Coupled Transformations of Shared Packed Parse Forests

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. SPPF (shared packed parse forest) is the best known graph representation of a parse forest (family of related parse trees) used in parsing with ambiguous/conjunctive grammars. Systematic general purpose transformations of SPPFs have never been investigated and are considered to be an open problem in software language engineering. In this paper, we motivate the necessity of having a transformation operator suite for SPPFs and extend the state of the art grammar transformation operator suite to metamodel/model (grammar/graph) cotransformations.

1 Motivation

Classically, parsing consumes a string of characters or tokens, recognises its grammatical structure and produces a corresponding parse tree [1,52]. However, sometimes we end up in situations when trees are not expressive enough. The most common scenarios include generalised parsing and Boolean grammar-based parsing. *Generalised parsing algorithms* (GLR [43], SGLR [44], GLL [39], RIGLR [38], etc) differ from the classic ones in dealing with ambiguities [7]: instead of trying to avoid, ignore or report them, ambiguous parses result in so called parse forests — sets of equally grammatically correct parse trees. In practice, these sets usually need to be filtered or ranked in order to make full use of the available tree-based approaches to program analysis and transformation. In *Boolean grammars* [34] and *conjunctive grammars* [33], we have conjunctive clauses in a grammar as first class citizens and must treat them properly when parsing, which means having special kinds of nodes in a parse tree whose descendant subtrees share leaves [35]. Both kinds of structures defined by these two related approaches conceptually are *parse forests*.

There have been various attempts in the past to represent parse forests. The earliest ones required a grammar to be in a Chomsky Normal Form [11] — theoretically a reasonable assumption since any context-free grammar can be normalised to CNF, but ultimately we need a parse forest for the original grammar, not for the normalised one, which would require bidirectional grammar transformations [46] to be coupled with tree and forest transformations, which is far from trivial.

The next attempt in representing parse forests revolved around tree merging [14]: such a parse forest representation would result in a tree-like DAG with all the edges of all the trees in the forest. This is obviously an overapproximation of the forest (see Figure 1), which requires additional information in order to be unfolded into a set of trees — in other words, in order for any sensible manipulation to happen. Obviously, having a data structure that requires so much nontrivial postprocessing overhead, is highly undesirable.

The best representation of a conceptual parse forest (a set of trees with equal lists of leaves) so far is a so-called *shared packed parse forest* [43, §2.4], SPPF from now on: its components are merged from the top until the divergent nodes, and due to maximal sharing the leaves and perhaps even entire subtrees grouping leaves together, are also merged. An example of such a graph is given on Figure 2. Formally, an SPPF is an acyclic ordered directed graph where each edge is a tuple from a vertex to a linearly ordered list of successors and each vertex may have more than one successor list. If V is a set of vertices, then edges are:

$$E = \{\langle v_i, (v_{i1}, v_{i2}, ..., v_{ik_i})\rangle \mid v_i \in V, v_{ij} \in V\} \subseteq V \times V^*$$

SPPF-like structures are used nowadays both in software language toolkits that allow explicit ambiguities (such as Rascal [22]) and those that allow explicit conjunctive clauses (such as TXL [42]). For a detailed view on the implementation details we refer the readers to a paper on ATerms [5]. However, the theory of their transformations is underdeveloped — this was pointed out as one of the major open problems in modern software language engineering by James Cordy and explained in his recent keynote at the OOPSLE workshop [3].

2 Transformation

For many years trees have been the dominant data structure for representing hierarchical data in software language processing. They are remarkably easy to define, formalise, implement, validate, visualise and transform. There are many ways to circumvent data representation as graphs by considering a tree together with a complementary component such as a relation between its vertices that would have turned a tree into a cyclic graph, as well as many optimisations of graph algorithms that work on skeleton trees of a graph. Take, for instance, traversing a tree — it can be done hierarchically from the root towards the leaves or incrementally from the leaves towards the root, each case guaranteed termination even if the traversal is not supposed to stop when a match is made. This naturally provides us with four traversal strategies found in metaprogramming: bottom-up-continue, bottom-up-break, top-down-continue and top-down-break [8,22]. More sophisticated and flexible traversal strategies exist (e.g., Nuthatch [2]), but the actual need for them is rather rare. For a detailed overview of visiting functions, strategic programming and typed/untyped rewriting we refer the readers to the work of van den Brand et al [9] and the bibliography thereof. This section is focused on finding existing techniques that can be or are in fact SPPF transformations.

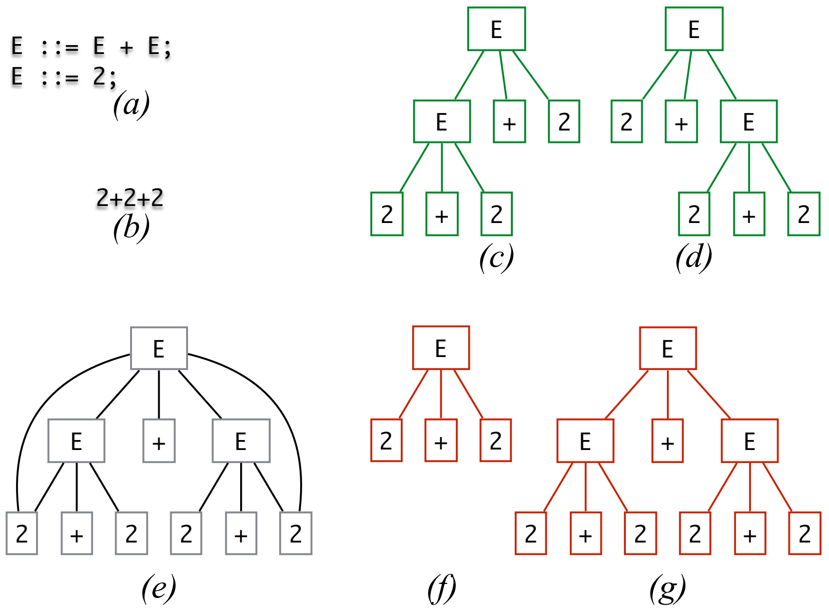
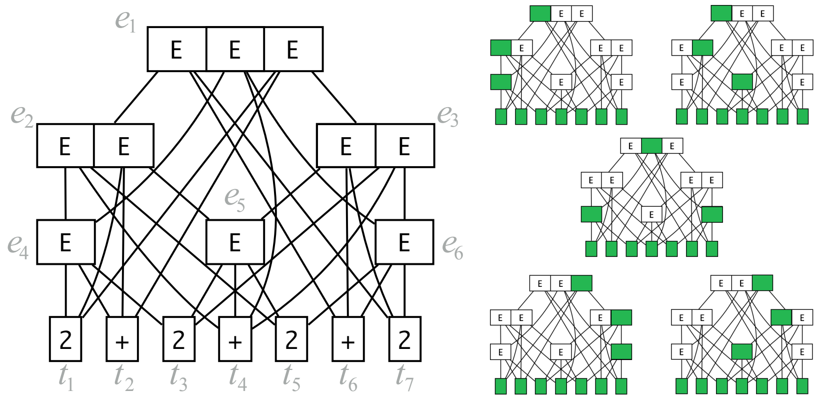


Fig. 1. Demonstration that the Earley representation overapproximates parse forests: **(a)** a simple ambiguous grammar example; **(b)** a term with ambiguous parse; **(c)**&**(d)** correct parses; **(e)** the graph representation of the forest suggested by Earley [14]; **(f)**&**(g)** incorrect parse trees that are well-formed according to the grammar (a) and covered by the parse tree representation (e), but not corresponding to the actual term (b).

2.1 Disambiguation

One of the relatively well-researched kind of SPPF transformations is disambiguation — it is commonly practised with ambiguous generalised parsing because static detection of ambiguity is undecidable for context-free grammars [10]. However, most of the time the intention of an average grammarware engineer is to produce one parse tree, so this line of research is mostly about leveraging additional sources of information to obtain a parse tree from a parse forest. There are three main classes of disambiguation techniques:

- ◇ Ordered choice, dynamic lookahead and other conventions aimed to *prevent* ambiguities altogether or avoid them. These are fairly static, relatively well-understood and widely used in TXL [12], ANTLR [37] and PEG [16].
- ◇ Follow/precede restrictions, production rule priorities, associativity rules and other annotations for local sorting (preference, avoidance, priorities) that help to prune the parse forest *during* its creation. Since these are algorithmic approaches in a sense that they modify the generation process of an SPPF and thus are not proper mappings from SPPFs to SPPFs, we will not consider



$$\{ \langle e_1,(e_2,t_6,t_7)\rangle, \langle e_1,(e_4,t_4,e_6)\rangle, \langle e_1,(t_1,t_2,e_3)\rangle, \langle e_2,(e_4,t_4,t_5)\rangle, \langle e_2,(t_1,t_2,e_5)\rangle, \langle e_3,(e_5,t_6,t_7)\rangle, \langle e_3,(t_3,t_4,e_6)\rangle, \langle e_4,(t_1,t_2,t_3)\rangle, \langle e_5,(t_3,t_4,t_5)\rangle, \langle e_6,(t_5,t_6,t_7)\rangle \}$$

Fig. 2. One the left, an SPPF graph resulted from parsing the input "2+2+2+2" with the grammar from Figure 1 (a). On the right, there are five parse trees in a forest, which are packed in a triple ambiguity, two of subgraphs of which have double ambiguities. All of them share leaves and subtrees whenever possible. Below the pictures we show its formal representation as an ordered directed graph.

them in the rest of the paper and refer to other sources primarily dedicated to them [7,4].

◇ Disambiguation filters that are run *after* the parsing process has yielded a fully formed SPPF: their main objective is to reduce the number of ambiguities and ultimately to shave all of them off, leaving one parse tree. An example of this would be how processing production rules marked for rejection is done for SGLR [7] and GLL [4] — even though recursive descent parsers can handle an equivalent construct (and-not clause) during parsing without any trouble [42].

Formally speaking, the first class never produces parse forests; the second class works with disambiguators (higher order functions that take a parser and return a parser that produces less ambiguous SPPFs) [7]; the third class uses filters (functions that take an SPPF and produce a less ambiguous SPPF) [23]. In some sources approaches with disambiguators are called "semantics-directed parsing" and approaches with filters are called "semantics-driven disambiguation" [6], since both indeed rely on semantic information to aid in the syntactic analysis. Disambiguation filters are still but a narrow case of SPPF transformation, but they have apparent practical application and are therefore well-researched.

2.2 Grammar programming

Grammar programming is like normal programming, but with grammars: there is a concrete problem at hand which can be solved with a grammar, which is then being adjusted until an acceptable solution emerges. A representative pattern here is working with a high level software artefact describing a language (we assume it to be a grammar for the sake of simplicity, but in a broad sense it can be a schema, a metamodel, an ontology, etc), from which a tool solving the problem at hand is inferred automatically.

There are at least three common approaches to grammar programming: manual, semi-automated and operator-based. *Manual grammar programming* involves textual/visual editing of the grammar file by a grammarware engineer. It is the easiest method in practice and is used quite often, especially for minor tweaks during grammar debugging. However, it leads to hidden inconsistencies within grammars (which require advanced methods like grammar convergence to uncover [31]), between changed grammars and cached trees (which demand reparsing) and between grammars and program transformations (which requires more manual labour). *Semi-automated grammar programming* adds a level of automation to that and thus is typically used in scenarios when a baseline grammar needs to be adjusted in different ways to several tasks (parsing language dialects, performing transformations, collecting metrics, etc). Usually the grammarware toolkit provides means to extend the grammar or rewrite parts of it — examples include TXL [12], GDK [24] and GRK [28]. Arguably the latter two of these examples also venture into the next category since they contain other grammar manipulation instruments like folding/unfolding. If we extend this arsenal with even more means like merging nonterminals, removing grammar fragments, injecting/projecting symbols from production rules, chaining/unchaining productions, adding/removing disjunctive clauses, permuting the order and narrowing/widening repetitions, we end up having an *operator suite* for grammar programming. The advantage of having such a suite lies in the simple fact that each of the operators can be studied and implemented in isolation, and the actual process of grammar programming will involve calling these operators with proper arguments in the desired order. Examples of operator suites include FST [29], XBGF [31], ΞBGF [46] and SLEIR [49].

2.3 Coupled transformation

We speak of coupled transformations when two or more kinds of mutually dependent software artefacts are transformed together to preserve consistency among them: usually one changes, and others co-evolve with it [27]. Naturally, the first coupled transformation scenario we should think of, involves an SPPF and a grammar that defines its structure. This change can be initiated from either side, let us consider both.

Assuming that we have a sequence of grammar transformation steps, we may want to execute them on the language instances (programs) as well, to make them compatible with the updated grammar. Such a need arises in the case of grammar convergence [30], when a relationship between two grammars is reverse engineered by programming the steps necessary to turn one into the other, and a co-transformation can help to migrate instances obtained with one grammar to fit with the other. For example, we could have a grammar for the concrete syntax and a schema for serialisation of the same data — a transformation sequence that strips the concrete grammar from elements not found in the schema (typically terminals guiding the parsing process such as semicolons and brackets), could also be coupled with a transformation sequence that removes the corresponding parts from the graphs defined by them (e.g., a parse tree and an XML document).

Consider another scenario where we have the change on language instances and want to lift it to the level of language definitions. An example could be found in program transformation, a common software engineering practice of metaprogramming. If we want a refactoring like extracting a method, renaming a variable or removing a go-to statement, it is easy and practical to express it in terms of matching/rewriting paradigm: in Spoofax [20], Rascal [22], TXL [12], ATL [19], XSLT [21], etc. However, a correct refactoring should preserve the meaning of the program, and the first step towards that is syntactic correctness of this program. For non-refactoring transformations found in aspect-oriented development, automated bug fixing and other areas, we still want to ascertain the extent to which the language is extended, reduced or revised. In the case of strongly typed metaprogramming languages, they will not allow you to create any ill-formed output, but the development process can lead you to *first* specify a breaking transform and *then* cotransform the grammar so that it "fits" — which is what coupled transformations are good for.

2.4 Explicit versus implicit

This was already mentioned before, but becomes a crucial point from now on: parse forests can arise from two different sources — conjunctive clauses in the grammar used for parsing and generalised parsing with ambiguous grammars. The latter case can be considered implicit conjunction, since it is present on the level of language instances but not on the grammar level. In that case, instead of a more cumbersome construction specifying a precise parse, we use a simpler grammatical definition which yields a forest. If a grammar is both conjunctive and ambiguous, this can lead to its both implicit and explicit conjunctive clauses to be found in SPPFs — with no observable difference on an instance level.

Similarly, some of the transformations will "collapse" conjunctions, making one branch of a clause equal to another. Formally, for an SPPF node to have several branches means existence of several edges in the form $\langle v_i, (v_{i1}, ..., v_{ik_i})\rangle$, $\langle v_i, (v'_{i1}, ..., v'_{ik'_i})\rangle$, etc. When a transformation results in all v_ij becoming equal to the corresponding v'_{ij}, such edges merge in the set. If such conjunctions represent ambiguities, this is disambiguation; if they represent parse views, it merges the views and makes them undistinguishable.

	Language preserved	Language extended	Language reduced	Language revised
SPPFs preserved	`bypass eliminate introduce import vertical horizontal designate unlabel anonymize deanonymize renameL renameS`	`addV addH define`		
SPPFs preserved or fail			`removeV removeH undefine`	
SPPFs refactored	`unfold fold inline extract abridge detour unchain chain massage distribute factor deyaccify yaccify equate rassoc lassoc renameN clone concatT splitT`	`appear widen upgrade unite removeC`	`disappear`	`abstractize project concretize permute renameT splitN`
SPPFs refactored or fail			`addC narrow downgrade`	`redefine replace reroot`
fail				`inject`

Fig. 3. The XBGF operator suite designed for convergence experiments [30,31,51] and updated here to the latest version of the GrammarLab. Columns of the table refer to the effects of the operators on the string language generated by a grammar; rows classify coupled effects on the SPPFs.

3 Grammar-based gardening

XBGF (standing for "transformations of BNF-like grammar formalism") was an operator suite for grammar programming originally developed for grammar recovery and convergence experiments [30,31] and used for various grammar maintenance tasks afterwards — e.g., for improving the quality and maturity of grammars in the Grammar Zoo [47,50]. It has operators like **eliminate**(n) that checks whether the given nonterminal n is referenced anywhere in the grammar, and if not, removes its definition harmlessly; or operators like **removeN**(x, y) that ensures that the nonterminal x is found in the grammar while y is not, and subsequently renames x to y; or even operators like **redefine**(p_k, p'_k) which removes all production rules p_k defining one nonterminal from the grammar and replaces them with rules p'_k defining the same nonterminal differently. These operators are relatively well-studied so that we can always make a claim about the effect that a transformation chain has on the language generated/accepted by the grammar. Originally [45, §7] XBGF operators were classified according to their preservation, increase, decrease or revision of the language within two semantics: the string semantics and the term semantics. The contribution of this section is their classification according to the coupled effect of the operators on the SPPFs — see Figure 3 for the overview.

3.1 SPPFs preserved

The best kind of coupled transformation is the trivial one where the initial transformation triggers no change in the linked artefacts.

3.1.1 Language-preserving operators

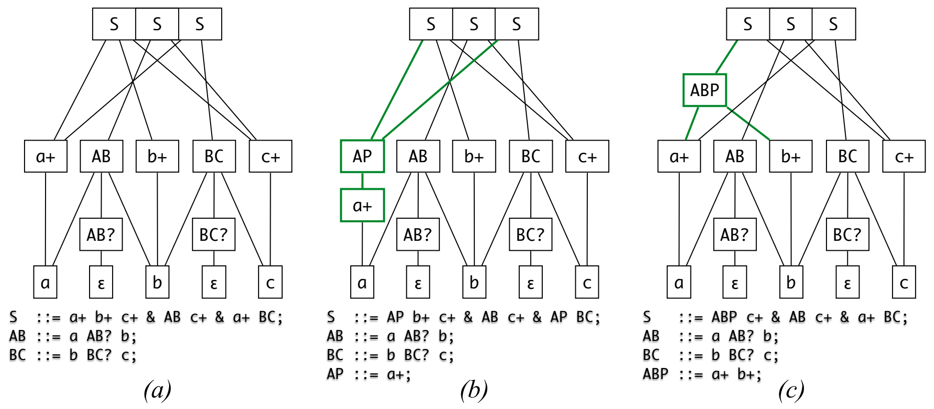
Many operators that preserve the (string) language associated with the grammar, also preserve the shared packed parse forests of the instances of this language. Consider, for instance, the **eliminate**(n) operator we have just introduced in the previous paragraph: essentially, it removes an unused construct. Since such a construct is unused in other production rules, it can never be reached from the root symbol, so it can also never occur in the graphs representing grammatically correct programs. Hence, any SPPF which was correct for grammar before the transformation, is still correct for the grammar with the unused part eliminated. Similarly, introducing a language construct that was not previously there and is not (yet) linked to the root, has no impact on the forests. The same argumentation holds for decorating operators that add/remove labels to/from rules of the grammar or their subexpressions, or rename them.

The last two operators seen in this cell on Figure 3 are **vertical**(n) and **horizontal**(n) — they facilitate switching between a horizontal style of grammatical definitions (i.e., "`A ::= B | C;`") and a vertical one (i.e., "`A ::= B; A ::= C;`") — some grammatical frameworks distinguish between them, but never on an instance level, since a realisation of a disjunction commits to one particular branch. Hence, these operators also have no impact on SPPFs.

3.1.2 Language-extending operators

In the same way rearranging alternatives in production rules discussed in the previous section, has no impact on SPPFs, strict language extension operators like **addV**(p) and **addH**(p) have no impact on the forests. Since disjunctive clauses are not explicitly visible in SPPFs, any tree or forest derived with the original grammar, also conforms to the transformed one — the coupled instance transformation is trivial.

There is even one operator which is very invasive on a grammar level while being entirely harmless on the instance level — **define**(p) is a variant of **introduce**(p) that adds a definition of a nonterminal that *is used* in some parts in the grammar reachable from the top symbol. Having such nonterminals (called "bottom nonterminals") in a grammar is not a healthy practice and is in general considered a sign of bad quality since it signals incompleteness [26,41,47]. However, if we assume for the sake of simplicity that the default semantics for an undefined nonterminal is immediate failure (or parsing, generation, recognition or whatever the goal we need the grammar for), we may view **define**(p) as a language-extending (not a language-revising) operator. Thus, if we do somehow obtain a well-formed SPPF for such a grammar, it means it was constructed while *avoiding* the bottom nonterminal in question — hence, introducing it is no different than adding any other unreachable part we have seen so far and as such has no effect on the SPPFs.



```
S  ::= a+ b+ c+ & AB c+ & a+ BC;
AB ::= a AB? b;
BC ::= b BC? c;
```

```
S  ::= AP b+ c+ & AB c+ & AP BC;
AB ::= a AB? b;
BC ::= b BC? c;
AP ::= a+;
```

```
S   ::= ABP c+ & AB c+ & a+ BC;
AB  ::= a AB? b;
BC  ::= b BC? c;
ABP ::= a+ b+;
```

Fig. 4. SPPF transformations coupled with extraction of a new nonterminal definition: **(a)** the original grammar and an SPPF of the term "`abc`"; **(b)** after applying **extract**(`AP::=a+;`); **(c)** after applying **extract**(`ABP::=a+ b+;`). The case of extracting one symbol is easier because an SPPF already has nodes for such derivations and it only needs to be chained; when more than one symbol is present in the right hand side of a production rule being extracted, then a new node is introduced for all matched patterns of use.

3.2 SPPFs preserved, if possible

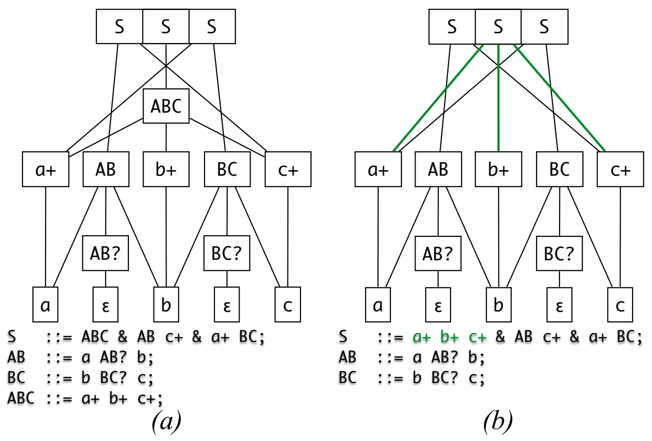
There are several cases when we do not know in advance whether the cotransformation of SPPFs will be possible: when it is, it is trivial.

3.2.1 Language-reducing operators

The operators **removeV**(p) and **removeH**(p) are the counterparts of **addV** and **addH** operators we have considered above, which remove alternatives instead of adding them. The effect of such a transformation on a given SPPF is easy to determine: if the alternative which is being removed, is exercised anywhere in the graph, the (co)transformation fails; if it is not, then no update of the forest is required.

Note that since all branches of the conjunctive clause are present in a given SPPF, their removal requires a (possibly failing) refactoring: hence, **removeC**(p) is considered later in §3.3.2.

The **undefine**(n) operator takes a valid nonterminal (defined and used within the grammar) and turns it into a bottom nonterminal (used yet not defined). It is a language reducing operator since its effect is a strict decrease in the number of possible correct programs: any parse graph containing a note related to the nonterminal n, becomes invalid. Hence, the coupled transformation for it checks whether such a node is indeed found in the given SPPF: if yes, the transformation fails; if not, it immediately succeeds without updating the SPPF.



```
S   ::= ABC & AB c+ & a+ BC;
AB  ::= a AB? b;
BC  ::= b BC? c;
ABC ::= a+ b+ c+;
```

```
S   ::= a+ b+ c+ & AB c+ & a+ BC;
AB  ::= a AB? b;
BC  ::= b BC? c;
```

Fig. 5. SPPF transformations coupled with inlining a nonterminal definition: **(a)** the original grammar and an SPPF of the term "`abc`"; **(b)** after applying **inline**(`ABC`). The inlining is fairly straightforward: the node in question is removed, and any previously incoming edge is replaced with the list of previously outgoing edges.

3.3 SPPFs refactored

In the next subsections we consider cases of less trivial coupled transformations, when language instances have to change to preserve conformance.

3.3.1 Language-preserving operators

Many transformation operators that preserve the language associated with a grammar, still have some impact on the parse graphs. When the impact is easy to calculate in advance and thus encode the coupled transformations as SPPF refactorings that are parametrised in the same way the grammar transformations are, we can run commands like **extract**(p) on both grammars and SPPFs.

Consider Figure 4(a). It shows a simple grammar of a non-context-free language $\{a^n b^n c^n \mid n > 0\}$ with three conjunctive views: the first one (`a+ b+ c+`) being the most intuitive and hence the most suitable for expressing patterns to be matched on programs; the remaining two being used to parse the language (which is well-known to be context-sensitive, so we *need* the power of two conjuncts to recognise it precisely). In a sense, the last two conjuncts represent a recogniser and the first one specifies a parser [40,42]. When a transformation command **extract**(`AP::=a+;`) is executed, the effect on the grammar is apparent: a new nonterminal is introduced and two occurrences of its right hand side are replaced with it. The effect on an SPPF is also quite easy to calculate: the node with **a**+ is replaced with a chain of two nodes (`AP` and `a+`); the incoming edges of the old node are connected as the incoming edges to the first one in the chain; the outgoing edges of the old node become the outgoing ones of the last in the chain (shown on Figure 4(b), changes in bold green). A slightly more complicated case is shown on Figure 4(c), where a new vertex needs to be created when we **extract**(`ABP::=a+ b+;`) because a symbol sequence `a+b+` did not correspond to any vertex in the old graph. For all vertices that had outgoing edges to both `a+` and `b+`, they got replaced by one edge to the new node. Figure 5 shows the opposite scenario of inlining a nonterminal in a grammar, coupled with "inlining" corresponding vertices in a graph by drawing edges through it.

Many other operators of this category from Figure 3 work similarly: **chain** replaces a node with a chain of two nodes; **fold** does the same folding we have seen above with **extract**, but without introducing a new nonterminal and possibly in a limited scope; **rassoc** and **lassoc** replace an iterative production rule with a recursive right/left associative one and thus stretches a node with multiple children into an unbalanced binary subtree; **concatT** and **splitT** merge or unmerge leaves, etc.

3.3.2 Language-extending operators

Above we have considered grammar transformation operators that add disjunctive clauses to the grammar, obviously extending the associated language. In the case of extended context-free grammars (regular right hand side grammars) that allow metasyntactic sugar like optionals ($x?$ effectively meaning $x|\varepsilon$) and regular closures (x^+ for transitive and x^* for reflexive transitive), the **widen**(e, e') operator is used to transform $x?$ to x^* or x to x^+, together with the **appear**(p) operator that transforms ε to $x?$ (effectively injecting an optional symbol). The coupled graph transformations for these cases usually boil down to inserting new vertices in the right places in order to keep the structural commitments up to date with the changed grammar.

An even less trivial case of language extension is called "upgrading" and involves replacing a nonterminal by an expression that can be reduced to it. For instance, in `A ::=` $\underline{\texttt{B}}$ `C; D ::= B|E;` we can upgrade `B` in `A` (underlined) to `D`. Such a transformation increases the string language associated with a grammar, as well as rearranges the relations between nonterminals. The coupled transformation for SPPF is still simple and inserts an extra vertex for `D` between `A` and `B` (`E` is still not present in the SPPF).

The **removeC**(p) operator that eliminates a conjunct, formally also increases the underlying language since any extra conjunct is possibly an extra condition to be met, and dropping it makes the combination weaker. Technically the coupled SPPF transform that removes a conjunct is a disambiguation filter, but it is not useful to count it as such since the ambiguity being removed is explicit (recall §2.4).

3.3.3 Language-reducing operators

The **disappear**(p) operator is used to transform $x?$ or x^* to ε. The coupled transformation on SPPFs for it exists, but is completely different from the ones being considered so far: it is inherently irreversible since if the SPPF in question actually contains the x? with x as a child node, then that x is removed and lost. This is contrasting to folding/unfolding vertices and rearranging the edges around them.

3.3.4 Language-revising operators

The operators **abstractize**(p) and **concretize**(p) eliminate and introduce terminals from production rules (a common practice when mapping abstract syntax to concrete syntax, hence the names). Since the terminals are present explicitly in the arguments, we can easily implement our coupled SPPF transformations by inserting leaves and connecting them to the appropriate places to the graph, or removing them. These transformations can have a big effect on the SPPF and are therefore more similar to the coupled transform from the previous paragraph. The **project** operator is a stronger version of **abstractize** or **disappear** that works on any symbol, but the transformation coupled with it, is the same: locate all the parts being removed from the grammar, remove them from the graph.

The rest of language revising operators are coupled with less invasive rearrangements of the parse graph: reordering edges (**permute**), updating the contents of the leaves (**renameT**) and splitting one nonterminal into several (**splitN**).

3.4 SPPFs refactored, when possible

Cotransformations from the previous section were necessary but could never fail: they were applicable to all possible graphs. Let us now move on to cotransformations that could seem successful on the grammar level but fail on the instance level (causing the combination to fail).

3.4.1 Language-reducing operators

The **narrow** operator (the reverse of **widen** discussed above) and the **downgrade** operator (the reverse of **upgrade**) become simple parse graph rearrangements, if the constructs in the SPPF happen to correspond to the new grammar, and fail otherwise. For instance, if a "wider" option is found in the SPPF, we have no automated way to update it.

The **addC** operator, on the other hand, shows us yet another class of coupled transforms: namely, the one requiring reparsing. Indeed, if the first branch of the conjunctive clause of `S` from Figure 4 were to be introduced as a transformation step, we would need to reconnect the left subnode of `S` to the appropriate children, which formally corresponds to parsing. In the current prototype implementation we reuse the existing parser — to the best of our knowledge, other frameworks like TXL [12] do the same — instead of exploring possibly more efficient alternatives.

3.4.2 Language-revising operators

The most brutal among language revising operators: **redefine** that replaces an entire nonterminal definition with a different one; **replace** doing the same for arbitrary subexpressions; **reroot** that changes the starting symbol of the grammar, — all require reparsing as a part of their coupled transformation steps.

3.5 Cotransformations destined to fail

Interestingly, there is one particular operator that is always doomed: **inject**(p) that works like **appear** but can insert any symbol anywhere in the grammar. In order to construct a coupled SPPF transformation for **inject**, we need to know how to connect the new node to its children, but this information is ultimately lacking from the operator parameters. The only cases where it could have worked, are already covered by other operators (e.g., injecting terminals is **concretize**, injecting possibly empty symbols is **appear**).

4 Related and future work

As said before, we are not the only ones trying to use computation models based on graphs instead of trees in software language engineering. It remains to be seen whether systematically using abstract syntax graphs [36] and general purpose graph transformation frameworks would be much different. In that case grammars can also be represented as graphs similar to Wirth's syntactic charts [32].

Our approach to couple instance transformations to grammar transformations and not vice versa has its counterparts in other technological spaces such as modelware [17] or XML [25] or databases [18], obviously with transformations of metamodels or schemata as the starting point. Coupled transformations in general have been re-explained to some extent in this paper, but there is a much more detailed introduction [27].

Grammar mutations [46,49] are systematic generalisations of grammar transformations used for this paper. There does not seem to be any fundamental problem in combining that generalisation with our couplings, but the implementation of coupled mutations remains future work.

The classification of coupled SPPF transformations from Figure 3 corresponds to the two kinds of negotiated evolution: "adaptation through tolerance" when SPPFs are preserved and "through adjustment" when they are refactored [48].

There is a lot of work on disambiguation, parse forest pruning and shaving, and it remains to be seen whether our approach can usefully complement similarly-minded techniques from that area such as van den Brand et al [6]'s implementation of disambiguation filters with term rewriting.

SPPF transformations could possibly be represented formally as classical graph replacement systems that rewrite nodes [15] or (hyper)edges [13]. One of the main objectives of presenting this paper at the workshop is to estimate potential usefulness of this approach.

5 Conclusion

In this paper, we have considered coupled transformations of grammars together with shared packed parse forests defined by these grammars. An implementation of a transformation operator suite was proposed. Each grammar change was coupled to one of the following: (1) no change in the parse graphs; (2) rearranging the graphs; (3) introducing new elements to graphs based on operator arguments; (4) reparsing; (5) imminent failure. This classification is complementary to the previously existing ones based on preserving, increasing, reducing or revising the semantics chosen for the grammar.

The examples given in the paper mostly refer to concrete grammars in the context of parsing, but the research was done with software language engineering principles, which means that the contribution is applicable to coupled evolution of grammars as well as ontologies, API, DSLs, XML schemata, libraries, etc. We have used Boolean grammars as the underlying formalism due to their power to represent non-context-free languages, ambiguous generalised parses and parse views in a uniform way. This is the first project involving coupled transformations of Boolean grammars.

The computation model proposed in this paper, can be used for formalisations and proofs of certain properties of transformation chains; for grammar-based convergence; for manipulating parse views and in general for tasks involving synchronous consistent changes to Boolean grammars and shared packed parse forests. This is an area of rapidly growing interest in the software language engineering community, and its limits, as well as the extent of its usefulness, remains to be examined.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1985)
2. Bagge, A.H., Lämmel, R.: Walk Your Tree Any Way You Want. In: ICMT. LNCS, vol. 7909, pp. 33–49. Springer (June 2013)
3. Bagge, A.H., Zaytsev, V.: Open and Original Problems in Software Language Engineering 2015 Workshop Report. SIGSOFT Software Engineering Notes 40(3), 32–37 (May 2015)
4. Basten, H.J., Vinju, J.J.: Parse Forest Diagnostics with Dr. Ambiguity. In: SLE'11. LNCS, vol. 6940, pp. 283–302 (2011)
5. van den Brand, M.G.J., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient Annotated Terms. Software: Practice & Experience 30(3), 259–291 (2000)
6. van den Brand, M.G.J., Klusener, A.S., Moonen, L., Vinju, J.J.: Generalized Parsing and Term Rewriting: Semantics Driven Disambiguation. ENTCS 82(3), 1–17 (2003)
7. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: CC. pp. 143–158 (2002)
8. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling Language Definitions: The ASF+SDF Compiler. ACM ToPLaS 24(4), 334–368 (2002)

9. van den Brand, M.G.J., Klint, P., Vinju, J.J.: Term Rewriting with Traversal Functions. ACM ToSEM 12(2), 152–190 (Apr 2003)
10. Cantor, D.G.: On the Ambiguity Problem of Backus Systems. Journal of the ACM 9(4), 477–479 (1962)
11. Chomsky, N.: On Certain Formal Properties of Grammars. Information and Control 2(2), 137–167 (1959)
12. Dean, T.R., Cordy, J.R., Malton, A.J., Schneider, K.A.: Grammar Programming in TXL. In: SCAM. IEEE (2002)
13. Drewes, F., Kreowski, H.J., Habel, A.: Handbook of Graph Grammars and Computing by Graph Transformation. chap. Hyperedge Replacement Graph Grammars, pp. 95–162. World Scientific Publishing Co., Inc. (1997)
14. Earley, J.: An Efficient Context-Free Parsing Algorithm. Ph.D. thesis, Carnegie-Mellon University (Aug 1968)
15. Engelfriet, J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. chap. Node Replacement Graph Grammars, pp. 1–94. World Scientific Publishing Co., Inc. (1997)
16. Ford, B.: Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In: POPL (Jan 2004)
17. Gîrba, T., Favre, J., Ducasse, S.: Using Meta-Model Transformation to Model Software Evolution. ENTCS 137(3), 57–64 (2005)
18. Henrard, J., Hick, J., Thiran, P., Hainaut, J.: Strategies for Data Reengineering. In: WCRE. pp. 211–220. IEEE (2002)
19. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like Transformation Language. In: OOPSLA. pp. 719–720. ACM (2006)
20. Kats, L.C.L., Visser, E.: The Spoofax Language Workbench. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) SPLASH/OOPSLA. pp. 237–238. ACM (2010)
21. Kay, M.: XSL Transformations (XSLT) Version 2.0. W3C Recommendation (23 January 2007), http://www.w3.org/TR/2007/REC-xslt20-20070123
22. Klint, P., Storm, T.v.d., Vinju, J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of SCAM. pp. 168–177. IEEE Computer Society (2009)
23. Klint, P., Visser, E.: Using Filters for the Disambiguation of Context-Free Grammars. In: Pighizzini, G., Pietro, P.S. (eds.) Proceedings of the ASMICS Workshop on Parsing Theory. pp. 1–20. Universitá di Milano (1994)
24. Kort, J., Lämmel, R., Verhoef, C.: The Grammar Deployment Kit. In: LDTA. ENTCS, vol. 65. Elsevier (2002), 7 pages
25. Lämmel, R., Lohmann, W.: Format Evolution. In: RETIS. vol. 155, pp. 113–134. OCG (2001)
26. Lämmel, R., Verhoef, C.: Semi-automatic Grammar Recovery. Software—Practice & Experience 31(15), 1395–1438 (Dec 2001)
27. Lämmel, R.: Transformations Everywhere. Science of Computer Programming (SCP) 52, 1–8 (2004)
28. Lämmel, R.: The Amsterdam Toolkit for Language Archaeology. In: ATEM'04. ENTCS, Elsevier (2005)
29. Lämmel, R., Wachsmuth, G.: Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment. In: LDTA. ENTCS, vol. 44. Elsevier (2001)
30. Lämmel, R., Zaytsev, V.: An Introduction to Grammar Convergence. In: Integrated Formal Methods (iFM). LNCS, vol. 5423, pp. 246–260. Springer-Verlag (Feb 2009)
31. Lämmel, R., Zaytsev, V.: Recovering Grammar Relationships for the Java Language Specification. Software Quality Journal (SQJ) 19(2), 333–378 (Mar 2011)

32. Martynenko, B.: Towards the 80th Anniversary of N. Wirth: Wirth's Syntactic Charts in the SYNTAX-Technology. In: SoRuCom. pp. 199–206. IEEE (Oct 2014)
33. Okhotin, A.: Conjunctive Grammars. Journal of Automata, Languages and Combinatorics 6(4), 519–535 (2001)
34. Okhotin, A.: Boolean Grammars. Information and Computation 194(1), 19–48 (2004)
35. Okhotin, A.: Conjunctive and Boolean Grammars: The True General Case of the Context-Free Grammars. Computer Science Review 9, 27–59 (2013)
36. Oliveira, B.C.d.S., Löh, A.: Abstract Syntax Graphs for Domain Specific Languages. In: PEPM. pp. 87–96. ACM (2013)
37. Parr, T., Fischer, K.: LL(*): the Foundation of the ANTLR Parser Generator. In: PLDI. pp. 425–436. ACM (2011)
38. Scott, E., Johnstone, A.: Generalized Bottom Up Parsers with Reduced Stack Activity. Computer Journal 48(5), 565–587 (2005)
39. Scott, E., Johnstone, A.: GLL Parsing. ENTCS 253(7), 177–189 (2010), LDTA'09
40. Scott, E., Johnstone, A.: Recognition is not Parsing — SPPF-style Parsing from Cubic Recognisers. Science of Computer Programming (SCP) 75(1-2), 55–70 (2010)
41. Sellink, M.P.A., Verhoef, C.: Development, Assessment, and Reengineering of Language Descriptions. In: CSMR. pp. 151–160. IEEE (Mar 2000)
42. Stevenson, A., Cordy, J.R.: Parse Views with Boolean Grammars. Science of Computer Programming (SCP) 97(1), 59–63 (2015), Special Issue on New Ideas and Emerging Results in Understanding Software
43. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1985)
44. Visser, E.: Scannerless Generalized-LR Parsing. Tech. Rep. P9707, University of Amsterdam (Jul 1997)
45. Zaytsev, V.: Recovery, Convergence and Documentation of Languages. Ph.D. thesis, Vrije Universiteit, Amsterdam, The Netherlands (Oct 2010)
46. Zaytsev, V.: Language Evolution, Metasyntactically. EC-EASST; BX 49 (2012)
47. Zaytsev, V.: Grammar Maturity Model. In: Pierantonio, A., Tamzalit, D., Schätz, B. (eds.) Ninth Workshop on Models and Evolution (ME 2014). pp. 42–51 (2014)
48. Zaytsev, V.: Negotiated Grammar Evolution. JOT; XM 13(3), 1:1–22 (Jul 2014)
49. Zaytsev, V.: Software Language Engineering by Intentional Rewriting. EC-EASST; SQM 65 (Mar 2014)
50. Zaytsev, V.: Grammar Zoo: A Corpus of Experimental Grammarware. Science of Computer Programming (SCP) 98, 28–51 (Feb 2015)
51. Zaytsev, V.: Guided Grammar Convergence. In: Poster proceedings of SLE'13 (2015), in print, available from CoRR: http://arxiv.org/abs/1503.08476
52. Zaytsev, V., Bagge, A.H.: Parsing in a Broad Sense. In: MoDELS. LNCS, vol. 8767, pp. 50–67. Springer (Oct 2014)