Building Program Understanding Tools Using Visitor Combinators

Arie van Deursen

CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
`http://www.cwi.nl/~arie/`

Joost Visser

CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
`http://www.cwi.nl/~jvisser/`

ABSTRACT

Program understanding tools manipulate program representations, such as abstract syntax trees, control-flow graphs, or data-flow graphs. This paper deals with the use of *visitor combinators* to conduct such manipulations. Visitor combinators are an extension of the well-known visitor design pattern. They are small, reusable classes that carry out specific visiting steps. They can be composed in different constellations to build more complex visitors. We evaluate the expressiveness, reusability, ease of development, and applicability of visitor combinators to the construction of program understanding tools. To that end, we conduct a case study in the use of visitor combinators for control-flow analysis and visualization as used in a commercial Cobol program understanding tool.

1998 ACM Computing Classification System: D.2.9, D.2.2, D.2.5, D.2.7

Keywords and Phrases: Program analysis, program comprehension, visitor design pattern, software visualization.

Note: Work carried out under projects SEN 1.1, *Software Renovation* and SEN 1.3, *Domain-Specific Languages*.

Note: To appear in Proceedings of the 10th International Workshop on Program Comprehension (IWCP), IEEE Computer Society, 2002.

1. Introduction

Program analysis and source models Program analysis is a crucial part of many program understanding tools. Program analysis involves the construction of source models from the program source text and the subsequent analysis of these models. Depending on the analysis problem, these source models might be represented by tables, trees, or graphs.

More often than not, the models are obtained through a sequence of steps. Each step can construct new models or refine existing ones. Usually, the first model is an (abstract) syntax tree constructed during parsing, which is then used to derive graphs representing, for example, control or data flow.

Visiting source models The intent of the visitor design pattern is to "represent an operation to be performed on the elements of an object structure. A visitor lets you define a new operation without changing the classes of the elements on which it operates" [8]. Often, visitors are constructed to *traverse* an object structure according to a particular built-in strategy, such as *top-down*, *bottom-up*, or *breadth-first*.

A typical example of the use of the visitor pattern in program understanding tools involves the traversal of abstract syntax trees. The pattern offers an abstract class *Visitor*, which defines a series of methods that are invoked when nodes of a particular type (expressions, statements, *etc.*) are visited. A concrete *Visitor* subclass refines these methods in order to perform specific actions when accepted by a given syntax tree.

Visitors are useful for analysis and transformation of source models for several reasons. Using visitors makes it easy to traverse structures that consist of many different kinds of nodes, while conducting actions on only a selected number of them. Moreover, visitors help to separate traversal from representation, making it possible to use a single source model for various sorts of analysis.

Visitor Combinators Recently, visitor *combinators* have been proposed as an extension of the regular visitor design pattern [14]. The aim of visitor combinators is to *compose* complex visitors from elementary ones. This is done by simply passing them as arguments to each other. Furthermore, visitor combinators offer full *control* over the traversal strategy and applicability conditions of the constructed visitors.

The use of visitor combinators leads to small, reusable classes, that have little dependence on the actual structure of the concrete objects being traversed. Thus, they are less brittle with respect to changes in the class hierarchy on which they operate. In fact, many combinators (such as the *top-down* or *breadth-first* combinators) are completely *generic*, relying only on a minimal *Visitable* interface. As a result, they can be reused for *any* concrete visitor instantiation.

Goals of the paper The concept of visitor combinators is based on solid theoretical ground, and it promises to be a powerful implementation technique for processing source models in the context of program analysis and understanding. Now

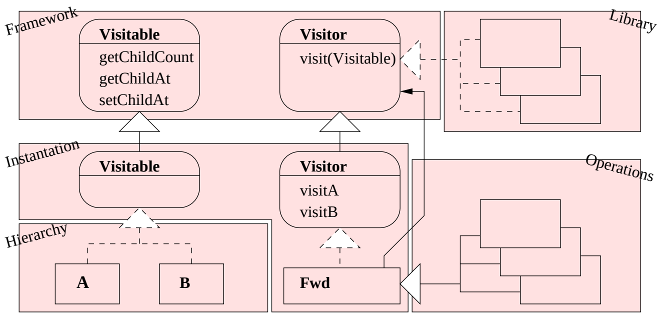


Figure 1. The architecture of JJTraveler.

this concept needs to be put to the test of practice.

We have implemented ControlCruiser, a tool for analyzing and visualizing intra-program control flow for Cobol. In this paper, we explain by reference to ControlCruiser how visitor combinators can be used to develop program understanding tools. We discuss design tactics, programming techniques, unit testing, implementation trade-offs, and other engineering practices related to visitor combinator development. Finally, we asses the risks and benefits of adopting visitor combinators for building program understanding tools.

2. Visitor Combinators

Visitor combinator programming was introduced in [14] and is supported by JJTraveler: a combination of a framework and library that provides *generic visitor combinators* for Java. This section briefly discusses the key elements of JJTraveler.

2.1. The architecture of JJTraveler

Figure 1 shows the architecture of JJTraveler (upper half) and its relationship with an application that uses it (lower half). JJTraveler consists of a *framework* and a *library*. The application consists of a class *hierarchy*, an *instantiation* of JJTraveler's framework for this hierarchy, and the *operations* on the hierarchy implemented as visitors.

Framework The JJTraveler framework offers two generic interfaces, *Visitor* and *Visitable*. The latter provides the minimal interface for nodes that can be visited. Visitable nodes should offer three methods: to get the number of child nodes, to get a child given an index, and to modify a given child. The *Visitor* interface provides a single *visit* method that takes any visitable node as argument. Each visit can *succeed* or *fail*, which can be used to control traversal behavior. Failure is indicated by a *VisitFailure* exception.

Library The library consists of a number of predefined visitor combinators. These rely only on the generic *Visitor* and *Visitable* interfaces, not on any specific underlying class hierarchy. An overview of the library combinators is shown in Figure 2. They will be explained in more detail below.

Name	Args	Description
Identity		Do nothing
Fail		Raise *VisitFailure* exception
Not	v	Fail if v succeeds, and v.v.
Sequence	v_1, v_2	Do v_1, then v_2
Choice	v_1, v_2	Try v_1, if it fails, do v_2
All	v	Apply v to all immediate children
One	v	Apply v to one immediate child
IfThenElse	c,t,f	If c succeeds, do t, otherwise do f
Try	v	*Choice(v,Identity)*
TopDown	v	*Sequence(v,All(TopDown(v)))*
BottomUp	v	*Sequence(All(BottomUp(v)),v)*
OnceTopDown	v	*Choice(v,One(OnceTopDown(v)))*
OnceBottomUp	v	*Choice(One(OnceBottomUp(v)),v)*
AllTopDown	v	*Choice(v,All(AllTopDown(v)))*
AllBottomUp	v	*Choice(All(AllBottomUp(v)),v)*

Figure 2. JJTraveler's library (excerpt).

Instantiation To use JJTraveler, one needs to instantiate the framework for the class hierarchy of a particular application. To do this, the hierarchy is turned into a visitable hierarchy by letting every class implement the *Visitable* interface. Also, the generic *Visitor* interface is extended with specific visit methods for each class in the hierarchy. Finally, a single implementation of the extended visitor interface is provided in the form of a visitor combinator *Fwd*. This combinator forwards every specific visit call to a generic default visitor given to it at construction time. Concrete visitors are built by providing *Fwd* with the proper default visitor, and overriding some of its specific visit methods.

Though instantiation of JJTraveler's framework can be done manually, automated support for this is provided by a generator, called JJForester [10]. This generator takes a grammar as input. From this grammar, it generates a class hierarchy to represent the parse trees corresponding to the grammar, the hierarchy-specific *Visitor* and *Visitable* interfaces, and the *Fwd* combinator. In addition to framework instantiation, JJForester provides connectivity to a generalized LR parser [2].

Operations After instantiation, the application programmer can implement operations on the class hierarchy by specializing, composing, and applying visitors.

The starting point of hierarchy-specific visitors is *Fwd*. Typical default visitors provided to *Fwd* are *Identity* and *Fail*. Furthermore, *Fwd* contains a method *visitA* for every class *A* in the hierarchy, which can be overridden in order to construct specific visitors. As an example, an *A*-recognizer *IsA* (which only does not fail on *A*-nodes) can be obtained by an appropriate specialization of method *visitA* of *Fwd(Fail)*.

Visitors are combined by passing them as (constructor) arguments. For example, *All(IsA)* is a visitor which checks that any of the direct child nodes are of class *A*, and *OnceTopDown(IsA)* is a visitor checking whether a tree contains any *A*-node. Visitors are applied to visitable objects through the *visit* method, such as *IsA.visit(myA)* (which does nothing), or

```
public class Sequence implements Visitor {
  Visitor v1;
  Visitor v2;
  public Sequence(Visitor v1, Visitor v2) {
    this.v1 = v1;
    this.v2 = v2;
  }
  public void visit(Visitable x) {
    v1.visit(x);
    v2.visit(x);
} }
```

Figure 3. The *Sequence* combinator.

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

Figure 4. The *Try* combinator.

IsA.visit(myB) (which fails).

2.2. A library of generic visitor combinators

Figure 2 shows high-level descriptions for an excerpt of JJ-Traveler's library of generic visitor combinators. A full overview of the library can be found in the online documentation of JJTraveler. Two sets of combinators can be distinguished: *basic* combinators and *defined* combinators, which can be described in terms of the basic ones as indicated in the overview. Note that some of these definitions are *recursive*.

Basic combinators Implementation of the generic visitor combinators in Java is straightforward. Figures 3 and 4 show implementations for the basic combinator *Sequence* and the defined combinator *Try*. The implementation of a basic combinator follows a few simple guidelines. Firstly, each argument of a basic combinator is modeled by a field of type *Visitor*. For *Sequence* there are two such fields. Secondly, a constructor method is provided to initialize these fields. Finally, the generic visit method is implemented in terms of invocations of the visit method of each *Visitor* field. In case of *Sequence*, these invocations are simply performed in sequence.

Defined combinators The guidelines for implementing a defined combinator are as follows. Firstly, the superclass of a defined combinator corresponds to the outermost combinator in its definition. Thus, for the *Try* combinator, the superclass is *Choice*. Secondly, a constructor method is provided that supplies the arguments of the outermost constructor in the definition as arguments to the superclass constructor method (`super`). For *Try*, the first superclass constructor argument is the argument of *Try* itself, and the second is *Identity*. The visit method is simply inherited from the superclass.

Recursive combinators In order to demonstrate how visitor combinators can be used to build recursive visitors with sophisticated traversal behavior, we will develop a new generic

```
public class TopDownWhile extends Choice {
  public TopDownWhile(Visitor v1, Visitor v2) {
    super(null,v2);
    setArgument(1,new Sequence(v1,new All(this)));
  }
  public TopDownWhile(Visitor v) {
    this(v,new Identity());
} }
```

Figure 5. The $TopDownWhile$ combinator.

visitor combinator $TopDownWhile(v_1, v_2)$.

$$\begin{aligned} &TopDownWhile(v_1,v_2) = \\ &\quad Choice(Sequence(v_1,All(TopDownWhile(v_1,v_2))),v_2) \end{aligned}$$

The first argument v_1 represents the visitor to be applied during traversal in a top-down fashion. When, at a certain node, this visitor v_1 fails, the traversal will not continue into subtrees. Instead, the second argument v_2 will be used to visit the current node. The encoding in Java is given in Figure 5. Note that Java does not allow references to `this` until after the `super` constructor has been called. For this reason, the first argument, which contains the recursion, gets its value not via `super`, but via the `setArgument()` method. Note also that the visitor has a second constructor method that provides a shorthand for calling the first constructor with *Identity* as second argument.

3. Cobol Control Flow

The example we use to study the application of visitor combinators to the construction of program understanding tools deals with Cobol control flow. Cobol has some special control-flow features, making analysis and visualization an interesting and non-trivial task. The analysis we describe is taken from DocGen (see [5]), an industrial documentation generator for a range of languages including Cobol, which has been applied to millions of lines of code.

Control flow in Cobol takes place at two different levels. A Cobol system consists of a series of *programs*. These programs can invoke each other using *call* statements. A Cobol system typically consists of several hundreds of programs.

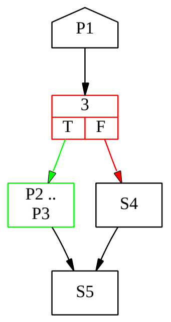
In this paper, we focus on control flow *within* a program, for which the *perform* statement is used. This perform statement is like a procedure call, except that no parameters can be passed (global variables have to be used for that). Typical programs are 1500 lines large, but is not uncommon to have individual programs of more than 25,000 lines of code, resulting in significant program comprehension challenges.

3.1. Cobol Procedures

Cobol does not have explicit language constructs for procedure calls and declarations. Instead, it has labeled *sections* and *paragraphs*, which are the targets of *perform* and *goto*

```
PROCEDURE DIVISION.
 P1. ACCEPT X
     IF X = "1"
        PERFORM P2 THRU P3
     ELSE
        PERFORM S4.
     STOP RUN.
 P2. DISPLAY "HELLO".
 P3. PERFORM S5.
 S4 SECTION.
 P4. DISPLAY "HI".
 P5. PERFORM S5.
 S5 SECTION.
     DISPLAY "WORLD".
```

(a) Cobol source



(b) Corresponding call graph

Figure 6. Example Cobol source and graph

statements. Perform statements may invoke individual sections and paragraphs, or *ranges* of them. A section can group a number of paragraphs, but this is not necessary.

Figure 6(a) shows an example program in which sections, paragraphs, and ranges are performed. Paragraph `P1` acts as the main block, which reads an input value X. If it is "1", the program invokes the range of paragraphs `P2` through `P3`. This range first prints HELLO, and then performs section `S5`, which prints WORLD. If the value read is not "1", the main program invokes just the section `S4`. This section consists of two paragraphs, of which `P4` displays HI, and `P5` invokes `S5` to display WORLD.

This example illustrates an important program understanding challenge for Cobol systems. Viewed at an abstract level the program involves four *procedures*: `P1`, the range `P2..P3`, `S4`, and `S5`. Paragraphs `P3`, `P4` and `P5` are not intended as procedures. This abstract view needs to be reconstructed by analysis, because the entry and exit points of performed blocks of code is determined not by their declaration, but by the way they are invoked in other parts of the program. In general, this makes it hard to grasp the control flow of a Cobol program, especially if it is of non-trivial size.

Typical, Cobol programmers try to deal with this issue by following a particular *coding standard*. Such a standard prescribes that, for example, only sections can be performed, or only ranges, or that perform...thru can only be used for paragraphs with names that explicitly indicate that they are the start or end-label of a range. Such standards, however, are not enforced. Moreover, especially older systems may have been subjected to multiple standards, leaving a mixed style for performing procedures. Again, it takes analysis in order to find out which styles are actually being used at each point.

The formal semantics of "perform P_1 thru P_n" is that paragraphs are executed starting with P_1 until control reaches P_n. In principle, this makes determining which paragraphs are actually spanned by a range a run time problem, which cannot necessarily be solved statically. In the vast majority (99%) of Cobol programs, however, ranges coincide with syntactic sequences. In this paper, we will assume that ranges are syntactically sequenced, and we refer to [6] for ways of dealing with dynamic ranges (where visitor combinators may well be applicable as well).

3.2. Analysis and visualization

To help maintenance programmers understand the control flow of individual Cobol programs, a tool is needed for analysis and visualization of a program's perform dependencies. From such a call graph, one could instantly glean which perform style is predominant, which sections, paragraphs or ranges make up procedures, and how control is passed between these procedures.

When discussing these procedure-based call graphs with maintenance programmers, they indicated that they would also like to know *under what conditions* a procedure gets performed. This gave raise to the so-called *conditional call graph* (CCG), an example of which is shown in Figure 6(b). These graphs contain nodes for procedures and conditionals, which are connected by edges that represent call relations and syntactic nesting relations. CCGs are part of the DocGen redocumentation system, in which these graphs are hyperlinked to both the sources and to documentation at higher levels of abstraction (see [5]).

Conditional call graphs are also a good starting point for computing detailed (per-procedure) metrics, as part of a systematic quality assurance (QA) effort. Example QA metrics include McCabe's cyclomatic complexity, fan-in, fan-out, deepest nesting level, coding style violations (goto's across section boundaries, paragraphs performing sections, or v.v.), dead-code analysis, and more.

4. ControlCruiser Architecture

We have implemented the analysis and visualization requirements just described using visitor combinators. The result is ControlCruiser, a Cobol analysis tool that provides insight into the intra-program call structure of Cobol programs. The tool employs several visitable source models, and performs various visitor-based traversals over them. This section discusses the ControlCruiser architecture; the next covers in detail how visitor combinators have been used in its implementation.

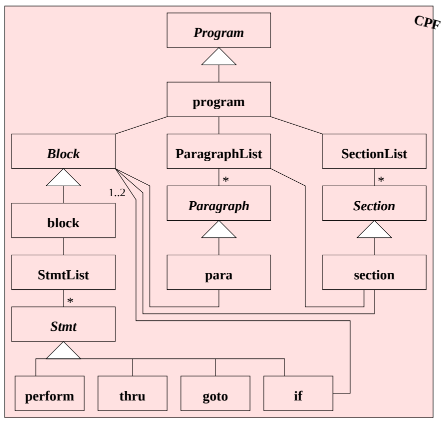
4.1. Initial Representation

The starting point for ControlCruiser is a simple language containing just the statements representing Cobol sections, paragraphs, perform statements, and conditional or looping constructs. An example of this *Conditional Perform Format* (CPF) is shown in Figure 7(a).

We obtain CPF from Cobol sources using a Perl script written according to the principles discussed in [4]. This script

```
PARA 2 P1
  IF 3
    THRU 4 P2 P3
  ELSE 5
    PERFORM 6 S4
  END-IF 7
END-PARA 9 P1
PARA 9 P2
END-PARA 10 P2
PARA 10 P3
  PERFORM 10 S5
END-PARA 11 P3
SECTION 11 S4
  PARA 12 P4
  END-PARA 13 P4
  PARA 13 P5
    PERFORM 13 S5
  PARA 14 P5
END-SECTION 14 S4
SECTION 14 S5
END-SECTION 15 S5
```

(a) CPF for Fig 6



(b) The generated CPF class hierarchy

Figure 7. Conditional Perform Format (CPF)

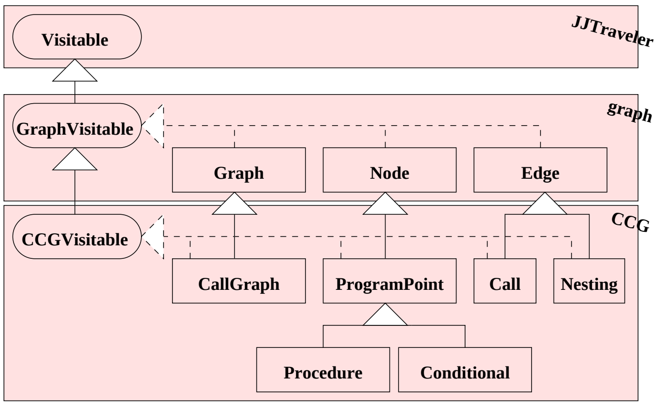


Figure 8. Class hierarchy for graph representations.

takes care of handling the tricky details of the Cobol syntax, such as scope termination of if-constructs.

The result is an easy to parse CPF file. We have written a grammar for the CPF format, and used JJForester to derive a class hierarchy for representing the corresponding trees. All nodes in such trees are of one of the types shown in Figure 7(b). Since these all realize the *Visitable* interface, we can implement all subsequent steps with visitor combinators.

4.2. Graph Representation

To analyze Cobol's control flow in an easy way, we have to create a graph out of the tree representation corresponding to Cobol statements. For this, we use an additional visitable source model which consists of two layers (see Figure 8).

The first layer is a generic graph model, with explicit classes for nodes, edges, and the overall graph providing entry points into the graph. Each of these classes implements a *GraphVisitable* interface, which is an extension of generic visitables. The classes are implemented such that the children of a node are defined as its outgoing edges, the children of an edge as its outgoing node, and the children of a graph as the collection of all nodes, thus making it possible to traverse a graph using visitor combinators. A forwarding visitor combinator taking a generic visitor as argument is provided as required (not shown).

The second layer is a specialization of the generic graph model to the level of control flow, called *Conditional Control Graphs* (CCGs). This representation contains classes for procedures, conditional statements, and different types of edges. Program points correspond to places in the original CPF tree, and have a pointer back to their originating construct. Each class implements the *CCGVisitable* interface. The forwarding combinator of CCG (not shown) contains three levels of forwarding. First, visit methods of classes low in the hierarchy (such as Procedure and Conditional) invoke a visit method higher up in the hierarchy (to ProgramPoint). Second, visit methods for top-level CCG classes forward to visit methods in a visitor at the generic graph level. Third, graph-specific visitors forward to generic visitors by default. Observe that thanks to this two-layer design, visitors designed for graphs can be reused to build visitors for CCGs. This will be demonstrated in Section 5.2.

4.3. Graph Construction

Constructing the CCG graph from the initial CPF tree representation is done using various visitors operating on CPF trees. In order to identify those paragraphs, sections and ranges that act as procedures, a visitor *PerformedLabels* is used to collect all performed labels and ranges. A second visitor *ConstructProcedures* then uses these to find the corresponding paragraphs or sections and to add procedure nodes to the graph. For ranges, the corresponding *list* of paragraphs or sections is collected.

After the procedure nodes are created, the *RefineProcedure* visitor is applied, in order to extend the graph with the conditionals and outgoing call edges of this procedure.

4.4. Graph Analysis

Once the CCG graph is constructed, it can be analyzed. For this, we use a number of visitors that operate on CCG graphs.

To visualize a CCG graph, we traverse it with a visitor that emits input for the graph-drawing back-end `dot`. This visitor is layered, as is the CCG class hierarchy on which it operates.

To compute metrics per procedure we have devised a number of collaborating visitors, shown in Figure 9. Most of these metrics are based on a *SuccessCounter(v)*, which, when visited, applies its argument *v* and increments a counter if this application was successful. An example application is the *McCabeIndex* combinator, which takes a visitor recognizing if-statements, and then counts the number of successes. Observe

Name		Description
SuccessCounter	*v*	Add one if *v* succeeds
CpfIfRecognizer		Succeed on CPF conditions
CcgIfRecognizer		Succeed on CCG conditions
...		Other recognizers
McCabeIndex	*i*	*SuccessCounter(i), i* an IfRecognizer
FanOut	*p*	*SuccessCounter(p), p* PerformRecogn.
GotoCounter	*g*	*SuccessCounter(g), g* GotoRecognizer
MaxNesting	*v*	Maximum nesting level of *v*-Recognizer
MaxNestedIf	*i*	*MaxNesting(i), i* an IfRecognizer

Figure 9. Selected Metrics Visitors

```
public class PerformedLabels extends cpf.Fwd {
 Set performedLabels = ...;
 Set performedRanges = ...;
 public PerformedLabels() {
  super( new Identity());
 }
 public void visit_perform(perform p) {
  performedLabels.add(p.getcallee());
 }
 public void visit_thru(thru x) {
  performedRanges.add(
   new Pair(x.getstartlabel(), x.getendlabel()));
}}
```

Figure 10. Collect performed labels.

that these metrics combinators are parameterized by recognizers: hence they can be applied to both the CPF and the CCG source models.

In a similar way we construct visitors for recognizing coding standards. For example, a visitor *MixedStyle* operates on the CCG format, and recognizes all call edges from *section* to *paragraph* or vice versa. Such edges indicate a mixed style, and usually are forbidden by coding standards.

5. ControlCruiser Implementation

In this section we discuss some of ControlCruiser's visitors in full detail. Due to space limitations, we limit ourselves to the visitors dealing with graph construction and visualization.

Collect performed labels Recall that perform statements come in two flavors: with and without *thru* clause. Consequently, we need to collect both individual labels, and pairs of labels. For this purpose we use a visitor combinator `PerformedLabels` with two collections in its state (see Figure 10). Note that there are no dependencies between the code in this visitor pertaining to pairs of labels and the code pertaining to individual labels. If desired, we could refactor this visitor into two even smaller separate ones, and re-join them with `Sequence` (visitor extraction).

To actually collect the labels from the input program `p`, we need to create the visitor, pass it to the generic `TopDown` combinator, and visit the tree with it:

```
public class CreateProcedures extends cpf.Fwd {
 CallGraph callGraph;
 Set performedLabels;
 public CreateProcedures(CallGraph g, Set labs){
  super(new Identity());
  ...
 }
 public void visit_section(section s) {
  addProc(s.getlabel(), s);
 }
 public void visit_para(para p) {
  addProc(p.getlabel(), p)
 }
 void addProc(String name, Visitable v) {
  if (performedLabels.contains(name)) {
   Procedure p = new Procedure(name,v);
   callGraph.addProcedure(p);
}}}
```

Figure 11. Create procedures for individual labels.

```
PerformedLabels pl = new PerformedLabels();
( new TopDown(pl) ).visit(p);
```

After the traversal has completed, we can obtain the performed labels and ranges via the instance variables of `pl`.

Paragraphs and Sections Every performed label corresponds to either a section or a paragraph. In order to create a procedure node with the proper link back to the CPF tree representing the procedure body, we use a visitor that triggers at individual sections and paragraphs (see Figure 11). It only actually creates a procedure node if the given label is one of the performed labels, which it receives at construction time. The created procedure nodes are added to a call graph, which is also provided at construction time. To ensure we will be able to retrieve the added nodes at a later stage, we assume they become direct *children* of the graph.

Again, this visitor can be passed to the TopDown combinator, in order to traverse the tree and collect the procedures. Below, however, we will see how we can make better use of combinators in order to avoid visiting too many nodes.

Ranges To construct procedure nodes for a pair of (start and end) labels, we collect those section or paragraph nodes that lie between those labels. For this purpose we have developed an auxiliary visitor (see Figure 12) which takes the start and end labels, and is triggered at each section or paragraph. If the start or end label is encountered, a boolean flag is switched, and paragraphs or sections visited are added to the list.

Given this auxiliary visitor, a visitor can be developed that constructs procedure nodes for pairs of labels (see Figure 13). This visitor triggers at ParagraphList and SectionList nodes. This is appropriate, because the sections and paragraphs spanned by a pair of labels must always occur in the same list. When such a list is encountered, the method `addSpannedASTs` is invoked to perform an iteration over the collection of label pairs. At each iteration, the `All` combinator is used to fire the auxiliary visitor `SpannedASTs` sequentially at all members of the current paragraph or section

```
public class SpannedASTs extends cpf.Fwd {
 VisitableList spannedASTs = new VisitableList();
 String startLabel;
 String endLabel;
 boolean withinRange = false;
 public SpannedASTs(String start, String end) {
  super(new Identity());
  ...
 }
 public void visit_para(para p) {
  addIfWithinRange(p.getlabel(), p);
 }
 public void visit_section(section s) {
  addIfWithinRange(s.getlabel(), s);
 }
 void addIfWithinRange(String label,
                       Visitable x) {
  if (label.equals(startLabel)) {
    withinRange = true; }
  if (withinRange) {
    spannedASTs.add(x); }
  if (label.equals(endLabel)) {
    withinRange = false;
}}}
```

Figure 12. Collect section and paragraph nodes spanned by a given pair of labels.

list. If this yields a non-empty result, a new procedure node is created and added to the graph.

Top Down *While* Finally, we can apply the developed visitors to the input program. This could be done with a simple top-down traversal. However, any nodes at the block level and lower would be visited superfluously, because our visitors have effect only on sections, paragraphs, and lists of these. To gain efficiency, we will use the `TopDownWhile` combinator instead. To detect blocks, we first define the following visitor (using an anonymous class):

```
Visitor isBlock
  = new Fwd(new Fail())
         { public void visit_block(block x) {} };
```

This visitor fails for all nodes, except blocks. We compose it with our procedure creation visitors to do a partial traversal:

```
graph = new CallGraph();
cp    = new CreateProcedures(graph,labels);
cr    = new CreateRanges(graph,ranges);
(new TopDownWhile(
       new IfThenElse(isBlock,
                      new Fail(),
                      new Sequence(cp,cr))
     ) ).visit(p);
```

Thus, at each node the `IfThenElse` combinator is used to determine whether a block is reached and the traversal should stop, or the visitors for procedure creation should be applied. Note that these two separate visitors are combined into one with the `Sequence` combinator. After this traversal, the graph `g` contains a node for every procedure reconstructed from the CPF tree. Each such procedure node contains a reference to the CPF subtrees that gave rise to it.

```
public class CreateRanges extends cpf.Fwd {
  CallGraph callGraph;
  Set todoRanges;
  public CreateRanges(CallGraph g, Set todo) {
    super(new Identity());
    ...
  }
  public void visit_ParaList(ParaList pl) {
   addSpannedASTs(pl);
  }
  public void visit_SectionList(SectionList sl) {
    addSpannedASTs(sl);
  }
  void addSpannedASTs(Visitable list) {
    Iterator pairs = todoRanges.iterator();
    while (pairs.hasNext()) {
      Pair pair = (Pair) pairs.next();
      VisitableList asts = getASTs(pair, list);
      if (! asts.isEmpty()) {
        addProc(pair.start, pair.end, asts);
  } } }
  VisitableList getASTs(Pair p, Visitable list) {
    SpannedASTs sa=new SpannedASTs(p.start, p.end);
    (new GuaranteeSuccess(new All(sa))).visit(list);
    return sa.spannedASTs;
  }
  void addProc(Pair p, VisitableList ast) {
  ...
} }
```

Figure 13. Create procedure for ranges

Construct program entry point We will not show the visitors for constructing the program entry point. They are similar to the creation of performed procedure nodes. An auxiliary visitor collects ASTs, starting from the top of the program, and stopping at the first `STOP RUN` statement or the first performed label. This implements the heuristic that performed sections and paragraphs are never part of the main procedure.

5.1. CCG Refinement

Now we have created the CCG's procedure nodes, we need to refine them by creating nodes that represent the conditions that occur in their bodies, and by adding nesting and call relations between the nodes. For these tasks, we have developed the `RefineProcedure` visitor (see Figure 14). For a given procedure node in the CCG, this visitor is used to create nodes and edges for the conditionals and performs contained in its AST.

For a perform or a perform-thru statement, it adds a call edge from the `caller` to the procedure node that corresponds to its label (pair).

For if statements, it first creates a new conditional node and adds a nesting edge from the `callee` to this new conditional node. It then *restarts* itself with two new starting points: one for the then branch, and another for the else branch. The restart invokes the `TopDownUntil` combinator to traverse these branches. Such restarts are a general mechanism that can be used when stack-like behavior is needed, for example

```
public class RefineProcedure extends cpf.Fwd {
 CallGraph graph;
 ProgramPoint caller;
 public RefineProcedure(CallGraph g,
                        ProgramPoint c) {
  super(new Fail());
  ...
 }
 public void visit_perform(perform perform) {
  String label = perform.getcallee();
  Procedure callee = graph.getProcedure(label);
  caller.addCallEdgeTo(callee);
 }
 public void visit_thru(thru x) {
  String s = x.getstartlabel();
  String e = x.getendlabel();
  Procedure callee = graph.getProcedure(s,e);
  caller.addCallEdgeTo(callee );
 }
 public void visit_if$(if$ x) {
  Conditional cond = graph.addConditional(x);
  caller.addNestingEdgeTo(cond);
  start(graph, cond.getThenPart());
  start(graph, cond.getElsePart());
 }
 public static void start(CallGraph graph,
                          ProgramPoint caller) {
  Visitable ast = caller.getAst();
  RefineProcedure rp
    = new RefineProcedure(graph, caller);
  (new GuaranteeSuccess(
           new TopDownUntil(rp))) . visit(ast);
}}}
```

Figure 14. Refine the CCG for a given procedure.

```
public class Visited implements Visitor {
  Set visited = new HashSet();
  public void  visit(Visitable x)
   throws VisitFailure {
    if (!visited.contains(x)) {
      visited.add(x);
      throw new VisitFailure();
} } }
```

Figure 15. The *Visited* combinator.

when dealing with nested constructs such as if statements.

We need to traverse the initial CCG to actually apply the `RefineProcedure` visitor at each procedure node. To prevent visiting nodes more than once and running in circles, we use the visitor `Visited` from JJTraveler's library (See Figure 15). This generic combinator keeps track of nodes already visited in its state. Now, to traverse the graph, we do a top-down traversal where each node that has not been visited yet is refined:

```
Visitor refine = new ccg.Fwd(new Identity()){
  public void visitProcedure(Procedure p) {
    RefineProcedure.start(graph, p);
  } };
(new TopDownWhile(
  new IfThenElse(new Visited(),
                 new Fail(),
                 refine)
```

```
public class GraphToDot extends graph.Fwd {
 Set dotStatements = new TreeSet();
 public GraphToDot() {
  super(new Identity());
 }
 public void visitNode(GraphNode n) {
  add(n+";")
 }
 public void visitEdge(DirectedEdge e) {
  add(e.inNode()+"->"+e.outNode()+";");
 }
 void add (String dotStatement) { ...  }
 public void printDotFile(String fname) {...}
}
```

Figure 16. Graph visualization.

```
public class CCGToDot extends ccg.Fwd {
 GraphToDot printer;
 public CCGToDot() {
  super(new GraphToDot());
  printer = (GraphToDot) fwd;
 }
 public void visitCall(Call c) {
  add(e.inNode()+"->"+e.outNode()
      +"[style=bold,color=blue];")
 }
 void add(String dotStatement) {
  printer.add(dotStatement);
 }
 public void printDotFile(String fname) {
  printer.printDotFile(fname);
}}
```

Figure 17. CCG visualization.

```
 ) ).visit( graph );
```

Note that we use an anonymous extension of the `Identity` visitor to invoke the `start()` method of the `RefineProcedure` visitor that does the actual refinement.

5.2. CCG visualization

The layered class hierarchy for graph representation allows us to implement a layered visualization visitor as well.

Visualizing generic graphs The visitor `GraphToDot` implements the construction of a representation in the `dot` input format for a given generic graph (see Figure 16). This visitor simply collects a set of `dot` statements, where an appropriate statement is added for each node and edge. After application of this visitor to each node and edge in a graph, the `printDotFile` method can be used to print the collected statements to a file.

Visualizing CCGs For our CCGs, the generic graph visualization does not suffice, because we want to generate different visual clues, for instance for call edges. For this purpose, we implemented `CCGToDot` (see Figure 17). Note that this visitor forwards to a generic `GraphToDot` visitor for all CCG elements but call edges. For these, the redefined visit method generates an adapted dot statement.

The visualization visitors are applied to the CCG in the exact same fashion as the `refine` visitor above. This calls for a refactoring of this traversal strategy into a reusable `GraphTopDown` combinator (extract strategy). We have added this combinator to JJTraveler's library.

6. Discussion

During the development of ControlCruiser we have learned many practical lessons about the use of visitor combinators for constructing program understanding tools. In this section we summarize some development techniques we have adopted and evaluate the benefits and risks of visitor combinator programming.

6.1. Development techniques

Separation of concerns Visitor combinators allow one to implement conceptually separable concerns in different modules, whilst otherwise they would be entangled in a single code fragment. As a result, these concerns can be understood, developed, tested, and maintained separately. Examples of (categories of) concerns we encountered include traversal, control, state, and testing (see below). Throughout all these concerns, we found it natural and beneficial to separate application-specifics from generics.

Testing and benchmarking We developed ControlCruiser following the extreme programming maxim of *test-first* design, which involves writing *unit tests* for every piece of code that can potentially fail. As a result, we wanted to test not only the compound visitors that are invoked by the application, but also each individual visitor combinator from which such compound visitors are composed.

To this end, we developed a testing combinator `LogVisitor`, which logs every invocation of its argument visitor into a special `Logger`. In combination with the standard unit testing utility JUnit, this testing combinator can be used to write detailed tests for hierarchy-specific visitors. To test the generic visitors of JJTraveler itself, we used a mock instantiation of JJTraveler's framework (with a single visitable class).

For detailed benchmarking, we needed to collect timing results, again not just on compound visitors, but also on individual visitor combinators. To this end, we created a specialization `TimeLogVisitor` of our testing combinator that measures and aggregates the activity bursts of its argument visitor. This enables us to separately measure the time consumed by different concerns, such as traversal and node action.

Failure containment When using visitor combinators that potentially fail, one needs to declare the `VisitFailure` exception in a `throws` clause. In many cases, the programmer knows from the context that such failure can actually never occur. Examples are the expressions $Try(Fail)$ and $TopDownWhile(Fail)$. To relieve the programmer from the burden of writing catch-throws contexts to contain such 'impossible' failures, we developed the combinator `GuaranteeSuccess`. Judicious placement of this combinator reduces code cluttering and makes code more self-documenting.

Class organization We have used several kinds of inner classes to improve code organization. For tiny visitors (no more than a few lines) we have used *anonymous* classes. For small visitors (no more than a few methods) that operate within the context of another visitor (i.e. using its state), we used *member* classes. This removes the need for additional instance variables and constructor method arguments.

6.2. Evaluation

Benefits Visitor combinators enable separation of concerns. This helps understanding, development, testing, and reuse. Combinators enable reuse in several dimensions. Within an application, a single concern, such as a particular traversal strategy or applicability condition, needs to be implemented only once in a reusable combinator. Across applications, visitors can be reused that capture generic behavior. Examples are the fully generic combinators of the JJTraveler library, but also the `DotPrinter` combinator that can be refined by any application that uses or even specializes the `graph` package on which this combinator operates.

A related benefit is robustness against class-hierarchy changes. Using visitor combinators, each concern can be implemented with explicit reference only to classes that are relevant to it. As a result, changes in other classes will not unduly affect the implementation of the concern.

In relation to other approaches to separation of concerns and object traversal, visitor combinators are extremely lightweight. Optionally, the JJForester tool can be used to instantiate JJTraveler's framework. However, visitor combinators do not essentially rely on tools. The required implementation of the (very thin) `Visitable` interface and the `Fwd` combinator is straightforward, and can easily be done by hand.

Risks Visitor combinators pose two risks with respect to performance. Firstly, the development of many little visitors may lead to many (relatively expensive) object creations. One should take care to keep these within reasonable limit. For instance, stateless combinators need only be created once. Stateful visitors can often be re-initialized to run again, instead of continually creating new ones.

Another performance penalty may come from heavy reliance on exceptions for steering visitor control. One should take care to choose the interpretation of `VisitFailure` such that failure is less common than success. E.g. one can use `TopDownWhile` with `Identity` as default, instead of `TopDownUntil` with `Fail` as default.

These performance risks can be combatted by profiling (maybe using `TimeLogVisitor`) and refactoring. Refactoring rules for combinators can often be described with simple equations. However, when we applied ControlCruiser to our code bases, including a 3,000,000 loc system, we did not experience performance problems. (in fact, the majority of the time was spent on parsing the CPF format, not on running the visitors on them).

7. Concluding Remarks

Related work We refer to [14] for a full account of related work in the areas of design patterns and object navigation approaches: of particular interest are the *extended* [7] and *staggered* [15] visitor patterns, and adaptive programming [11] for expressing "roadmaps" through object structures. The origins of visitor combinators can furthermore be traced back to *strategic term rewriting*, in particular [13].

Traversals in the context of reverse engineering tools are discussed by [3], who provide a top-down analysis or transformation traversal. Their traversals have been generalized in the context of ASF+SDF in [1]. Similar traversals are present in the Refine toolset [12], which contains a pre-order and post-order traversal. In both cases, only a few traversal strategies are provided, and little support is available for composing complex traversals from basic building blocks or controlling the visiting behavior.

In the field of program understanding and reengineering tools *exchange formats* have attracted considerable attention since 1998 [16]. Visitor combinators provide an interesting perspective on such formats. Instead of focusing on the underlying structure, visitor combinators make assumptions on what they can observe in a structure. By minimizing these assumptions, for example by trying to use the generic *Visitable* interface, the reusability of these combinators is maximized.

One of the outcomes of the exchange format research is the Graph Exchange Language GXL [9]. Visitor combinators are likely to be a suitable mechanism for processing GXL representations. This requires generating directed graph structures that implement the *Visitable* interface from GXL schema's, similar to the way JJForester generates visitable trees from context free grammars and to the way our graph package implements the visitable interface.

Contributions We have demonstrated that visitor combinators provide a powerful programming technique for processing source models. We have given concrete examples of instantiating the visitor combinator framework provided by JJTraveler, and of developing complex program understanding visitors by specialization and combination of JJTraveler's combinator library. We have applied the developed visitors to a large code base to establish feasibility and scalability of the approach. Finally, we have summarized the development techniques surrounding visitor combinator programming and we have made an assessment of the risks and benefits involved.

Availability JJTraveler, JJForester, and ControlCruiser can be downloaded from `www.jjforester.org/`.

Acknowledgements We would like to thank Huub de Hesselle, Paul Klint and Leon Moonen for reading drafts of our paper.

References

[1] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, 2001.

[2] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[3] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Sc. of Comp. Progr.*, 36(2–3), 2000.

[4] A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *International Workshop on Program Comprehension*, pages 90–97. IEEE, 1998.

[5] A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.

[6] J. Field and G. Ramalingam. Identifying procedural structure in cobol programs. In *Workshop on Program analysis for software tools and engineering; PASTE*, pages 1–10. ACM Press, 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] R. Holt, A. Winter, and A. Schürr. GXL: Toward a standard exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 162–171. IEEE Computer Society, 2000.

[10] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Electronic Notes in Theoretical Computer Science*, 44(2), 2001. Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA).

[11] 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.

[12] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Comm. of the ACM*, 37(5):58–70, 1994.

[13] E. Visser, Z. 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).

[14] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings.

[15] John Vlissides. Visitor in frameworks. *C++ Report*, 11(10), November 1999.

[16] S. Woods, L. O'Brien, T. Lin, K. Gallagher, and A. Quilici. An architecture for interoperable program understanding tools. In *6th International Workshop on Program Comprehension (IWPC)*, pages 54–63. IEEE, 1998.