Object Grammars:
Compositional & Bidirectional Mapping Between Text and Graphs

Tijs van der Storm[1,2], William R. Cook[3] and Alex Loh[3]

[1] Centrum Wiskunde & Informatica (CWI)
[2] INRIA Lille Nord Europe
[3] University of Texas at Austin

Abstract. *Object Grammars* define mappings between text and object graphs. *Parsing* recognizes syntactic features and creates the corresponding object structure. In the reverse direction, *formatting* recognizes object graph features and generates an appropriate textual presentation. The key to Object Grammars is the expressive power of the mapping, which decouples the syntactic structure from the graph structure. To handle graphs, Object Grammars support declarative annotations for resolving textual names that refer to arbitrary objects in the graph structure. Predicates on the semantic structure provide additional control over the mapping. Furthermore, Object Grammars are compositional so that languages may be defined in a modular fashion. We have implemented our approach to Object Grammars as one of the foundations of the Ensō system and illustrate the utility of our approach by showing how it enables definition and composition of domain-specific languages (DSLs).

1 Introduction

A grammar is traditionally understood as specifying a language, defined as a set of strings. Given such a grammar, it is possible to recognize whether a given string is in the language of the grammar. In practice it is more useful to actually parse a string to derive its meaning. Traditionally parsing has been defined as an extension of the more basic recognizer: when parts of the grammar are recognized, an action is invoked to create the (abstract) syntax tree. The actions are traditionally implemented in a general-purpose programming language.

In this paper we introduce *Object Grammars*: grammars that specify mappings between syntactic presentations and graph-based object structures. *Parsing* recognizes syntactic features and creates object structures. Object grammars include declarative directives indicating how to create cross-links between objects, so that the result of parsing can be a graph. *Formatting* recognizes object graph features and creates a textual presentation. Since formatting is not uniquely specified, an Object Grammar can include formatting hints to guide the rendering to text.

The second problem addressed in this paper is modularity and composition of Object Grammars. Our goal is to facilitate construction of domain-specific languages (DSLs). It is frequently desirable to reuse language fragments when creating new languages. For example, a state machine language may require an expression sub-language to represent constraints, conditions, or actions. In many cases the sublanguages may also

be extended during reuse. We present a generic merge operator on that covers both reuse and extension of languages.

The contributions of this paper can be summarized as follows:

- We introduce *Object Grammars* to parse textual syntax into object graphs.
- Cross references in the object structure are resolved using *declarative paths* in the Object Grammar.
- Complex mappings can be further controlled using *predicates*.
- We show that Object Grammars are both *compositional* and *bidirectional*.
- The entire system is self-describing.

The form of Object Grammars presented in this paper is one of the foundations of Ensō, a new programming system for the definition, composition and interpretation of external DSLs[4].

2 Object Grammars

In domain-specific modeling, a software system is modeled using a variety of dedicated languages, each of which captures the essence of a single aspect. In textual modeling [27], models are represented as text, which is easy to create, edit, compare and share. To unlock their semantics, textual models must be parsed into a structure suitable for further processing, such as analysis, (abstract) interpretation or code generation.

Many domain-specific models are naturally graph structured. Well-known examples include state machines, workflow models, petri nets, network topologies and grammars. Nevertheless, traditional approaches to parsing text have focused on tree structures. Context-free grammars, for instance, are conceptually related to algebraic data types. As such, existing work on parsing is naturally predisposed towards expression languages, not modeling languages. To recover a semantic graph structure, textual references have to be resolved in a separate name-analysis phase.

Object Grammars invert this convention, taking the semantic graph structure (the model) as the primary artifact rather than the parse tree. Hence, when a textual model is parsed using an Object Grammar, the result is a graph. Where the traditional tree structure of a context-free grammar can be described by an algebraic data type, the graphs produced by object grammars are described by a *schema*. In Ensō, a schema is a class-based information model [26], similar to UML Class Diagrams [29], Entity Relationship Diagrams [8] or other meta-modeling formalisms (e.g., [5]).

There is, however, an *impedance mismatch* between grammars (as used for parsing), and object-oriented schemas (to describe structure). Previous work has suggested the use of one-to-one mappings between context-free grammar productions and schema classes [1, 40]. However, this is leads to tight coupling and synchronization of the two formats. A change to the grammar requires a change to the schema and vice versa. Object Grammars are designed to bridge grammars and schemas without sacrificing flexibility on either side. This bridge works both ways: when parsing text into object model and when formatting a model back to text. An Object Grammar specifies a mapping between

[4] `http://www.enso-lang.org`

syntax and object graphs. The syntactic structure is specified using a form of Extended Backus-Naur Form (EBNF) [41], which integrated regular iteration and optional symbols into BNF. Object Grammar extend BNF with constructs to declaratively construct objects, bind values to fields, create cross links and evaluate predicates.

2.1 Construction and field binding

The most fundamental feature of Object Grammars is the ability to declaratively construct objects and assign their fields with values taken from the input stream. The following example defines a production rule named `P` that parses the standard notation `(x, y)` for cartesian points and creates a corresponding `Point` object.

```
P ::= [Point] "(" x:int "," y:int ")"
```

The production rule begins with a *constructor* `[Point]` which indicates that the rule creates a `Point` object. The literals `"("`, `","` and `")"` match the literal text in the input. The *field binding* expressions `x:int` and `y:int` assign the fields `x` and `y` of the new point to integers extracted from the input steam. The classes and fields used in a grammar must be defined in a *schema* [26]. For example, the schema for points is:

```
class Point   x: int  y: int
```

Any pattern in a grammar can be refactored to introduce new non-terminals without any effect on the result of parsing For example, the above grammar can be rewritten equivalently as

```
P  ::= [Point] "(" XY ")"
XY ::= x:int "," y:int
```

The `XY` production can be reused to set the `x` and `y` fields of any kind of object, not just points.

The Object Grammars given above can also be used to format points into textual form. The constructor acts as a guard that specifies that points should be rendered. The literal symbols are copied directly to the output. The field assignments are treated as selections that format the `x` and `y` fields of the point as integers.

2.2 Alternatives and Object-Valued Fields

Each alternative in a production can construct an appropriate object. The following example constructs either a constant, or one of two different kinds of `Binary` objects. The last alternative does not construct an object, but instead returns the value created by the nested `Exp`.

```
Exp ::= [Binary] lhs:Exp op:"+" rhs:Exp
     |  [Binary] lhs:Exp op:"*" rhs:Exp
     |  [Const] value:int
     |  "(" Exp ")"
```

This grammar is not very useful, because it is ambiguous. To resolve this ambiguity, we use the standard technique for encoding precedence and associativity using additional non-terminals.

```
Term ::= [Binary] lhs:Term op:"+" rhs:Fact   |  Fact
Fact ::= [Binary] lhs:Fact op:"*" rhs:Prim   |  Prim
Prim ::= [Const] value:int                   |  "(" Term ")"
```

This grammar refactoring is independent of the schema for expressions; the additional non-terminals (`Term`, `Fact`, `Prim`) do not have corresponding classes. Object grammars allow ambiguous grammars: as long as individual input strings are not ambiguous there will be no error. Thus the original version can only parse fully parenthesized expressions, while the second version handles standard expression notation.

During formatting, the alternatives are searched in order until a matching case is found. For example, to format `Binary(Binary(3,"+",5),"*",7)` as a `Term`, the top-level structure is a binary object with a $*$ operator. The `Term` case does not apply, because the operator does not match, so it formats the second alternative, `Fact`. The first alternative of `Fact` matches, and the left hand side `Binary(3,"+",5)` must be formatted as a `Fact`. The first case for `Fact` does not match, so it is formatted as a `Prim`. The first case for `Prim` also does not match, so parentheses are added and the expression is formatted as a `Term`. The net effect is that the necessary parentheses are added automatically, to format as `(3+5)*7`.

2.3 Collections

Object Grammars support regular symbols to automatically map collections of values. For example, consider this grammar for function calls:

```
C ::= [Call] fun:id "(" args:Exp* @"," ")"
```

The regular repetition grammar operator $*$ may be optionally followed by a separator using `@`, which in this case is a comma. The `args` field of the `Call` class is assigned objects created by zero-or-more occurrences of `Exp`. A collection field can also be explicitly bound multiple times, rather than using the $*$ operator. For example, `args:Exp*` could be replaced by `Args?` where `Args ::= args:Exp (","Args)?`.

For formatting, the regular operators $*$ and `+` provide additional semantics, allowing the formatter to perform intelligent grouping and indentation. A repeated group is either formatted on one line, or else it is indented and broken into multiple lines if it is too long.

2.4 Reference Resolution

In order to explain path-based reference resolution in Object Grammars, it is instructive to introduce a slightly more elaborate example. Consider a small DSL for modeling state machines. Figure 1 displays three representations of a simple state machine representing a door that can be opened, closed, and locked. Figure 1(a) shows the state machine in graphical notation. The same state machine is rendered textually in Fig. 1(b). Internally,

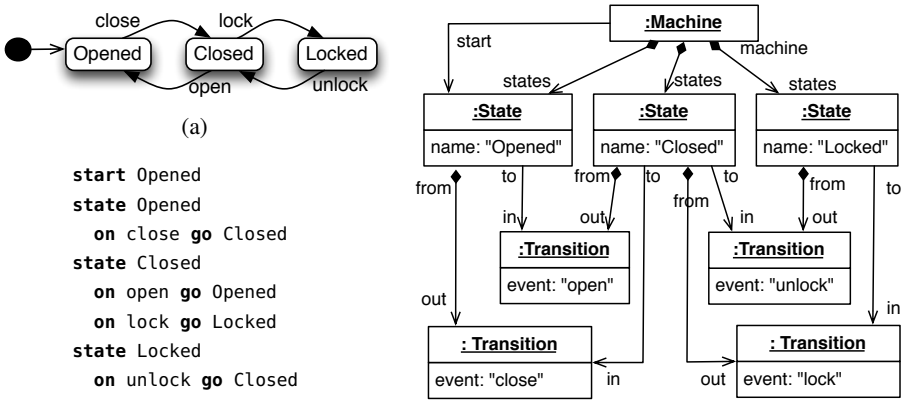

Fig. 1. (a) Example state machine in graphical notation, (b) the state machine in textual notation, and (c) the internal representation of the state machine in object diagram notation

```
class Machine
  start  : State
  states ! State*
```

```
class State
  machine: Machine / states
  name   # str
  out    ! Transition*
  in     : Transition*
```

```
class Transition
  event # str
  from  : State / out
  to    : State / in
```

Fig. 2. Ensō schema defining the structure of state machine object graphs

the machine itself, its states and the transitions are all represented explicitly as objects. This is illustrated in the object diagram given in Fig. 1(c).

The object diagram conforms to the State Machine schema given in Fig. 2. The schema consists of a list of named classes, each having a list of fields defined by a name, a type, and some optional modifiers. For example, the `Machine` class has a field named `states` which is a set of `State` objects. The `*` after the type name is a modifier that marks the field as many-valued. The # annotation marks a field as a primary key, as is the case for the `name` field of the State class. As a result, state names must be unique and the `states` field of `Machine` can be indexed by name. The `/` annotation after the `machine` field indicates that the `machine` and `states` are *inverses*, as are `from`/`out` and `to`/`in`. The `!` modifier indicates that the field is part of the *spine* (a minimal spanning tree) of the object graph. If a spanning tree is defined, then all nodes in a model must be uniquely reachable by following just the spine fields. The spine gives object models a stable traversal order. Note that in the example schema, the `start` field is not part of the spine, since the start state must be included in the set of all `states`.

The textual representation in Fig. 1(b) uses *names* to represent links between states, while the graphical presentation in Fig. 1(a) uses graphical edges so names are not needed. When humans read the textual presentation in Fig. 1(b), they immediately resolve the names in each transition to create a mental picture similar Fig. 1(a).

```
start M
M ::= [Machine] "start" \start:</states[it]> states:S*
S ::= [State] "state" name:sym out:T*
T ::= [Transition] "on" event:sym "go" to:</states[it]>
```

Fig. 3. Object Grammar to parse state machines

Figure 3 shows an Object Grammar for state machines[5]. It uses the reference `</states[it]>` to look up the start state of a machine and to find the the target state of a transition. The path `/states[it]` starts at the root of the resulting object model, as indicated by the forward slash `/`. In this case the root is a `Machine` object, since `M` is the start symbol of the grammar, and the `M` production creates a `Machine`. The path then navigates into the field `states` of the machine (see Fig. 2), and uses the identifier from the input stream to index into the keyed collection of all states. The same path is used to resolve the `to` field of a transition to the target state.

References and Paths In general, a reference <*p*> represents a lookup of an object using the path *p*. Parsing a reference always consumes a single identifier, which can be used as a key for indexing into keyed collections. Binding a field to a reference thus results in a cross-link from the current object to the referenced object.

```
Path ::= [Anchor] type:"."
       |  [Anchor] type:".."
       |
[Sub] parent:Path? "/" name:sym Subscript?
Subscript
::= "[" key:Key "]"
Key  ::= Path | [It] "it"
```

Fig. 4. Syntax of paths.

The syntax of paths is given in Fig. 4. A path is anchored at the current object (`.`), at its parent (`..`), or at the root. In the context of an object a path can descend into a field by post-fixing a path with `/` and the name of the field. If the field is a collection, a specific element can be referenced by indexing in square brackets. The keyword **`it`** represents the string-typed value of the identifier in the input stream that represents the reference name.

The grammar of schemas, given in Fig. 5, illustrates a more complex use of references. To lookup inverse fields, it is necessary to look for the field within the class that is the type of the field. For example, in the state machine schema in Fig. 1(b), the field `from` in `Transition` has type `State` and its inverse is the `out` field of `State`. The path for the type is `type:</types[it]>`, while the path for the inverse is `inverse:<./type/fields[it]>`, which refers to the type object. To resolve these paths, the parser must iteratively evaluate paths until all paths have been resolved.

To format a path, for example `/states[it]` in Fig. 3, the system solves the equation `/states[it]`=*o* to compute **`it`** given the known value *o* for the field. The resulting name is then output, creating a symbolic reference to a specific object.

[5] The field label `start` is escaped using \ because `start` is a keyword in the grammar of grammars; cf. Section 2.7.

```
start Schema
Schema ::= [Schema] types:TypeDef*
TypeDef ::= Primitive | Class
Primitive ::= [Primitive] "primitive" name:sym
Class ::= [Class] "class" name:sym Parent? defined_fields:Field*
Parent ::= "<" supers:</classes[it]>+ @","
Field ::= [Field] name:sym Kind type:</types[it]> Multiplicity? Annot?
Kind ::= "#" { key }   |   "!" { spine }   |   ":"
Multiplicity ::= "*" { many && optional }
               | "?" { optional }
               | "+" { many }
Annot ::= "/" inverse:<./type/fields[it]> |   "=" computed:Expr
```

Fig. 5. Schema Grammar

2.5 Predicates

The mapping between text and object graph can further be controlled using predicates. Predicates are constraint expressions on fields of objects in the object graph. During parsing, the values of these fields are updated to ensure these constraints evaluate to `true`. Conversely, during formatting, the constraints are interpreted as conditions to guide the search for the right rule alternative to format an object.

Predicates are useful for performing field assignments that are difficult to express using basic bindings. For instance, Ensō grammars have no built-in token type for boolean values to bind to. To write a grammar for booleans, one can use predicates as follows:

```
Bool ::= [Bool] "true"  { value }
       |  [Bool] "false" { !value }
```

Predicates are enclosed in curly braces. When the parser encounters the literal "`true`" it creates a `Bool` object and set its `value` field to true. Alternatively, when encountering the literal "`false`" the value field is assigned false, to satisfy the constraint that `!value` is true.

When formatting a `Bool` object, the predicates act as guards. The grammar is searched for a constructor with a fulfilled predicate or no predicate at all. Thus, a `Bool` object with field `value` set to true prints "`true`" and one with field `value` set to false prints "`false`".

A more complex example is shown in the Schema Grammar of Fig. 5. The classes and fields used in the grammar are defined in the Ensō Schema Schema [26]. The production rule for `Multiplicity` assigns the boolean fields `many` and `optional` in different ways. For instance, when a field is suffixed with the modifier "`*`", both the `many` and `optional` fields are assigned to values that make the predicate true; in this case both `optional` and `many` are set to true. Conversely, during formatting, *both* `many` and `optional` must be true in the model in order to select this branch and output "`*`".

2.6 Formatting Hints

Object Grammars are bidirectional: they are used for reading text into an object structure and for formatting such structure back to text. Since object structures do not maintain the layout information of the source text, formatting to text is in fact pretty-printing, and not unparsing: the formatter has to invent layout. As mentioned above, default lines breaks and indenting are generated based on the repeated expressions (marked with $*$ or +).

The layout can be further controlled by including formatting hints directly in the grammar. There are two such hints: suppress space (`.`) and force line-break (`/`). They are ordinary grammar symbols so may occur anywhere in a production.

The following example illustrates the use of `.` and `/`.

```
Exp ::= name:sym | Exp "+" Exp | "(".Exp.")"
Stat ::= Exp.";" | "{" / Stat* @/ / "}"
```

Spaces are added between all tokens by default, so the dot (`.`) is used to suppress the spaces after open parentheses and before close parentheses around expressions. Similarly, the space is suppressed before the semicolon of an expression-statement. The block statement uses explicit line breaks to put the open and close curly braces, and each statement, onto its own line. Note that the `Stat` repetition is *separated by* line-breaks (`@/`) during formatting, but this has no effect on parsing.

2.7 Lexical syntax

Ensō's Object Grammars have a fixed lexical syntax. This is not essential: Object Grammars can easily be adapted to scannerless or tokenization-based parser frameworks. For Ensō's goal, a fixed lexical syntax is sufficient. Furthermore, it absolves the language designer of having to deal with tokenization and lexical disambiguation.

First of all, whitespace and comments are completely fixed: spaces, tabs and newlines are ignored. There is one comment convention, `//` to end of line. Second, the way primitive values are parsed is also fixed. In the examples we have seen the **`int`** and **`sym`** symbols to capture integers and identifiers respectively. Additional types are **`real`** and **`str`** for standard floating point syntax and strings enclosed in double quotes.

The symbol to capture alpha-numeric identifiers, **`sym`**, is treated in a special way, since it may cause ambiguities with the keyword literals of a language. The parser avoids such ambiguities in two ways. First, any alpha-numeric literal used in a grammar is automatically treated as a keyword and prohibited from being a **`sym`** token. Second, for both keyword literals and identifiers a longest match strategy is applied. To use reserved keywords as identifiers they can be escaped using `\`. An example of this can be seen in the state machine grammar of Fig. 3, where the `start` field name is escaped because start is a keyword in grammars.

3 Self-Description

The Ensō framework is fully self-describing and Object Grammars are one of the foundations that make this possible. Grammars and schemas are both first-class Ensō

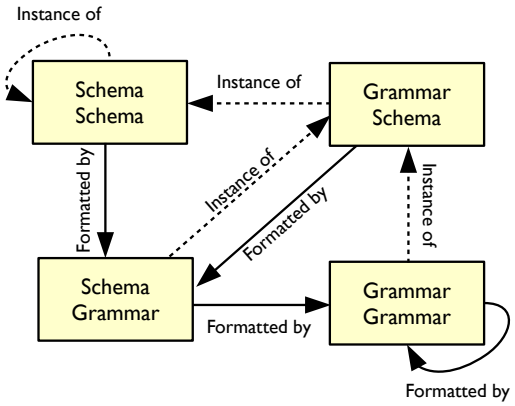

Fig. 6. The four core schema and grammar models

models [24], just like other DSLs in the system. In Ensō, all models are an *instance* of a schema, and grammar and schema models are no exception. Schemas are instances of a "schema of schemas", which is in turn an instance of itself. For grammars the relation is *formatting*. For example, the state machine grammar of Fig. 3 *formats* state machine models. Similarly, the grammar of grammars (Fig. 7) formats itself. The grammar of schemas (Fig. 5) parses and formats schemas. The schema of grammars (Fig. 8) instantiates grammars, and is formatted using the grammar of schemas. The schema of schemas and its relationship to other schemas are explained further in a companion paper [26]. These four core models and their relations are graphically depicted in Fig. 6.

Self-description provides two important benefits. First, the interpreters that provide the parsing and formatting behavior for Object Grammars can be reused to parse and format the grammars themselves. The same holds for Schema factories that are used to construct object graphs typed by a schema: the schema of schemas is just a schema that allows the creation of schemas, including its own schema. Second, by representing core languages grammar and schema as the first-class models of Fig. 6, they become amenable to extension in the same way just like ordinary models. For instance, both the Schema Schema and the Grammar Grammar reuse a generic expression language (cf. Section 5). The self-described nature of Ensō poses interesting bootstrapping challenges. However, we consider this to be outside the scope of this paper.

3.1 Grammar Grammar

The formal syntax of Object Grammars is specified by the Grammar Grammar defined in Fig. 7. A grammar consists of the declaration of the start symbol and a collection of production rules. There are two types of rules: concrete rules and abstract rules. Both types are identified by their `name`, which identifies the non-terminal that is introduced. A concrete rule has a body that consists of one or more alternatives separated by (`|`) as defined in the `Alt` rule. For an abstract rule, the body is bound through composition with

```
start Grammar
Grammar  ::= [Grammar] "start" \start:</rules[it]> rules:Rule*
Rule     ::= [Rule] name:sym "::=" arg:Alt
           | [Rule] "abstract" name:sym
Alt      ::= [Alt] alts:Create+ @"|"
Create   ::= [Create] "[" name:sym "]" arg:Sequence  |  Sequence
Sequence ::= [Sequence] elements:Field*
Field    ::= [Field] name:sym ":" arg:Pattern        |  Pattern
Pattern  ::= [Lit] value:str
           | [Value] kind:("int" | "str" | "real" | "sym" | "atom")
           | [Ref] "<" path:Path ">"
           | [Call] rule:</rules[it]>
           | [Code] "{" code:Expr "}"
           | [Regular] arg:Pattern "*" Sep?  { optional && many }
           | [Regular] arg:Pattern "+" Sep?  { many }
           | [Regular] arg:Pattern "?"       { optional }
           | [NoSpace] "."
           | [Break] "/"
           | "(" Alt ")"
Sep      ::= "@" sep:Pattern
abstract Path
abstract Expr
```

Fig. 7. The Grammar Grammar: an Object Grammar that describes Object Grammars

```
class Grammar            start: Rule        rules: Rule*
class Rule               name: str          arg: Alt?
                         grammar: Grammar / rules
class Pattern
class Alt < Pattern      alts: Pattern+
class Sequence < Pattern elements: Pattern*
class Create < Pattern   name: str          arg: Pattern
class Field < Pattern    name: str          arg: Pattern
class Lit < Pattern      value: str
class Value < Pattern    kind: str
class Ref < Pattern      path: Path
class Call < Pattern     rule: Rule
class Code < Pattern     expr: Expr
class NoSpace < Pattern
class Break < Pattern
class Regular < Pattern  arg: Pattern        sep: Pattern?
                         optional: bool      many: bool
```

Fig. 8. The Grammar Schema

another grammar. `Path` is an example of an abstract rule, which was defined in Fig. 4. See Section 5 for more discussion of grammar composition.

The grammar rules use the standard technique for expressing precedence of grammar patterns, by adding extra non-terminals. An alternative is a `Sequence` of `Patterns` possibly prefixed by a constructor (`Create`), which creates a new object that becomes the current object for the following sequence of patterns. If there is no constructor, the current object is inherited from the calling rule. The `Patterns` in a sequence can be `Field` bindings or syntactical symbols commonly found in grammar notations, such as literals, lexical tokens, non-terminals, regular symbols, and formatting hints.

Since the grammar of grammars is itself an Ensō model, it is accompanied by a schema of grammars. This is shown in Fig. 8. Note that the different forms of regular operators in the grammar are represented by a single class `Regular` with boolean properties to define the number of repetitions.

There is something very elegant and appealing about the concise self-description in the Grammar Grammar. For example the `Create` and `Field` rules both explain and use the creation/binding syntax at the same time. The `Ref` and `Call` rules seem to be inverses of each other, as the body of a `Call` is defined by a reference, and the body of a `Ref` is a call to `Path`. The normal level and meta-level are also clearly separated, as illustated by the various uses of unquoted and quoted operators (`|` vs. `"|"`, $*$ vs. `"*"`, etc).

4 Implementation

The implementation of Ensō is a collection of interpreters for the DSLs that are used in the system. Currently, these interpreters are implemented in the Ruby programming language [14]. In contrast to most systems, which are based on generating code for a parser by compiling a grammar, Ensō uses dynamic interpretation of grammars. The same applies to schemas: a "factory" object interprets a schema to dynamically create objects and assign fields [26].

These are the two interpreters relevant for the purpose of this paper:

$$parse : (S : Schema) \rightarrow Grammar_S \rightarrow String \rightarrow S \tag{1}$$

$$format : Grammar_S \rightarrow S \rightarrow String \tag{2}$$

The *parse* function takes a value S of type Schema, a grammar (compatible with S), and a string, and returns a value of type S. Note that *parse* is dependently typed: the value of the first argument determines the type of the result, namely S. The *format* function realizes the opposite direction: given a grammar compatible with S and an value of type S it produces a textual representation.

4.1 Parsing

The parser is implemented as an interpretive variant of the GLL algorithm [33]. GLL is a general parsing algorithm. As a result it supports infinite lookahead and supports the general class of context-free grammars. Tokenization of the input stream happens on the fly, during parsing. When a certain token type is expected on the basis of the state of

```
def build(t, ob=nil, f=false, vs=[], ps=[])
 l = t.label
 case l.schema_class.name
 when :Sequence then t.kids.each { |k|
    build(k, ob, false, vs, ps)
  }
 when :Create then t.kids.each { |k|
    build(k, ob=@factory.make(l.name))
  }
 when :Field then t.kids.each { |k|
    build(k, ob, true, vs=[], ps=[])
  }
  vs.each { |v| update(ob, l.name, v) }
  ps.each { |p| @fixes << [p, ob, l.name] }
 when :Lit then vs << t.value if f
 when :Value
  vs << convert(t.value, l.kind)
 when :Ref
  ps << subst_it(t.value, l.path)
 when :Code
  l.code.assert(ob)
 else then t.kids.each { |k|
    ob = build(k, ob, f, vs, ps)
  }
 end
 return ob
end
```

```
def fixup(root)
 begin
  later = []; change = false
  @fixes.each do |path, obj, fld|
   x = path.deref(root, obj)
   if x then
    # the path can be resolved
    update(obj, fld, x)
    change = true
   else
    # if not, try it later
    later << [path, obj, fld]
   end
  end
  @fixes = later
 end while change
 unless later.empty?
  raise "Fix-up error"
 end
end
```

Fig. 9. Pseudo Ruby code for building the spine and fix-up of cross-links

the parser, the scanner is asked to provide this token at the current position of the input stream. If it delivers, parsing continues, otherwise, an alternative branch in the grammar will be taken. If there are no remaining branches, a parse error is issued. The result of a successful, non-ambiguous parse is a concrete syntax tree where the nodes are annotated with grammar patterns (e.g., `Sequence`, `Create` etc.—see Fig. 8).

If parsing is successful, the object-graph is constructed from the concrete syntax tree in two steps. First, the spine of the object graph is created. This is shown in the left-hand side of Fig. 9. The `build` algorithm recursively traverses the syntax tree and depending on the label of a node, creates objects and assigns fields. The first argument to `build` is the syntax tree, `ob` represents the "current" object; `f` indicates if field assignment can be performed. Finally, `vs` and `ps` collect values and paths respectively.

For constructor directives, a factory is called to create an object of the right class. The created object becomes the new current object when recursing down the tree. In the case for `Field` nodes, the values collected in `vs` are directly assigned to the current object. The paths `ps` are recorded as "fixes" to the current object for the current field in the global variable `@fixes`; these fixes are applied later to create cross-links.

Both `Literals` and `Values` (tokens) are simply added to the collection of values `vs`. `Values` are first converted to the expected type; the value of a literal is recorded literally, but only when the node is directly below a field binder. When a reference is encountered (`Ref`) the special keyword **`it`** is substituted for the name that has been parsed, and the result is added to `ps`. Finally, predicates (`Code`) are asserted in the context of the current object so that the referenced fields are appropriately set.

In the second step, the path-based references are resolved in an iterative fix-point process. This is shown in the right-hand side of Fig. 9. The fix-point process ensures that dependencies between references are dynamically discovered. If in the end some of the paths could not be resolved—for instance because of a cyclic dependency—an error is produced.

4.2 Formatting

Formatting works by matching constructor and field binding specifications in an Object Grammar against objects. In essence, the formatter searches for a minimal rendering that is compatible with the object graph. When the class in a constructor directive matches the class of the object being formatted, the object is formatted using the body of the production alternative. If formatting fails when recursing through the grammar, the formatter backtracks to select a different production alternative. If no suitable alternatives can be found, an error is raised.

Literals are formatted directly to the output, and fields are selected from the object. The formatter creates an intermediate formatting structure that includes the pretty printing hints of the grammar. This structure is then formatted to text using Wadler's prettier printer algorithm [39].

5 Language Composition

Modular language development presupposes a composition operator to combine two language modules into one. For two grammars, this usually involves taking the union of their production rules, where the alternatives of rules with the same name are combined. To union Object Grammars in such a way, it is also necessary to merge their target schemas so that references to classes and fields in both languages can be resolved.

In Ensō, composition of grammars and schemas are both accomplished using the same generic merge operator $\cdot \lhd \cdot$. This operator can be characterized as an overriding union where conflicts are resolved in favor of the second argument. Since a language is defined by its schema and grammar, the composition of a base language B with an extension E is given by $B_{grammar} \lhd E_{grammar}$ and $B_{schema} \lhd E_{schema}$.

5.1 Merge

The algorithm implementing $\lhd$ is shown in pseudo Ruby code in Fig. 10. There are two passes in the merge algorithm. In the first pass, `build` traverses the spine of the object graph `o1` to create any new object required. If `build` encounters an object in `o2` but none at the same location on the spine in `o1`, it creates a new copy of that object and

```
def merge_into(type, o1, o2)
  build(type, o1, o2, memo = {})
  link(type, true, o1, o2, memo)
end

def build(type, a, b, memo)
  return if b.nil?
  memo[b] = new = a || type.new
  type.fields.each do |fld|
    ax = a && a[fld.name]
    bx = b[fld.name]
    if fld.type.Primitive? then
      new[fld.name] = bx
    elsif fld.spine
      if !fld.many
        build(fld.type, ax, bx, memo)
      else
        ax.outer_join(bx) do |ai, bi|
          build(fld.type, ai, bi, memo)
        end
      end
    end
  end
end
```

```
def link(type, spine, a, b, memo)
  return a if b.nil?
  new = memo[b]
  return new if !spine
  type.fields.each do |fld|
    ax = a && a[fld.name]
    bx = b[fld.name]
    next if fld.type.Primitive?
    if !fld.many? then
      val = link(fld.type, fld.spine, ax, bx, memo)
      new[fld.name] = val
    else
      ax.outer_join(bx) do |ai, bi|
        x = link(fld.type, fld.spine, ai, bi, memo)
        unless new[fld.name].include?(x)
          new[fld.name] << x
        end
      end
    end
  end
  return new
end
```

Fig. 10. Pseudo Ruby code for the generic $\lhd$ operator

attaches it to the graph of `o1`. Primitive fields from `o1` are always overridden by the same fields of `o2`, allowing the extension to modify the original language. Pairs of objects are merged by merging the values of each field. Collections are merged pair-wise according to their keys; `outer_join` is a relational join of two collections, matching up all pairs of items with equivalent keys and pairing up the remaining items with `nil`. At the same time, the first pass also establishes a mapping `memo`, between each object in `o2` and the corresponding object in the same spine location in `o1`.

In the second phase, non-spine fields—those without the `!` modifier—are made to point to their new locations. The object graph is once again traversed along the spine, but this time `link` looks up `memo` for each non-spine field in order to find the updated target object.

5.2 Composition in Ensō

Many of the current set of languages in Ensō are defined by composing two or more language modules. Fig. 11 shows how Ensō languages are related with respect to language composition. Each edge in the diagram represents an invocation of $\lhd$. The arrow points in the direction of the result. For instance, the Stencil and Web languages are, independently, merged into the Command language. As a result both Stencil and

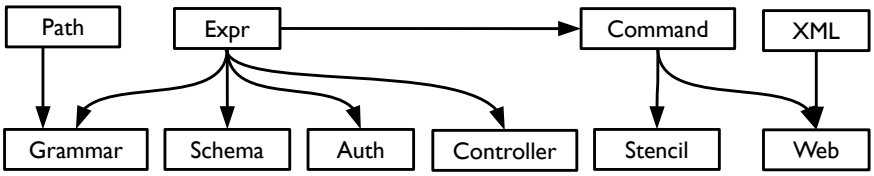

Fig. 11. Language composition in Ensō. Each arrow $A \rightarrow B$ indicates an invocation of $B \lhd A$.

Web include, and possibly override and/or extend the Command language. If a language reuses or extends multiple other languages, the merge operator is applied in sequence. For instance, Grammar is first merged into Path, and then merged into Expr.

The core languages in Ensō include both the Schema and Grammar languages, as well as Stencil, a language to define graphical model editors. Additionally, Ensō features a small set of library languages that are not deployed independently but reused in other languages. An example of a library language is Expr, an expression language with operators, variables and primitive values. It is, for instance, reused in Grammar for predicates and in Schema for computed fields. Command is a control-flow language that captures loops, conditional statements and functions. The Command language reuses the Expr language for the guards in loops and conditional statements. Another example is the language of paths (Path), shown in Fig. 4, which provides a model to address nodes in object graphs.

The reuse of Expr and Path are examples of a simple embedding. The languages are reused as black boxes, without modification. The composition of Command with Stencil and Web, however is different. Stencil is created by adding language constructs for user-interface widgets, lines, and shapes to the Command language as valid primitives. The Command language can now be used to create diagrams. A similar extension is realized in the Web language: here a language for XML element structure is mixed with the statement language of Web. The extension works in both directions: XML elements are valid statements, statements are valid XML content. The Piping and Controller languages are from a domain-specific modeling case-study in the domain of piping and instrumentation for the Language Workbench Challenge 2012 [25]. Fig. 11 only shows the Controller part which reuses Expr.

An overview of the number source lines of code (SLOC) is shown in Table 1(a). We show the number for the full languages in Ensō as well as the reused language modules (Path, Command, Expr and XML). A language consists of a schema, a grammar and an interpreter. The interpreters are all implemented in Ruby. Table 1(b) shows the reuse percentage for each language [17]. This percentage is computed as $100 \times \#SLOC_{\text{reused}}/\#SLOC_{\text{total}}$. Which languages are reused in each case can be seen from Fig. 11. As can be seen from this table, the amount of reuse in schemas and grammars is consistently high, with the exception of the Piping language, which does not reuse any language. It shows that the merge operator is powerful enough to combine real languages in a variety of ways, with actual payoff in terms of reuse.

Language	Schema	Grammar	Interpreter
Grammar	53	31	1243
Schema	30	20	667
Stencil	51	26	1387
Web	79	43	885
Auth	28	16	276
Piping	80	22	306
Controller	26	14	155
Path	14	6	222
Command	39	26	265
Expr	47	30	91
XML	10	6	47

(a)

Reuse Percentages			
Language	Schema	Grammar	Interpreter
Grammar	54%	54%	20%
Schema	61%	60%	12%
Stencil	63%	68%	20%
Web	55%	59%	31%
Auth	63%	65%	25%
Piping	0%	0%	0%
Controller	64%	68%	37%

(b)

Table 1. SLOC count (a) and reuse percentages (b) for schemas, grammars and interpreters of the languages currently in Ensō

6 Related work

The subject of bridging modelware and grammarware is not new [1,40]. In the recent past, numerous approaches to mapping text to models and vice versa have been proposed [13, 16, 19, 21, 23, 27, 28]. Common to many of these languages is that references are resolved using globally unique, or hierarchically scoped names. Such names can be opaque Unique Universal Identifiers (UUIDs) to uniquely identify model elements or key attributes of the elements themselves [18]. The main difference between these approaches and Object Grammars is that Object Grammars replace the name-based strategy by allowing arbitrary paths through the model to find a referenced object. This facilitates mappings that require non-global or non-hierarchical scoping rules. Below we discuss representative systems in more detail.

The Textual Concrete Syntax (TCS) language supports deserialization and serialization of graph-structured models [21]. Field binders can be annotated with `{refersTo =` ⟨*name*⟩`}`, which binds the field to the object of the field's class with the field ⟨*name*⟩ having the value of the parsed token. Rules can furthermore be annotated with `addToContext` to add it to the, possibly nested, symbol table. The symbol table is built after the complete source has been parsed to allow forward references. Only simple references to in-scope entities are allowed, however. Path-based references of Object Grammars allow more complex reference resolving, possibly across nested scopes. TCS aims to have preliminary support for pretty printing directives to control nesting and indentation, spacing and custom separators. However, these features seem to be unimplemented.

Xtext is an advanced language workbench for textual DSL development [13]. The grammar formalism is restricted form of ANTLR so that both deserialization and serialization is supported. Xtext supports name-based referencing. To customize the name lookup semantics Xtext provides a Scoping API in Java. Apart from the use of simple names, Xtext differs from Object Grammars in that, by default, linking to target objects is performed lazily. Again, this can be customized by implementing the appropriate

interfaces. Xtext is said to support a limited form of modularity through grammar mixins. For lexical syntax Xtext provides a standard set of terminal definitions such as INT and STRING, which are available for reuse.

EMFText is an Ecore based formalism similar to Xtext grammars [9, 19]. EMFText, however, supports accurate unparsing of models that have been parsed. For models that have been created in memory or have been modified after parsing, formatting can be controlled by pretty printing hints similar to the `.` and `/` symbols presented in this paper. The grammar symbol `#`*n* forces printing of the *n* spaces. Similarly, `!`*n* is used for printing a line-break, followed by *n* indentation steps.

In the MontiCore system both metamodel (schema) and grammar are described in a single formalism [23]. This means that the non-terminals of the grammar introduces classes and syntactic categories at the same time. Grammar alternatives are declared by non-terminal "inheritance". As a result, the defined schema is directly tied to the syntactic structure of the grammar. The formalism supports the specification of associations and how they are established in separate sections. The default resolution strategy assumes file-wide unique identifiers, or syntactically hierarchical namespaces. This can be customized by programming if needed.

The Textual Concrete Syntax Specification Language (TCSSL) is another formalism to make grammars metamodel-aware [15]. It features three kinds of syntax rules: CreationRules which function like our `[Create]` annotations,—SeekRules, which look for existing objects satisfying an identifying criterion,—and SingletonRules, which are like CreationRules, but only create a new object if there is no existing object satisfying a specified criterion. The queries used in SeekRules seem more powerful than simple, name-based resolution; it is however unclear from the paper how they are applied for complex scenarios. TCSSL furthermore allows code fragments enclosed in double angular brackets (`<<>>`) but it is unclear how this affects model-to-text formatting.

Discussion The requirements for mapping grammars to metamodels were first formulated in [20]: the mapping should be customizable, bidirectional and model-based. The Object Grammars presented in this paper satisfy these requirements. First, the mapping is customizable because of asynchronous binding: the resulting structures are to a large extent independent of the structure of the grammar. Path-based referencing and predicates are powerful tools to control the mapping, but admit a bidirectional semantics so that formatting of models back to text is possible. Formatting can be further guided using formatting hints. Finally, Object Grammars are clearly model-based: both grammars and schemas are themselves models, self-formatted and self-described respectively. A comparative overview of systems to parse text into graph structures that conform to class-based metamodels can be found in [18].

To our knowledge, Object Grammars represent the first approach to mapping between grammars and metamodels that supports modular combination of languages. Xtext, EMFText, TCS, MontiCore, and TCSSL are implemented using ANTLR. ANTLR's LL(*) algorithm, however, makes true grammar composition impossible. Object Grammars, on the other hand, *are* compositional due to the use of the general parsing algorithm GLL [33]. Moreover, the use of a general parsing algorithm has the advantage that there is no restriction on context-free structure. For instance, the designer of a modeling

language does not have to worry about whether she can use left-recursion or whether her grammar is within some restricted class of context-free grammars, such as LL(k) or LR(k). As a result, Object Grammars can be structured in a way that is beneficial for resulting structure, without being subservient to a specific parsing algorithm. Object Grammars share this freedom with other grammar formalisms based on general parsing, such as SDF [36] and Rascal [22].

The way references are resolved in Object Grammars bears resemblance to the way attributes are evaluated in attribute grammars (AGs) [30]. AGs represent a convenient formalism to specify semantic analyses, such as name analysis and type checking, by declaring equations between inherited attributes and synthesized attributes. The AG system schedules the evaluation of the attributes automatically. Modern AG systems, such as JastAdd [11] and Silver [35], support reference attributes: instead of simple values, such attributes may evaluate to pointers to AST nodes. They can be used, for instance, to super-impose a control-flow graph on the AST. Reference resolving in Object Grammars is similar to attributes: they are declarative statements of fact, and the system—in our case the *parse* function— decides how to operationally make these statements true. Object-grammars are different, however, in the sense that the object graph is first-class, and not a decoration of an AST. Moreover, path-based references only allow navigating the object graph without performing arbitrary computations. Extending Object Grammars with AG style attributes is an area for further research.

Modular language development is an active area of research. This includes work on modular extension of DSLs and modeling languages [34, 37, 38], extensible compiler construction [4, 11], modular composition of lexers [7] and parsers [6, 32], modular name analysis [10] and modular language embedding [31]. Object Grammars support a powerful form of language composition through the generic merge operation ($\lhd$) applied in tandem to both grammars and schemas. The merge operator covers language extension and unification as discussed in [12]. In essence, merge captures an advanced form of inheritance similar to feature composition [2, 3]. However, merge currently applies only syntactic and semantic *structure*. To achieve the same level of compositionality at the level of *behavior*, i.e. interpreters, is an important direction for further research.

7 Conclusion

Object Grammars are a formalism for bidirectional mapping between text and object graphs. Unlike traditional grammars, Object Grammars include a declarative specification of the semantic structure that results from parsing. The notation allows objects to be constructed and their fields to be bound. Paths specify cross-links in the resulting graph structure. Thus the result of parsing is a graph, not a tree. Object Grammars can also be used to format an object graph into text.

Our implementation of Object Grammars in Ensō supports arbitrary context-free grammars. This is required when composing multiple grammars together. We have shown how Object Grammars are used in Ensō to support modular definition and composition of DSLs.

Acknowledgements We thank Atze van der Ploeg and the anonymous reviewers for their comments on earlier drafts of this paper.

References

1. Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Technical Report 606, Turku Centre for Computer Science (2004)
2. Apel, S., Hutchins, D.: A calculus for uniform feature composition. ACM Trans. Program. Lang. Syst. **32**(5) (2008) 19:1–19:33
3. Apel, S., Kastner, C., Lengauer, C.: FeatureHouse: Language-independent, automated software composition. In: Proceedings of the International Conference on Software Engineering (ICSE). (2009) 221–231
4. Avgustinov, P., Ekman, T., Tibble, J.: Modularity first: a case for mixing AOP and attribute grammars. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM (2008) 25–35
5. Bąk, K., Czarnecki, K., Wąsowski, A.: Feature and meta-models in Clafer: mixed, specialized, and coupled. In: Proceedings of the 3rd international conference on Software Language Engineering (SLE'10), Springer (2011) 102–122
6. Bravenboer, M., Visser, E.: Parse table composition. In: Proceedings of the International Conference on Software Language Engineering (SLE). Revised selected papers. Volume 5452 of LNCS., Springer (2009) 74–94
7. Casey, A., Hendren, L.: MetaLexer: a modular lexical specification language. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM (2011) 7–18
8. Chen, P.P.: The Entity-Relationship Model—Toward a Unified View of Data. ACM Transactions on Database Systems **1**(1) (1976)
9. DevBoost: EMFText: concrete syntax mapper `http://www.emftext.org/`.
10. Ekman, T., Hedin, G.: Modular name analysis for Java using JastAdd. In: Proceedings of the International Summerschool on Generative and Transformational Techniques in Software Engineering (GTTSE), Springer (2006) 422–436
11. Ekman, T., Hedin, G.: The JastAdd system—modular extensible compiler construction. Sci. Comput. Program. **69**(1-3) (December 2007) 14–26
12. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the International Workshop on Language Descriptions, Tools and Applications (LDTA). (2012)
13. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: OOPSLA Companion (SPLASH), ACM (2010) 307–309
14. Flanagan, D., Matsumoto, Y.: The Ruby Programming Language. O'Reilly (2008)
15. Fondement, F., Schnekenburger, R., Gérard, S., Muller, P.A.: Metamodel-aware textual concrete syntax specification. Technical Report LGL-2006-005, EPFL (December 2006)
16. Fondement, F.: Concrete syntax definition for modeling languages. PhD thesis, EPFL (2007)
17. Frakes, W., Terry, C.: Software reuse: metrics and models. ACM Comput. Surv. **28**(2) (1996) 415–435
18. Goldschmidt, T., Becker, S., Uhl, A.: Classification of concrete textual syntax mapping approaches. In: Proceedings of the European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA). Volume 5095 of LNCS. (2008) 169–184
19. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Model Driven Architecture—Foundations and Applications (ECMDA-FA). Volume 5562 of LNCS. Springer (2009) 114–129
20. Jouault, F., Bézivin, J.: On the specification of textual syntaxes for models. In: Eclipse Modeling Symposium, Eclipse Summit Europe 2006. (2006)
21. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM (2006) 249–254

22. Klint, P., van der Storm, T., Vinju, J.: Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE (2009) 168–177
23. Krahn, H., Rumpe, B., Völkel, S.: Integrated definition of abstract and concrete syntax for textual languages. In: Proceedings of the International Conference On Model Driven Engineering Languages And Systems (MoDELS). Volume 4735 of LNCS. Springer (2007) 286–300
24. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: OOPSLA Companion (OOPSLA), ACM (2006) 602–616
25. Loh, A.: Piping and instrumentation in Ensō. Language Workbench Challenge Workshop at Code Generation 2012 (March 2012) http://www.languageworkbenches.net/index.php?title=LWC_2012. Accessed June 11th, 2012.
26. Loh, A., van der Storm, T., Cook, W.R.: Managed data: Modular strategies for data abstraction http://www.cs.utexas.edu/~wcook/Drafts/2012/ensodata.pdf. Submitted.
27. Merkle, B.: Textual modeling tools: overview and comparison of language workbenches. In: OOPSLA Companion (SPLASH), ACM (2010) 139–148
28. Muller, P.A., Fondement, F., Fleurey, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-driven analysis and synthesis of textual concrete syntax. Software and System Modeling **7**(4) (2008) 423–441
29. Object Management Group: Unified Modeling Language Specification, version 1.3. OMG, http://www.omg.org (March 2000)
30. Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. ACM Comput. Surv. **27**(2) (1995) 196–255
31. Renggli, L., Denker, M., Nierstrasz, O.: Language boxes: bending the host language with modular language changes. In: Proceedings of the International Conference on Software Language Engineering (SLE). Volume 5969 of LNCS., Springer (2010) 274–293
32. Schwerdfeger, A.C., Van Wyk, E.R.: Verifiable composition of deterministic grammars. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM (2009) 199–210
33. Scott, E., Johnstone, A.: GLL parsing. Electr. Notes Theor. Comput. Sci. **253**(7) (2010) 177–189
34. Van Wyk, E.: Aspects as modular language extensions. In: Proc. of Language Descriptions, Tools and Applications (LDTA). Volume 82.3 of Electronic Notes in Theoretical Computer Science., Elsevier Science (2003)
35. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Science of Computer Programming **75**(1–2) (January 2010) 39–54
36. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
37. Völter, M., Solomatov, K.: Language modularization and composition with projectional language workbenches illustrated with MPS. In: Proceedings of the International Conference on Software Language Engineering (SLE). Revised selected papers. Volume 6563 of LNCS., Springer (2010)
38. Völter, M., Visser, E.: Language extension and composition with language workbenches. In: OOPSLA Companion (SPLASH), ACM (2010) 301–304
39. Wadler, P.: A Prettier Printer. In: The Fun of Programming. Palgrave Macmillan (2003)
40. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Proceedings of the Satellite Events at the MoDELS Conference, Springer (2006) 159–168
41. Wirth, N.: What can we do about the unnecessary diversity of notation for syntactic definitions? Commun. ACM **20**(11) (1977) 822–823