopDown* is the outermost symbol in the definition of *DefUse*, the latter is implemented by extending the former. The fields def and use store the references to the visitor parameters, for reference by the methods getUndefined and getUnused. These use simple set operations to compute the result sets we wanted to obtain.

```
public interface Collector extends AnyVisitor {
  public Set getSet();
}
```

```
public class DefUse extends TopDown {
  Collector def;
  Collector use;
  public DefUse(Collector def, Collector use) {
    super(new Sequence(def,use));
    this.def = def;
    this.use = use;
  }
  public Set getUndefined() {
    HashSet result = new HashSet(use.getSet());
    return result.removeAll(def.getSet());
  }
  public Set getUnused() {
    HashSet result = new HashSet(def.getSet());
    return result.removeAll(use.getSet());
  }
}
```

Figure 19: A language-independent combinator for def-use analysis.

Instantiation To instantiate the generic algorithm for GraphXML, we need to provide visitors that collect defined and used GraphXML entities. In the domain of graphs, nodes can be considered 'definitions' when they occur as the source of a directed edge. Likewise, they can be considered 'uses' when they occur as target. A node that occurs as source but not as target is a *root* of the graph. Conversely. a node that occurs as target, but not as source is a *sink*. Determining roots and sinks corresponds to determining unused definitions and undefined uses.

Figure 20 shows the relevant fragment of the GraphXML syntax, and the code that instantiates the generic def-use analysis for GraphXML. The classes `CollectTarget` and `CollectSource` are the required implementations of the interface *`Collector`*. Their definitions are similar. The combinator `CollectTarget`, for example, is a specialization of the GraphXML-specific identity combinator, which is defined as *Fwd(Identity)*. It redefines a single visit method: the one corresponding to the syntax rule for targets. The redefined visit method simply adds the target it encountered to the local `Set` field. Given these collector classes, the generic def-use analysis can be instantiated for GraphXML as follows:

DefUse(CollectSource,CollectTarget)

This is conveyed by the test method in Figure 20.

The def-use analysis is only one example of a generic algorithm that can be programmed with visitor combinators. In Stratego, generic algorithms have been defined e.g. for graph transformation and analysis, and for substitution, renaming and unification [22, 21].

6. Application

In the previous sections, we have seen small, tutorial examples of programming with (generic) visitor combinators. In this section, we discuss our experiences with applying them on a larger scale.

Our primary objective in developing visitor combinators has been to support the employment of object-oriented programming technology in the domain of language processing. Section 6.1 presents JJForester, a tool that supports the use of visitor combinators in this domain. Section 6.2 discusses the application to COBOL control flow analysis, in the context of legacy system redocumentation [6].

```
EdgeAttribute := 'source' '=' AttValue
               | 'target' '=' AttValue
```

```
public class CollectTarget extends Fwd
                           implements Collector {
  Set targets = new Set();
  public CollectTarget() {
    super(new Identity());
  }
  public void visit_Target(Target attValue) {
    targets.add(attValue);
  }
  public Set getSet() {
    return targets;
  }
}
```

```
public void testDefUseVisitor(GraphXML g) throws VF {
  DefUse defuse = new DefUse(new CollectSource(),
                             new CollectTarget());
  g.accept_Any(defuse);
  System.out.println("Sinks: "+defuse.getUndefined());
  System.out.println("Roots: "+defuse.getUnused());
}
```

Figure 20: Retrieving the roots and sinks of a GraphXML document by GraphXML-specific instantiation of the generic def-use combinator. The class `CollectSource` is not shown. It is similar to `CollectTarget`.

We conjecture that our techniques are equally applicable to any domain that involves traversal of (heterogeneous) object structures, tree-shaped or not (see also Section 7).

6.1 Tool support

As mentioned above, the only syntax-specific code that is needed for programming with visitor combinators, are the specific interfaces *`Visitor`* and *`Visitable`*, the accept methods, and the combinator `Fwd`. Rather than writing this code manually, it can be generated from tree or syntax definitions. We have extended our parser and visitor generator JJForester [14] with this functionality. The architecture of this tool is shown in Figure 21. The input to JJForester consists of a grammar specified in the syntax definition formalism SDF [9]. SDF is supported by a parse table generator, and a *generalized* LR parser generator. These tools are available as stand-alone components, which are reused by JJForester to generate a parse table at compilation time, and to parse input terms at run time. The SDF grammar is also passed to the code-generation component of JJForester, which emits the Java classes that instantiate the visitor combinator framework. The user programs against the generated code and the framework. The framework, the generated code and the user code can be compiled to byte code, and run on a virtual machine. During run time, calls to parse methods will lead to invocations of the parser, which returns abstract syntax trees (ASTs). These ASTs are passed to factory methods to build the actual object structure.

In Section 5, we added the possibility of generic specialization to the possibility of syntax-specific, per production specialization. For a many-sorted syntax, an intermediate level of genericity is conceivable: per sort specialization. This can be realized by inserting another set of interfaces, and another forwarding combinator, like `Fwd`. JJForester supports a simplified variation on this scheme, where both layers of forwarding are done in a single `Fwd` combinator.

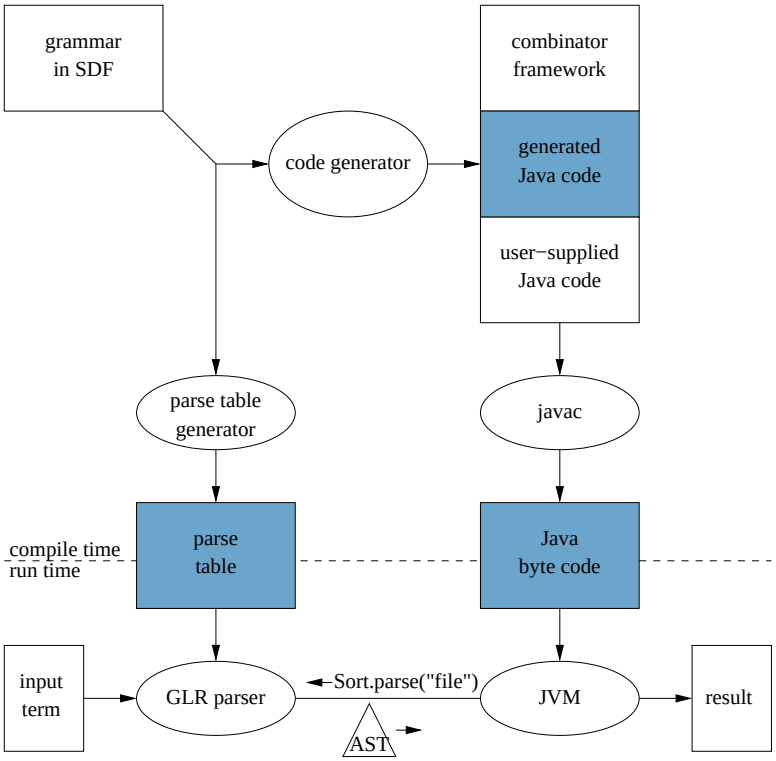

Figure 21: Architecture of JJForester: tool support for visitor combinators. Ellipses are tools, shaded boxes are generated code.

6.2 COBOL control-flow analysis

Visitor combinators are currently being applied for reimplementation and further development of DocGen, a documentation generator for COBOL [6]. Given the sources of a COBOL legacy system, DocGen produces hypertext documentation that offers browsable views of the system at various abstraction levels. One of these views are clickable control-flow graphs. We have used visitor combinators to reimplement the underlying control-flow analysis component. Furthermore, we have started to develop components for further control-flow-based analyses, such as complexity analysis and dead-code analysis.

The control-flow visualization of DocGen is performed in three phases:

Extraction The essential control-flow information is extracted from COBOL source by means of lexical analysis. This involves e.g. recognition of *perform* statements, *conditionals*, and *section* boundaries.

Analysis The control-flow information is analyzed to generate a control-flow graph represented in the *dot* format. The analysis involves e.g. resolving references to called sections, and determining the calling scope.

Graph layout The graph layout tool *dot* [13] is used to calculate position information for the nodes and edges of the graph, and to produce an image of the graph.

The extraction and analysis components have both originally been implemented in the regular expression-based scripting language Perl [25]. The control-flow information is communicated between them via a special-purpose exchange format, which goes by the name of *conditional perform format*. For further development of the documentation generator, reimplementation of the analysis component in a main stream, general purpose language, was deemed necessary.

The first step towards the reimplementation of the analysis component was the development of a grammar for the conditional perform format. The resulting SDF grammar contains 5 context-free non-terminals, and 11 context-free productions. Secondly, the grammar was fed to JJForester to generate a parse table and the necessary Java code to instantiate the visitor combinator framework. This resulted in 2350 lines of Java code (including constructor methods, factory methods, parse methods, accept methods, getters, and setters).

To reimplement the analysis component, 6 visitor combinators were written, totaling 200 lines of Java code. To compare, the Perl implementation consisted of 400 lines. Apart from various basic visitor combinators, the derived combinators `DownUp` (see Figure 14) and `DefUse` (see Figure 19) were reused from the framework. We briefly sketch some techniques we used.

- During traversal of a conditional perform term, the analyzer accumulates graph elements in a visitor with state. After the traversal, the graph elements are used to construct the actual result graph.
- During traversal, the current scope is maintained on a stack. This stack is again provided as the state of a visitor. Auxiliary combinators for pushing and popping scopes on and off the stack are passed as *down* and *up* parameters to the `DownUp` combinator.
- When several children of a given parent must be traversed with different scopes, the traversal is stopped in the parent, and restarted for each child. Stopping the traversal is realized with by setting the *stop* argument of `TopDown` to a visitor combinator that tests for the stop condition. Restarts are realized by invocations of `accept` methods with the current visitor (`this`) as argument.
- The main traversal needs some global information about which labels (procedure names) are actually performed (called). This global information is collected in a symbol table during a separate traversal, preceding the main analysis. The table is passed to the visitor that implements

the main analysis by appropriately initializing its state. To implement the first traversal, we instantiated the `DefUse` combinator, by passing reusable combinators that collect callers (use) and callees (def).

So far, we applied the new analysis component to three legacy systems, with a total of 440 thousand lines of code. The corresponding conditional perform files total 75 thousand lines. Using Java 1.2 on a 300Mhz Sun ULTRA-5 workstation with 256Mb of memory, the analysis component took under 30 minutes to process these (including parsing and tree construction).

With respect to the original Perl implementation, the new implementation with visitor combinators in Java has a number of advantages. The amount of code that needs to be written is significantly smaller, easier to understand and to maintain. The extensibility and reusability of the Java code is also much better. These benefits of the reimplementation payed off during further development of the documentation generator. For example, further control-flow based analyses, such as dead code analysis and complexity analysis, could now be implemented by re-instantiation of the traversal combinator of the control-flow visualizer.

7. Concluding remarks

We reconstructed the basic strategy combinators of Stratego [23] in the object-oriented setting as a suite of basic visitor combinators (reusable classes). These visitor combinators remove the limitations of the classic visitor pattern with respect to composability of visitors and with respect to traversal control. Additionally, we refactored the visitor combinator design pattern into a syntax-independent framework, and a design pattern for instantiating the framework for a specific syntax. This visitor combinator framework opens the door to a new style of generic object-oriented programming, where new frameworks are built by composition of basic visitor combinators of the basic framework.

7.1 Evaluation

How does programming visitor combinators compare with programming with ordinary visitors, or without visitors at all?

Explicit stack maintenance Visitors *iterate* over an object structure. Consequently, they can not use the call stack to pass data. Instead, the state of the visitor is used for this. When, at back-tracking, the state needs to be restored, this is not done automatically as it would when the call stack would be used. Instead, state restoration at back-tracking needs to be done explicitly by the visitor. When using visitor *combinators*, the stack maintenance can be done by separate, reusable combinators. This technique was successfully applied in the reimplementation of the control-flow visualizer (Section 6).

Performance Using visitor combinators introduces the overhead of forwarding of method calls between combinators. Of course, the precise amount of forwarding overhead is strongly dependent on the particular constellation of combinators, and on the ratio of combinator code. On the other hand, the additional traversal control can be used to construct more efficient traversal strategies. In our benchmarks experiments and in the COBOL control-flow applications, we did not experience performance problems. For a complete picture of performance consequences, more experience and experiments are needed.

Paradigm shift The style of programming with visitors is one step removed from the 'natural' method passing style of object-oriented programming, where data and operations on the data are encapsulated in the same object. The style of programming with visitor combinators is yet one more step removed from this 'natural' style. In fact, programming with visitor combinators can be considered a *paradigm shift*, as it not only separates data from operations, but also introduces the technique of growing programs, not by adding classes and methods, but by composing compound classes from basic ones.

Robustness A well-know problem of the visitor design pattern is that visitors are *brittle* with respect to changes in the class hierarchy on which they operate. When, for instance, a class is added to the hierarchy, all previously defined visitors need to implement an additional visit method. To some extent, default `Visitor` implementations provide isolation against such changes [8].

Our visitor combinators actually are such default implementations. By inheriting from them, user-defined combinators only need to depend on those fragments of the class hierarchy (or syntax) that are relevant to the functionality they implement. By making the combinators syntax-independent, we have even made these default implementations robust against changes in the class hierarchy. Finally, the generator concentrates all syntax-dependence in the syntax itself. For instance, one simply adds a new production to the syntax, and the generator takes care of updating the class hierarchy, the `Visitor` class, and the `Fwd` combinator.

7.2 Generic traversal across paradigms

The generic traversal combinators presented in this paper are the result of transposing concepts that have (recently) been developed in other paradigms. The most important source of inspiration is the strategic term rewriting language Stratego [23]. Our visitor combinators are reconstructions of the primitive strategy combinators of Stratego. Similar primitives can also be found in the rewriting calculus and its extensions [5]. However, both these languages are untyped (though a proposal for an appropriate type system has recently been drafted [15]). Previously, we reconstructed Stratego's primitives in the strongly typed functional language Haskell [16], and these typed strategies guided the design of our visitor combinators. Thus, incarnations of the concept of strategy combinators, including combinators for generic traversal, are now available in three different programming paradigms.

Strategy combinators are closely related to folds. Many-sorted folds can be constructed by specialization and combination of fold combinators just like strategies [17]. The main difference between folds and strategies is that the former employ a fixed bottom-up traversal strategy. The relation between strategies and folds is discussed in more detail in [16].

7.3 Related work

Traversal control The *hierarchical* visitor pattern [1] employs a visitor interface with two methods per visitable class: one to be performed upon entering the class, and one to be performed before leaving it. This pattern allows hierarchical navigation (keeping track of depth) and conditional navigation (cutting off traversal below a certain point). As Figure 14 demonstrates, visitor combinators can be used to achieve such traversal control, and much more.

In adaptive programming, and its implementation by the Demeter system [18], a notion is present of traversal *strategies* for object structures. These strategies should not be confused with the strategies and strategy combinators of the Stratego language which inspired our visitor combinators. Demeter's strategies are high-level descriptions of paths through object graphs in terms of source node, target node, intermediate nodes, and predicates on nodes and edges. These high level descriptions are translated (at compile time) into 'dynamic roadmaps': methods that upon invocation traverse the object structure along a path that satisfies the description. During traversal, a visitor can be applied. The aim of these strategies is to make classes less dependent on the particular class structure in which they are embedded, i.e. to make them more robust, or adaptive. Unlike our visitor combinators, Demeter's strategies are *declarative* in nature and can not be executed themselves. Instead, traversal code must be generated from them by a constraint-solving compiler. On the other hand, while reducing commitment to the class structure, Demeter's strategies do not eliminate all references to the class structure. Visitor combinators allow definition of *fully* generic traversals.

To complement Demeter's declarative strategies, a domain-specific language (DSL) has been proposed to express recursive traversals at a lower, more explicit level [19]. This traversal DSL sacrifices some compactness and adaptiveness in order to gain more control over propagation and computation of results, and to prevent unexpected traversal paths due to underspecification of traversals. With respect to our visitor combinators, this traversal DSL provides cleaner support for *recursive* traver-

sals; no explicit stack managment is needed. On the other hand, visitor combinators are more generic, extensible and reusable, and they offer more traversal control. Also, they do not essentially rely on tool support.

Generics The separation of specifics and generics in the visitor pattern is addressed by Vlissides' *staggered* visitor pattern [24], and the *extended visitor* pattern supported by the SableCC tool [7]. Here the aim of this separation is to allow extension of the syntax without altering existing (visitor) code. In the extended visitor pattern of SableCC, the generic visitor interface does not contain any methods. In the staggered pattern, the generic visitor contains a generic visit method, similar to our `visit_Any`. The main difference with our approach is that in these patterns forwarding from specific to generic visit methods is done in the *`Visitor`* class, while we do it in a separate reusable combinator `Fwd`. In the presence of `Choice`, the `Fwd` combinator allows not only extension of a syntax, but also merging of several syntaxes.

The `Walkabout` class [20] makes essential use of reflection (including, but not limited to RTTI) to model generic visiting behavior. The class performs a traversal through an object structure. At each node it reflects on itself to ascertain whether it contains a visit method for the current node. If not, it uses reflection to determine the fields of the current node and calls itself on these. The authors report high performance penalties for the extensive reliance on reflection. The benefit is that no (syntax-specific) accept methods, visitor interface, or visitor combinators need to be supplied. The `Walkabout` class implements a fixed top-down traversal strategy, which is cut off below nodes for which the visitor fires (i.e. *DownUp(Identity,v,Identity)*, see Figure 14). It is not restricted to tree shaped object structures.

7.4 Future work

Implementation The support currently offered by JJForester has a restraining effect. Classes are generated in their entirety from a grammar. Programmers can not add their own methods or fields. This may be desirable, for instance, to allow decoration of the tree that is being traversed. To offer more flexibility, JJForester should additionally generate such decoration fields, or *weave* the generated methods into given classes (a form of aspect-oriented programming [12]).

For several visitor combinators we gave definitions in mathematical notation, as well as in Java. Obviously, a language extension could be defined which allows combinator definitions in this concise, mathematical style. An implementation of such an extension would map these high-level combinator definitions to their more verbose counterparts in the base language. Such a language extension would be amenable to optimization by means of source-to-source transformation, on the basis of the algebraic equations that hold between visitor combinators.

Extensions and alternatives It would be convenient to include some special-purpose visitors into the combinator set, e.g. `ToString`, `Equals`, and `Clone` visitors. These would help to address common traversal scenarios such as pretty-printing, and non-destructive transformation. Currently, only a `ToString` combinator is generated by JJForester.

The ATerm library [4] supports (space) efficient representation and exchange of generic trees through maximal subtree sharing. We want to design a variation of our combinator suite that makes use of the ATerms for representation of both generic and syntax-specific trees. This would enable us to deal with exceptionally large trees. As maximal subtree sharing does not mix well with destructive tree operations, the combinators would need to be reformulated to use visit and accept methods that return (modified) trees.

Document processing, like language processing, essentially involves tree traversal. The GraphXML example of Section 5.4 illustrates how (DTD-aware) document processing can be done with visitor combinators. We want to compare our techniques with existing proposals for XML-document traversal (*cf.* DOM [2], XSLT [3]), and investigate whether these could benefit from the combinator approach.

Until now, we assumed tree shaped object graphs. This restriction is not essential. When it is

removed, the need arises for some mechanism to mark nodes as visited, and to ensure termination (in case of cycles). It remains to be investigated whether the current set of combinators needs to be modified to accomodate such a mechansism.

Availability JJForester is free software, developed at the CWI. It can be downloaded from http://www.jjforester.org/. The distribution includes the visitor combinator framework, the code generator, and examples.

Acknowledgments Thanks to Arie van Deursen for his scrutiny. Jan Heering, Merijn de Jonge, Paul Klint, Tobias Kuipers, Leon Moonen, and Eelco Visser gave useful feedback. I am indebted to Ralf Lämmel for our ongoing cooperation on the development of ideas underlying this paper.

References

1. Portland pattern repository. http://www.c2.com/cgi/wiki.
2. Document Object Model (DOM) Level 1 Specification Version 1.0, W3C Recommendation. Oct. 1998.
3. XSL Transformations (XSLT) Version 1.0, W3C Recommendation. Nov. 1999.
4. M. Brand, H. Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
5. H. Cirstea and C. Kirchner. Introduction to the rewriting calculus. Rapport de recherche 3818, INRIA, Dec. 1999.
6. A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings of the International Conference on Software Maintenance (ICSM '99)*. IEEE Computer Society, 1999.
7. E. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS USA 98 (Technology of Object-Oriented Languages and Systems)*. IEEE, 1998.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
9. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
10. I. Herman and M. Marshall. GraphXML – An XML-based graph description format. In *Symposium on Graph Drawing (GD 2000)*, volume 1984 of *LNCS*, pages 52–62. Springer, 2000. A full grammar for GraphXML can be found at http:/www.program-transformation.org/gb/.
11. P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *POPL '97: 24th Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 15–17 Jan. 1997.
12. G. Kiczales, J. Lamping, et al. Aspect-oriented programming. In *Proceedings of ECOOP'97*, number 1241 in LNCS. Springer Verlag, 1997.
13. E. Koutsofios. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, Nov. 1996.
14. T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In *Workshop on Language Descriptions, Tools and Applications (LDTA)*, To appear in ENTCS, 2001.

15. R. Lämmel. Typed Generic Traversal. Technical report, CWI, 2001. To appear.

16. R. Lämmel and J. Visser. Type-safe functional strategies. Manuscript in preparation. Preliminary version in draft proceedings of the Scottish Functional Programming Workshop, St Andrews, 2000.

17. R. Lämmel, J. Visser, and J. Kort. Dealing with large bananas. In J. Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

18. K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, July 1997.

19. J. Ovlinger and M. Wand. A language for specifying recursive traversals of object structures. In *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 70–81, 1999.

20. J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

21. E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming*, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

22. E. Visser. *The Stratego Library (version 0.4.22)*. Institute of Information and Computing Sciences, Utrecht, The Netherlands, 2001. Most recent version at http://www.stratego-language.org/.

23. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

24. J. Vlissides. Visitor in frameworks. *C++ Report*, 11(10), Nov. 1999.

25. L. Wall, R. L. Schwartz, T. Christiansen, and S. Potter. *Programming Perl*. Nutshell Handbook. O'Reilly & Associates, 2nd edition, 1996.