Visitor Optimization Revisited – Realizing Traversal Graph Pruning by Runtime Bytecode Generation

Markus Lepper
semantics gGmbH Berlin, Germany

Baltasar Trancón y Widemann
Nordakademie Elmshorn, Germany
semantics gGmbH Berlin, Germany

Abstract

Visitors and Rewriters are a well-known and powerful design pattern for processing regular data structures in a declarative way, while still writing imperative code. The authors' "umod" model generator creates Java data models from a concise and algebraic notation, including code for visitor skeleton classes according to traversal annotations. User visitors are derived from these, overriding selected generated methods with payload code. All branches of the visiting trajectory that are not affected can thus be safely pruned according to control flow analysis. In the first version [7], the pruning was implemented by dynamic case distinction. Here we have developed a new solution employing code generation at runtime.

2012 ACM Subject Classification Software and its engineering → Software performance; Mathematics of computing → Paths and connectivity problems; Theory of computation → Program analysis

Keywords and phrases Visitor Pattern, Generative Programming, Control Flow Analysis, Reflection, Runtime Code Generation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.20

Supplementary Material *Software (Source Code)*: `http://bandm.eu`

1 Introduction, Context

Visitors and Rewriters are a well-known and powerful design pattern for processing regular data structures in a declarative way, still writing imperative code. See Gamma et al. [4], Palsberg et al. [13], and VanDrunen et al. [18] for the foundations; see Nanthaamornphong et al. [11] for an empirical study encouraging their use, and for a survey on such studies. Pati and Hill [14] give a survey on extensions and alternatives of the original pattern – the umod approach as described below would classify generally as their "dynamic type" and "acyclic", its multiphase variant as a "hierarchical" one. Petrashko et al. [15] present a full-fledged compiler architecture as a sequence of visitor/rewriter phases and emphasize the optimization needs for compilers in practical engineering. Taking a bird's-eye view upon complete model definitions, and planning and optimizing traversals by algebraic means has been a main topic in "Adaptive Programming" [3][9].

The authors' "umod" model generator creates Java data models from a concise and algebraic notation [7]. (For more related work prior to 2011 please refer here.) The tool generates source text files for the classes which realize the elements of the model. The factor between the lines of codes of an umod source and the generated Java is in the range of 1:25 to 1:40. Such a reduction significantly improves order, robustness, and maintainability in software projects, and is a strong argument in favour of source text generation.

```
1  MODEL M =
2     VISITOR 0 Simple ;
3     VISITOR 1 Rewriter IS REWRITER ;
4     VISITOR 2 Visitor2 MULTIPHASE ;
5
6  TOPLEVEL CLASS
7    A
8        a1  int <-> B1 ! V 0/0 1/0 2/0 ;
9        a2  A          ! V         2/1 ;
10   | B1
11       b1  OPT A      ! V     1/0     ;
12       b1b SEQ B1     ! V 0/0         ;
13   | B2
14       b2  int -> D   ! V 0/0 1/0 2/0 ;
15       b2b OPT B2     ! V 0/1 1/1     ;
16   D
17       d   int  = "17"
18   | E
19 END MODEL
```

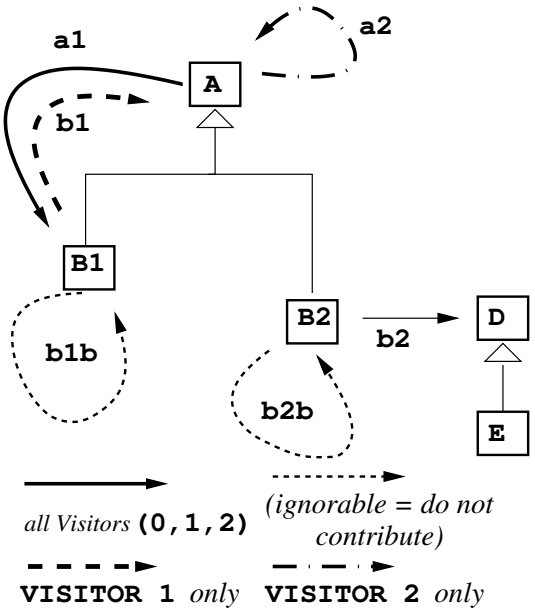

Figure 1 A simple example model definition.

Models realized by umod are collections of instances of these classes, called *model elements*, created one after the other explicitly by the programmer, with host language constructor calls, and linked together by field values. (For technical details please refer to the original publication [7].) Umod is currently the basis for about twenty small to medium-scale projects in academy and industry.

Figure 1 shows an example: Uppercase identifiers are names of class declarations, lower case are instance fields. Indentation with the "`|` " operator indicates inheritance/subclassing.

The types of fields are primitive types, references to model classes, references to external classes, or free applications of the constructors `SEQ`, `MAP/->`, `REL/<->`, `OPT` and tupling/`*`. These behave fully compositionally, allowing complex nestings like "`((SEQ int) * string) <-> (int -> OPT int)`". The generated code includes safe constructor and setter methods that reject spurious `null` values, serialization, visualisation, etc.

Much of the power of umod comes with the generated code for visitors and rewriters. in the second halves of the source lines 8, 9, 11, 12, 14, and 15 in Figure 1, the annotations of form "`V` *a*`/`*b*" define *traversal plans*: The value of *a* is a numeric identifier of such a plan, the order of the numbers *b* gives the sequential order of visiting the fields of this class.

Umod models are not restricted to tree shape – their fields with reference values can span arbitrary graphs, possibly including cycles. Any straightforward solution like the classical `Walkabout` [13] cannot cope with these. The explicit selection of the fields to be followed allows for cycle-free traversal plans (= "spanning trees"), much more efficiently implemented.[1]

Using these plans, the generation of visitor classes of different types (plain or multiphase, rewriters, cyclic rewriters, visualizers etc.) can be declared, as shown in lines 2–4 of the Figure.

[1] Theory and implementation of cyclic visitors and rewriters is much more challenging, see [8].

2 Optimization of Visitor Traversal

As known from other visitor frameworks, the visitor pattern is employed by (A) deriving a class from the abstract visitor's base class (here: source generated by umod), (B) overriding a dedicated, overloaded `visit` method for some selected classes with payload code, and finally (C) invoking some top-level `visit` method with the object (graph/model) to be processed.

The `visit` methods of the generated base class perform just the traversal, according to the traversal plan. They can explicitly be called from the payload code where appropriate.

When campaigning to introduce generative tools into practice, the gains in production speed and maintainance always stand against a (presumed or real) loss in execution speed. Therefore all potentials for *optimization* should be explored. All descending calls which never (transitively) reach a payload method, according to the selected traversal plan and the overridings in this particular programemr-defined visitor subclass, can thus be pruned for optimization.

The overridden payload methods are inherited along two axes: first by a visitor from its superclass, then (by explicit casting in the generated code) among the visitees. Descending into one class valued field successively follows the traversal plan of all ancestors of this class, and potentially any of its descendant classes. The right side of Figure 1 shows the UML graph with the associations selected by the different traversal plans.

Must a request to visit an instance of class `B1` really be satisfied when there is only one single payload definition for class `E`? Not with traversal plan `0`. But in plan `1` each `B1` has a direct reference to an object of type `A` (in UML terms: an "association"), which can be an instance of `B2`, which has an association to `D`, and any instance of this could be an `E`. In plan `2` such a reference to `B2` is even more indirect, namely inherited by `B1` from `A`.

The *pruning condition* states for every field of every class in the umod definition whether descending into its value can possibly reach a payload method. It is calculated for one particular programmer-defined visitor by combining (a) the subclass relation of the model classes, (b) the map from all field definitions to the sets of all model classes which occur in their type, (c) the set of fields selected by the traversal plan, and (d) the set of model classes with an overridden visit method. (a)–(c) are explicit in the model definition, and hence known to umod when generating the Java source; (d) is specific to the application-defined visitor class, and extracted at runtime from the loaded class by means of *reflection*.

The expressions for $\mathsf{classGuard}_U$ and $\mathsf{fieldGuard}_U$ in Figure 2 give the the pruning condition per class or per field, resp., as needed for the two variants of the optimization. Most of it can be calculated at model compilation time. An intermediate step is to condense the traversal graph into strongly connected components (SCCs) of classes, and all further graph analysis is performed on these rather than individual model classes.

In the original implementation [7], the class loading code generated an array of boolean flags indexed by field ids, and the generated visitor code contained an explicit test before descending into the current value of a particular field – see the value of $\mathsf{fieldGuard}_U$ in Figure 2. This variant is called AOT in the following, because all code is created *ahead of time*.

A first result of this project was an empirical performance improvement in the range of 10–20% for a realistic application.

A second result was a methodological warning to software developers: The analysis of the reachability relation turned out to be quite more complex than expected. For example: The associations from a class to itself `B1.b1b` and `B2.b2b` do not contribute to the set of reachable classes and can be ignored, see Figure 1. But `A.a2` does, because of subclassing. Thus any attempt to apply traversal pruning "manually", when coding, is not advisable:

C // = *all model classes*
F // = *all fields, by ids unique across all classes*
$\mathsf{super} : C \nrightarrow C$ // *maps class to its superclass, if not a root class*
$\mathsf{fieldOf} : F \to C$ // *maps field to containing class*
$\mathsf{refersTo} : F \leftrightarrow C$ // *links fields to the model classes appearing in its type*
$\mathsf{select}_n : 1 \leftrightarrow F$ // = *fields selected by the traversal plan* n
$\mathsf{subSup} : C \leftrightarrow C = \mathsf{super}^* \cup \mathsf{super}^{\sim *}$
$\mathsf{reaches}_n : C \leftrightarrow C = (\mathsf{subSup} \fatsemi \mathsf{fieldOf}^{\sim} \fatsemi \mathrm{ID}_{\mathsf{select}_n} \fatsemi \mathsf{refersTo})^*$
$\mathsf{scc}_n : C \to C$ // *each SCC is represented by one of its members*
$\mathsf{reaches}_n \cap \mathsf{reaches}_n^{\sim} = \mathsf{scc}_n \fatsemi \mathsf{scc}_n^{\sim}$
$\mathsf{payload}_U : 1 \leftrightarrow C$ // = *for one particular programmer defined visitor* U*: the set of*
// *all classes for which the visit method is overridden.*
// *Let the underlying traversal plan of* U *be* p*.*
$\mathsf{classGuard}_U : 1 \leftrightarrow C = \mathsf{payload}_U \fatsemi \mathsf{subSup} \fatsemi \mathsf{scc}_p \fatsemi \mathsf{reaches}^{\sim} \fatsemi \mathsf{scc}_p^{\sim}$
$\mathsf{fieldGuard}_U : 1 \leftrightarrow F = \mathsf{classGuard}_U \fatsemi \mathsf{refersTo}^{\sim}$
// $\_ \fatsemi \_$ *is relational composition.* $\_^{\sim}$ *is inversion.* $\_^*$ *is reflexive–transitive closure*
// $ID\_$ *is the identity restricted to a set.* 1 *is any singleton type, used for modelling sets.*

Figure 2 The umod visitor optimization, symbolically.

```
protected void action(E elem) {
  ...
  if (fieldGuard[idOfField]) {match(elem.field);}
  ...
}
```

Figure 3 Old implementation of pruning, AOT variant.

The strength of the visitor pattern is its declarative nature, which brings compositionality, maintainability, and automated adaption to model changes, all of which are jeopardized by manual intervention.

3 Pruning by Runtime Code Generation

In the original implementation (AOT), the pruning has been realized by explict guards on the caller side; see Figure 3. Although somewhat redundant, this appears to be a more reasonable choice for performance than placing guards on the callee side: it is not likely for the just-in-time compiler to inline a method that contains a potentially large conditional statement, and hence unnecessary calls would be expected.

More recently, we have been experimenting with the potential of dynamic program specialization by runtime bytecode generation for staged meta-programming [16, 17]. This allows for a distinct implementation strategy called JIT (because part of the code is created just in time, at runtime): In a subclass of the given programmer-defined visitor, methods that constitute entry points for unaffected SCCs can be overridden with code that returns immediately; see Figure 4. Modern JIT compilers can be trusted to inline such small method bodies aggressively, thereby fully eliminating the need for caller-side checks. As a bonus, contrarily to the AOT implementation, visitor optimization has zero cost, regarding both

```
// if (!classGuard[idOfS]) generate the following:
@Override
protected void action(S subElem) {
  return;
}
```

Figure 4 New JIT variant of pruning.

```
new MyVisitor() {
  ...
}.compile().match(rootElement);
```

Figure 5 Explicit use of specialized visitors.

code size and runtime performance, in cases where no pruning can be applied. Furthermore, the partial pruning of fields with complex types, such as `A -> B` where `B` is affected but `A` is not, is supported.

For this strategy to take effect, the visitor object that is actually invoked needs to be replaced by a quasi-clone, an instance of a suitably specialized subclass generated on demand by the umod runtime library. This is almost transparent to the programmer; see Figure 5: Every visitor class provides a method `compile()` that creates and returns such a quasi-clone. Code is generated only once per programmer-defined visitor class, and subsequently cached. If the compiler is switched off or otherwise not available, `compile()` falls back to simply returning the original, such that programmer code may work as intended, only with suboptimal performance.

The runtime support in the umod library consists essentially of the inherited base method `compile()`, and has been implemented in some 25 lines of code, using the LLJava-live code generator library [17]. LLJava-live provides a concise builder API and efficients generator for JVM bytecode to be loaded straightaway into the running application. All Java classes, including inner, local and anonymous classes can be subclassed.

For the umod specialization mechanism to work as expected, three preconditions must be met: First, the visitor class to be specialized must have an accessible constructor that takes no user-visible arguments. Second, necessary configuration with setters must be performed *after* compilation. Third, the Java 9+ module system precludes access to the captured variables of nested classes across class loaders. Hence, support for specialization of nested classes, as in Figure 5, requires the application to be loaded with a specific class loader provided by LLJava-live. In the next section this class loader is used also for the other variants, for fair comparison.

4 Evaluation

In order to compare the dynamic and the generator-based implementations of pruning with each other and the naïve version in detail, we have constructed some small benchmark tests. As the umod model for these experiments we use DTM, a semantic model of XML DTDs [2]. In DTM, parameter entities, cross-references between elements, attribute lists, and XML namespaces are resolved. The umod source defines 29 model classes in about 100 lines of code. All experiments operate on two fixed DTM document objects, namely the XHTML-1.0-Strict DTD [20] and the SVG-1.1 DTD [19] (978/5664 LoC in original form, respectively).

Table 1 Benchmark results (XHTML-1.0-Strict).

Case	Naïve	AOT	JIT
nop	40.06 μs ± 0.41%	0.05 μs ± 2.08%	0.10 μs ± 6.12%
nms	70.89 μs ± 0.85%	47.69 μs ± 0.86%	49.62 μs ± 1.05%
rep	37.00 μs ± 0.42%	21.64 μs ± 0.54%	7.94 μs ± 0.43%
req	39.23 μs ± 0.46%	21.67 μs ± 0.50%	20.67 μs ± 0.38%
few	43.11 μs ± 0.39%	0.96 μs ± 0.31%	1.78 μs ± 1.07%

Table 2 Benchmark results (SVG-1.1).

Case	Naïve	AOT	JIT
nop	70.97 μs ± 1.16%	0.06 μs ± 1.64%	0.42 μs ± 68.11%
nms	205.70 μs ± 5.07%	198.14 μs ± 2.94%	194.81 μs ± 4.22%
rep	87.40 μs ± 1.75%	67.44 μs ± 0.69%	7.60 μs ± 0.51%
req	86.10 μs ± 1.46%	71.37 μs ± 0.61%	74.36 μs ± 1.12%
few	95.74 μs ± 1.38%	1.09 μs ± 0.18%	2.05 μs ± 1.02%

Each benchmark case consists of a single visitor class that performs a simple query on the data, and hence overrides no more than two methods: *nop* does nothing at all; *nms* collects the set of all names in the document (167/340); *rep* counts the elements with repeatable content, i.e., which use the operator `+/*` (17/63); *req* counts the required attributes (13/40); *few* counts the elements with fewer than five attributes (3/4).

Running times are estimated as wallclock times, measured with `System.nanoTime()` precision. Each query is repeated $N = 50000$ times, after a JIT compiler warmup phase of the same length. A fresh visitor instance is created in each iteration, and its `compile()` method is invoked. The reported numbers are median values and median absolute deviations. All measurements have been performed on a system with a Core i5-10210U CPU, running Ubuntu 20.04 and OpenJDK 11.0.16.

The results indicate that the performance improvement by code generation are largely comparable with those by dynamic pruning, and that both can be quite dramatic for "sparse" visitors operating on stratified models, where a lot of pruning can occur. It is unclear how much just a more aggressive JIT compiler would improve the naive approach.

The instantiation of a runtime-generated visitor subclass in the JIT variant requires the use of reflection, and thus incurs some initial overhead; as can be seen in the *nop* case, the overhead is generally small but potentially erratic. By contrast, generated code performs significantly better in the *rep* case, where pruning occurs much more sparingly and at deeper levels of nesting than in the other cases.

5 Umod Visitors Meet Strategic Programming

Strategic Programing (SP), as developed by Eelco Visser and others, is in the first line a *concept*, comprising term definitions, requirements, categories, abstract algorithms, etc. In a second step this concept can be "incarnated" in programming languages of different paradigms: functional, object oriented, data driven, etc.; see Figure 7 in [6], and more in Figure 6 in [10].

Umod differs from most other approaches to visitor generation: It is a *domain-specific language (DSL)* with own syntax, semantics and implementation. But it is neither realized by genuine means of the host language (*Embedded DSL, EDSL*) as in the "Scrap your boilerplate" projects [3] [5] and the Scala "Miniphases" [15]. Nor are its syntax and that of the host language merged into the input format of a dedicated compiler, as with "Tom". [1] Instead, the DSL is processed totally independently from the programmer's Java source, by which its translation result finally will be called – in so far similar to other visitor generators [9], [12], and [18]. But umod does not only generate visitor code, but the complete class sources which make up the model definition, including constructors and setter methods, both with null check, visualisation, de/serialization, uncurrying and implicit creation of nested containers, etc.

The generated code of the different variants of visitors carries out the fundamental operations only: traversal and cycle detection anyhow, plus the tedious task of clone generation and management by the rewriters. All other functionality must be added by the programmers explicitly, by the host language means they are familiar with.

Nevertheless, the theory of Strategic Progamming can sensibly be applied also to the visitors of umod, for classification and clarification of its relations to other concepts. Applying the check list from [6], p. 171, yields:

Genericity Being Java class definitions, visitors can take part in the standard Java type parametrization, and can be constructed for models which are themselves parametric. (Both is currently not yet implemented, but does not impose fundamental problems.) A generic handling is *not* possible for *field names*. Nevertheless, inheritance, as discussed below, has turned out sufficient for adaptation and re-use of visitors with variants of models.

Specificity Since the activities are defined by genuine host language code operating on statically typed arguments, all their details are accessible.

Compositionality Being Java class definitions, umod visitors are *not* compositional in the strictest sense, as demanded for strategies. But they profit from the host language's *inheritance* rules along two axes: The generated code calls as default the visiting methods of the argument's superclass, and visitors can be derived from visitors. This allows their refinement, heavily employed in programming practice, see for instance Figure 6.[2]
Visitors are normal objects and thus first class citizens: they can be passed as arguments and used by other visitors. This is a weak form of compositionality, as usual in object oriented languages, see Figure 7.

Traversal The sequential order of the fields on one level of class definition can/must be specified explicitly by the user, see Figure 1. These sequences are concatenated starting with the most specific class upwards to the most general. The content of Java collections is visited by the standard iterators. For instances, the traversal plan `1` in Figure 1 will for each instance of `B2` first visit the objects of type `D` from the map in field `b2`, sorted by the keys, then the reference to `b2b`, if not null, and finally all `B1`s contained in `a1`.
Alternatively, the order can always be overridden by host language means, see Figure 8.

Partiality Any umod visitor can always be applied to any Java object by the method `visit(Object o)`, which by default does nothing. All activities are controlled by the type (= Java class) of the visited model component. In particular, visitor source text is robust against extensions and re-organisations of the model definition.

First-class means that "they can be named, can be passed as arguments, etc." Being Java classes and objects, Visitors are first-class in this sense.

[2] The complete, runnable source code of these examples will be available at `http://bandm.eu`.

```
class Guarded extends MyVisitor {
  @Override public void visit(B2 b) {
    if (b.b2.size()>10)
      super.visit(b);
  }
}
```

Figure 6 Injection of a further guard into some Visitor `MyVisitor` of model `M`.

The classification grid from the section "Rich Variation Points" in [6] is especially useful for identifying the use cases which could be better supported:

Transformation vs. Query Any kind of **query** can be carried out by our basic variant of visitor with *register variables*, see Figure 9.

For transformations there are dedicated variants called `REWRITER` and `COREWRITER`. The former can deal with sharing, internally and between input and output. The latter can deal with cycles transparently. The programmer is responsible for the correct choice, see Figure 10.

Single vs. Cascased Traversal "Cascaded" / "nested" application of visitors is possible by host language means, see Figures 7.

Top-Down vs. Bottom-Up Traversal This is supported by the `MULTIPHASE` visitor which allows independent definitions of the methods `pre` and `post`, which are called before and after descending, resp. The generated code calls as default the visiting methods of the argument's superclass,

Depth-First vs. Breadth-First Rraversal The usual implementation for visitors is recursive descent and thus depth-first search. If required, the code generator could be extended for breadth-first traversal. Emulation with the current umod code is inconvenient.

Left-to-Right Traversal and vice versa See the discussion of "Traversal" above, and Figure 8.

Types vs. General Predicates as Milestones "Milestones" in this context comes from "Adaptive Programming" [6], is used like "guards" in SP, and decides which model elements shall be processed or not. Umod realizes types = Java classes as static and declarative guards; the programmer can add dynamic and imperative guards by calculated values, see Figure 6.

Full vs. Single-Hit vs. Cut-Off Traversal Figures 11 and 12 show two ways for preemptive termination of a traversal. The first uses the exception mechanism of the host language, while the second completes all pending activities in the normal way, supressing all further descents by a built-in guard variable. The costs of both solutions must be judged by the programmer.

Fixpoint by Equality Test vs. Fixpoint by Failure Applying a umod rewriter means to perform the specified work once. There is no support for automatic fixpoint iteration.

Local Choice vs. Full Backtracking vs. Explicit Cut Umod rewriters support local choice: the commitment to rewritten sub-models can be mixed with branching control flow. Full backtracking is not supported. However, combining local choice with compositionality (sub-rewriters) is fairly expressive.

Traversal With Effects (Accumulation, Cloning, etc.) All effects are under control of the programmer, see the discussions above.

```
class Main extends Visitor {
  public Main(Visitor sub){ this.sub = sub;}
  final Visitor sub ;
  @Override public void visit(B2 b) {
    if (b.b2.size()>10)
      sub.visit(b);
    super.visit(b);
  }
}
```

Figure 7 Visitor calling another visitor.

```
class MyVistor extends Simple {
  @Override public void descend(B2 b) {
    // no call to super.descend(b)
    match(b.b2b); match(b.a); match(b.b2);
  }
  @Override public void descend(B1 b) {
    for (int i = b.b1b.size()-1; i>=0; i-=2)
      match(b.b1b.get(i));
    descend((A)b1);
  }
}
```

Figure 8 Overriding the traversal plan by explicit code.

```
int sum = new Simple() {
  int accu = 0 ;
  public int process(Object a) {
    match(a); return accu;
  }
  @Override public void visit(D d) {
    accu += D.d;
  }
}.process(rootElement);
```

Figure 9 Query realized with a umod visitor.

```
A copy = new Rewriter() {
  final D d17 = new D(17);
  @Override public void visit(D d) {
    replace(d17);
  }
}.rewrite(rootElement);
```

Figure 10 Rewriting realized with a umod visitor.

```
new Visitor {
  class Ex extends RuntimeException{}
  int result = -1;
  public int process(Object a) {
    try { match(a); } catch (EX ex) {}
    return result;
  }
  @Override public void visit(D d) {
    if (d.d > 10) {
      result = d.d;
      throw new Ex();
    }
    super.visit(d);
  }
}.process(rootElement);
```

Figure 11 Terminating a traversal abruptly by an exception.

```
new Visitor2 { // is of MULTIPHASE type
  { hasPre = hasDescend = true; }
  int result = -1;
  public int process(Object a) {
    match(a); return result;
  }
  @Override public void pre(D d) {
    if (d.d > 10) {
      result = d.d;
      hasPre = hasDescend = false;
    }
  }
}.process(rootElement);
```

Figure 12 Terminating a traversal by switching off the descending mechanism.

6 Conclusion

We have demonstrated how the optimization of visitors based on a declarative data model, namely by pruning of unaffected node types according to a static analysis, can be implemented in a purely object-orientied spirit, by transparently subclassing programmer code. The novel implementation technique requires sophisticated tool support for runtime bytecode generation, but results in leaner code and compares well against the naïve baseline and the previous dynamically pruning implementation in empirical benchmarks.

Our approach shows that heterogeneous coding styles and technologies can co-operate in a natural, efficient and well-arranged way, namely declarative model definition and source text generation in the large, imperative coding by the programmer in the middle, and automated low-level bytecode generation in the small. The use of global structural knowledge from the model specification for code optimization purposes appears as a nice supplement and complement to the rather local optimization heuristics of current JIT compilers.

Applying two classification grids from Strategic Programming helps to identify the use cases which are not yet optimally supported by umod.

References

1 Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007. doi:10.1007/978-3-540-73449-9_5.

2 Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition)*. W3C, 2006. URL: http://www.w3.org/TR/2006/REC-xml11-20060816/.

3 Alcino Cunha and Joost Visser. Transformation of structure-shy programs: applied to xpath queries and strategic functions. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, pages 11–20. ACM, 2007. doi:10.1145/1244381.1244385.

4 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.

5 Ralf Lämmel. Scrap your boilerplate with xpath-like combinators. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 137–142. ACM, 2007. doi:10.1145/1190216.1190240.

6 Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic programming meets adaptive programming. In William G. Griswold and Mehmet Aksit, editors, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003, Boston, Massachusetts, USA, March 17-21, 2003*, pages 168–177. ACM, 2003. doi:10.1145/643603.643621.

7 Markus Lepper and Baltasar Trancón y Widemann. Optimization of visitor performance by reflection-based analysis. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations, ICMT 2011*, volume 6707 of *LNCS*, pages 15–30. Springer, 2011. doi:10.1007/978-3-642-21732-6_2.

8 Markus Lepper and Baltasar Trancón y Widemann. Rewriting object models with cycles and nested collections. In *ISoLA 2014, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 445–460. Springer-Verlag, 2014. doi:10.1007/978-3-662-45234-9_31.

9 Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004. doi:10.1145/973097.973102.

10 Ralf Lämmel, Eelco Visser, and Joost Visser. The essence of strategic programming. Unpublished, January 2002. URL: https://www.researchgate.net/publication/277289331_The_Essence_of_Strategic_Programming.

11 Aziz Nanthaamornphong and Rattana Wetprasit. Evaluation of the visitor pattern to promote software design simplicity. *Jurnal Teknologi*, 77(9):61–77, November 2015. doi:10.11113/jt.v77.6186.

12 Johan Ovlinger and Mitchell Wand. A language for specifying recursive traversals of object structures. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 70–81, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/320384.320391.

13 Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *International Computer Software and Applications Conference (Compsac '98)*. ieee, 1998. doi:10.1109/CMPSAC.1998.716629.

14 Tanumoy Pati and James H. Hill. A survey report of enhancements to the visitor software design pattern. *Software Practice and Experience*, 44(6), 2014. doi:10.1002/spe.2167.

15 Dmitry Petrashko, Ondrej Lhotak, and Martin Odersky. Miniphases: compilation using modular and efficient tree transformations. In *Proc. PLDI 2017*, pages 201–216. ACM, 2017. doi:10.1145/3062341.3062346.

16 Baltasar Trancón y Widemann and Markus Lepper. Improving the performance of the Paisley pattern-matching EDSL by staged combinatorial compilation. In *Declarative Programming and Knowledge Management*, volume 12057 of *LNAI*, pages 268–285. Springer, 2019. doi:10.1007/978-3-030-46714-2.

17 Baltasar Trancón y Widemann and Markus Lepper. LLJava live at the loop – a case for heteroiconic staged meta-programming. In *Proc. MPLR 2021*, pages 113–126, 2021. doi:10.1145/3475738.3480942.

18 Thomas VanDrunen and Jens Palsberg. Visitor-oriented programming. In *Proc. FOOL-11*, 2004.

19 W3C. *Scalable Vector Graphics (SVG) 1.1*, 2nd edition, 2011. URL: http://www.w3.org/TR/SVG11/.

20 W3C HTML Working Group. *XHTML 1.0 The Extensible HyperText Markup Language*, 2 edition, 2002. W3C Recommendation. URL: http://www.w3.org/TR/2002/REC-xhtml1-20020801.