Source Model Analysis using the JJTraveler Visitor Combinator Framework

Arie van Deursen[*,‡]

Joost Visser[¶]

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

SUMMARY

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.

KEY WORDS: Program analysis, program comprehension, visitor design pattern, software visualization.

1. Introduction

Source Models Many reverse engineering, program understanding and reengineering tools operate by constructing so-called source models from the program source text, followed by an appropriate

[*]Correspondence to: CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

[†]This is a substantially revised version of our earlier paper: A. van Deursen and J. Visser. Building Program Understanding Tools Using Visitor Combinators. In *Proceedings 10th International Workshop on Program Comprehension (IWPC'02)*, pages 137-146, IEEE Computer Society, 2002.

[‡]E-mail: Arie.van.Deursen@cwi.nl. Also affiliated with Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Software Evolution Research Laboratory (SWERL), Mekelweg 4, 2628 CD Delft, The Netherlands

[¶]Departamento de Informática, Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal. E-mail: Joost.Visser@di.uminho.pt analysis of these models. Different source models can vary in their abstraction level, involving, for example, a representation relatively close to an abstract syntax tree, or, alternatively, consisting of high-level architectural structures such as package containment and module dependencies. Depending on the analysis problem, these source models can be represented by tables, trees, or graphs. Typically, 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.

Object-Oriented Source Model Analysis Objects form a natural way for representing such source models, offering appropriate abstraction mechanisms, classes for organizing model elements, and object manipulation and navigation for operating on the model. In order to implement a range of operations on an object-oriented source model, the Visitor design pattern can be used. 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" [1]. 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 it gets accepted by a given syntax tree.

Visitors are useful for analysis and manipulation 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 make it possible to add new forms of analysis easily, without modifying the class hierarchy representing node types. The implementation of these different analysis can be isolated in individual classes, rather than being scattered over the various node types.

Visitor Combinators Recently, visitor *combinators* have been proposed as an extension of the regular visitor design pattern [2]. These visitor combinators offer an explicit separation between traversal (object navigation) strategies, and the actual operations performed on each object. 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 the theoretical grounds of *strategic programming*, and it promises to be a powerful implementation technique for processing source models in the context of program analysis and understanding. Now this concept needs to be put to the test of practice.

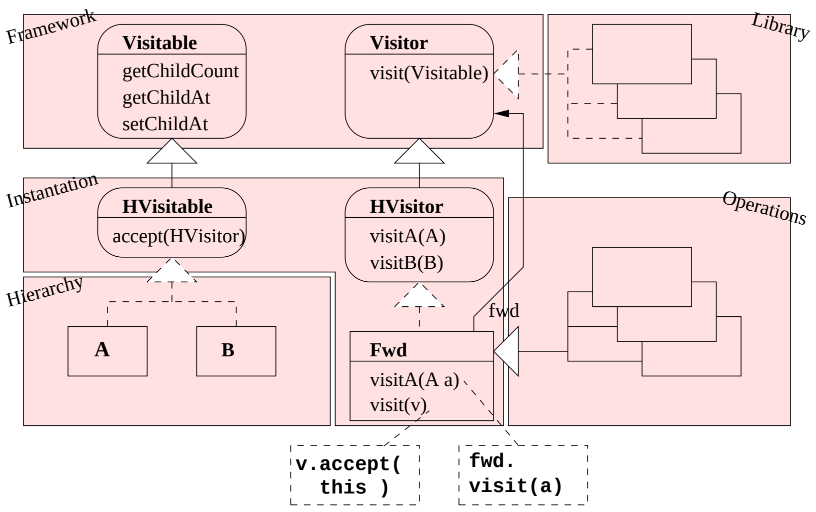


Figure 1. The architecture of JJTraveler. Rounded boxes indicate interfaces, square boxes are classes. Inheritance is indicated by lines with triangular connectors. These are dashed if the inheritance relation is implementation of an interface rather than specialization of a class. Dashed boxes indicate implementation notes.

To that end, 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 construct and manipulate source models as used in program understanding and reverse engineering 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 [2] and is supported by JJTraveler: a combination of a framework and library that provides generic visitor combinators for Java. This section provides a completely updated account of JJTraveler, and discusses several library extensions that were made during the development of ControlCruiser.

2.1. The JJTraveler Architecture

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.

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	Do v_1, if it fails, do v_2
All	v	Apply v sequentially to all immediate children until it fails
One	v	Apply v sequentially to all immediate children until it succeeds
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).

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.

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.

To use JJTraveler, one needs to instantiate the framework for the class hierarchy of a particular application. This first of all requires specializing the visitor and visitable interfaces to hierarchy-specific ones, called *HVisitor* and *HVisitable* in Figure 1. In particular, the *HVisitor* interface contains distinct visit methods for each class in the hierarchy.

Secondly, a default 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 – typically *Identity* if for most nodes nothing needs to be done – and overriding some of the specific *Fwd* methods to obtain the required behavior for selected node types.

Finally, the class-hierarchy must be made visitable. To turn a class into a visitable class, it must implement the hierarchy-specific *HVisitable* interface. In addition to the generic visitable methods, this interface provides an `accept` method, which calls the appropriate visit method in the hierarchy-specific *HVisitor*. The `accept` method realizes the so-called double-dispatch functionality of the Visitor pattern: it selects a visit method to be executed, based both on the visitor object and the object being visited.

For more details about instantiation of the JJTraveler framework we refer the reader to Appendix 9 where the full Java code is provided for an example instantiation involving a toy hierarchy.

```
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) throws VisitFailure {
    v1.visit(x);
    v2.visit(x);
} }
```

Figure 3. The *Sequence* combinator.

Though instantiation of JJTraveler's framework can be done manually, automated support for this is provided by a generator, called JJForester [3]. 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 *HVisitor* and *HVisitable* interfaces, and the *Fwd* combinator. In addition to framework instantiation, JJForester provides connectivity to a generalized LR parser [4].

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 whether all 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 *IsA.*`visit`*(myB)* (which fails).

2.2. A library of generic visitor combinators

Figure 2 shows high-level descriptions for an excerpt of JJTraveler'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 provide the primitive buliding blocks for visitor combination. They include unary combinators *Identity* and *Fail*, as well as binary operators such as *Sequence* and *Choice*. The full implementation for *Sequence* is shown in Figure 3; the body of the `visit` method for selected other basic combinators is shown in Figure 4.

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.

Identity()	`; // skip`
Fail()	`throw new VisitFailure();`
Sequence(v1, v2)	`v1.visit(x); v2.visit(x);`

Choice(v1, v2)	`try { v1.visit(x); }`
	`catch (VisitFailure vf) {`
	`  v2.visit(x);`
	`}`

All(v)	`for (int i=0; i<x.getChildCount(); i++) {`
	`  x.getChildAt(i).visit(v);`
	`}`

Figure 4. Implementations for Selected Basic Visitors

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

Figure 5. The *Try* combinator.

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.

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 (see Figure 2). 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.

An example of a recursively defined visitor is *TopDown(v)*, which in Figure 2 is defined as

$$TopDown(v) = Sequence(v, All(TopDown(v)))$$

Thus, *TopDown* first applies *v* to the current node, and then recursively applies the top down strategy to each of the children of the current node, yielding a depth-first traversal of a tree visited.

Visitor combinators can be used to build recursive visitors with all sorts of sophisticated traversal behavior. As an example, during our work on ControlCruiser, we encountered the need for a variant of

```
public class DoWhileSuccess implements Visitor {
  private Visitor dws;
  public DoWhileSuccess(Visitor cond,
                        Visitor action,
                        Visitor atBorder) {
    dws = new IfThenElse( cond,
                          new Sequence(action, new All(this)),
                          atBorder );
  }
  public void visit(Visitable x) throws VisitFailure {
    dws.visit(x);
} }
```

Figure 6. The *DoWhileSuccess* combinator.

top down which only traverses downwards as long as a the nodes visited meet certain criteria. Thus, we extended JJTravelers library with the generic visitor *DoWhileSuccess*, defined as

DoWhileSuccess(cond, action, atBorder) =
    IfThenElse(cond,
        Sequence(action, *All*(*DoWhileSuccess*(cond, action, atBorder))),
        atBorder)

The first argument of this visitor is a condition, the second the action to be performed as long as the condition holds, and the third the action to be performed on the border nodes at which the traversal stops going downward. This behavior is recursively repeated top down for all nodes, as long as the condition holds, as expressed in the *Sequence* expression.

The encoding in Java of this visitor is shown in Figure 6. It makes use of an instance variable to store the combinator expression, which includes the recursion via the reference to `this`. Alternatively, the inheritance guideline discussed above can be adopted, in which case *DoWhileSuccess* extends *IfThenElse*. This requires an extra method to circumvent the fact that Java cannot access `this` in calls to the `super` constructor.

Given this visitor, we can define the following convenient shorthands which we will use later in the paper:

DoWhileSuccess(c, a)	=	*DoWhileSuccess*(c, a, *Identity*)
TopDownWhile(c, b)	=	*DoWhileSuccess*(c, *Identity*, b)
TopDownUntil(c, b)	=	*DoWhileSuccess*(*Not*(c), *Identity*, b)
TopDown(a)	=	*DoWhileSuccess*(*Identity*, a, *Identity*)

Observe that the notion of failure can be used in various ways. In the examples above, failure is used as a Boolean flag, failure meaning false and success meaning true. Another use of fail is for the termination of a full traversal. In the example above this would happen if the *action* would fail at some point. If one wants to be sure that the action doesn't fail, it can be surrounded by a *Try* combinator — which never fails. Observe that the fact that failure is implemented as an exception is hidden in the basic combinators. Thus, defined combinators need not consist of *try ... catch* clauses, but can be composed from the basic combinators.

2.3. Extensions

The library of JJTraveler is still evolving. Any combinator that can be expressed just in terms of the generic Visitable and Visitor interfaces, is a candiate for inclusion in the JJTraveler library. During the development of ControlCruiser, we constructed a number combinators that are now part of the library. One example is the *DoWhileSuccess* combinator discussed above.

Another combinator we added encapsulates the *VisitFailure* exception. This exception plays an essential role to control traversal behavior. In many cases, however, we can predict that such an exception will never escape the outermost visit method call. For example, the expression *Not(Fail)* will never fail, nor will *TopDown(Identity)*. To document such cases, and in order to avoid unnecessary try-catch statements and throw-declarations, we introduced the *GuaranteeSuccess* combinator. This combinator catches any VisitFailure exception, and turns it into a Java `RunTimeException`, which should never occur and hence need not be declared. Judicious placement of this combinator reduces code cluttering and makes code more self-documenting.

Furthermore, we extended JJTraveler with a test sub-package, providing support for testing generic visitor combinators. This package includes the simplest possible instantation of the generic visitable interface, a *LogVisitor(v)* combinator which maintains a trace of all nodes visited by *v*, and an extension of the JUnit *TestCase* class (see [5]) in order to provide dedicated support for constructing, running, tracing, and checking visitor combinators. This makes it possible to adopt a systematic unit testing approach when developing generic visitor combinators.

3. Cobol Control Flow

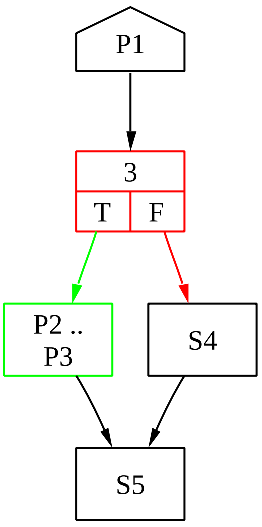
The example we use to study the application of visitor combinators to the construction of program understanding tools deals with Cobol control flow. Cobol has some special control-flow features, making analysis and visualization an interesting and non-trivial task. The analysis we describe is taken from DocGen (see [6]), 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.

```
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 conditional call graph. Note that the conditional node shows that the condition occurs at line number 3.

Figure 7. Example Cobol source and graph

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 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 7(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`. After performing each section, paragraph, or range of them, control is returned to the statement after the perform that invoked them. In the example, this means that control will finally return to the `STOP RUN` statement at the end of paragraph `P1`. 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.

Typically, 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 an undecidable problem. In this paper, we will assume that ranges are syntactically sequenced, which corresponds to the way Cobol programmers generally understand and apply this construct, and which has grown into a widely accepted Cobol programming convention. If this assumption does not hold, paragraph P_n will be reached from P_1 in another way than via fall through. In that case, our approach will infer a procedure that is too large (containing too many paragraphs). Our approach could be easily extended to issue warnings for some (though not all) situations in which this can happen. We refer to [7] for ways of dealing with dynamic ranges. We conjecture that visitor combinators can applied successfully to implement the algorithms described by [7].

3.2. Analysis and visualization

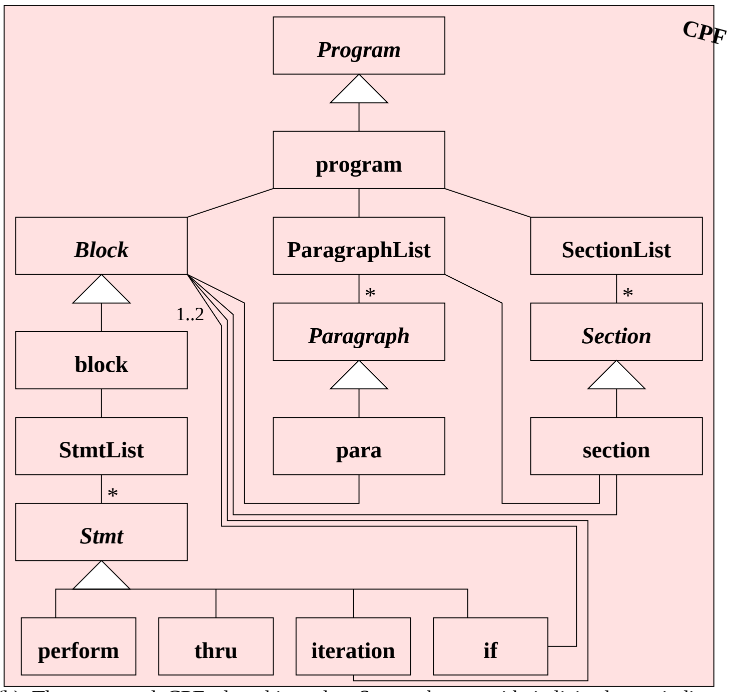
To help maintenance programmers understand the control flow of individual Cobol programs, a tool is needed for analysis and visualization of a program's perform dependencies. From such a call graph, one could instantly glean which perform style is predominant, which sections, paragraphs or ranges make 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 Figures 7(b) and 17. These graphs contain nodes for procedures and conditionals (if-then-else, iteration, and case statements), which are connected by edges that represent call relations and nesting relations. Return of control is not modeled by an explicit edge, but subsumed in the edge that represents a call. In essence, a CCG summarizes the control-dependencies of procedure calls. The visualization of these graphs is such that only control statements that affect calls are shown, leading to a compact representation suitable for large Cobol programs as well. The details of the CCG format will be presented in Section 4.3. 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 [6].

Conditional call graphs can also be used as part of a systematic quality assurance (QA) effort, for example for computing quality related metrics at the system, program, and procedure level. Example QA metrics include McCabe's cyclomatic complexity, fan-in, fan-out, and the deepest conditional nesting level. Moreover, they can be used to detect certain coding style violations, such as the use of goto's across section boundaries, or the mixed use of both sections and paragraphs as perform target — both of which are forbidden in current Cobol programming methodologies.

```
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
  END-PARA 14 P5
END-SECTION 14 S4
SECTION 14 S5
END-SECTION 15 S5
```

(a) CPF for Fig 7



(b) The generated CPF class hierarchy. Square boxes with italicized text indicate abstract classes.

Figure 8. Conditional Perform Format (CPF)

4. Visitable Program Representations

We have used visitor combinators to implement the analysis and visualization requirements just described. The result is ControlCruiser, a program analysis tool that can provide 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 visitable program representations used in ControlCruiser; the next sections covers in detail how visitor combinators have been used for the purpose of graph construction and analysis.

4.1. Initial Tree Representation

The starting point for ControlCruiser is a simple language containing just the statements representing Cobol sections, paragraphs, perform statements, and conditional constructs. An example of this Conditional Perform Format (CPF) is shown in Figure 8(a). This representation can be thought of as a simplified Cobol abstract syntax tree, stripped to include only statements relevant for the intra-program control flow.

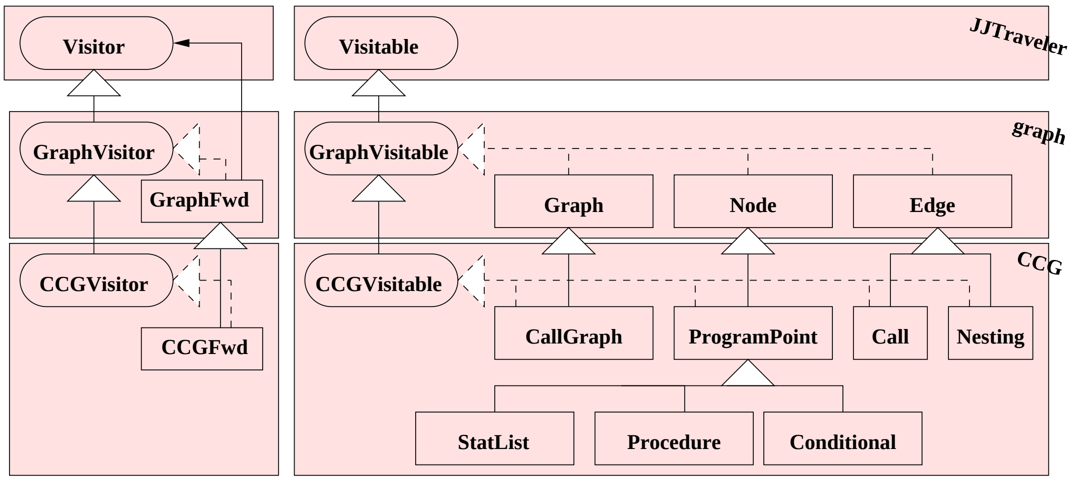


Figure 9. Class hierarchy for graph representations.

We obtain CPF from Cobol sources using a Perl script written according to the principles discussed in [8]. This script takes care of handling the tricky details of the Cobol syntax, such as scope termination of nested if- and loop-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 8(b). Since these all realize the *Visitable* interface, we can implement all subsequent steps with visitor combinators.

An alternative way to obtain CPF from Cobol sources would be to employ a full-blown Cobol parser. The syntax tree produced by this parser could be traversed by a visitor to construct the CPF trees. Visitor combinators could be employed to constuct this CPF constructing visitor. The difficulty of this approach is the lack of availability of Cobol parsers, and the wide variety of existing Cobol dialects. The Perl script is tolerant with respect to differences between dialects, but parsers generally need to be explicitly tuned for each dialect. The use of *island grammars* to remediate such intolerance is subject of ongoing investigation [9, 6]. For languages that are more rigorously standardized, this alternative avenue for obtaining CPF is not problemetic, and we have actually done so for extraction of CPF-like structures from Java.

4.2. Visitable Graphs

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 9).

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 the hierarchy-specific *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 child of an edge is its outgoing node,
- and the children of a graph consists of the collection of all nodes.

This makes it possible to traverse a graph using visitor combinators. Furthermore, a hierarchy-specific *GraphVisitor* interface (not shown) is provided, as well as the default forwarding implementation *GraphFwd*. These can be used to implement various general graph algorithms, such as determining root nodes, connectivity, etc.

4.3. Conditional Call Graphs

The generic graph layer is specialized to represent control and call dependencies giving rise to Conditional Call Graphs (CCGs). The nodes in a CCG represent procedures, statement lists, and control statements, as illustrated in Figure 9. Procedures and statement lists both can have any number of outgoing edges, which can be of two types. Call edges represent a call from a statement list to a procedure. Nesting edges represent a syntactic containment between a statement list and a control statement. Control statements represent statements that transfer control, such as if-then-else, case, and iteration. These statements have a fixed number of outgoing nesting edges to statement lists, one for each branch.

Procedures are statement lists that can be retrieved by name. All nodes are program points, and can have a pointer back to their originating construct in the CPF tree.

The forwarding combinator of CCG (not shown) provides the default behavior of a CCG visitor. This involves three levels of forwarding, capturing the interplay between the various representations, starting at the CCG format and ending at the generic visitable perspective.

1. First, visit methods of classes that have a superclass within the visitable hierarchy invoke the visit method of their immediate superclass.
2. Second, visit methods for top-level CCG classes forward to visit methods in a *GraphVisitor* at the generic graph level.
3. Finally, graph-specific visitors forward to their generic visitors provided to them at constructor time.

This layered design makes it possible to reuse *GraphVisitors* when building a *CCGVisitor* by forwarding to the graph visitor. This will be illustrated in Section 6.3 where we build a CCG visualizer that delegates default behavior to a graph visualizer.

Observe that conditional call graphs are language-independent: we are using them for Java as well as Cobol analysis. If language-specific issues are to be taken into account an additional bottom layer can be created. As an example, our ControlCruiser implementation currently has a small Cobol-specific layer to cater for the particularities of sections, paragraphs, and ranges.

```
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 Range(x.getstartlabel(), x.getendlabel()) );
  }
  public void apply(Program p) {
    (new GuaranteeSuccess (new TopDown(this))).visit(p);
} }
```

Figure 10. Collect performed labels.

5. Graph Construction

We will first explore how to use visitor combinators for the purpose of constructing the conditional call graph. This phase involves the application of visitor combinators to abstract syntax trees in CPF format. Our treatment covers visitor combinator programming basics (to familiarize the reader with the concepts) as well as more advanced techniques (such as restarting visitors). In the next section we will then explore how visitor combinators can be applied to graph traversals as well.

Observe that the graph construction itself is non-trivial: it involves the *reconstruction* of the procedural structure of the Cobol program. For this, the labels that are actually performed need to be selected and turned into graph nodes. Moreover, conditional structures should be organized in a part-of relationship with those recovered procedures instead of with the existing labels.

5.1. Tree Construction

To parse a CPF file into a corresponding syntax tree with root node of type *Program*, we simply call a factory method `Program.parseFile()` generated by JJForester. All nodes in this tree are of one of the types shown in Figure 8(b). Since these all implement the *Visitable* interface, we can build all subsequent steps with visitor combinators.

5.2. Label Collection

The visitor for collecting labels that are used as target in PERFORM statements is shown in Figure 10. This is a visitor for the CPF hierarchy, and as such it extends the CPF forwarding visitor *cpf.Fwd*. The *PerformedLabels* constructor invokes the *cpf.Fwd* constructor with *Identity* as argument, indicating that by default nothing needs to be done.

```
public class CreateProcedures extends cpf.Fwd {
  CallGraph callGraph;
  Set performedLabels;
  public CreateProcedures(CallGraph g, Set labels){
    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)) {
      callGraph.addProcedure( new Procedure(name,v) );
} } }
```

Figure 11. Create procedures for individual labels.

Recall that perform statements come in two flavors: with and without THRU clause. Consequently, there are two cases for which dedicated behavior is required, corresponding to two method definitions in Figure 10. The method bodies simply update the corresponding collection of labels or ranges.

To actually collect the labels from the input program `p`, we need to pass the *PerformedLabels* visitor to the generic *TopDown* combinator, and visit `p` with it. The method taking care of this is called `apply`. It makes use of the *GuaranteeSuccess* combinator to emphasize that this traversal can never fail. Observe that in principle, the *PerformedLabels* visitor can be combined with any (generic) combinator. We recommend to include an `apply` method for the typical way of invocation when assembling functionality from combinators which is not intended to be reusable in other contexts.

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 ones, *PerformedIndividuals* and *PerformedRanges*, and obtain the combined behavior via

Sequence(PerformedRanges, PerformedIndividuals)

5.3. Procedure Reconstruction

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. Recall from Section 4.2 that the nodes comprise the children of a graph, so that we indeed will be able to retrieve the added nodes at a later stage via a traversal over the graph object.

```
public class SpannedASTs extends cpf.Fwd {
  VisitableList spannedASTs = new VisitableList();
  boolean withinRange = false;
  public SpannedASTs(Range range)
    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(range.start)) { withinRange = true; }
    if (withinRange) {
      spannedASTs.add(x);
      if (label.equals(range.end)) {
        throw new VisitFailure("Range Found");
  } } }
  public SpannedASTs apply(Visitable list) {
    (new GuaranteeSuccess(new Try(new All(this)))).visit(list);
    return this;
} }
```

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

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.

To construct procedure nodes for perform *ranges*, we need to collect the section or paragraph between the start and end label. For this purpose we have developed an auxiliary visitor (see Figure 12) which triggers at each section or paragraph. When the given start label is encountered, paragraphs or sections visited are added to the list. At the end label, a failure is generated to indicate that further traversal is not necessary. This visitor is typically combined with *All*, in order to check all elements of a paragraph or section 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, since it follows from our assumption that ranges are syntacticly recognizable that 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 *SpannedASTs* combinator is activated to collect the corresponding paragraphs or sections. If this yields a non-empty result, a new procedure node is created and added to the graph. If the closing paragraph of a range is not found, we could easily generate a warning, but this has not been implemented.

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,

```
public class CreateRanges extends cpf.Fwd {
  public CreateRanges(CallGraph graph, Set todoRanges) {
    super(new Identity());
    ...
  }
  public void visit_ParaList(ParaList pl) {
    addSpannedASTs(pl);
  }
  public void visit_SectionList(SectionList sl) {
    addSpannedASTs(sl);
  }
  void addSpannedASTs(Visitable list) {
    Iterator ranges = todoRanges.iterator();
    while (ranges.hasNext()) {
      Range range = (Range) ranges.next();
      VisitableList body = getBody(list, range);
      if (! body.isEmpty()) {
        addProc(range, body);
  } } }
  public VisitableList getBody(list, range) {
    return (new SpannedASTs(range)).apply(list).spannedASTs;
  }
  void addProc(Range p, VisitableList ast) {
    ...
} }
```

Figure 13. Create procedure for ranges

because our visitors for procedure creation have effect only on sections, paragraphs, and lists of these. To gain efficiency, we will use the `DoWhileSuccess` 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:

```
cp = new CreateProcedures(graph,labels);
cr = new CreateRanges(graph,ranges);
(new DoWhileSuccess(
        (new Not( isBlock )),
        (new Sequence(cp, cr))
     ) ) . visit(program);
```

Thus, both separate combinators are combined in a sequence, and applied as long as we did not enter a block. After this traversal, the graph 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.

The visitors for constructing the program entry point are similar to the creation of performed procedure nodes and are not shown. An auxiliary visitor collects ASTs, starting from the top of the

```
public class RefineProcedure extends cpf.Fwd {
  public RefineProcedure(CallGraph graph, ProgramPoint caller) {
    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) {
    ...
  }
  public void visit_if$(if$ x) {
    Conditional cond = graph.addConditional(x);
    caller.addNestingEdgeTo(cond);
    restart(cond.getThenPart());
    restart(cond.getElsePart());
  }
  public RefineProcedure apply() {
    Visitable body = caller.getAST();
    (new GuaranteeSuccess(new TopDownUntil(this))).visit(body);
    return this;
  }
  public void restart(ProgramPoint newCaller) {
    (new RefineProcedure(graph, newCaller)).apply();
} }
```

Figure 14. Refine the CCG for a given procedure. Note that the symbol `$` has been appended to `if`, because it is a reserved keyword in Java.

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.

5.4. Constructing Conditionals and Edges

Our next visitor analyzes the various procedure bodies, and constructs the appropriate conditional structures together with call edges to other performed procedures (see Figure 14). This visitor provides interesting behavior in that it needs to construct a recursive structure, which requires a *restarting* visitor at every recursive call.

The *RefineProcedure* visitor itself fires at perform statements and conditional nodes such as if-then-else statements. For perform statements it simply creates a call edge to the performed procedure. For conditional statements, it creates the appropriate node and edge, and then restarts, creating a new visitor incarnation for the true and the false branches in order to find outgoing edges of these branches.

This visitor is intended for combination with *TopDownUntil*, which traverses downward until success. For that reason, *RefineProcedure* fails by default (it inherits from *Fwd(Fail)*): the only nodes that succeed are the perform and conditional nodes, which should stop current top down traversal.

```
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("First visit");
} } }
```

Figure 15. The *Visited* combinator.

The *RefineProcedure* visitor itself must be invoked for each procedure. Since the visitable graph model chosen (see Section 4) ensures that the *All* combinator just visits each node in the graph, this can be achieved via:

```
Visitor refine = new ccg.Fwd(new Identity()){
  public void visitProcedure(Procedure p) {
    (new RefineProcedure(graph, p).apply();
} };
(new All(refine)).visit(graph);
```

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

6. Graph Analysis

In the previous sections, we have seen how to apply traversals over trees in order to construct a graph. In this section, we take this constructed graph as starting point, and use visitors in order to traverse the graph. Our examples are concerned with metrics collection and graph visualization.

6.1. Graph Combinators

From a visitor combinator perspective, graph traversals offer a number of interesting issues. For one thing, graph nodes may have multiple incoming edges. Thus, directly applying the regular *TopDown* combinator to a graph will have the effect that a single node may be visited more than once. Therefore, we introduce the (fully generic) *Visited()* combinator (see Figure 15) which indicates (via success) whether a node has been visited before.

This combinator is also helpful when traversing cyclic graphs. It is typically used in the following context, which applies the action visitor exactly once to every node. Observe that we operate on the generic visitable level, independent of any specific graph representation.

*WhileNotVisited(*action*) = DoWhileSuccess(Not(Visited),* action*)*

6.2. Collecting Metrics

To collect control flow related metrics from our graphs, we can again benefit from a number of fully generic visitors. To illustrate this, we will construct a visitor for computing McCabe's cyclomatic complexity. In the "simplified complexity calculation" as discussed by Watson and McCabe [10], this amounts to counting binary decision predicates.[†] This involves the following visitors:

- *SuccessCounter(v)*
  Whenever *v* succeeds, increment an internal counter.
- *IfRecognizer()*
  Succeed on an CCG conditional statement, and fail on all other statements
- *McCabeCounter() = WhileNotVisited(SuccessCounter(IfRecognizer()))*
  Compute the number of if statements in the graph visited. To obtain a language-independent McCabe counter, we can turn the specific *IfRecognizer* into a parameter.

This *McCabeCounter* computes the cyclomatic complexity for a full graph: the *WhileNotVisited* combinator traverses every edge it encounters. For reporting and quality assurance, we will typically be interested in metrics *per procedure* rather than aggregated for the full program. This involves the following visitors:

- *WithinScope()*
  Fail on call edges, and succeed on the (remaining) nesting edges or conditional nodes, which do not exit the scope of the current procedure.

This combinator can then be used in the following expression, which counts conditionals within the scope of a particular procedure.

```
McCabeCounter() =
    DoWhileSuccess(
        Sequence( Not(Visited), WithinScope() ),
       SuccessCounter(IfRecognizer) )
```

This visitor should then be activated for all procedures in the graph, for example via the *All* combinator.

In a similar manner we can collect the maximum nesting depth of conditional statements. In legacy systems if-then-else structures with a nesting level higher than 10 are not uncommon, and can be a major source of errors. To identify them, we use a fully generic *NestingDepth(r)* visitor, which is parameterized by a visitor *r* to recognize relevant nesting constructs. In our case, *r* will typically be the *IfRecognizer* we encountered before. The *NestingDepth(r)* visitor counts nested occurrences of nodes on which *r* succeeds, and returns the maximum nesting level. Since this involves recursive behavior, this combinator is based on restarts as discussed in Section 5.4. In order to provide distinct counts per procedure, the combinator can be given a second argument indicating when to stop, similar to the way in which the *WithinScope* visitor is used in the McCabe computation.

[†]For ease of exposition, we follow the simplified computation in this paper: the actual implementation of ControlCruiser adopts the complete computation also covering multi-branch statements and loops.

```
public class GraphToDot extends graph.Fwd {
  ...
  public GraphToDot(DotFile dotFile) {
    super(new Identity());
  }
  public void visitNode(GraphNode n) {
    dotFile.add(n+";")
  }
  public void visitEdge(DirectedEdge e) {
    dotFile.add(e.inNode() + "->" + e.outNode() + ";");
  }
  public void apply(Graph g) {
    (new GuaranteeSuccess(new WhileNotVisited(this))).visit(g);
  }
}
```

Figure 16. Graph visualization.

6.3. Graph Visualization

We conclude our presentation ControlCruiser by discussing different ways in which our graphs can be visualized. Again, this introduces interesting new visitor combinator programming techniques.

Visualizing a graph involves a simple mapping from the *Graph* object structures to the textual representation of `dot` that is part of the GraphViz suite offered by AT&T [11]. To facilitate this, we extend *GraphFwd(Identity)*, and redefine the methods `visitNode` and `visitEdge` to emit the appropriate lines to a DotFile object. Combining this visitor with *WhileNotVisited*, and applying it to a *Graph* object yields the dot representation, as shown in Figure 16.

If we want to generate specialized visual clues for certain CCG elements, we can reuse this *GraphToDot* via forwarding. Thus, a *CcgToDot* combinator implements a refinement of *ccg.Fwd* to offer specific visualizations of, for example, procedures and conditional statements. In all other cases, the default behavior of *ccg.Fwd* is used, which is to forward to the *GraphToDot* visitor provided, which takes care of visualizing remaining nodes and edges in a standard way.

An example of the resulting conditional call graph for an existing Cobol program is shown in Figure 17. The EVAL nodes represent case statements (called "evaluate" statements in Cobol), which occur in our full implementation as subclass of the Conditional class. Recall that only perform edges and conditional statements are shown in the graph: this explains why some of the if-then-else statements have no outgoing edges (their then- or else-part probably just consist of assignments, not of calls or extra conditional logic).

For large programs, a graph including all conditionals may be come too cluttered for understanding purposes. Thus, we would like to provide an alternative visualization without the conditions. The visitor combinator programming technique to achieve this is to apply a *filter* on the graph before we visualize it. Such a filter does not reconstruct a graph, but forwards only selected visit events to another *action* visitor. In our case, the required *CallFilter* only forwards visits to procedure nodes and call edges. Since existing call edges can depart from a conditional node instead of a procedural node, the filter

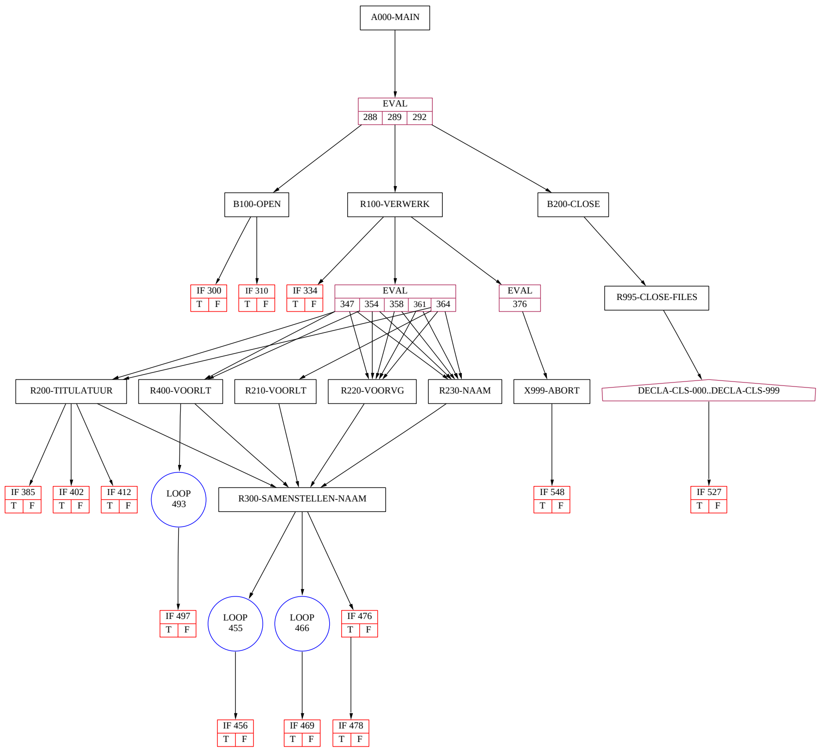


Figure 17. Example real life Conditional Call Graph

keeps track of the local scope, creates a new call edge having that procedure as departing point, and tells the action visitor that this new edge was visited.

We can reuse this filter for different purposes, such as computing the fan-in (number of different callers of a procedure) and fan-out (number of different calls made in a procedure) metrics, which also requires call edges between procedures rather than between statements and procedures. The actual computation can be simplified by introducing yet another filter which takes care of filtering out duplicated edges in (arbitrary) graphs. The resulting fan-in and fan-out computation operates on arbitrary graphs as well, and is independent of the CCG structure.

Last but not least, we can combine the visualization and the metrics computations, resulting in a graph in which the node height indicates a selected metric of interest. As an example, Figure 18 shows the graph for the same program as used in Figure 17. In this case, we have applied the *CallFilter*, so that all call relations are shown at the procedure level. The height of each procedure in the figure is derived from the fanout value computed for that procedure.

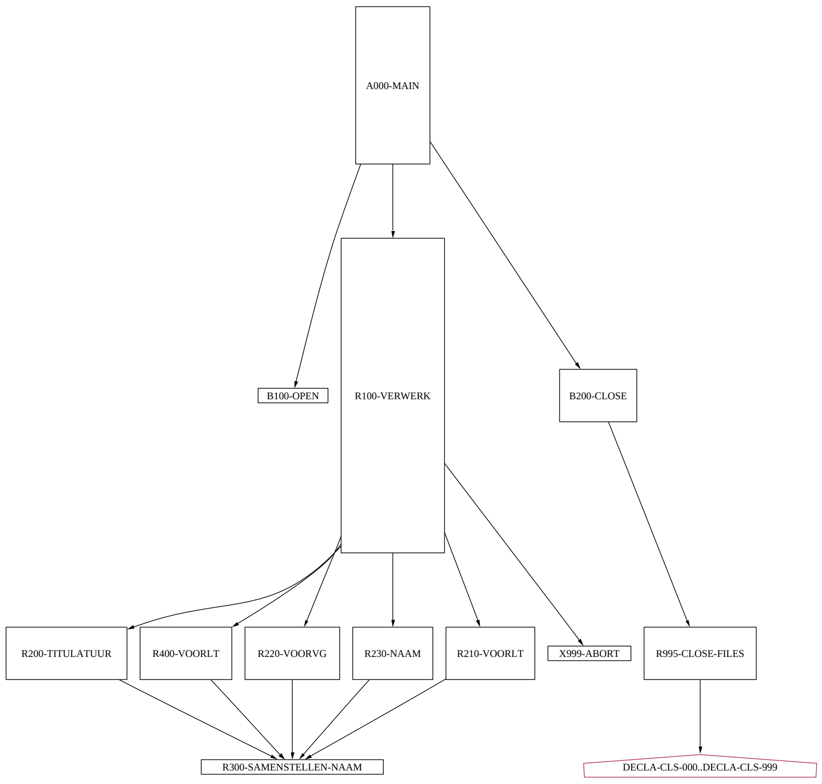


Figure 18. Graph of Figure 17 combined with metrics collection and conditional filtering.

7. Discussion

7.1. Visitor Programming Techniques

In the preceding section, we have applied visitor combinators to the construction of graphs representing control flow information and to the subsequent analysis of these graphs. In the course of developing these combinators, we have identified a number of useful techniques for applying visitor combinators in practice:

`apply` method A combinator is typically written with a particular application context in mind, such as a top down traversal. We adopted a convention to provide each combinator with an `apply` method reflecting this typical application.

filters Various forms of behavior can be obtained by composing complex combinators from a series of filters. The visitors involved in such a filter sequence need not be aware of the filtering mechanism, and can be developed as if they are directly visiting a graph. If necessary, the filters can generate new nodes and edges on the fly, providing even more flexibility.

layers If different but related visitable hierarchies are needed, these can be organized as separate layers. As an example, ControlCruiser includes separate layers for graphs in general and control-flow specific conditional call graphs. We have illustrated how to setup the inheritance and forwarding relations between visitors and visitables respectively. As a result, visitors developed for the higher layer are directly reusable for the lower layers via forwarding.

restarts Passing a visitor over a tree using a traversal combinator hides the recursion inherent in the traversal. We have shown how to use a restarting visitor for cases where recursion must be explicit, for example when constructing recursive structures.

7.2. Visitor Programming Benefits

While implementing ControlCruiser, we found the key benefit of programming using visitor combinators to be in the explicit separation of object navigation strategies (traversal) from object manipulation. Such a separation of concerns has the following advantages:

- Traversal behavior can be understood, developed, tested, and maintained separately separately from the actual object manipulation.
- Analysis behavior can be combined with different traversals. As an example, our McCabe combinator can be applied to a full program or repeatedly for each procedure.
- Traversal behavior can be reused across implementations. For example, we have reused a number of generic traversal combinators such as *TopDownWhile* or *Try*, and we have contributed new generic combinators such as *WhileNotVisited* and *SuccessCounter*.

In our implementation, we extensively used this separation of concerns to include extensive testing. We have developed the combinators in ControlCruiser in a test-driven manner, resulting in separate JUnit [5] tests for each combinator. Moreover, we have extended JJTraveler with explicit support for testing generic combinators.

Other important benefits of visitor combinators include:

- Complex traversal behavior can be composed from simple building blocks, expressing defined combinators in terms of basic ones.
- Behavior can be expressed at various levels of abstraction, ranging from hierarchy-specific combinators to fully generic visitors. Levels we encountered include *CCG*, *Graph*, and *Visitable*. Combinators for the different levels can be seamlessly combined via forwarding.
- Visitor combinators are robust against hierarchy changes.
- Visitor combinators offer refined control over the parts of a tree that should be visited and those that can be skipped. As an example, in Section 5 we introduced an *isBlock* visitor to stop a top down traversal whenever a block was entered.

7.3. Visitor Programming Risks

Making hierarchies visitable may incur some cognitive overhead, in that a larger number of classes must be understood and maintained. The potential maintenance costs should be balanced by the benefits gained from using visitors as discussed in the previous section. Turning a class hierarchy *H* of *N* classes into a visitable hierarchy, requires the following additional classes and methods:

- Two extra interfaces *HVisitor*, *HVisitable*, and an extra class *HFwd* implementing the former interface;
- Four extra methods in each class in *H* in order to implement the *HVisitable* interface (consisting of the methods `getChildAt`, `setChildAt`, `getChildCount`, and `accept`). Each of these methods can usually be implemented in a single statement. Observe that the child access methods in some cases can be obtained via (implementation) inheritance: in our case these methods are only defined in the *Graph* hierarchy, and inherited in the *CCG* refinement.

Note that in many cases, the extra methods and classes can be *generated*, thus reducing the maintenance penalties. For example, we have used JJForester [3] to generate the fully visitable CPF hierarchy of Figure 8.

A second cause of additional classes is the fact that the use of visitor combinators encourages the developer to split traversals into many small steps. Where appropriate, anonymous inner classes or member classes can be used to deal with such small classes.

An account of visitor combinator performance risks is presented in the next section. In some cases, visitor combinators can be optimized by reducing the number of exceptions thrown (which is a relatively costly operation in Java). For example, forwarding to *Fail* in a *TopDownUntil* is more expensive than forwarding to *Identity* in a *TopDownWhile*. Performance can be improved by choosing an interpretation of *VisitFailure* such that failure is less common than success. In particular, using a *Not* in a traversal condition will cause an exception to be thrown for every visited node. For that reason, the implementation of the *WhileNotVisited* in the ControlCruiser libary uses an explicit *Unvisited* combinator, instead of the *Not(Visited)* discussed in Section 6.1.

7.4. Performance Evaluation

In order to analyze the potential performance penalties of using visitor combinators, we set up a simple experiment concerning the computation of the unscoped McCabe counting as discussed in Section 6.2. We wrote three different ways to perform this computation, with the following characteristics:

visitor combinators In the combinator implementation, we computed the McCabe index using the following equation

McCabeCounter() = WhileNotVisited(SuccessCounter(IfRecognizer()))

plain visitor In the plain visitor implementation, we used a traditional visitor as described by [1]. Comparing the combinator and visitor solution provides us with insight in the additional costs of using a combinator solution.

We implemented this traditional visitor as a refinement of *CcgFwd*, overriding the forwarding behavior by recursive calls implementing the (in that case fixed) top down traversal.

direct In the direct implementation, we did not write a visitor at all: instead we directly traversed the tree in a for loop. Comparing this implementation with the plain visitor, provides us with insight in the costs of using the visitor design pattern, in particular the cost of the double dispatch involved.

Our implementation of this direct traversal makes use of the generic *Visitable* interface, using Java's `instanceof` operator to determine whether an if-then-else node has been found.

These three ways of computing McCabe have been applied to the conditional perform graphs of 87 Cobol programs comprising a system totaling over 100,000 lines of code. The results are shown in Figure 19. The X-axis shows size in lines of code for each of the 87 programs; the Y axis shows the visiting time in milliseconds. Per program, the visiting time in the combinator, plain and direct implementations are provided. In our experiments we used the Java compiler and virtual machine from Sun's Java 2 SDK Standard Edition, version 1.4.

When analyzing these figures, the following issues should be considered:

- We expect the combinator approach to be the most expensive one: For each node, a chain of delegating actions has to be conducted (forwarding, for example, visits from Conditional via ProgramPoint, Node, GraphVisitable, and Visitable to the default Identity combinator).
- Our primary interest is in the time needed for *visiting*. Therefore, the data shown does not include the time needed for parsing and graph construction.
- All three McCabe computations simply do one pass over the full conditional perform graph, and are of complexity $O(|V|+|E|)$, where V is the set of vertices in the graph and E the set of nodes. Thus, as the Cobol programs get larger, we expect a linear growth in visiting time. The figure shows the linear regression for the three approaches.
- Outliers can be explained by the fact that individual times will depend on factors that differ per Cobol program, such as the number of conditional statements, the total number of statements, and the number of (perform) relations between them. Each of these factors is strongly correlated with program size, making size a natural unit for doing our performance analysis.

Based on these considerations, we can perform a linear regression analysis on the scatter plot in Figure 19. This shows that the performance differences between the plain and direct visitors are very small. Moreover, from the derived equations for the combinator and classical visitor performance data, we can conclude that the combinator overhead is given by $0.0512/0.0068$ making combinators roughly 7.5 times as expensive.

7.5. Object-oriented language processing

At first glance, the object-oriented programming paradigm may seem to be ill-suited for language processing applications. Terms, pattern-matching, algebraic datatypes are typically useful for language processing, but are not native to an object-oriented language like Java. More generally, the reference semantics of objects seems to clash with the value semantics of terms in a language. Thus, in spite of Java's many advantages with respect to e.g. portability, maintainability, and reuse, its usefulness in language processing is not evident.

In this paper and in an earlier case study [3], we have shown that, in the presence of visitor combinators, object-oriented programming can be a powerful instrument when developing language processing applications. In fact, some specific strengths of object-orientation with regard to declarative paradigms can be mentioned. For example, graph-shaped (rather than tree-shaped) source models can be represented directly as object graphs. Stateful traversals can be implemented naturally with object state rather than resorting to sophisticated encodings such as monads or propagation patterns. And class hierarchies can be extended or refined more easily than algebraic data types.

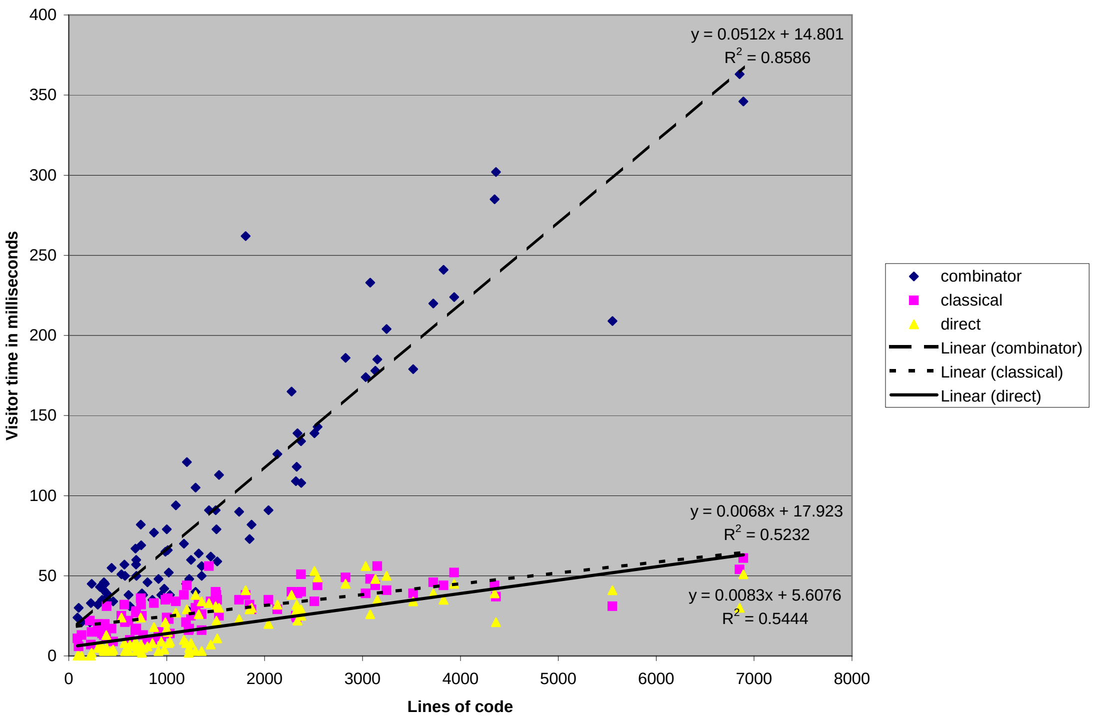


Figure 19. Performance of various visitor forms.

Still, some features of declarative languages remain desirable when applying object-oriented programming to language processing problems. For instance, the modeling of visitor combinators could have been more elegant and more concise if in Java we could directly dispose of higher-order functions and (parametric or rank-2) polymorphism (see also Section 8.2). Also, pattern-matching on term representations could remove some of the verbosity of our approach.

8. Related work

8.1. Source Model Analysis

Selected source model analysis methods include relational algebra in which relations are combined using operators such as intersection, transitive closure, and so on [12]; the GUPRO approach based on extended entity relationships and a *Z*-like graph specification language [13]; reflection models aimed at connecting source models with high-level models [14]; Rigi's Tcl-based command language for navigating through hierarchical graphs [15]; and meta-model based analysis in G^{SEE} [16]. A recent overview is provided in [17]. Observe that in none of these approaches traversals are first-class citizens that can be explicitly manipulated and combined.

In the context of program transformations and reengineering, abstract syntax tree traversal is discussed by [18], who provide a top-down traversal for analysis and transformation purposes. Their traversals have been generalized in the context of ASF+SDF in [19]. Similar traversals are present in the Refine toolset [20], 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 for controlling the visiting behavior.

8.2. Strategic Programming

The origins of visitor combinators can be traced back to strategic term rewriting, in particular [21]. In this style of programming, tree rewrite rules are provided in combination with explicit strategies determining the order in which the rewrite rules are applied. The language Stratego[‡] can be used as a dedicated language for strategic programming. This language focuses on tree transformations rather than just tree analysis. In principle, our visitor combinator approaches can also be used for transformation, but so far we have used them for analysis only. Observe that our analysis involves graphs instead of pure trees.

Term rewriting strategies have also been integrated into typed functional programming, where strategy combinators can be implemented directly as a particular kind of higher-order function [22]. The corresponding Haskell-centered bundle, Strafunski, for generic programming and language processing has also been applied to re- and reverse engineering problems [23] including a very restricted form of Cobol control-flow graphs as discussed more generally in the present paper.

Programming with visitor combinators and programming with function combinators for generic traversal as well as programming with term rewriting strategies can all be seen as an instance of the same idiom: strategic programming. These instances exhibit different strengths. The visitor combinator approach is optimally integrated with the object-oriented paradigm which allows among others dealing with graphs as useful in our problem domain. The functional approach requires encoding effort when dealing with graphs but is very concise because of pattern matching and true combinator style. The Stratego approach offers several features that particularly address concerns in a program transformation context.

8.3. Combinator libraries and generic algorithms

The visitor combinator library of JJTraveler, as discussed and used in the present paper, can be related to combinator libraries in other languages and paradigms. In functional programming, combinator libraries have been developed for a wide variety of general-purpose and domain-specific purposes. Among these are monadic programming [24], parsing [25], pretty-printing [26], document processing [27], web authoring [28], and robotics [29]. In general, functional languages that support higher-order functions can be used to create embedded languages as combinator libraries [30].

[‡]See `http://www.stratego-language.org/`

The C^{++} Standard Template Library (STL, [31]) is a library of collections and algorithms for these collections. It employs some higher-order programming techniques, such as iterators and function objects. Using these techniques, the STL offers generic container algorithms, which the user can instantiate for different kinds of containers with different element types. Thus, both JJTraveler's visitor combinator library and the STL employ generic programming techniques to offer a library of composable and extendable generic algorithms.

8.4. Visitor Design Pattern Extensions

The *hierarchical* visitor pattern [32] 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). Visitor combinators can be used to achieve such traversal control, and much more.

In adaptive programming, and its implementation by the Demeter system [33], 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. A comparison between adaptive and strategic programming is provided by [34].

To complement Demeter's declarative strategies, a domain-specific language (DSL) has been proposed to express recursive traversals at a lower, more explicit level [35]. 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. Being based on code generation, this traversal DSL provides explicit support for recursive traversals, thus avoiding the need for our restarting visitors. 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.

JJTraveler offers a separation between generic and hierarchy-specific operators. This separation is also addressed by Vlissides' *staggered* visitor pattern [36], and the *extended visitor* pattern supported by the SableCC tool [37]. 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 method. 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 [38] uses reflection to achieve what is called *shape polymorphism*. The *Walkabout* 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 (which corresponds to *DownUp(Identity,v,Identity)* in the JJTraveler library).

9. Concluding Remarks

In this paper, we have explored the use of visitor combinators for the purpose of source model analysis in for example reverse engineering and program comprehension tools. Our main contributions include:

- An up to date account of visitor combinators in general and the JJTraveler framework in particular.
- A detailed discussion of a visitor combinator case study, instantiating the generic JJTraveler framework to Cobol program analysis and visualization. The case study includes layered visitable graph representations, graph construction, analysis, and visualization, and a number of new generic visitor combinators.
- A description of a range of techniques and guidelines that can be used during visitor combinator programming.
- An analysis of the risks and benefits of visitor combinator programming, including an experiment for the specific measurement of potential combinator performance penalties.

The key benefit of applying visitor combinator programming to source model is the treatment of traversals as first class citizens, and the possibility to create complex traversals simply by combining basic building blocks.

Future work includes the further expansion of JJTraveler and ControlCruiser. An interesting question is how data can be passed between combinators in a functional style, as opposed to the referencing of instance variables as done in the current paper. In the context of reverse engineering, we anticipate to extend ControlCruiser with hierarchical graphs and graph analyses as required for software architecture reconstruction (conform, e.g., [39]). In the context of program analysis, we will explore how to extend ControlCruiser to cater for various algorithms as described by, e.g., [40].

Availability

JJTraveler can be downloaded from `http://www.cwi.nl/projects/MetaEnv/`. Control-Cruiser is still expanding, and available for research purposes on request.

Acknowledgements We would like to thank Huub de Hesselle, Paul Klint, Ralf Lämmel, Leon Moonen, and Jurgen Vinju for reading drafts of our paper, as well as the anonymous Software Practice & Experience reviewers for their detailed feedback.

REFERENCES

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
2. J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings.
3. T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Science of Computer Programming*, 47(1):59–87, November 2002.
4. 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.
5. K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.
6. A. van Deursen and T. Kuipers. Building documentation generators. In *International Conference on Software Maintenance, ICSM'99*, pages 40–49. IEEE Computer Society, 1999.
7. 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.
8. A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In *International Workshop on Program Comprehension*, pages 90–97. IEEE, 1998.
9. L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, 2001.
10. A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical Report 500-235, NIST Computer Systems Laboratory, 1996.
11. E. R. Gansner, E. Koutsofios, S. North, and K-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.
12. R. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *5th Working Conference on Reverse Engineering, WCRE'98*, pages 210–219. IEEE Computer Society, 1998.
13. B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *Proceedings 3d European Conference on Software Maintenance and Reengineering (CSMR).*, pages 42–50. IEEE Computer Society, 1999.
14. G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.
15. K. Wong, S.R. Tilley, H.A. Müller, and M.-A.D. Storey. Structural redocumentation: a case study. *IEEE Software*, 12(1):46–54, 1995.
16. J.-M. Favre. A new approach to software exploration: Back-packing with g^{see}. In *Proceedings 6th European Conference on Software Maintenance and Reengineering (CSMR).*, pages 251–262. IEEE Computer Society, 2002.
17. R. I. Bull, A. Trevros, A. J. Malton, and M. W. Godfrey. Semantic grep: Regular expressions + relational abstraction. In A. van Deursen and E. Burd, editors, *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE Computer Society, 2002.
18. 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.
19. M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, 2001.
20. 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.
21. 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).
22. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. of Practical Aspects of Declarative Programming 2002 (PADL'02)*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.
23. R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming 2003 (PADL'03)*, LNCS. Springer-Verlag, January 2003.
24. Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.
25. Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1998.
26. John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925. Springer Verlag, 1995.
27. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.
28. P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, pages 192–208, 2002.

```
public interface TreeVisitable extends Visitable {
  public void accept(TreeVisitor visitor) throws VisitFailure;
}
```

```
public interface TreeVisitor extends Visitor {
  public void visitLeaf(Leaf leaf) throws VisitFailure;
  public void visitFork(Fork fork) throws VisitFailure;
}
```

Figure A1. Hierarchy-specifi c Visitor and Visitable interfaces.

29. P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.
30. D. Swierstra, P. Azero, and J. Sariava. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.
31. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.
32. Portland pattern repository. `http://www.c2.com/cgi/wiki`.
33. 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.
34. R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *Proceedings Aspect-Oriented Software Development (AOSD03)*. ACM, 2003.
35. 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.
36. John Vlissides. Visitor in frameworks. *C++ Report*, 11(10), November 1999.
37. 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.
38. 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.
39. I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *21st International Conference on Software Engineering, ICSE-99*, pages 555–563. ACM, 1999.
40. T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, 1998.

APPENDIX A. Example instantiation of the JJTraveler framework

In this appendix we provide the full Java code of an instantiation of the JJTraveler framework for a toy hierarchy. The hierarchy contains a composite interface `Tree` which is implemented by two concrete classes `Fork` and `Leaf`. The hierarchy-specific extensions of the Visitor and Visitable interfaces are given in Figure A1. Figure A2 shows the hierarchy-specific forwarding combinator. Finally, the hierarchy itself is given in Figure A3. Note that the classes in this hierarchy have been made visitable, in the sense that they implement the Visitable interface, via the TreeVisitable interface.

```
public class TreeFwd implements TreeVisitor {
  private Visitor visitor;
  public TreeFwd(Visitor visitor) {
    this.visitor = visitor;
  }
  public void visit(Visitable visitable) throws VisitFailure {
    if (visitable instanceof TreeVisitable) {
      ((TreeVisitable) visitable).accept(this);
    } else {
      throw new VisitFailure();
  } }
  public void visitLeaf(Leaf leaf) throws VisitFailure {
    visitor.visit(leaf);
  }
  public void visitFork(Fork fork) throws VisitFailure {
    visitor.visit(fork);
} }
```

Figure A2. Forwarding combinator for the Tree hierarchy.

```
public interface Tree extends TreeVisitable {
  // Composite pattern
}
```

```
public class Leaf implements Tree {
  public void accept(TreeVisitor visitor) throws VisitFailure {
    visitor.visitLeaf(this);
  }
  public int getChildCount() {
    return 0;
  }
  public Visitable getChildAt(int i) {
    return null;
  }
  public Visitable setChildAt(int i, Visitable child) {
    return this;
  }
}
```

```
public class Fork implements Tree {
  private Tree left;
  private Tree right;
  public Fork(Tree left, Tree right) {
    this.left = left;
    this.right= right;
  }
  public void accept(TreeVisitor visitor) throws VisitFailure {
    visitor.visitFork(this);
  }
  public int getChildCount() {
    return 2;
  }
  public Visitable getChildAt(int i) {
    switch (i) {
      case 0 : return left;
      case 1 : return right;
      default : return null;
  } }
  public Visitable setChildAt(int i, Visitable child) {
    switch (i) {
      case 0 : left = (Tree) child; return this;
      case 1 : right = (Tree) child; return this;
      default : return this;
} } }
```

Figure A3. The class-hierarchy, made visitable.