Compositional Language Engineering Using Generated, Extensible, Static Type-Safe Visitors

Robert Heim[1(✉)], Pedram Mir Seyed Nazari[1], Bernhard Rumpe[1,2], and Andreas Wortmann[1]

[1] Software Engineering, RWTH Aachen University, Aachen, Germany
heim@se-rwth.de
[2] Fraunhofer FIT, Aachen, Germany
http://www.se-rwth.de
http://www.fit.fraunhofer.de

Abstract. Language workbenches usually produce infrastructure to represent models as abstract syntax trees (AST) and employ processing infrastructure largely based on visitors. The visitor pattern suffers from the expression problem regarding extensibility and reuse. Current approaches either forsake static type safety, require features unavailable in popular object-oriented languages (e.g., open classes), or rely on procedural abstraction and thereby give up the object-oriented data encapsulation (the AST) itself. Our approach to visitors exploits knowledge about the AST and generation of statically type-safe external visitor interfaces that support extensibility in two dimensions: (1) defining new operations by implementing the interface and (2) extending the underlying data structure, usually without requiring adaptation of existing implemented visitors. We present a concept of visitor development for language engineering that enables an adaptable traversal and provides hook points for implementing concrete visitors. This approach is applicable to single DSLs and to language composition. It thus enables a transparent, easy to use, and static type-safe solution for the typical use cases of language processing.

Keywords: Visitor pattern · Compositional language engineering · Language workbenches

1 Introduction

Many language workbenches (LWBs) employ context-free grammars (CFGs) to describe modeling languages [4]. From these grammars, LWBs derive the abstract syntax of languages in form of abstract syntax trees (ASTs). Parsers process models into instances of the ASTs. Processing ASTs requires to define traversal algorithms on the underlying tree data structure. Separating these algorithms from operations that perform on the AST liberates from reimplementing traversal strategies for different operations. The visitor design pattern [6] enables such separation by providing a traversal definition and `visit` methods that act as

A. Wąsowski and H. Lönn (Eds.): ECMFA 2016, LNCS 9764, pp. 67–82, 2016.
DOI: 10.1007/978-3-319-42061-5_5

hook points for AST node instances during traversal of the AST. For each visited AST node the traversal algorithm calls the appropriate `visit` method which then performs the specified operations. Thereby, the visitor pattern facilitates to *add new operations* on data structures, while *adding new types* (for instance when new productions are added to the related languages' AST) to data structures is effortful. In the original visitor pattern, visitor implementations must be extended with an additional `visit` method for each added data type. The problem of supporting extensibility in both dimensions is also known as the *expression problem* [20]. Approaches to it either require features unavailable in popular object-oriented languages (e.g., mixins [5] or open classes [3]), demand advanced type systems [15], forsake static type safety [2,16], or rely on procedural abstraction – and thereby abandon the object-oriented data encapsulation of the AST [14].

We contribute a concept to generate visitor infrastructures from CFGs that support language engineers with *statically type-safe* interfaces to work on the AST for model analysis and transformation. The visitor infrastructure facilitates development of reusable model processing infrastructures for single languages and supports integration of visitor implementations for combined languages. It is based on the single-dispatch language Java as the most popular object-oriented language[1] and we demonstrate a realization within the language workbench MontiCore [10][2].

Section 2 introduces MontiCore and its language processing mechanisms. Afterwards, Sect. 3 presents the visitor infrastructure generation approach for single languages and Sect. 4 for combined languages. Section 5 discusses our approach, before Sect. 6 debates related work and Sect. 7 concludes.

2 Preliminaries

MontiCore [10] is a language workbench for the engineering of compositional modeling languages. It provides an extended CFG format for integrated specification of concrete and abstract syntax. From these grammars, MontiCore derives the Java AST classes of a language and its parsers instantiate these classes to represent processed models.

We present quintessential concepts of MontiCore's grammar by the example of the grammar for class diagrams depicted in Fig. 1: it begins with the keyword `grammar` (l. 1), followed by its name and a body in curly brackets. The body contains productions to describe the structure of the `CD` language. The grammar's main production is `CDDef`, the definition of a class diagram consisting of the model keyword `classdiagram` (everything in quotation marks is part of the concrete syntax only), a name, classes, and associations (l. 2). The production `Name` is part of

[1] http://www.tiobe.com/index.php/tiobe_index.

[2] MontiCore is open source (https://github.com/MontiCore/monticore) and running visitor examples as described in this paper are available online (http://www.se-rwth.de/materials/mcvisitors/).

MCG

```
1 grammar CD {
2   CDDef = "classdiagram" Name "{" ( Class | Association )* "}";
3   Class = "class" Name ( "extends" super:Name)? "{" Method* "}";
4   interface Method;
5   MethodSignature implements Method = type:Name Name ParameterList ";";
6   ParameterList = "(" (Parameter || ",")* ")";
7   Parameter = type:Name Name;
8   // associations ...
9 }
```

Fig. 1. An exemplary MontiCore grammar for definition of simplified class diagrams.

MontiCore primitives. The body of a class diagram is delimited by curly brackets and contains arbitrary many (denoted by the star operator `*`) associations and classes in arbitrary order (via disjunction operator `|`). The production for classes begins with the model keyword `class`, followed by a name, optionally followed by the keyword `extends` with another name, and a body delimited by curly brackets (l. 3). The body contains arbitrary many instances of `Method`, which is an interface production (l. 4) that is implemented by the production `MethodSignature` (l. 5). Thus, `MethodSignature` can be used whenever a `Method` is required – for instance in the `Class` production (l. 3). This enables to add new implementing productions to the grammar a-posteriori without modifying the interface production itself, which is essential for language inheritance (see Sect. 4). The production `MethodSignature` consists of a type, a name, and a list of parameters. The `ParameterList` (l. 6) consists of comma-separated `Parameter` instances in brackets[3], where each `Parameter` (l. 7) has a type and a name. From the grammar depicted in Fig. 1, MontiCore generates an AST node class for each production. Figure 2 depicts these classes. The names of AST classes begin with "AST" and are followed by the name of the production they are derived from (such as `ASTCDDef`). The non-terminal `Name` results to a field `name` of Java type `String` in the AST class. MontiCore enables to define the names of AST fields, as for example via `type:Name` in the production `Parameter`. This results in the field `String type` of `ASTParameter`. References to other non-terminals in a production become associations between the corresponding AST classes. They have the same cardinalities as specified in the grammar. For instance, `CDDef` uses the non-terminal `Class`, thus `ASTCDDef` is associated to `ASTClass`. For the interface production `Method`, MontiCore generates the interface `ASTMethod`. As the production `MethodSignature` implements the interface production `Method`, its AST class `ASTMethodSignature` implements `ASTMethod` as well.

3 Generating the Visitor Pattern as DSL Infrastructure

We exploit knowledge on the automated generation of AST node classes to produce visitor interfaces for CFGs. These interfaces prescribe separate methods

[3] For a production `P` and a separator `s` the expression `(P || "s")*` denotes an arbitrary count of `P` separated by `s`. There is no `s` at the end.

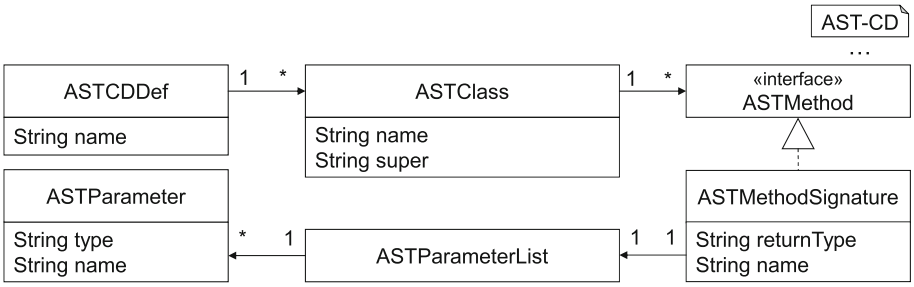

Fig. 2. The AST node classes MontiCore derives from the `CD` grammar of Fig. 1.

for traversal and visiting that are connected by methods to handle their interaction. For each production of the CFG, the generated visitor interface yields the methods `visit`, `endVisit`, and `handle` to handle operations on these nodes. For concrete class nodes, we also generate a `traverse` method for subtree traversal. All methods have default implementations[4] and hence do not require an implementation. Instead, they are best understood as hook points.

Derived from the `CD` language presented in Fig. 1, the generated visitor interface `CDVisitor` is as depicted in Fig. 3. The methods `visit` and `endVisit` for an AST node type enable to process instances of that node before and after its traversal, respectively. By default they do nothing and hence the default implementation is an empty method body (omitted in Fig. 3).

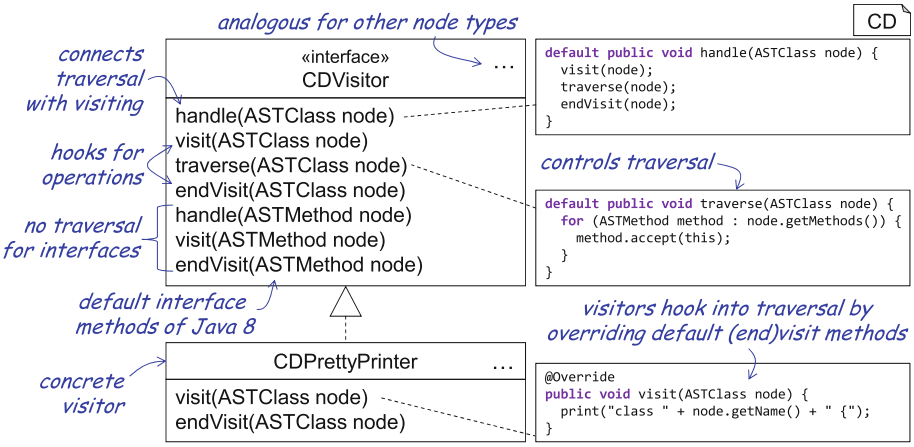

Fig. 3. The `CDVisitor` interface generated for the `CD` language and a concrete visitor.

Reusing a traversal algorithm can be achieved by implementing it in a super type that is meant to be inherited by concrete visitor implementations.

[4] Default implementations are available since Java 8.

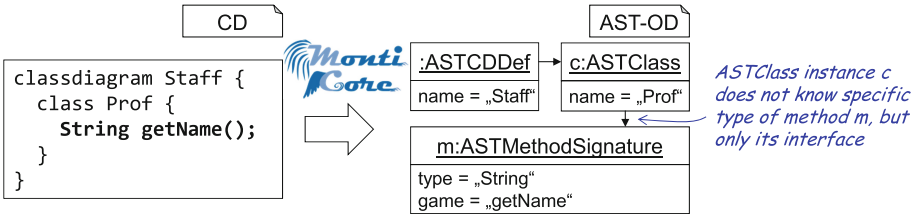

Fig. 4. A `CD` model and the resulting AST instance with an `ASTClass` not knowing that `m` is of type `ASTMethodSignature`.

By providing such an implementation in a class it requires software components that are based on the visitor infrastructure to extend it. This, however, contradicts flexibility in software design during language engineering as the component then is defined as a specialization of the visitor infrastructure and multi-inheritance is not supported in Java. Also, with respect to semantics, it is better to leave the specialization characteristics open to the language engineer. For example, a pretty printer could specialize an abstract pretty printer that defines constants regarding whitespaces. To this effect, MontiCore's visitor infrastructure is not based on classes but on interfaces which enable easier integration with other software components. This becomes even more relevant in language composition (cf. Sect. 4), when concrete visitors require extension and composition.

The top-right side of Fig. 3 illustrates the default implementations of the `CDVisitor` interface: The `handle` method takes care of visiting and traversal. The `traverse` method implements a climb-down strategy (e.g., order) to traverse the children. This separation enables to easily change the traversal order in subclasses while shielding the developer from involuntary changing the overall traversal strategy or missing to call the `visit` methods. The `CDVisitor` interface does not provide traversal methods for AST nodes of interface types derived from interface productions of the CFG, because interfaces do not have children. AST nodes of interface types (e.g., `ASTMethod`) are never instantiated. Instead, MontiCore enforces that each CFG contains at least one implementation for each interface production. Consequently, there always exists a concrete AST node class that implements the interface. Figure 4 elucidates this with a `CD` model (left) and the resulting AST instance (right). The `CD` model contains the class `Prof`, which has a method `getName()`. Although the `ASTClass` instance `c` has an instance of `ASTMethodSignature`, the class `ASTClass` knows this instance only via its interface `ASTMethod`.

For traversing the children of the `ASTClass` associated `ASTMethod` instances have to be considered. To call the most specific `handle` method for each of the children, a mechanism to calculate the most specific type of each child is required. This mechanism should not make use of type-introspection, but instead dispatch dynamically by itself. To simulate double dispatch in the single dispatch language Java, MontiCore generates an interface with a single `accept` method responsible

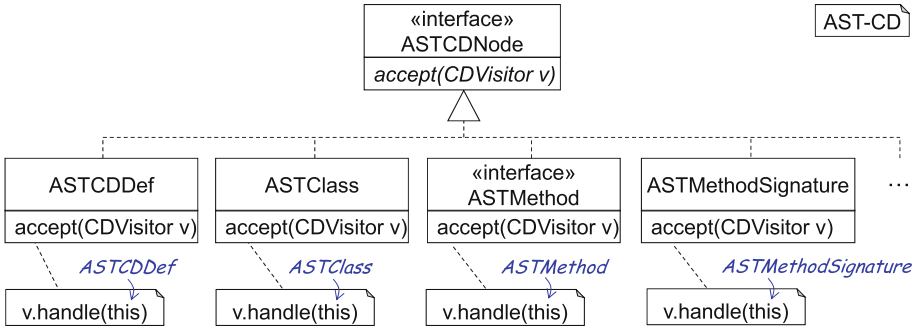

Fig. 5. All AST node types implement the `accept` method for the language's visitor.

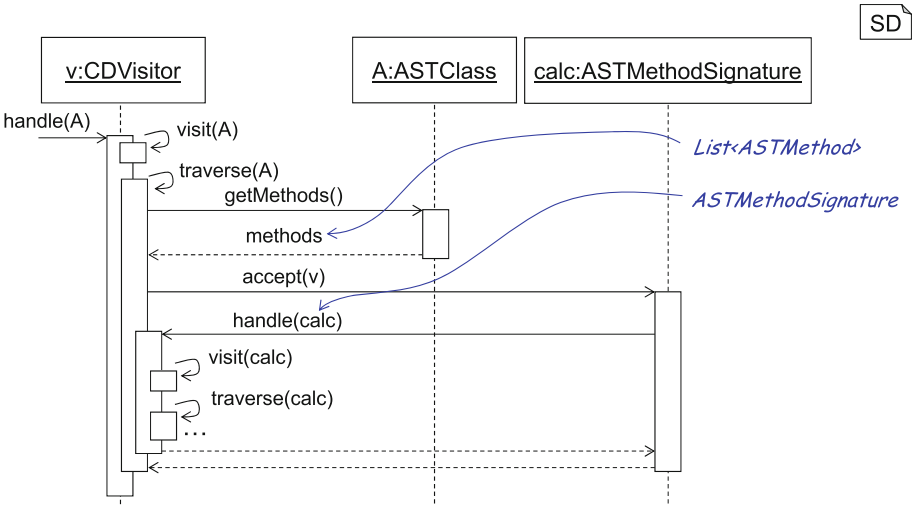

Fig. 6. Traversing a class requires identifying the runtime type of the methods in order to call the most specific `handle` methods. To this effect, a double dispatch is simulated.

for calling the visitors' `handle` methods with the most specific type. Figure 5 shows this interface and its implementations. The purpose of the `accept` methods is to dispatch to the most specific `handle` method of the given `CDVisitor` for each AST node type. Consequently, MontiCore generates AST nodes in such a way that they dispatch on the language's visitor to the `handle` method with themselves as argument (i.e. `this`). Although the implementation looks similar among the AST types, it differs in the specific type of `this`. Thereby, the most specific `handle` method is called. The sequence diagram in Fig. 6 illustrates the double dispatching of `ASTMethod`s when handling the `ASTClass` shown in Fig. 4.

Until now, it was assumed that defining operations on a language's AST always relates to the most specific node types. This, however, is not true: For example, defining an operation to count all methods should be based on the

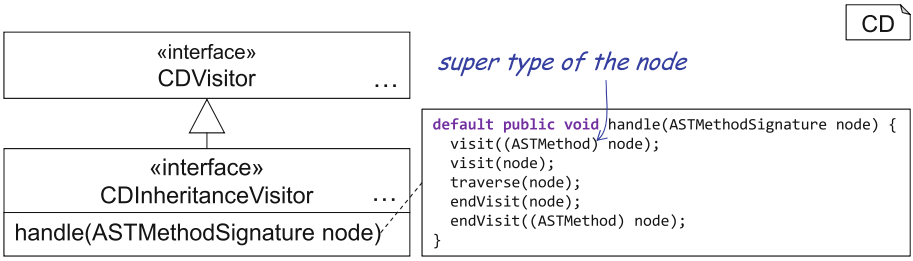

Fig. 7. The inheritance visitor visits nodes also in all their super types.

`visit` method for `ASTMethod`, but the double dispatching always calls the most specific `visit` hooks during traversal. Hence, such a method counter must hook into all `visit` methods for nodes that refine `Method`. While in the example only one production (`MethodSignature`) is effected, MontiCore enables implementing interface productions multiple times. All specific `visit` hooks would require to implement the same code. To prevent such code repetitions, MontiCore provides an extended visitor interface called *inheritance visitor* (see Fig. 7). While the formerly described visitor interface only calls `visit` hooks for the specific types, the inheritance visitor exploits knowledge about the grammars and the relation between their rules to call all `visit` methods that a node type is applicable for. This requires casting the nodes (cf. Fig. 7) in the `handle` methods, but is statically type-safe and more importantly generated. By extending this inheritance visitor instead of the common one, operations can be defined on all node types.

Implementing visitors using the presented infrastructure is straightforward and statically type-safe. It does not require knowledge about the double dispatch mechanism when depth-first traversal suffices. Adapting traversal requires calling `accept` methods and, hence, some knowledge about the visitor pattern.

For example, pretty printers, which transform an AST instance to a string representation, often are implemented as visitors. Figure 3 (bottom) depicts an excerpt of a pretty printer for the `CD` language. The `CDPrettyPrinter` implements the visitor interface `CDVisitor` and thus inherits the default implementations for traversing and handling a `CD` AST. By overriding `visit` and `endVisit` methods it hooks into the default traversal to execute operations on specific nodes.

4 The Visitor Pattern for Compositional Languages

In software language engineering, non-invasive reuse of languages and related infrastructures can greatly improve development. In this context, extending and composing visitors is of particular interest. The following section describes an extension of the `CD` language and demonstrates how its visitors can be easily reused in the sub language. Afterwards, we describe how visitors of multiple super languages can be composed.

MCG

```
1 grammar CDWithConstructor extends CD {
2   ConstructorSignature implements Method = Name ParameterList ";"
3 }
```

Fig. 8. The `CDWithConstructor` language extends the `CD` language by constructors.

4.1 Extending Concrete Visitors for Language Extension

MontiCore supports language extension [8], where sub languages inherit the productions of their parents. For instance, Fig. 8 depicts the `CDWithConstructor` language, which extends `CD` (l. 1) and introduces constructors (l. 2). Constructors implement the interface production `Method`, enabling to use these whenever an instance `Method` is required.

The resulting AST is illustrated in Fig. 9. The AST nodes `ASTMethod` and `ASTParameterList` (not shown in the figure) of `CD` are reused by the new `ASTConstructorSignature` AST node type of `CDWithConstructor`. MontiCore also produces the interface `CDWithConstructorVisitor` for the sub language. Generated visitor interfaces extend the visitor interfaces of all their super languages to inherit their default implementations. Here, the visitor interface `CDWithConstructorVisitor` extends `CDVisitor` and adds default implementations for the new node type `ASTConstructorSignature`. Using this inheritance relation all default implementations of the super language's visitor interface are reused. Hence, when implementing a visitor for the new language the aggregated default implementations are available. Consequently, the pretty printer for the sub language (`CDWithConstructorPrettyPrinter`) can be implemented by extending the `CDPrettyPrinter` of the super language (l. 1 of Fig. 10). It thereby reuses the pretty printing for all nodes of the super language and only adds pretty printing for constructors. For example, the reused production `ParameterList` (l. 2 in Fig. 8) is printed using the inherited implementation of `CDPrettyPrinter`.

Visitors of a super language are unaware of new AST node classes introduced in sub languages. For example, the pretty printer of the `CD` language is able to handle `ASTClass` nodes. An AST instance of the sub language (i.e. a model) reuses the exact same class and hence it is possible to hand this model to the `handle(ASTClass)` method of the `CDPrettyPrinter`. However, double dispatching the children of type `ASTMethod` to their specific types is only possible for types defined in the super language itself. In case of a `ASTConstructorSignature` the most specific type known by the super language's visitor is `ASTMethod`. Hence `handle(ASTMethod)` would be executed. This is unintuitive, because the most specific type at runtime is `ASTConstructorSignature`. Consequently, MontiCore forbids directly applying a visitor implemented for a specific language on any of its sub languages. A compiler cannot statically check this which leads to MontiCores convention to only run visitors on their own language. This also means that a concrete visitor implementation must be adjusted to be reused on a sub language, even if nothing changes. This, however, is very easy and requires minimal

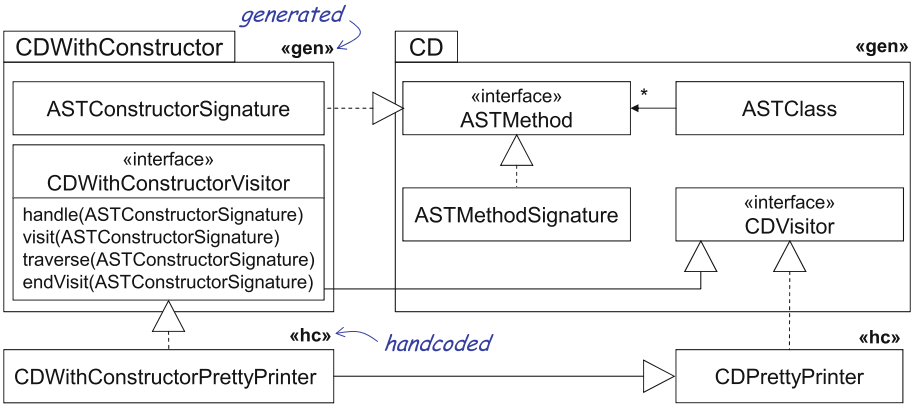

Fig. 9. A subset of `CD`'s and `CDWithConstructor`'s language infrastructure.

Java

```
1  class CDWithConstructorPrettyPrinter extends CDPrettyPrinter implements
       CDWithConstructorVisitor {
2    @Override
3    public void visit(ASTConstructorSignature node) {
4      print(node.getName());
5    }
6    @Override
7    public void endVisit(ASTConstructorSignature node) {
8      print(";");
9    }
10 }
```

Fig. 10. Extending a concrete visitor of a single super language is straightforward. Only the additional nodes must be considered. For nodes of the super language, the super language's pretty printer `CDPrettyPrinter` is reused.

glue code. For example, reusing the pretty printer without adding anything is as easy as defining a class similar to Fig. 10, but with an empty class body.

Another issue with this implementation is that extension of AST types in sub languages results in an unexpected `accept` call. For example, the node type `ASTConstructorSignature` implements the interface `ASTMethod` and thereby inherits `CD`'s `accept` method with the visitor parameter being of static type `CDVisitor`. The `CDWithConstructorVisitor` calls this method when traversing the `ASTClass` instead of the intuitively expected `accept` method with parameter of static type `CDWithConstructorVisitor` This occurs because the inherited traversal is defined in the super language. This traversal calls `accept` on the `ASTMethod` children. While the type of the child is dispatched dynamically, choosing the `accept` method within it uses method overloading based on the static type of the visitor that defines traversal (i.e. `CDVisitor`). Consequently, the wrong `accept` method is executed. This cannot be solved by simulating another double dispatch, because the super language never statically is aware of types of a sub language. Hence, this is a limitation of our approach. Our solution to

MCG

```
1 grammar Automaton {
2   Automaton =  "automaton" Name "{" (State | Transition)* "}";
3   State = "state" Name ";";
4   Transition = from:Name "-" input:Name ">" to:Name ";";
5 }
```

Fig. 11. The MontiCore grammar for a language to model automata.

this is overriding the `accept` method for visitor interfaces of super languages in affected AST types (e.g., `accept(CDVisitor)` of `ASTConstructorSignature`). The generated implementation checks at runtime, whether the given visitor stems from the correct sub language. In this case, the call is delegated to the correct `accept` method by casting the visitor to the specific type. We argue, that this solution still is a good tradeoff between static type safety and flexibility in reuse, because (a) the AST types and this mechanism are generated and (b) the visitor interfaces remain statically type-safe. Implementing as well as reusing concrete visitors do neither require manual type-introspection nor casts. Also, the former problem does not occur when reusing non-terminals of super languages as done with `ParameterList`. Here, traversal (of `ASTConstructorSignature`) resides in the sub language and the generated traversal is aware of relations to the super language and consequently is statically type-safe.

4.2 Composing Concrete Visitors During Language Embedding

MontiCore enables language embedding by inheritance of grammars and provides adaptation mechanisms on a symbolic level [8,9]. These adaptation mechanisms depend on visitor composition for, e.g., building symbol tables. The same holds true for other software components such as validation or pretty printers. In this section we first show the embedding of automata into the `CD` language on the grammar level and then demonstrate the resulting visitor interfaces as well as composing existing pretty printers of both the `CD` language as well as the pretty printer of the `Automaton` language to a new pretty printer for the integrated language. Figure 11 depicts the `Automaton` language that describes automata (l. 2) using states (l. 3) and transitions (l. 4). We assume an implemented `AutomatonPrettyPrinter` (analogous to `CDPrettyPrinter`) that implements the `AutomatonVisitor` interface to pretty print an automaton.

Figure 12 depicts the MontiCore grammar that embeds automata into class diagrams. Automata are integrated as methods by implementing the interface production `Method` of the `CD` language. Figure 13 shows the resulting structure. The `ASTAutomatonEmbedding` implements the `ASTMethod` interface of the `CD` language and has a `ASTAutomaton` of `Automaton` language as child. The visitor interface of the new language extends both super language's visitor interfaces to inherit their default implementations.

Implementing a pretty printer for this language cannot make use of the approach shown in Sect. 4.1, because Java does not allow multi-inheritance of classes

MCG

```
1 grammar CDWithAutomaton extends CD, Automaton {
2   AutomatonEmbedding implements Method = Automaton;
3 }
```

Fig. 12. The `CDWithAutomaton` language's grammar extends the grammars of the languages `CD` and `Automaton` and provides a production that embeds automata as methods.

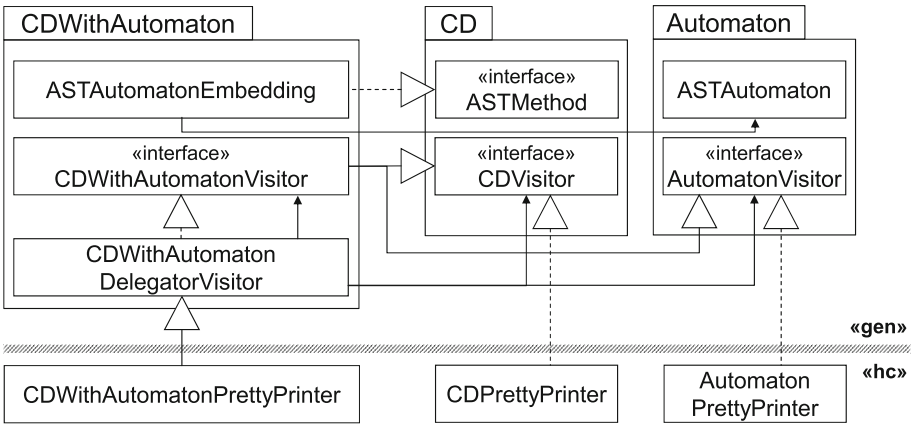

Fig. 13. A subset of the resulting infrastructure when embedding the `Automaton` language into the `CD` language.

and hence, it is not possible to extend both languages' pretty printers. Instead, MontiCore generates a visitor implementation for the new language, that is capable of composing visitors of all super languages using a delegator pattern (cf. `CDWithAutomatonDelegatorVisitor` in Fig. 13). It provides setters for visitors of all (potentially transitive) super languages. MontiCore exploits knowledge of the AST model by generating this delegator visitor in such a way that it delegates all `handle`, `traverse`, and `visit` calls to the concrete super visitor of the language that the current node stems from. This behavior is adjustable by extending this delegator visitor and overriding the corresponding methods.

As this approach to composition is based on delegation, it requires *inversion of control*. For example, when handling a node the delegator delegates the handling to the visitor that is registered for that specific node. Executing the `visit` hooks and traversal, however, should not directly take place in the delegate, but the delegator should control these operations as well. In MontiCore this inversion of control is called the *realThis pattern*. It describes that in composable objects (such as concrete visitors) the `this` reference should not be used, but instead a `realThis` reference. A composer then can set this `realThis` reference to itself and thereby gain control over all methods within the delegates. Consequently, a composer must implement a super set of the aggregated interfaces of all composed objects, which holds true for the visitor infrastructure as visitor interfaces

```java
public interface CDVisitor {
  public CDVisitor getRealThis();
  public void setRealThis(CDVisitor realThis);

  default public void handle(ASTClass node) {
    getRealThis().visit(node);
    getRealThis().traverse(node);
    getRealThis().endVisit(node);
  }

  default public void traverse(ASTClass node) {
    for (ASTMethod method : node.getMethods()) {
      method.accept(getRealThis());
    }
  }
  // ...
}
```

Fig. 14. To support composition visitors must always use `realThis` instead of `this`.

inherit from the visitor interfaces of all super languages and only visitors of the corresponding languages are meant to be composed.

To enable such a composition the delegates must always use the `realThis` reference. It is accessible through a `getRealThis()` method defined in a super interface. In case of visitors this is the visitor interface. A setter for the `realThis` reference then enables changing it. The composed delegates must implement both methods to manage the `realThis` reference, which initially equals `this`. Figure 14 depicts the actual default implementations of the visitor interface that enable visitor composition by using `getRealThis()` (ll. 6–8, 13).

For transitive delegation (e.g., composing already composed visitors) a delegator must transitively ensure the correct `realThis` reference when itself gets composed. This is achieved by transitively setting `realThis` for all delegates in case that the own `realThis` changes.

Figure 15 depicts an example by composing the concrete pretty printer visitors of `CD` and `Automaton` to a pretty printer for the new language. The composition extends the delegator visitor `CDWithAutomatonDelegatorVisitor` (l. 2) that provides statically typed methods for the composition (used in ll. 4–5).

```java
class CDWithAutomatonPrettyPrinter
                          extends CDWithAutomatonDelegatorVisitor {
  public CDWithAutomatonPrettyPrinter() {
    setCDVisitor(new CDPrettyPrinter());
    setAutomatonVisitor(new AutomatonPrettyPrinter());
    setCDWithAutomatonVisitor(new EmptyCDWithAutomatonVisitor());
  }
}
```

Fig. 15. Implementing a pretty printer for the combined language reuses existing pretty printers of the super languages. For the new AST node `ASTAutomatonEmbedding` nothing is printed, which is why only an empty implementation is chosen (l. 6).

A setter for a visitor of the own language (l. 6) ensures that the new AST node `ASTAutomatonEmbedding` can be traversed. In this example the pretty printer is solely based on the pretty printers of the super languages and does not require any additional output for the embedding production. Hence, an empty visitor is used that only inherits the default implementation from the visitor interface, but does not hook into any of its methods.

5 Discussion

The visitor pattern and its derivatives suffer from the expression problem [20] and so does our approach. Nevertheless, the presented infrastructure makes minimal use of type-introspection that is (a) hidden when implementing and composing visitors and (b) automatically generated and, thus, less error prone. To this effect, our main contribution is enabling language engineers to implement visitors in a statically type-safe fashion. Also, the generated visitor interfaces are semantically bound to the languages to support comprehensibility. Experience with language engineering has shown that depth-first traversal (with the same order of child traversal as they occur in the grammar) is sufficient for most model processing tasks. Hence, handcrafted visitor implementations, which inherit traversal and hooks from the generated interface, usually do not require adjustments. When necessary, traversal can be adapted in specific visitors by overriding the inherited default traversal. Being interface-driven, our approach furthermore does not enforce implementing operations on the AST by extending visitors, which enables developers to use the inheritance relation to flexibly integrate visitors with other software components.

The main limitation of the presented visitor infrastructure resides in overridden traversals. While our approach enables adaptation of traversal by overriding the default implementation, it might require manual adjustments when the CFG of the language changes.

When a non-terminal is added on the right-hand side of a production, the corresponding AST node gets a new child that must be traversed. In this case, manual adaptation of overridden traversals in concrete visitors is required, which a compiler does not identify statically.

However, removing complete productions or changing their names can be statically identified and evolution efforts can be minimized by employing refactoring mechanisms. Nonetheless, our approach is affected by the expression problem as changing the AST of a language might require adaptation of all its visitors. When depth-first traversal is sufficient this rarely occurs, because default implementations are generated.

MontiCore supports overriding non-terminals in sub languages. The full-qualified names of the generated AST nodes include the languages' names and hence are unique. Consequently, the generated visitor interfaces support overridden non-terminals since they are based on the full-qualified names of the corresponding AST nodes as well.

6 Related Work

In contrast to established visitor combinators [18] our solution is statically type-safe in liberating language developers from manually casting generic types (such as `AnyVisitable` [18]) to specific types. Instead, our solution provides a statically type-safe visitor interface that supports visitor composition. It enables implementing a concrete visitor by hooking into a predefined traversal for specific AST nodes, but also enables to adapt the traversal for specific visitors.

The original visitor pattern [6] describes the traversal algorithms as part of the data structure (within the `accept` methods). This prohibits adjusting traversal in specific AST visitors as they all share the same AST implementation. Oliveira [13] distinguishes between *internal visitors* that define traversal within the data structure and *external visitors* that define it in the visitors. Also, the original visitor pattern stores the result of a visitor run as state in the visitor. This *imperative* calculation is distinguished by Oliveira [13] from a *functional* style, where all corresponding methods aggregate and return results. Based on this categorization MontiCore's visitor infrastructure is external and imperative.

Various approaches that solve the underlying expression problem rely on mechanisms not available to popular object-oriented languages: Advanced type systems can solve the expression problem [15] and enable to implement visitors as type-safe reusable components. Other approaches employ mixins [5] or open classes [3] to overcome the expression problem. However, utilizing such features requires to forsake existing language workbench infrastructure and enforces engineers to learn less supported languages.

To circumvent the cyclic dependency between the visitor interface and the data structure the Acyclic Visitor pattern [11] splits all visit methods into their own data-specific visitor interface. Thereby, the different visitor interfaces are semantically bound to a specific data type, but they require type-introspection to cast the generic visitor interface to the specific one. Consequently, this approach is similar to the one in MontiCore, with the difference that MontiCore semantically binds visitor interfaces to a language instead of their concrete nodes.

Another recent solution to the underlying expression problem, that is applicable in common object oriented languages, is given by Object Algebra (OA) [14]. It, however, gives up representation of a language's AST as types. Instead, using constructor overloading the AST for a given model is only implicitly constructed during a concrete calculation on it. It thereby introduces a different approach to language engineering which requires language engineers to change their understanding of DSL implementation in general. Currently, there is limited experience [7,17,21] about OA's main advantages and limitations and hence it is not yet clear whether such investment pays off for language engineers.

Other LWBs, such as Xtext [1], build on the Eclipse Metamodeling Framework (EMF) using Ecore models to describe the AST [12,19]. EMF trees are traversed using tree iterators[5] that require clients to implement the method

[5] http://download.eclipse.org/modeling/tmf/xtext/javadoc/2.9/org/eclipse/xtext/nodemodel/BidiTreeIterator.html.

`getChildren(Object)`, which defines an iterator over all children of the object. The parameter is of the most generic type `Object` and returning one iterator for all children requires to cast them to a common super type as well. Thus, using this infrastructure depends on type-introspection in client-code to differentiate between specific AST node types. Commonly, Xtext-based DSLs use Xtend [1] for implementing code generators. Xtend is a Java dialect that compiles into Java code and claims to enable multi-dispatching. However, the multi-dispatching is implemented using type-introspection in switch statements[6].

7 Conclusion

We demonstrated a concept to derive visitor infrastructures from context-free grammars to support language engineers to work with ASTs for model analysis, transformations, and code generation. It separates AST traversal from operations that hook into the traversal. In order to define new operations on ASTs, it provides language engineers with generated statically type-safe visitor interfaces that foster reuse during language composition and allow for traversal adaptation if required. With the infrastructure being interface-driven, developers may flexibly integrate other software components with concrete visitor implementations. To this effect, we described how a sophisticated combination and extension of software patterns support compositional language engineering and presented a realization in the language workbench MontiCore.

References

1. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham (2013)
2. Carlisle, M.C., Sward, R.E.: An Automatic "Visitor" Generator for Ada. Ada Letters (2002)
3. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for Java. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (2000)
4. Erdweg, S., et al.: The state of the art in language workbenches. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 197–217. Springer, Heidelberg (2013)
5. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and Mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1998)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Professional, Boston (1995)
7. Gouseti, M., Peters, C., van der Storm, T.: Extensible Language Implementation with Object Algebras (Short Paper). SIGPLAN Not. (2014)

[6] https://eclipse.org/xtend/documentation/202_xtend_classes_members.html.

8. Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., Wortmann, A.: Composition of heterogeneous modeling languages. In: Desfray, P., Filipe, J., Hammoudi, S., Pires, L.F. (eds.) MODELSWARD 2015. CCIS, vol. 580, pp. 45–66. Springer, Heidelberg (2015)
9. Hölldobler, K., Nazari, P.M.S., Rumpe, B.: Adaptable symbol table management by meta modeling and generation of symbol table infrastructures. In: Proceedings of the Workshop on Domain-Specific Modeling (2015)
10. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(5), 353–372 (2010)
11. Martin, R.C., Riehle, D., Buschmann, F. (eds.): Pattern Languages of Program Design 3. Addison-Wesley Longman Publishing Co., Inc., Boston (1997)
12. Merkle, B.: Textual modeling tools: overview and comparison of language workbenches. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (2010)
13. Oliveira, B.C.S.: Genericity, Extensibility and Type-Safety in the Visitor Pattern. Oxford University, Oxford (2007)
14. Oliveira, B.C.D.S., Cook, W.R.: Extensibility for the masses: practical extensibility with object algebras. In: Proceedings of the 26th European Conference on Object-Oriented Programming (2012)
15. Oliveira, B.C.D.S., Wang, M., Gibbons, J.: The visitor pattern as a reusable, generic, type-safe component. In: SIGPLAN Notices (2008)
16. Palsberg, J., Jay, C.B.: The essence of the visitor pattern. In: Proceedings of the 22Nd International Computer Software and Applications Conference (1998)
17. Rendel, T., Brachthäuser, J.I., Ostermann, K.: From object algebras to attribute grammars. In: SIGPLAN Notices (2014)
18. Visser, J.: Visitor combination and traversal control. In: SIGPLAN Notices (2001)
19. Vlter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages (2013). dslbook.org
20. Torgersen, M.: The expression problem revisited. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 123–146. Springer, Heidelberg (2004)
21. Zhang, H., Chu, Z., Oliveira, B.C.D.S., Storm, T.V.D.: Scrap Your boilerplate with object algebras. In: SIGPLAN Notices (2015)