Optimization of Visitor Performance by Reflection-Based Analysis

Markus Lepper [a]
Baltasar Trancón y Widemann [b]

[a] <semantics/> GmbH, Berlin
post@markuslepper.eu
[b] Universität Bayreuth
Baltasar.Trancon@uni-bayreuth.de

Abstract. Visitors are a well-known and powerful design pattern for processing regular data structures and for combining declarative and imperative coding styles. The authors' umod model generator creates Java data models from a concise and algebraic notation. It is primarily used to model intermediate representations of computer languages. The user defines visitor code by extending skeleton classes, which are generated according to traversal annotations in the model. Since the generated code on its own executes the pure traversal and no semantic side-effects, traversals are redundant unless some user-overridden method is eventually invoked. We present a reflection-based control flow analysis to detect this situation and prune the traversal transparently. With a well-stratified model, this may lead to substantial increase in performance.
Keywords: Visitor Pattern, Generative Programming, Control Flow Analysis, Reflection

1 Introduction

1.1 Visitors

Visitors are a well-known and powerful design pattern for processing regular data structures and for combining declarative and imperative coding styles [3]. The principle is as follows: For evaluating or transforming a certain *instance* of a *data model definition* (existing in a certain *hosting* computer language, normally an object-oriented one), there is some *visitor class* definition (in a strict sense "external" to the model itself) which executes a *traversal* of the model instance. This means that one or more methods (defined in this visitor class) with reserved names are called recursively with all or some of the objects that make up the *elements* of the model instance. The selection and sequential order of these method incarnations follows the graph of the model instance, as defined and realized by the references among the model elements.

There are many different variants of this principle, see Sect. 5. In particular, different phases or transformation steps of the visiting process can be identified by distinct method names. Commonly, the *base visitor code* performs only the traversal, nothing more. Intended *semantics* (i.e. evaluation or transformation of the model) are realized by deriving user-defined, specialized visitors from these base classes, overriding the appropriate method definitions.

1.2 Optimization and Generative Programming

The base class(es) for the visitors are general-purpose traversal code, and do not "know" which subgraphs of a given model instance really need to be visited for a certain user-defined visitor. The optimization algorithm presented here prunes all paths which *surely do not reach any user-defined code*, and consequently do not contribute to the custom semantics of the visitor.

This principle can of course also be applied manually, and indeed often is, e.g. by defining different base visitors for different traversal situations, or by explicitly overriding descending code with a "no-operation". But as the example in Sect. 4.1 will show, the effects of these manipulations can possibly turn out to be very surprising. Hence evolution, incremental definition and refactoring of models may become impossible in a safe and efficient way.

In general, maintenance and evolution of models and the related visitor classes can be made especially safe and efficient if the code for both is *generated* automatically from some concise model definition. The authors' `umod` is a tool for this kind of generative programming. It generates multiple base visitor classes, each combining a certain *kind of processing* with a user-definable *traversal directive*. In this context, the optimization is based on traversal directive information extracted in the model *compilation* phase, combined with the collection of method signatures specific to the user-defined, derived visitor classes, extracted at *class-initialization* time via *reflection*.

2 The Algorithm in General

2.1 Prerequisites for the Algorithm

Our algorithm is suitable for all modeling frameworks where (1) the class collection for a certain model is finite and structured by references and inheritance, (2) visitors with a certain semantics are defined by deriving from basic visitors which realize mere traversal, and (3) the visiting sequence of the latter is defined by a sequence of field selections per class. This applies to our own `umod` implementation, as described in the Sect. 3.2, to many other modeling frameworks, and to XML document type definition languages, see Sect. 4.3.

The following subsections describe these prerequisites and the algorithm in a semi-formal way. A survey of the employed notation is found appendix A.

2.2 Models

Let $\mathcal{C}_0$ be a finite set of predefined, external classes. Any certain model M defines a finite set of classes C_M, the *model element classes*, i.e. the classes of the objects which make up the model instances. C_M is disjoint from $\mathcal{C}_0$. There is a total superclass function $\texttt{extends} : C_M \to (C_M \cup \mathcal{C}_0)$, which must be free of cycles, as usual.

Let ident be the (infinite) set of valid identifiers. Each class definition from C_M is assigned a finite set of *field definitions* by $\mathit{fieldDefs} : C_M \to (\mathsf{ident} \nrightarrow \mathcal{T})$. Assume there is no shadowing of field names. Then the inheritance closure of this map, denoted by *fieldTypes*, is defined recursively as

$$\mathit{fieldTypes}(c) = \begin{cases} \{\} & \text{if } c \in \mathcal{C}_0 \\ \mathit{fieldDefs}(c) \cup \mathit{fieldTypes}(\texttt{extends}(c)) & \text{otherwise} \end{cases}$$

The set of types $\mathcal{T}$ contains primitive predefined types and reference types to external classes from $\mathcal{C}_0$, which are not relevant to our topic. But $\mathcal{T}$ of course also includes simple references to one single instance of a certain model class from C_M, and also to aggregate types referring to more than one instance of more than one model class, like "$\texttt{MAP}\, T_1 \;\texttt{TO}\; (T_2 \texttt{*} T_3)$". The cycle from class definitions via field types back to classes is closed by a relation $\mathit{containedClasses} : \mathcal{T} \leftrightarrow C_M$ which relates each type to all model classes appearing in its definition.

The *set of all field definitions* of a certain model M can be represented by set of pairs $F_M : C_M \leftrightarrow \mathsf{ident}$ which is defined by simply forgetting the types of the defined fields:

$$F_M = \mathit{fieldDefs} \mathbin{⨾} \mathsf{dom}$$

Let Q_M be a certain *model instance* of the model M. In practice, this is realized by a finite collection of objects of the underlying programming language which complies to the structural declarations in M. By the actual values of all those fields which point to instances of element classes, this collection describes an *arbitrarily shaped, labeled directed graph.*

2.3 Visitors

A *visitor* is an imperative program construct which reads and modifies some Q_M, enhanced by some *arbitrary state space* $\mathcal{S}$. Transformations of $\mathcal{S}$ may include local visitor fields updates, local side-effects, or even global side-effects like I/O.[1] For our optimization it is required that visitors with semantic actions are derived by method overriding from *basic visitors.* Each of these does not perform any transformation of the state space, but realize only one certain traversal of the model instance, i.e. the successive visiting of the model objects. Its behavior is thus equivalent to a *traversal selection*, which is simply the selection of those fields, the visitor will follow[2], given as a subrelation $R_n \subseteq F_M$. Let V_n^{G} be the basic visitor corresponding to R_n

[1] In the following formulae, the modification of Q_M is restricted not to modify the traversed fields, because this would make the mathematical modeling unnecessarily complicated. But in theory, there is no real difference for the optimization algorithm.

[2] Indeed, in most frameworks the user defines a *sequential order* of traversal, but we can abstract from this for the purpose of this paper.

From V_n^{G}, the user defines the semantic transformations by deriving one or more *user-defined visitors* $V_n^{\mathrm{U0}}, V_n^{\mathrm{U1}}, \ldots$ by subclassing, i.e. method overriding. Let the collection of all possible visitors of the model M be V_M, and V_n all those based on R_n.

$$\begin{array}{lcl}
\mathit{transformationType} = V_M \times Q_M \to (\mathcal{S} \nrightarrow \mathcal{S}) & & \\
\mathtt{match}, \mathtt{descend} & : & \mathit{transformationType} \\
\mathit{getAction} & : & V_M \to (C_M \to \mathit{transformationType}) \\
\mathtt{getClass} & : & Q_M \to C_M \\
\mathtt{match}\ (v, q) & = & \big(\mathit{getAction}(v)(\mathtt{getClass}(q))\big)(v, q)
\end{array}$$

The purpose of simple visitors is to evaluate a state transformation $(s \mapsto s') \in \mathtt{match}(v, q)$. The transformation operation $\mathtt{match}(v, q)$ depends on the lookup function $\mathit{getAction}(v)$. This function delivers the transformation for a certain visitor and a certain model element class.

In case of the basic visitors, this “transformation” is nothing more than a complicated construction of the “identity”; for all $c \in C_M$, we have

$$\mathit{getAction}(V_n^{\mathrm{G}})(c) = \mathtt{descend}_{c,n}$$

The operation $\mathtt{descend}_{c,n}$ of *transformationType* realizes the traversal of the model, following all fields selected by R_n in class definitions c.

Let $f_1, f_2, \ldots, f_k$ be the sequence of field identifiers from the traversal selection R_n, restricted to $\mathit{fieldDefs}(c)$, i.e., to all fields appearing in a certain class definition. Let $\mathtt{get}(q, f)$ be the semantic operation to look up the current value of a certain field named f in a model element instance q. This value is of type $\mathit{fieldTypes}(\mathtt{getClass}(q))f$, and can be a direct reference to a model element, or of some aggregate type like `SET` or `REL`, referring to model elements indirectly. Let $q_{f,1}, q_{f,2}, \ldots, q_{f,m}$ be all the model element instances which are referred to in this value. Then the code for traversing the model is specified by

$$\begin{array}{rcl}
\mathtt{descend}_{c,n}(v_n, q) & = & \mathtt{descend}_{\mathtt{extends}(c),n}(v_n, q) \\
& & \fatsemi \mathtt{match}\big(v_n, \mathtt{get}(q, f_1)\big) \fatsemi \ldots \fatsemi \mathtt{match}\big(v_n, \mathtt{get}(q, f_k)\big)
\end{array}$$

where $\mathtt{match}\big(v_n, \mathtt{get}(q, f)\big)$ is just an abbreviation for the aggregated operation $\mathtt{match}(v_n, q_{f,1}) \fatsemi \mathtt{match}(v_n, q_{f,2}) \fatsemi \ldots \fatsemi \mathtt{match}(v_n, q_{f,k})$. This code constitutes *a first axis of inheritance*, namely on the model's element classes which serve as *arguments* of the visitor methods.

Some important properties of the **basic visitors' code** can directly be concluded:

(1) If the selected traversal sequence applied to a certain model instance Q_M leads to any *cycles* in the traversal path, then the top-level `match` will not terminate.
(2) But if there are no cycles, it *will* terminate, because $\mathtt{get}()$ and $\mathit{getAction}()$ are *total* functions.

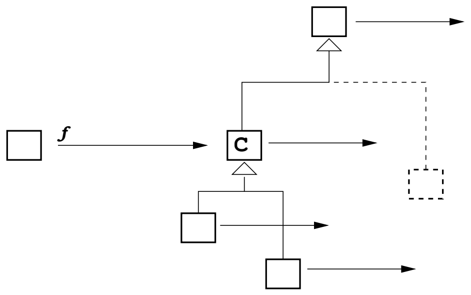

Fig. 1. Inheritance of Associations and Paths

(3) The state variable $s : \mathcal{S}$ is not changed at all, but simply passed through. So each generated visitor V_n^G does not perform any useful work on its own. See the next section for the far-reaching optimization potential that arises from this deceivingly simple property.

A **user-defined visitor** is specified as

$$V_n^{\mathrm{U}m} : V_n \times (C_M \nrightarrow\!\!\!\!\rightarrow \mathit{transformationType})$$

The projection functions `baseVisitor` $: V_n^{\mathrm{U}m} \to V_n$ and $\mathit{actionDefs} : V_n^{\mathrm{U}m} \to (C_M \nrightarrow\!\!\!\!\rightarrow \mathit{transformationType})$ return its first or second component, resp. Its purpose is to re-define the transformations only on the model element classes to be analyzed or modified. This is accomplished by completing the equation system above by

$$\mathit{getAction}(V_n^{\mathrm{U}m}) = \mathit{getAction}\big(\texttt{baseVisitor}(V_n^{\mathrm{U}m})\big) \oplus \mathit{actionDefs}(V_n^{\mathrm{U}m})$$

This constitutes *a second axis of inheritance*, namely w.r.t. the hierarchy of user-defined visitors, rooted at the basic visitors. Again, the inheritance relation must be free of cycles, as usual.

Of course, in all implementations following this scheme, the code delivered by *actionDefs* can make use of ("call") the `match` code of the super-class visitor, including the `descend` of the basic visitor.

2.4 Optimization

Due to observation (3) from the list above basic visitors never change the state, only user-defined code does. Therefore **it is sound to prune all paths on which no user-defined code will ever be executed for any model instance.** So the optimization aims at eliminating the redundant calls to `descend`() that will never reach any user-defined code, for a given Visitor $V_n^{\mathrm{U}m}$. It is necessary to follow a certain field f, iff at least one class appears in the domain of $\mathit{getAction}(V_n^{\mathrm{U}m})$, an instance of which is reachable through f, either directly or indirectly.

To decide this question, we have to consider *inheritance*, as depicted in Fig. 1: The field f as declared pointing to an instance q_c of some model element class c (any "association" in UML nomenclature) may point, in a certain model instance, to an instance of any subclass of c. So, in a first step, all subclasses and all superclasses must be tested whether they appear in the domain of *actionDefs* of any $V_n^{\mathrm{U}m}$, which means that the corresponding action method is overridden by the user. Furthermore, all relevant fields of all subclasses of c, and of all superclasses of c must be included for extending the possible paths which start at f and which have to be checked recursively for user code. Associations starting from "siblings" and "nephews" of the declared target of f need not be considered, see the dotted lines in Fig. 1, since the declaration of the field as of type c does not allow instances of those classes in any valid model instance.

In other words: As soon as the $c \in C_M$ are no longer seen as one level of declaration, but interpreted as an *extensional collection*, i.e., as representatives for all possible run-time objects q for which "q `instanceof` c" holds, then the fields of all super- and subclasses must be included when constructing their connectivity graph. This graph is the first basis for the further search for classes subject to user code. It is calculated as

$$\begin{aligned} &\mathit{includeFieldsOf} : C_M \leftrightarrow C_M \\ &\mathit{includeFieldsOf} = \mathtt{extends}^* \cup (\mathtt{extends}^*)^{\sim} \end{aligned}$$

Please note that the transitive and symmetric closure must be applied in exactly this nesting order, for including all ancestors and descendants while excluding siblings and nephews. We consider this the central "trick" of our algorithm.

Independent from this relation is the question which class *declarations* are related by association through fields. This is the second basis for the search paths. It depends on the selected traversal R_n and is calculated by

$$\begin{aligned} \mathit{decClassToDecClass}_n &: C_M \leftrightarrow C_M \\ \mathit{relevantFields}_n &= F_M \cap R_n \\ \mathit{relevantTypes}_n &= \big\{ \big(c, \mathit{fieldTypes}(c)(i)\big) \bigm| (c,i) \in \mathit{relevantFields}_n \big\} \\ \mathit{decClassToDecClass}_n &= \mathit{relevantTypes} \mathbin{⨾} \mathit{containedClasses} \end{aligned}$$

where $\mathit{containedClasses}(t)$ is simply the set of all model classes which appear in a certain field type t, as defined in Sect. 2.2.

Composing these two axes, we obtain a finite adjacency relation which is subject to standard SCC analysis.[3] Each of the recognized *strongly connected components* is represented simply by some natural number:

[3] A strongly connected component (SCC) is a maximal nonempty subgraph in which there is a path between each pair of members. In this context, the SCC analysis serves as "data compression" to minimize the information which has to be carried over from compile-time to run-time. It is orthogonal to the semantics of the optimization because no two classes in one and the same SCC can behave differently w.r.t. the algorithm. SCC analysis can be performed by standard algorithms like TARJAN's or GABOW's, in linear time w.r.t. the number of edges.

$$
\begin{array}{lcl}
\mathit{connected}_n & : & C_M \leftrightarrow C_M \\
\mathit{connected}_n & = & \mathit{includeFieldsOf} \mathbin{;} \mathit{decClassToDecClass}_n \mathbin{;} \mathit{includeFieldsOf} \\
\texttt{class2scc}_n & : & C_M \rightarrow \mathbb{N} \quad \textit{// partition} \\
\mathit{sccToScc}_n & : & \mathbb{N} \leftrightarrow \mathbb{N} \quad \textit{// quotient of the SCC analysis of } \mathit{connected}_n \\
\texttt{field2sccs}_n & : & F_M \leftrightarrow \mathbb{N} \\
\texttt{field2sccs}_n & = & \mathit{fieldTypes} \mathbin{;} \mathit{containedClasses} \mathbin{;} \texttt{class2scc}_n
\end{array}
$$

Up to this point, the analysis uses only information from model class definitions and the traversal selections for the basic visitors (R_n). Its results are identical for all semantic visitors based on the same R_n.

The next step of analysis requires the knowledge of the semantic visitors:

$$
\begin{array}{lcl}
\mathit{defedClasses} & : & V_M \leftrightarrow C_M \\
\mathit{defedClasses} & = & (\mathit{actionDefs} \mathbin{;} \mathsf{dom}) \cup (\texttt{baseVisitor} \mathbin{;} \mathit{defedClasses}) \\
\texttt{fieldFlags} & : & V_M \leftrightarrow F_M \\
\texttt{fieldFlags} & = & \mathit{defedClasses} \mathbin{;} \texttt{class2scc} \mathbin{;} \texttt{field2sccs}_n^{\sim}
\end{array}
$$

So $\mathit{defedClasses}(v)$ is the set of all model element classes which appear as an argument to some `action()` method overwritten by the user, i.e. the set of those an action method has been "defined for". From this set, the set of all overwritten SCCs is inferred, and from this the set of fields that need to be followed.The traversal code contained in V_n^{G} is instrumented with corresponding conditionals, so that it only calls "$\texttt{match}\big(V_n^{\mathrm{U}m}, \texttt{get}(\ldots, f)\big)$" if $(V_n^{\mathrm{U}m}, f) \in \texttt{fieldFlags}$.

3 Implementation

3.1 The meta-tools Context

meta-tools [15] is a collection of Java and XML-based tools for generative programming, compiler generation, text processing etc. The basic philosophy of all components is to relieve the programmer from tedious and error-prone routine by generating code from semantic models, but at the same time preserving the freedom of arbitrary usage of the underlying programming language as far as possible, for smooth integration with a wide range of tools and software development processes. Hand-written code and generated code are not woven together, but cooperate using the conventional language features for modularization, in particular inheritance.

Typical components of meta-tools are `format` (a framework for human-readable text and code layout, enhancing HUGHES's pretty-printing combinators [5] substantially), `metajava` (seamlessly integrated counterpart of Java reflection for *generating* source code), `tdom` (a *strictly typed* XML document object model [16]), or `option` (compiler for command-line style and GUI style parametrization of applications). These tools have been successfully employed in a variety of mid-scale projects.

3.2 The `umod` Tool

Among these tools the `umod` compiler is a central means for generating `Java` code for general-purpose data models. This includes the classes for the model's elements, safe constructors and setter methods (primarily w.r.t. the illegal value `null`), various methods for visualization and (de-)serialization and different kinds of visitors. Fig. 2 shows a simple model definition source file[4], presenting the most important features of the compiler:

- The syntax of the model definition is designed for compactness.
- Source code of model element classes is generated for

  ```
  abstract class A extends java.lang.Object
  class B1 extends A
  class B2 extends A
  class D extends java.lang.Object
  ```
- Class `B1` has algebraic semantics: the `equals()` predicate and the corresponding `hashCode()` are defined structurally by field-wise recursion, and all instances are *immutable*—no `set()` methods will be generated, but instead `with()` methods that create a modified copy.
- These classes have fields which point to different types of containers (maps, relations, sequences), for which empty instances are created automatically, and which have setter functions checking for illegal `null` values.
- The directive "`C 0/0`" following a field declaration generates constructor code which initializes this field. Constructor code is *inherited*, and 0-ary constructors are provided whenever appropriate.

3.3 `umod` Visitors

The basic visitors V_n^{G} from Sect. 2.3 are also generated by the tool.

Visitor strategies are defined in the same source document with the model, to ensure the consistency of necessary changes as required by certain software development processes. e.g. rapid prototyping. and by later maintenance. In the generated code however, model classes do not depend on visitor classes.

Visitor code generation is controlled by the "`VISITOR` ..." statements, as in Fig. 2. This syntax combines a *traversal directive* and a *visitor kind*.

The traversal directive n is constructed by annotating a field definition with "`V` *n*`/`...". It specifies the field references a visitor shall follow when traversing the model, i.e. it defines the traversal selection R_n which is employed in definition of the basic visitor's behavior in Sec. 2.3.

Fig. 3 shows a simplified realization of the visitor class `Simple`, as defined in the model. (The actual code generated by `umod` looks slightly different.) The visitor *kind* distinguishes between simple visitors, multi-phase visitors, rewriters, co-rewriters (which can deal with cycles), printers, Swing tree builders for visualization, XML encoders, etc. Only basic visitors are generated by the tool.

[4] Please ignore the graying-out of two text lines until reading Sect. 4.1.

The subsequent definition of the semantic visitors ($V_n^{\mathrm{U}m}$) is done on source-text level, using the normal Java inheritance techniques, and tools and processes of the user's choice.

3.4 Implementation Of The Visitor Optimization Algorithm

All analysis up to the calculation of `class2scc`$_n$ and $sccToScc_n$ is done at **model code-generation time**. Every model class is part of an SCC, and every field is linked to the SCC that constitutes the root of the subgraph of classes which can be potentially reached by following the actual value of this field. Using the "`ops`" libraries of $^{\text{meta}}$-tools for high-level algebraic programming, the specifications from Sect. 2.4 are implemented almost literally. The results are encoded into appropriate `static final` data structures, and passed over to the Java compiler, and thus to **execution time**.

At class loading the analysis is completed, since the calculation of *defedClasses* and `fieldFlags` requires the knowledge of the user-defined semantic visitor code. In our framework, this code can come from any source, could even be generated on the fly, therefore the byte code must be analyzed. Since the interaction between generated and user-defined code is restricted to method overriding, *reflection* is sufficient. When the first instance of the visitor class $V_n^{\mathrm{U}m}$ is constructed, its Java class object is queried via reflection for all methods with a matching signature. Whenever such a method is recognized as defined at an inheritance level lower than that of the *generated* basic visitor, then it is classified as user-defined and recorded in *defedClasses*. From this set, the set of all overwritten SCCs is inferred, and from this the set of fields that need to be followed.

The results are stored in a central cache, and retrieved on each subsequent constructor call. The values of these sets vary with every user defined visitor class. Since these are written independently from the tool, this little overhead of copying constant bitset values from a dynamic storage is required.

4 Examples

4.1 Simple Example, Continued

Returning to the toy model definition presented in Fig. 2, assume there is a user-defined visitor $V_n^{\mathrm{U}1}$ which only overrides the method for `action(D)`, and it has to be decided whether a call to `match(A.get_a1())` is necessary. Note that this call could be rather expensive, since `a1` can contain references to an unknown number of `B1` objects. The relevant associations are depicted in Fig. 4.

If $n = 0$, meaning that the visitor is derived from V_0^{G}, the base visitor generated from traversal code `0`, then this match *does not* need to be called: The only way to reach an object of class `D` is via `B2.b2`, but all associations starting from any `B1`, namely `B1.b1b` and `A.a1`, stay in the collection of `B1` objects. This is different when deriving from V_1^{G} and V_2^{G}, for different reasons: $V_1^{\mathrm{U}m}$ follows additionally field `B1.b1` of type `A`. The actual value of this field *could* be of type B2, thus `action(D)` can be reached.

```
MODEL M =
   VISITOR  0 Simple ;
   VISITOR  1 Rewrite IS REWRITER ;
   VISITOR  2 Visitor2 ;

TOPLEVEL CLASS
  A ABSTRACT
      a1  int <-> B1    !       V 0/0 1/0 2/0 ;
      a2  A             ! C 0/0 V         2/1 ;
  | B1 ALGEBRAIC
      b1  OPT A         !       V     1/0     ;
      b1b SEQ B1        !       V 0/0         ;
  | B2
      b2  int -> D      !       V 0/0 1/0 2/0 ;
      b2b OPT B2        !       V 0/1 1/1     ;
  D
      d   int  = "17"
END MODEL
```

Fig. 2. A simple example model definition

```
public class Simple {
  public void match(Object x) {
    if (x instanceof A)
      match((A)x);
    else if (x instanceof D)
      match((D)x);
    else
      action_foreignObject(x);
  }
  public void match(A x) {
    if (x instanceof B1)
      match((B1)x);
    match((B2)x);  // closed world
  }
  public void match(B1 x) {
    action(x);
  }
  public void match(B2 x) {
    action(x);
  }
  public void match(D x) {
    action(x);
  }
  protected void action(A x) {
    for (B1 sub : x.a1.range())
      match(sub);
    match(x.a2);
  }
  protected void action(B1 x) {
    action((A)x);
    for (B1 sub : x.b1b)
      match(sub);
  }
  protected void action(B2 x) {
    action((A)x);
    for (D sub : x.b2.values())
      match(sub);
    if (x.b2b != null)
      match(x.b2b);
  }
  protected void action(D x) {}
   //no call of match() for "int"
  protected void
    action_foreignObject(Object x)
    {}
}//class Simple
```

Fig. 3. Generated code for a simple visitor according to traversal directive "`V 0/..`".

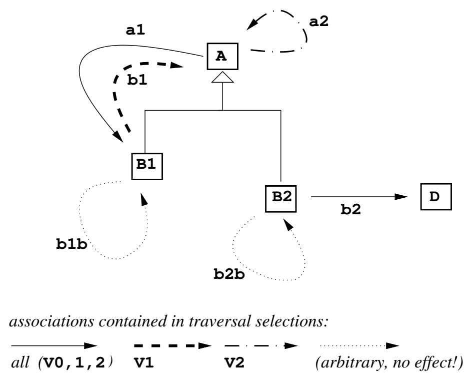

Fig. 4. The associations in the example model

The consequence of adding `A.a2` with traversal code `2` may seem somehow surprising at a first glance: While the "loops" at the "leaf classes" `B1.b1b` and `B2.b2b` are totally irrelevant for the analysis (indicated by the graying-out of the source text lines and the dotted arrows in the graphic!), a loop at a non-leaf class is *not*: of course, the current visited object is already an instance of `A` (remember, we are discussing whether `match(A.get_a1())` needs to be executed). But this new association `A.a1` does not only add some redundant self-reference, but also a way to reach a `B2` from any instance of `B1`, which changes the graph of possible paths dramatically. This illustrates that applying this kind of optimization *manually* would be rather error-prone, esp. when later maintenance requires adaption of the model's structure.

4.2 A Real-World Application

We have applied the optimization to a real-world programming language compiler created with extensive use of the $^{\text{meta}}$-tools: The Tofu language is a pure functional language with a powerful type system based on the *calculus of constructions*. It features polymorphism, type-level functions and dependent types, but is still *total*: All well-typed functions terminate for all inputs. Tofu is intended as executable semantics, and hence prototype implementation and test oracle, for mathematically rigorous software documentation (cf. [17]). The Tofu compiler uses a generated parser and a `umod`-generated semantical model of the language. All compiler passes are implemented as visitor-style transformations on the model (mostly of the rewriter kind). The compiler translates Tofu to a low-level, untyped applicative intermediate representation (functional "assembly language") and eventually to Java using the facilities of `metajava`.

The structure of the internal `umod`-model of Tofu makes it a natural test case for visitor optimization: The model is, in its current version, strictly 2-stratified, with references from a *namespace* level (modules, declarations, definitions) to an *expression* level (terms, function abstractions, types), but not vice versa. In

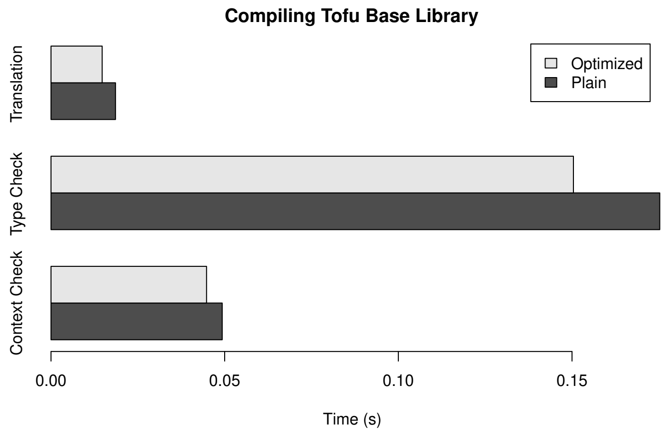

Fig. 5. Run times for major phases of the Tofu compiler, compiling the Tofu base library. Shown figures are means of real durations (`System.nanoTime`) of 100 runs after a warm-up phase of 10 runs. Relative improvement is approx. 9% (Context Check), 14% (Type Check) and 21% (Translation), resp. (14% cumulatively). A total of 5639 subtrees are pruned per run.

total there are 33 classes with 44 declared fields (not counting inheritance) in the model.

Fig. 5 shows some benchmark results. They were obtained on an Athlon 64 X2 5000 dual-core machine and the Sun JDK 1.6.0_13 client VM with 200MiB of heap space. The execution times of the following phases have been measured: *Context Check* – removes syntactic sugar, resolves references, establishes scoping of variables. *Type Check* – verifies type consistency (including termination) of all function definitions, infers implicit type parameters. *Translation* – performs type erasure, constant folding, inlining and lambda lifting; generates low-level (LISP-like) functional code.

All of these rely heavily on the visitor pattern, invoking 21 user-defined visitor classes. Invoked on the Tofu base library (757 lines of code), a total of 3625 visitors are instantiated. The measurements show consistent improvement of the optimized version of the generated visitor code over the plain version of approximately 14%. The overhead of a dynamic check of `fieldFlags` for each field value is included in the calculation. The check allows to prune the traversal of a subtree of unknown size in 5693 cases. The additional overhead of the execution-time phase of our algorithm is difficult to measure exactly because of its interference with class loading, but can be estimated as at most 4 ms, i.e. 5% of execution time.

The measurements have been obtained as the real execution times (as reported by successive calls to `System.nanoTime()`) of the respective compiler phase. To reduce the impact of system load and random effects, the compilation has been repeated $K + N$ times, interspersed with calls to `System.gc()`.

The first $K = 10$ runs served as a warm-up phase for the class loader and JIT compiler. The following $N = 100$ runs were averaged.

We conclude that the achieved gain in efficiency is satisfactory. Most of the pruned subtrees are subgraphs that represent Tofu expressions, traversed by visitors that only provide custom actions for the namespace level of Tofu. While the same efficiency (and even more, considering the overhead of dynamic pruning checks) can be achieved by user-controlled explicit pruning, the automatic optimization is superior from the software-engineering perspective, because the stratification properties of the Tofu model could be altered, or ruined entirely, even by small changes (cf. "`A.a2`" in Fig. 4). For a model of realistic size, the transparency provided by automatic visitor optimization is a valuable feature.

4.3 Possible Applications on Other Stratified Data

The amount of stratification and unidirectional references in a model is an indicator for the expected gain from applying our algorithm. Our analysis of two wide-spread XML models has shown significant potential. We have recorded the number of SCCs and the maximal path length in the SCC graph (generation count), which serve as measures of stratification in relation to the number of elements.

Document Type	Path Length	SCCs	Elements
XHMTL 1.0	6	22	77
DocBook 5.0	18	77	362

5 Related Work

The related work we have found is either rather narrow or rather broad, depending on the criterion: There is only one work dealing with automated pruning ([8]) and few dealing with pruning by programming. On the other hand there are many approaches to automated visitor code generation, very different and hardly comparable. We decided to give here the broader view, because all these papers contribute valuable aspects.

The "visitor pattern" itself appears to have risen out of folklore, with [3] being the first publication to use this term explicitly. In [1] there is a more recent and more exact description.

In [10] "`Walkabout`" is presented, a versatile extension of the visitor pattern, avoiding the need of the "closed-world assumption". It uses Java reflection for run-time inquiry and even for *method invocation*, being substantially less efficient than generated code. We did not find practical applications. A combinator library for visitors is presented in [14]. It allows, among other useful things, explicit and hand-coded pruning of subtrees. It contains combinators like "Choice(v1,v2) — Try v1. If it fails, try v2". This catalog could be a valuable suggestion for further evolution of the `umod` output. A more recent paper on "strategic programming" is [6], which discusses the need of a language-independent traversal control very thoroughly. Automated pruning is not discussed, but could be integrated into this context in a rather natural way.

The only paper we found which addresses (among more general questions of traversal control) a very similar topic, namely automated pruning of visiting sequences directed by constraints, is [8]. Its context is the Demeter system [7], which translates graph models into Java code and other programming languages. It resembles our `umod` approach w.r.t. compactness and algebraic flavor. Belonging to aspect-oriented programming, this approach is independent from a particular language as back-end and includes proprietary language constructs for visiting strategies and algorithms. Our approach leaves this intentionally to Java constructs, while supporting also non-syntactic data types like sets and maps, with their Java-specific semantics.

Based again on a standard programming language is the approach in [4]: an architecture based on C++ template instantiations and the "standard template library" (STL) for re-using visitor code for translating domain specific models into different back-ends. This meets our approach w.r.t. re-using traversal code and separating phases and tasks. The issue of automated pruning is not addressed there.

The approaches in [9] and in [13] are even more similar to ours, since Java code is generated. The main difference is that `umod` visitors are generated as part of generated Java source, but in [9] the visitors are generated to operate with independently created Java classes. Consequently, the visitor description language is much more complicated and deals with details (e.g. calculation results and method signatures) which our tool leaves to the subsequent "manual" refinement process, using plain Java language means.

Another ambitious visitor code generation system is presented in [13], together with formal semantics and corresponding proofs. Again, we do less (w.r.t. verification and provable correctness), but allow more (namely arbitrary transformations in the course of $S \twoheadrightarrow S$, e.g. I/O). In [11], all possible execution sequences of a certain visitor are translated into a context-free grammar. In a second step, this approach would also allow automated pruning, but this is not addressed in the paper. The approach in [2] resembles our `umod` code generation, but is again more versatile on the back-end (not focused on a single programming language) and less versatile on the front-end (models must be "trees"). Whether the visitors perform any optimization is not clear from the documentation.

As a **conclusion**, our approach seems to take a middle road between the others: One one hand, it has exact algebraic specification of the data model, and computability of certain model properties needed for the optimization. On the other hand, it is not a self-contained language for the complete definition of models and their transformations, but a pragmatic solution for generating "smart boilerplate" model code in Java, leaving open all the possibilities of the language the programmer is accustomed to use freely.

Acknowledgments

Many thanks to the anonymous reviewers for valuable hints. These have helped to improve the text substantially.

References

1. P. Buchlovsky and H. Thielecke. “A Type-theoretic Reconstruction of the Visitor Pattern”. In: Mathematical Foundations of Programming Semantics (MFPS’05), vol. 155. ENTCS. Elsevier, 2006.
2. A. Demakov. *TreeDL*, 2007. URL: `http://treedl.org/`.
3. E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
4. J. H. Hill and A. Gokhale. *Using Generative Programming to Enhance Reuse in Visitor Pattern-based DSML Model Interpreters*. Tech. rep. IEEE Trans. SP, 2007.
5. J. Hughes. “The Design of a Pretty-printing Library”. In: *Advanced Functional Programming*. Springer, 1995, pp. 53–96.
6. R. Lämmel, E. Visser, and J. Visser. *The Essence of Strategic Programming*. 2002.
7. K. Lieberherr. *Demeter: Aspect-Oriented Software Development*, URL: `http://www.ccs.neu.edu/research/demeter/`.
8. K. Lieberherr, B. Patt-Shamir, and D. Orleans. “Traversals of object structures: Specification and Efficient Implementation”. In: *ACM Trans. Program. Lang. Syst.* 26.2 (2004), pp. 370–412.
9. J. Ovlinger and M. Wand. “A language for specifying recursive traversals of object structures”. In: *ACM SIGPLAN Notices* 34 (10 1999).
10. J. Palsberg and C. B. Jay. *The Essence of the Visitor Pattern*. 1997.
11. M. Schordan. “The Language of the Visitor Design Pattern”. In: *Journal of Universal Computer Science* 12.7 (2006), pp. 849–867.
12. J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1988. URL: `http://spivey.oriel.ox.ac.uk/~mike/zrm/`.
13. T. van Drunen and J. Palsberg. “Visitor-oriented programming”. In: *Proc. FOOL-11*. 2004.
14. J. Visser. “Visitor Combination and Traversal Control”. In: *Proc. OOPSLA 2001*. ACM Press, 2001, pp. 270–282.
15. B. Trancón y Widemann and M. Lepper. *The BandM Meta-Tools User Documentation*. 2010. URL: `http://bandm.eu/metatools/docs/usage/`.
16. B. Trancón y Widemann, M. Lepper, and J. Wieland. “Automatic Construction of XML-based Tools seen as Meta-programming”. In: *Automated Software Engineering* 10.1 (2003), pp. 23–38.
17. B. Trancón y Widemann and D. L. Parnas. “Tabular Expressions and Total Functional Programming”. In: *IFL 2007*. Revised Selected Papers. Springer, 2008, pp. 219–236.

A Mathematical Notation

The employed mathematical notation is fairly standard, inspired by the Z notation [12]. The following table lists some details:

$A \rightarrow B$	The type of the *total* functions from A to B.
$A \nrightarrow B$	The type of the *partial* functions from A to B.
$A \ \text{⇻} \ B$	The type of the *partial and finite* functions from A to B.
$A \leftrightarrow B$	The type of the relations from A to B.
$\mathsf{ran}\, a, \mathsf{dom}\, a$	Range and domain of a function or relation.
$r^{\sim}$	The inverse of a relation
r^{*}	The reflexive-transitive closure of a relation
$r \mathbin{⨾} s$	The composition of two relations: the smallest relation s.t. $a \ r \ b \wedge b \ s \ c \Rightarrow a \ (r \mathbin{⨾} s) \ c$
$r \oplus s$	Overriding of function or relation r by s. Pairs from r are shadowed by pairs from s: $r \oplus s = \big(r \setminus (\mathsf{dom}\, s \times \mathsf{ran}\, r)\big) \cup s$

Functions are considered as special relations, i.e. sets of pairs, like in "$f \cup g$".