Object Parsing Grammars with Composition

Stefan Sobernig
Institute for Information Systems and New Media, WU Vienna, Welthandelsplatz 1, A-1020 Vienna, Austria

Keywords: Language-product Line, Parsing Expression, Object Graph, Grammar Reuse, Grammar Transformation.

Abstract: An Object Parsing-Expression Grammar (OPEG) is an extension of parsing expression grammars (PEG) including generator expressions to directly produce object graphs from parsed text. This avoids typical abstraction mismatches of intermediate parse representations (e.g., decomposition mismatches). To develop language families via extension, unification, and extension compositions, OPEGs can be composed—without preplanning and with unmodified reuse. Composability is established by supporting both forming basic grammar unions and performing grammar transformations between two or more OPEGs (e.g., rule extraction, symbol rewriting). These transformation operators assist developers in mitigating the consequences of the non-disjointness under composition of parsing expressions (e.g., language hiding). An implementation of OPEGs is available as part of the multi-DSL development system DjDSL.

1 INTRODUCTION

Developing variable software languages shifts emphasis from developing and analysing a single language to developing and to analysing composable development artefacts for a language family. This ambition gave rise to approaches to language-product line engineering (Méndez-Acuña et al., 2016; Kühn et al., 2015; Jézéquel et al., 2015; Liebig et al., 2013) and their supporting multi-language development systems. Their shared goals are to minimise preplanning effort as well as, at the same time, to reuse development artefacts and language tooling in an unmodified manner.

The composability of language-definition artefacts (e.g., definitions of abstract and concrete syntaxes, context conditions, behaviour, and test cases) is a pre-requisite for a variable design and implementation of a language (Erdweg et al., 2015). Composability is also required to adhere to the conditions of a modular de- and re-composition, for example, preserving modular comprehensibility of grammars and grammar extensions (Johnstone et al., 2014). Language composition must be tackled at different levels of language and processing (abstract syntax, context conditions, behaviour implementation). The emphasis in this paper is on composition of concrete-syntax definitions.

Syntax-level composition involves two or more concrete-syntax definitions (e.g., production or parsing grammars) to become combined. Text written using the combined syntax must be processed—by grammar-based parser generators, grammar interpreters, or parser combinators—as if it were produced from or recognised by distinct syntax definitions. Each syntax piece is thought of as conforming to one of the source syntaxes, or a mix of source-syntax fragments. The definition of a resulting parser (by a composed grammar or by a parser combinator) is ideally formed by referencing the source definitions, rather than cloning them. This has the benefit of tracking any modification in the source definitions without further intervention by the developer (e.g., providing patch code to the generated parser). Reusing the source definitions without modification is the key objective.

Rendering syntax definitions composable presents important challenges (Degueule, 2016, Section 3.3). The main challenge is ambiguity under composition. Ambiguity can arise as an unwanted consequence of a composition: Two unambiguous grammars may enter a composition and turn into an ambiguous composed grammar (Diekmann and Tratt, 2014; van der Storm et al., 2014). Ambiguous parsing is particularly critical when each parse presents different and possibly contradicting interpretations in terms of the underlying semantics. But also earlier, when constructing higher-level parse representations such as an abstract-syntax graph (ASG), each valid parse may translate into a different ASG (van der Storm et al., 2014, Section 2.2).

In parsing expression grammars (PEG), ambiguity under composition takes a characteristic form: lan-
Listing 1: An excerpt from an Object Parsing-Expression Grammar (OPEG), showing two parsing rules in EBNF-like notation. The first rule exemplifies the inline mapping of concrete-syntax elements to object-classes (Event) and their properties (name).

```
E ← `Event` ON name:<alnum>+ ;
void: ON ← WS 'on' WS;
```

**guage hiding.** Parsing expressions from two or more grammars, when combined, may result in a composed recogniser that hides a language’s syntax unintentionally.

This paper documents fine-grained grammar transformations to prevent unintended language hiding from happening when composing Object Parsing-Expression Grammars (OPEG). OPEGs are a general PEG extension for defining, in one, a concrete syntax as well as a mapping between concrete syntax and an object-oriented primary abstract syntax (ASG). For production grammars, these became known as object grammars (van der Storm et al., 2014).

The fine-grained grammar transformations introduced in this paper include rule extractions with and without symbol rewriting, transitive symbol rewriting as well as rule removals (see Section 3).

A proof-of-concept implementation, the running examples as well as the code listings in this paper are available from a supplemental Web site as an executable tutorial\(^1\).

## 2 Parsing to Object Graphs

A parsing expression grammar (PEG; Ford 2004) is defined as a 4-tuple \( G = (N, T, R, e_S) \). \( N \) denotes the finite set of non-terminals, \( T \) is the finite set of terminals, \( R \) is the finite set of rules, and \( e_S \) is the start expression. Each rule \( r \in R \) is a pair \((A, e)\) typically written as a maplet \( A \xleftarrow{e} \), with \( A \in N \) and \( e \) being another parsing expression.

A parsing expression defines a pattern to match (recognise) and, if matched, to consume a specified fragment of input. A parsing expression is defined using the empty string \(\epsilon\), the sets of terminals and non-terminals \((N, T)\), as well as operator expressions summarised in Table 1 (1–12).

The meaning of a PEG is given by a recognising program (Grune and Jacobs, 2010, Section 15.7). A recognition program is a program for recognising and structuring (incl. tokenising, parsing) a string. The (operational) meaning of a PEG-based recognition program can be thought of that works left-right, top-down to recognise, and if recognised, to consume the matched input. The interpreter always consumes the longest possible matched prefix of some input. A given parsing expression is said to succeed when it consumes what it has recognised; if an expression fails (i.e., it does not recognise anything), it consumes nothing from the input. This is even so when some of its sub-expressions have succeeded.

### Table 1: Overview of the operators available for OPEG/PT parsing expressions. The operators correspond to those of other parsing-expression language implementations, e.g., Rats! (Grimm, 2006), APEG (Reis et al., 2015), Arpeggio (Dejanović et al., 2016). Note that \(\epsilon\) (epsilon) stands for matching the empty string. OPEG extensions are **generators** (13–14; see Section 2.2). Other extensions are inherited from the reused parsing virtual machine (PT); rule modifiers syntax-tree generation (e.g., void, leaf, value); character classes (e.g., `<alnum>`, `<digit>`, `<digit>`).

<table>
<thead>
<tr>
<th>op</th>
<th>description</th>
<th>desugared</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(e_1, e_2)</td>
<td>sequence</td>
</tr>
<tr>
<td>2</td>
<td>(e_1 / e_2)</td>
<td>prioritised (ordered) choice</td>
</tr>
<tr>
<td>3</td>
<td><code>d</code></td>
<td>literal character</td>
</tr>
<tr>
<td>4</td>
<td><code>abc</code></td>
<td>literal string</td>
</tr>
<tr>
<td>5</td>
<td>([A-z0-9])</td>
<td>character ranges</td>
</tr>
<tr>
<td>6</td>
<td><code>.</code></td>
<td>any character</td>
</tr>
<tr>
<td>7</td>
<td><code>(e)</code></td>
<td>sub-expression (group)</td>
</tr>
<tr>
<td>8</td>
<td><code>?</code></td>
<td>optional expression</td>
</tr>
<tr>
<td>9</td>
<td><code>*</code></td>
<td>inclusive-or (zero-or-more)</td>
</tr>
<tr>
<td>10</td>
<td><code>+</code></td>
<td>inclusive-or (one-or-more)</td>
</tr>
<tr>
<td>11</td>
<td><code>?e</code></td>
<td>not predicate</td>
</tr>
<tr>
<td>12</td>
<td><code>?e</code></td>
<td>and predicate</td>
</tr>
<tr>
<td>13</td>
<td><code>\'c\' e</code></td>
<td>instantiation generator</td>
</tr>
<tr>
<td>14</td>
<td><code>f:e</code></td>
<td>assignment generator</td>
</tr>
<tr>
<td>15</td>
<td><code>f:</code></td>
<td>query generator</td>
</tr>
</tbody>
</table>

\(^1\)https://github.com/mrcalvin/djdsl

Parsing expressions can contain operator expressions and operator behaviours not available in other parsing approaches. Most importantly, for a given expression, alternate subexpressions are tried in their order of definition. The first one to succeed wins, any remaining ones are discarded. This is referred to as a prioritised or ordered choice (see operator 2 in Table 1). Prioritised or ordered choice has been documented as the key discriminator between PEG and CFG (Mascarenhas et al., 2014). On top, the choice operator gives rise to all difficulties associated with PEG regarding composition: ambiguity handling and language hiding.
2.1 Language Hiding

Parsing expressions build on ordered choices and unlimited lookahead. These preclude the possibility of ambiguous parses, but incur the risk of unwanted language hiding. Language hiding is a practical consequence of the absence of general semi-disjointness of a choice expression (Schmitz, 2006) for the scope of the language matched by a PEG.

Consider appending a parsing expression $e_2$ as an alternate to an existing parsing rule $S \leftarrow e_1$, yielding $S \leftarrow e_1/e_2$ as a result of composing two PEG. This ordered choice is commutative only if $e_1$ and $e_2$ are semi-disjoint expressions, that is, they succeed in consuming input from two languages that are semi-disjoint. Otherwise, the choice is not commutative and the order of composition becomes essential for the parsing result.

Intuitively, $e_1$ and $e_2$ are semi-disjoint if $e_1$ does not overlap with any prefix also recognised by $e_2$ (Schmitz, 2006, Section 4). Disjointness must also hold for any super-expression that contains $e_1/e_2$ like $(e_1/e_2)e_2$. The parsing rule $S \leftarrow (\cdot a a^* / a^*) a^*$, with $aa$ as parsing expression $e_1$ and $a$ as $e_2$. This rule will successfully consume one input: $aaa$. Input $aa$ will be rejected, on the ground that the $e_1$ ($aa$) is tested first, rejecting any input not having a third $a$. When flipping the order between $e_1$ and $e_2$ from $(\cdot a a^* / a^*)$ to $(\cdot a^* / a a^*)$, only $aa$ will be consumed and $aaa$ becomes now rejected.

Covering all input, i.e., the language \{aa, aaa\}, in this one example with a single expression requires an informed re-arrangement. First, the sub-expression $(e_1/e_2)$ must be moved to the right yielding $e_2(e_1/e_2)$. This is equivalent to writing $(e_2e_1)/(e_2e_2)$ according to the distributive property of the ordered choice (Ford, 2004, Section 3.7). This way, $e_2$ cannot fail the super-expression unconditionally, the second alternate will be tested when $e_2$ fails the first one. Second, within the sub-expression, it must be taken care that the expression consuming more of the input (the longer prefix) on success is tested first. This re-arrangement yields $S \leftarrow a^*$ (\(\cdot a a^* / a^*\)).

To summarise: Language hiding occurs when a (greedy) alternate of a choice expression prevents a later alternate from being applied to inputs that it could otherwise succeed on. This is also called a preemptive prefix capture (Redziejowski, 2008). See also Section 3.7 in Ford 2004.

Listing 2: Miss Grant is told to maintain a secret compartment in her bedroom. This compartment requires a particular sequence of actions from her side to become unlocked for her to open. The corresponding state-machine models the modal behaviour of the software-based compartment controller, reacting to Miss Grant’s input actions (Fowler, 2010, Section 1.1.1).

2.2 Object Parsing Expressions

A parsing grammar can contain extended parsing expressions to process the consumed syntactic structure into an object graph. This way, an OPEG definition lays out two-in-one: (a) input recognition and (b) mapping the recognised input onto objects, their fields, and non-hierarchical relationships between the mapped objects. This is shared spirit with object (production) grammars (van der Storm et al., 2014).

In what follows, the extensions to parsing grammars are highlighted by referring to the running example of modelling the state machine driving “Miss Grant’s Controller”. In later sections, this is referred to as the State-Machine Definition Language (SMDL) notation. Listing 2 depicts the concrete-syntax snippet of a state-machine definition.

Object parsing expressions are only covered to the extent necessary to provide a general background and to render the contributions on grammar transformations in Section 3 understandable (object generation, alternates, associations, and references). For a comprehensive coverage including details on multi-valued properties and non-positional parsing, see Sobernig (2020).

Object Generation. Parsing rules in parsing grammars can contain special-purpose expressions at their RHS that compute one or several instantiations of object-classes when their rule is applied. These expressions are referred to as instantiation generators (see Table 1, operator 13). Listing 1 shows a gram-
mar excerpt with two rules E and ON, with WS handling and discarding whitespace characters (the WS rule not being shown). Rule E consumes trigger-event definitions for state machine transitions of the form on doorClosed (line 5, Listing 2). It features the rule element Event enclosed by single grave accents (`...`). This is an instantiation generator that will translate into an instantiation call for an object-class Event.

To become useful, a parsing rule can be extended to include assignment generators (see Table 1, operator 14). These generators mark recognised and consumed values from the processed input as values to become assigned to the properties of objects created by an instantiation generator. Listing 1 shows the example of an assignment generator for a property name. The so-generated assignment binds any value returned from applying the parsing expression <alnum>+, that is, a string of at least one alphanumerical character. In the example, this value denote the event’s name.

Alternates. Each alternate at a RHS of a parsing rule, i.e., the operand parsing expressions of an ordered choice, can define an instantiation generator. The instantiation generators can point to the same or different object-classes. Listing 3 demonstrates how two alternative writing styles for transitions (i.e., on-go vs. go-on) could be defined as alternates.

In accordance with the semantics of ordered choices in parsing grammars, only the generator as part of the matching choice branch will be evaluated. For all but the transition definition on line 15, Listing 2, the first alternate applies; the second alternate applies then to the input on line 15.

Associations and References. Assignment generators allow a developer to relate objects, as defined by instantiation generators, in two ways: First, an assignment generator refers to a bare parsing expression. The result computed by this parsing expression will be bound as value of an assignment. Given the hierarchical relationship between parsing expressions, objects are therefore related in a manner reflecting the parsing hierarchy. A StateMachine references its State instantiations, each State maintains Transitions that, again, reference a trigger Event. This web of relations corresponds to the parsing procedure.

Second, assignment generators can be used to relate objects independently from the parse. This is required because an abstract-syntax graph typically involves some form of circular initialisation (Servetto et al., 2013). This refers to associations (references) established between objects beyond those induced by the parse, i.e., at different times of a parse. Circularity requires, to be fully resolved, that all objects to enter circular relationships have been fully initialised before.

Lines 2 and 4 in in Listing 2 exemplify a circular dependency between two declaration statements. Setting the start state to idle is in the preamble of the definition (line 2). The state, however, is about to be defined later (line 4). To defer the assignment, to a moment the remainder of the object graph with all states including idle has been constructed, an assignment operator can be extended into a second form. This second form nests a parsing expression with a query generator (see Table 1, operator 15). In Listing 4, the assignment generator for the property start is assigned a parsing expression that contains such a query expression: $\$$root states$0$.

A query expression allows for navigating and for accessing the object graph under construction. The first word of a query (e.g., the command) roots the query in the object graph: $\$$root refers to the top-level object corresponding to the root of the parse tree. $\$$parent refers to the ancestor object according to the parse tree. $\$$self is the self-reference to the receiver of the assignment. In addition, a query generator can refer to the parse matches of the surrounding parsing expression in a positional manner. For example, in Listing 4, $\$$0 will bind the first value computed by the first sub-expressions <alnum> at position 0 of the sequence expression.

The result of evaluating the query expression in an environment that provides values for the predefined variables (e.g., root, parent, 0) is then assigned to the property denoted by the assignment generator. The generated assignment, however, is deferred to a moment when all objects are guaranteed to being existing, according to the underlying parse.

3 COMPOSING PARSING EXPRESSIONS

A grammar composition relates a receiving grammar and one or more composed grammars, with grammars as defined in Section 2. A composition produces a
resulting grammar (Johnstone et al., 2014). The fundamental unit of composition is a grammar rule as a pair of a non-terminal as the rule’s LHS and a parsing expression as the RHS. A collection of rules having the identical LHS, but different RHS are said to be alternates. The multi-sets of all rules \( R \) of a receiving grammar and one or more composed grammars enter the composition. Composition operations on the rules sets fall into three coarse-grained groups: overriding, combination, and restriction (Johnstone et al., 2014). Concrete variations of these operation types then propagate between rules and into the sub-expression level (e.g., alternate selection).

In Section 3.1, concrete composition operations for OPEGs are introduced. This also highlights specifics to PEG (as opposed to production grammars). The fit of the composition operations for different types of language compositions is elaborated on in Sections 3.2 through 3.4.

3.1 Merges and Transforms

Object parsing grammars can be connected using a merges relationship. A receiving grammar can merge one or several composed grammars to obtain a resulting grammar, with and without intermittent grammar transformations between the merged ones. Composition starts from a disjoint union of the rules set of the receiving and composed grammars. To obtain a disjoint union, all symbols at the RHS and the LHS of the rules are qualified by their origin grammars. This union of all rules is the basis for transformations (extracts, rewrites) that yield the resulting OPEG. As a default, if no transformations have been defined, a simple union with override is performed.

The grammar definition in Listing 5 defines a merge relationship between two grammars: \( G_1 \) acts as the receiving, \( G_0 \) as the composed one.

OPEGs allow for multiple levels of defining merges relationships. Before computing a resulting grammar, the collection of composed definitions is turned into an unambiguous linear order. This linear order preserves local-precedence orders. Violations (e.g., circular merges) under linearisation are signalled at definition time\(^2\). This linearisation is then used to resolve dependencies between rules and as the basis for the subsequent transformations. The merges relationship does not directly determine which kind of composition operation is to be performed between the receiving and composed grammars. This is achieved in a separate step.

In addition to establishing a merges relationship, the receiving grammar can also define a script of grammar transformations to implement different composition operations. These include simple union with override in the absence of transformations, as well as different variants of extraction and of restriction in the presence of transformations. Transformations are defined as a dedicated section of a Grammar definition (see Listing 5). The transformations can be called repeatedly, causing a flush of the resulting grammar and a rerun of any transformations.

The composition behaviour in presence of transformations is implemented on the procedure illustrated in an informal manner in Figure 1 (steps a–d).

First, the set of rules of the input grammars \( (G_0, G_1) \) are processed to turn the non-terminals names into qualified names (a): A qualified non-terminal is a non-terminal whose name is prefixed by the name of the owning grammar. For example, non-terminal \( A \) becomes qualified as \( G_0::A \). Non-terminal \( B \) so becomes \( G_0::B \). Second, a union operation is performed with precedence for rules from the receiving grammars over those of the composed one (b). Due to prior name qualification, this represents effectively a disjoint-union operation. This has the consequence that the original sets of rules enter the intermediate set of rules in an unaltered and in a complete fashion. Third, the defined transformations (e.g., extraction, restriction) are performed on this intermediate set of rules (c). In Figure 1, the example refers to the removal of the rule with the LHS non-terminal \( G_0::B \). Fourth, after completion, standard grammar cleaning

---

\(^2\)Our OPEG implementation represents parsing grammars as object-classes: The merges relationship is therefore derived from an ordinary subclass-superclass relationship. This way, OPEGs can leverage the built-in \( C3 \) linearisation (Barrett et al., 1996).

\(^3\)Johnstone et al. (2014) refer to this auxiliary transformation as introducing name hygiene.
is performed, most importantly: unrealisable (including undefined) non-terminals are dropped, then unreachable non-terminals are removed (d).

As for the actual grammar transformations supported in step (c) of Figure 1, an overview of the available operators is presented in Table 2. An (1) extract w/o rewrite (⇐) selects the RHS expression of the referenced rule (e.g., G0::A) and introduces it into the receiving rules set. Introduction refers to either creating a new rule G1::A with the extracted RHS or appending the selected RHS as an additional alternate to an existing rule. An (2) extract w/ rewrite (⇒) proceeds as the extract. Additionally, it renames any non-terminals reachable at the extracted RHS expression using the prefix of the receiving grammar (e.g., substituting prefix G1::A for G0::A). The (3) transitive variant of extract w/ rewrite (⇒) additionally imports any rules providing definitions for the extracted and renamed RHS non-terminals. These rule dependencies are satisfied from the pool of linearised composed grammars. Finally, (4) resulting grammars can be restricted by using the removal operator (⇒). A removal affects an entire rule or a rule alternate. To selectively insert an extracted RHS as a new alternate, the operators 1–3 allow for defining an insertion position, e.g., A ⇐ G0::A 0. The extracted RHS expression becomes inserted at the first (zero-based) position. That is, it is prepended as a new alternate to a rule, if existing. The position qualifier defaults to prepending extracted expressions⁴.

Generators. The generators for instantiations, assignments, and queries as part of object parsing expressions are integral parts of the parsing expressions also under transformation. Generators become combined, extracted, and removed with the surrounding parsing expressions or sub-expressions (alternates) according to the stipulated behaviour of the first four operators (1–4). This is particularly important for choice expressions. At their top level, generators are elements of each alternate and can become inserted or removed during a transformation affecting the respective alternate. However, to reference and to reuse parsing (sub-)expressions without their generators (e.g., to attach matches to an alternative generator), there are two transform operators that operate on the plain expressions, without generators (see operators 5 and 6 in Table 2). An (5) extract w/o generators (⇐) selects the RHS expression of the referenced rule (e.g., G0::A), omitting any generators, and introduces it into the receiving rules set (see also operator 1). An (6) extract w/ rewrite w/o generators (⇒) performs the extraction/introduction and patches the namespace prefixes (see also operator 2), again, omitting any generators. See Sections 3.2–3.4 for concrete applications of these two generator-free operators.

As already explained, in absence of transformations, a merges relationship defaults to a union with override. When forming the union, the receiving rules take precedence over the composed ones.

The operand values consumed by the six operators listed in Table 2 must be qualified (G0::A) or unqualified non-terminal names (A). Unqualified names, both on the left-hand side and on the right-hand side, will be narrowed by automatically prepending the enclos-

---

⁴This default is a consequence of the issue of language hiding in PEG.
Object Parsing Grammars with Composition

3.2 Syntax Extension

A language developer composes a base language, e.g., for defining graphs using DOT in Listing 6, with a language extension. A language extension is an incomplete language fragment which depends directly on the base language for completion (in terms of the concrete syntax, the abstract syntax, and the behaviours; Erdweg et al. 2012). For example, graph definitions using DOT node and edge statements should be extended to model edge-weighted graphs (see Listing 7). For this, the DOT notation must be extended accordingly to support attribute statements, so that edge weights can be captured as attributes (see Figure 2a).

Listing 7 shows an object parsing grammar including \textit{generators} for instantiations, assignments, and queries recognising graph definitions w/o weight attributes: the base notation. The object parsing grammar establishes three correspondences between parsing expressions (their matches) and a class model of graphs:

1. Matches obtained by NodeStmt map to instantiations of the Node class.
2. Matches obtained by EdgeStmt map to instantiations of the Edge class.
3. Matches obtained by the top-level or start rule \texttt{G} map to instantiations of the Graph class.

In addition, the Node instantiations must be initialised to the provided node names. The Edge instantiations must obtain references to the Node instantiations identified by the node names given in DOT edge statements. Finally, all Edge instantiations must be assigned to the edges property of the Graph.

To allow for edge statements to carry attribute statements in-between brackets defining edge weights (see lines 7 and 8 in Listing 6), among others, the base grammar can be composed with a grammar extension. This extension can be realised in different manners using OPEGs. Options include a straightforward union between two OPEGs or a grammar transformation. In the following, OPEG transforms are exhibited.

As for extra rules (the \textit{receiving} grammar’s rules in Listing 8a, the rules \texttt{WeightAttr} and \texttt{Weight} de-
Consider the transforms shown in Listing 8b, one step (line) at a time: Recall that in presence of transforms, merging produces a disjoint union of two sets of rules, with all rules and non-terminals being prefix-qualified by their originating grammars (see Figure 1, step b). Therefore, at the start, there will be effectively two EdgeStmt rules in the intermediate set of rules: Dot::EdgeStmt from the base grammar and ExtDot::EdgeStmt from the extension grammar (see Listing 8a).

Based on this intermediate set, the transforms can be used to extract the RHS of Dot::EdgeStmt, park it in a helper rule for ExtDot::CoreEdge, and reference this helper from the RHS of ExtDot::EdgeStmt. This corresponds to what is achieved by the \( \leftrightarrow \) transform of Listing 8. Then, to integrate this revised EdgeStmt with the remainder of the composed grammar, the transitive-extraction transform (\( \Rightarrow \)) is used to draw the entirety of the composed grammar into the namespace of the receiving grammar. This starts from the start symbol Dot::S. The operation will pick up the previously defined, revised ExtDot::EdgeStmt. This, in turn, activates the added syntax for weight attributes via ExtDot::CoreEdge.

Syntax extensions using explicit OPEG transformations, rather than a union with override (also supported), avoids duplicated rules. Besides, all changes to the base (composed) grammar will be automatically tracked by the extended (resulting) one. In addition, accidental overrides are avoided by maintaining the merged sets of rules in separate namespaces.

3.3 Syntax Unification

Composing two or more, otherwise free-standing and independent languages and their syntaxes has been referred to as language unification. Unification of the two or more languages mandates that they maintain their standalone functional properties, and still, when composed, interoperate without modifying their implementation. A unification must preserve the composed syntaxes, leaving them intact and unmodified. In this setting, the same basic composition operations provided between two or more OPEGs apply, as for language extension or extension compositions. Important differences arise from the fact that unintended or accidental overrides, for instance, are much more likely. This is because one syntax definition might contend with the other’s symbol names, whitespace conventions, and reserved literals (e.g., keywords).

Consider two separately developed languages. These are a Boolean and comparison expression language (BCEL) and a state-machine-definition language (SMDL). See also Figure 2. The BCEL is a candidate of a functional kernel language (Volter, 2018) to become unified with SMDL to implement guarded transitions. A guarded transition is a transition that is annotated by a guard expression and whose firing is controlled by the prior evaluation of the attached guard expression. If the guard is evaluated to true at that time, the transition is enabled, otherwise, it is disabled and will not fire. Listing 9 shows two transitions, one with and the other without a guard expression.
A unification is marked by two or more composed grammars being merged by a receiving (unifying) grammar. The running example requires the developer to define a receiving grammar (e.g., Guarded-MGC) that merges the BCEL's grammar and the SMDL grammar. The definitional content of the unifying grammar is documented by Listing 10. Guard expressions are attached to the Transition instantiations.

Intuitively, the unification is achieved in three transformational steps (see Listing 10b): First, a revised rule definition for transition definitions is provided. This rule derives from the original definition via the rewrite transform named OrigT. Note that generators are excluded from the rewrite transform. This is to avoid duplication of the instantiation generator for Transition in the resulting grammar. The revamped rule T becomes extended by an assignment generator guard. This will establish the guard reference between an instance of Transition and an instance of Expression. Second, the assignment generator is related to the Expression rule of the BCEL grammar. This rule becomes referenced on the second to last line of Listing 10. Third, and finally, the entire SMDL rules set is dragged into the resulting grammar (on the last line).

These transforms resemble closely the ones for a syntax extension, regarding the provision of an extension point in the state-machine language (OrigT). The main difference comes with the referencing of a syntax element from the second composed language (extract w/o rewrite): BCEL::Expression. As this happens to be the start symbol of BCEL, the entire BCEL rules set is effectively incorporated into the resulting grammar. This is achieved in a way that avoids conflicts with the state-machine rules set (thanks to prefixing of non-terminals).

To summarise, OPEGs with transforms allow for the unanticipated, the unmodified, and the controlled reuse of two independently developed syntaxes to form a unified syntax.

### 3.4 Syntax-extension Composition

Extension composition captures situations in which two or more syntax extensions can be composed with one another or can be co-present as an extension to a base syntax. Two or more extensions may be composed incrementally (step-wise) into a base language, one at a time. Alternatively, the extensions can be composed first, and the resulting grammar becomes merged once with a base grammar. The first is sometimes referred to as incremental extension composition, the second as extension unification. OPEGs support both variants of syntax-extension composition, with incremental compositions being a flavour of syntax extensions as described in Section 3.2. In what follows, the emphasis is on extension unification. As the name implies, this combines aspects of syntax extension (Section 3.2) and syntax unification (Section 3.3).

In an extension unification, the points of departure are the extension grammars per se. First, the extensions become composed, then, as a last step the unified (resulting) grammar is merged into the base grammar (see Figure 2c). The unified extension grammar is, as its merged extension grammars, abstracted, that is they must be completed by merging a base grammar.

Consider the example of a second syntax extension to the DOT-like graph-modelling language introduced in Section 3.2. This extension’s aim is to add a colour attribute to edge definitions. Colour attributes carry 3-digit hex codes of colours as part of edge definitions, as depicted in Listing 11.

The basic flow of an extension unification is exemplified in Listing 12 for a colour- and weight-enabled graph syntax. The two main steps are identified as steps (2) and (3). In step (2), a unified extension is created by composing the two grammars documented (as excerpts) in Listings 13 (for weight attributes) and in Listing 14 (for colour attributes). The actual unification of the underlying rules sets in accomplished by
# 1) weighted extension
Grammars create\n-\start EdgeStmt $weightedGrmStr$

# 2) unified (coloured+weighted) extension
Grammars create\n-\start EdgeStmt $weightedGrmStr$

# 3) base + unified extension
Grammars create FinalGrm \n-\start G \n-merges [WeightedExtGrm] $colouredGrmStr$
{ \n    EdgeStmt $\Rightarrow$ WeightedExtGrm::EdgeStmt
}

Listing 12: An actual extension unification implementation. First, two grammars embody the extension grammars: WeightedExtGrm, ColouredWeightedExtGrm. The latter reifies the unified grammar of the two extensions. Finally, the FinalGrm becomes composed from the unified extensions and the original DOT grammar (whose definition is not shown here).

Listing 13: Excerpt from the extension grammar for the weighted feature, as used for extension unification.

\begin{verbatim}
# rules
EdgeStmt ← 'Edge' CoreEdge
WeightAttr ← OSQBRACKET WEIGHT
EQ weight:Weight;
Weight ← 'Weight'
value:<digit>+
# deferred
CoreEdge ← '('
void: WS ← ''
\end{verbatim}

Listing 14: Excerpt from the extension grammar for the coloured feature, as used for extension unification.

See WS and CoreEdge in Listings 13 and 14 for examples. In step (3), the base (DOT) grammar is merged together with the unified extension (ColouredWeightedExtension) into a completed and operative grammar (FinalGrm). From this final grammar, a parser can be derived.

Two details are noteworthy about this final compositional step: First, this final receiving grammar does not introduce any new rules. Second, it is the resulting grammar of the unified extension becoming merged into the final grammar, and not the receiving grammar of the unification itself. See line 16 of Listing 12.

As for the first detail: There are no dedicated rules for the final grammar because its rules set is populated purely from running grammar transformations (see lines 18–22 in Listing 12). This is not only permitted in OPEGs, but also corresponds to the nature of an extension unification. The first three transforms (lines 18–20) provide actual definitions for the deferred non-terminals coming with the unified extensions (WS and CoreEdge). Without the upfront removal of the ε-placeholders, the resulting grammar would remain dysfunctional. Recall that definitions present in the grammars under composition are turned into alternates of a combined rule. The subsequent two lines 21 and 22 load the sets of rules of the two composed grammars into the final resulting one. This is equivalent to the use of the transitive extract/rewrite transformation, as applied for syntax extension and for syntax unification.

To recap, extension unification differs from stepwise syntax extensions in that at the time of composing the extensions, they are treated in isolation from any base grammar. Most importantly, any undefined or deferred non-terminal definitions must be provided either by the extension grammars themselves or any intermediate (unified) grammar (at least in terms of placeholders). This complicates a unification, as com-
pared to repeated syntax extensions. However, uni-
fication also presents immediate benefits. One up-
side is that the extension unification is defined under a
closed-world assumption: The start symbols point
to parsing rules introduced by the extension gram-
mars; any deferred non-terminals are clearly marked
as such\(^5\). Another consequence is that an exten-
sion unification is symmetric as opposed to incremen-
tal composition. Extensions are composed as peers,
without one taking precedence over the other. In addi-
tion, extension unification provides more control over
a syntax composition (see derivatives in Section 4).

4 DISCUSSION

Language Hiding Revisited. Language hiding
(a.k.a. pre-emptive prefix capture) is a practical con-
sequence of the absence of general semi-disjointness
of a choice expression for the scope of the language
matched by a PEG (see Section 2.1). This is counter
the otherwise practical, advantageous consequence of
PEGs precluding ambiguity. Language hiding has im-
plications for composition operations as introduced
in Section 3.1, in that alternates become automatic-
ly (combination) or selectively added (extraction
w/ and w/o insertion position). Any added alternate
may unintentionally hide others, and, therefore, im-
portant fragments of the matched language.

The implementation of OPEGs tries to minimize
unintended language hiding for two or more com-
posed grammars, by applying precautionary defaults:
For example, alternates introduced by DSL exten-
sions are prepended to those of the receiving gram-
mar. This follows from the assumption that, in exten-
sions, the aim is to capture longer prefixes. Beyond
that point, manual inspection (Redziejowski, 2018)
and fine-grained control during composition are sup-
ported (explicit alternate positioning).

Grammar Cleaning. In Section 3.1, it was estab-
lished that techniques for reducing (“cleaning”) object parsing grammars from unrealisable and un-
reachable non-terminals is a building block for mod-
elling and implementing certain composition opera-
tions (see Figure 1). For production grammars (CFG),
this is a matter of static analysis (Aho and Ullman,
1972, Section 2.4.2). For parsing grammars, in the
general case, finding useless (non-recognising, unde-
fined, and unused) non-terminals is known to be un-
decidable (see Grune and Jacobs 2010, p. 507 and
Ford 2004, Section 3.5). This is, again, due to the is-
ue of non-disjointness of the ordered-choice operator
(see Section 2.1): The evaluation of some alternate is
conditional on the success or failure of its preceding
alternates.

The parsing expression with two alternates ‘a’ /
‘ab’ is pathological because it will only recognise
a in all inputs prefixed by a single a (e.g., aa, ab).
The second alternate is realisable (i.e., it recognises a
literal string) but is effectively shadowed by the first
alternate (‘a’). Therefore, even if an alternate expres-
sion can be statically marked as realisable (i.e., it does
recognise and possibly consume at least one terminal
on the input stream), it may be actually unreachable
in the order of any evaluation of a given choice ex-
pression.

(Practical) Workarounds are tool-supported man-
ual inspection (Redziejowski, 2018) or leveraging
higher-level CFG that are transformed into corre-
ponding PEG, said being well-behaved having only
choice expressions containing alternates then known
to be disjoint (Mascarenhas et al., 2014). None of
these apply to automated grammar cleaning, however.
For the scope of this work, a conservative approxima-
tion is applied. An approximative cleaning of pars-
ing grammars will lead to false negatives. A false
negative is a non-terminal marked as realisable that
may still turn out unreachable, conditionally. There-
fore, for an OPEG, it is not able to obtain fully re-
duced grammars and fully optimised parsers. How-
ever, the approximation is sufficient for cleaning re-
sulting grammars from composition artefacts, such as
non-terminals becoming undefined.

Additional Composition Types. Beyond the basic
types covered in Section 3, one can realise language
restrictions and higher-order extension unifications
(derivatives).

By default, and to maintain closure under com-
position, rules are combined as alternates. This ef-
ectively widens the matching space of the resulting
grammar as compared to the composed one. If the re-
resulting syntax should be restricted to disallow previ-
ously allowed syntax elements, one must restrict the
resulting grammar by removing alternates explicitly.
This restriction can be achieved by employing the re-
move operator (⇒; see Table 2, operator 4).

In syntax-extension composition, a developer
must realise a dual goal. A developer must (a) pro-
vide for coordination code to accommodate the two
co-present syntax extensions. In addition, the devel-

\(^5\)These are not only matters of definitional clarity, but
also an implementation-level requirement: Once composed,
the resulting grammar will be cleaned from useless non-
terminals. To prevent this from happening, there must be
placeholder definitions.
oper must to (b) implement the coordination code in a way that both syntax extensions remain deployable in isolation from each other. OPEGs help achieve this double goal by derivative extension composition. Coordination code can be provided as a dedicated grammar (derivative grammar, Liu et al. 2006). This grammar provides for extra rules and transforms to resolve unwanted interactions such as syntax failures of two or more syntax extensions (composed grammars).

5 RELATED WORK

The relevant context is set by approaches to composable and modular grammars. Grammar (definition) reuse without modification (Erdweg et al., 2012) is referred to, but mainly with respect to the limitations perceived at the time. These include conflicting lexers (token scanners) vs. scannerless parsing and parser generators being limited to single and closed grammar definitions.

To overcome these limitations, first contributions included syntax modules of the series of Syntax Definition Formalisms (SDF, SDF2, SDF3; Visser 1997), the grammar-inclusion mechanism by TXL (Cordy, 2006) and grammar imports by ANTLR (Parr, 2013, pp. 257). These approaches turned grammars into open definitions. SDF starting with version 2 introduced parametrised syntax modules. Modules can import from each other. When an imported module exposes non-terminals or terminals as named module parameters, they can be bound under different names in the importing module. The same can be achieved in SDF using explicit renaming, without formal parameters. SDF is contained by a number of language development systems, including Spoofax and RascalMPL. SDF also addressed composability issue by operating on scannerless and generalised parsing (i.e., scannerless GLR). It is noteworthy that the support for parametrised modules has been discontinued starting from SDF3.

TXL allowed a developer to place rules over different files, rooted under one start symbol though. In addition, TXL provided for a referne to replace or add a new alternate to a given rule. ANTLR, as elaborated on in this section, applies a union-with-override technique, with particularities regarding different types of definition artefacts. As ANTLR serves as the parsing infrastructure for several language development systems such as Xtext (Bettini, 2013), MontiCore (Krahn et al., 2010), MetaDepth (Meyers et al., 2012), grammar imports have seen uptake.

Throughout Section 3, the PEG-based system Rats! was referred to, mainly because Rats! provides for basic grammar composition on the basis of rules and alternates using dedicated transformations (add, delete, append). In addition, Grimm (2006) highlights important barriers to composing PEG-based syntax definitions (e.g., ordering).

6 CONCLUDING REMARKS

This paper departs from the foundations of advanced parsing expression grammars (PEG) and delivers object parsing-expression grammars (OPEGs) to define—in one—a concrete syntax and the mapping to an object-oriented primary abstract syntax (language model). The double aim is to avoid common abstraction mismatches of parse representations (e.g., decomposition mismatches) and to render the extended parsing grammars composable. The extended parsing grammars support different composition techniques under the umbrella of a uniform framework: simple grammar unions and fine-grained grammar transformations. The latter transformations are backed by well-defined operators taking as input the sets of parsing rules of two or more OPEGs at different levels (e.g., rule-wise, alternates) to form a valid OPEG as their output. The transformation procedure and the provided operators enable developers to mitigate the consequences of unwanted language hind by PEGs under composition. This coverage of robust composition techniques is shown to be necessary to enable a developer to implement the different grammar compositions relevant for realising language-product lines: extensions, unification, extension composition, and derivative grammars.

REFERENCES


Degueule, T. (2016). Composition and Interoperability
for External Domain-Specific Language Engineering. Theses, Université de Rennes 1 [UR1].