Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2013
Online at http://www.jot.fm.

Negotiated Grammar Evolution

Vadim Zaytsev[ab]

a. Software Analysis & Transformation Team (SWAT), Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands

b. Universiteit van Amsterdam, Amsterdam, The Netherlands

Abstract In this paper, we study controlled adaptability of metamodel transformations. We consider one of the most rigid metamodel evolution formalisms — automated grammar transformation with operator suites, where a transformation script is built in such a way that it is essentially meant to be applicable only to one designated input grammar fragment. We propose a new model of processing unidirectional programmable grammar transformation commands, that makes them more adaptable. In the proposed method, the making of a decision of letting the transformation command fail (and thus halt the subsequent transformation steps) is taken away from the transformation engine and can be delegated to the transformation script (by specifying variability limits explicitly), to the grammar engineer (by making the transformation process interactive), or to another separate component that systematically implements the desired level of adaptability. The paper investigates two kinds of different adaptability of transformation (through tolerance and through adjustment), explains how an existing grammar transformation system was reengineered to work with negotiations, and contains examples of possible usage of this negotiated grammar transformation process.

Keywords Tolerance, soft computing, grammar transformation, metamodel evolution, extreme modelling.

1 Motivation

Some metamodel transformation formalisms and instruments are more adaptable than others. One of the most rigid ones is grammar transformation with operator suites. Within this approach, a collection of well-defined transformation operators with well-understood semantics is provided, and those operators are supplied with arguments and the input grammar, so that the output grammar can be derived automatically. The transformation scripts are stored in the form of, in fact, partially evaluated operators, for which the arguments have already been provided, but the input grammar is not a part of such a transformation script. Thus, for example, if **renameN** is an operator that changes the name of one nonterminal symbol, then **renameN**(a, b) is a valid transformation command. However, suppose that the symbol a disappears from the original grammar (due to some evolution happening concurrently: renaming, unfolding, slicing, etc) — this makes the command of renaming it, inapplicable directly. This tight coupling between the shape of the input grammar fragment and the transformation step that is supposed to work on it, makes programmable grammar transformations rather fragile and prevents effective manipulation of such a system. (We say "fragment" to emphasize that the grammar transformation scripts are not necessarily applicable to only one specific grammar, but rather to any grammar that includes the expected fragment that satisfies a certain set of constraints. However, such "fragment" is not always sequential, it is in fact more of a slice — for instance, in the abovementioned example with renaming a to b, the applicability condition concerns presence of any production rules defining or referring to a).

The focus of this paper is specifically on adaptable grammar transformation approaches. Prior research on adaptability in grammarware mostly concerns adaptation of grammars towards a specific cause [DCMS02, HRK11, KLV02, Läm01, Läm05, LW01, LZ11, ZLvdS$^+$14]; while adaptation and co-adaptation of grammar transformation *scripts* remains a much less popular topic [Läm04, LR03], thus far from being convincingly covered. In particular, no previous work on programmable grammar manipulation with operator suites [Läm01, LV01, LW01, KLV02, Läm05, LZ11, Zay12b] considered grammar transformation adaptability explicitly.

In section 2 we revisit some background aspects, mostly related to grammar programming with the XBGF operator suite, and its model-related properties, and frame the contributions in a broader context by discussing research topics directly linked, relevant or conceptually close to the presented work. We note that the problem being solved is not at all specific to the XBGF which was used as the backend for our prototypes [ZLvdS$^+$14]. Coarse grammar transformations redefining nonterminals entirely or adding new production rules to them, which are commonly found in metaprogramming frameworks [DCMS02, KLV02] and parser combinator libraries [SD96, Swi01], are robust to a greater extent. However, there is usually no control over the kind of adaptation we will experience: tolerance or adjustment — section 3 follows with introducing and discussing both. Finer grammar transformations that can, for example, fold a symbol sequence as a definition of a new nonterminal (cf. the example at the end of section 5) or change one particular repetition from the "one or more" kind to the "zero or more", that are possible with frameworks like GRK [Läm05] or FST [LW01], are also prone to any kind of change in the source fragment of the input grammar, and easily are rendered inapplicable without a clearly traceable way to prevent it, so for them the addressed problem stands just as firm.

In section 4 we propose a method for making grammar transformation scripts more adaptable. In short, the method entails clear separation of applicability assertions from the actual transformation actions, and reformulating the former in a way that allows it to send suggestions back to the user instead of simply refusing to work. Section 4.3 provide details on the prototype implementation of the proposed method, which is publicly available for inspection and replication in its entirety through the open source repository of Software Language Processing Suite [ZLvdS$^+$14]. In section 5 we list some advantages and possible uses of the proposed model. The paper is concluded with section 6 briefly revisiting all main contributions.

Sections 3 and 4 extend the material previously presented at the Extreme Modelling Workshop at MoDELS 2012 [Zay12c].

2 Background and related work

BGF, or BNF-like Grammar Format, is being used within various grammar-related projects at least since 2009 [LZ09], as a common internal format for storing grammars. Most of these projects are available for inspection at the open source repository of Software Language Processing Suite, or SLPS [ZLvdS+14]. The expressiveness of BGF is comparable to that of an EBNF dialect: in fact, it was designed specifically to cover features typically found in various EBNF dialects: it contains terminals, nonterminals, repetitions, sequences, choices, etc [Zay12a]. Since implementation details are not the main focus of this paper, BGF fragments are intentionally left out.

XBGF, or Transformations of BGF, is of greater importance for us. It is an operator suite consisting of over 50 different operators, originally developed for grammar convergence [LZ09, Zay11] and received multiple applications since then [LZ11, ZL11, Zay12b, Zay14c]. Its complete description is available as a reference manual[1], an XML Schema definition[2], a Prolog interpreter[3] and a Rascal interpreter[4] [ZLvdS+14]. The behaviour of many operators is rather sophisticated, but for the purpose of reading this paper, the awareness of the following operators will suffice:

- **bypass**() — a trivial operator that takes no parameters and propagates the input grammar without changing it;
- **factor**(x, y) — an implementation of basic algebraic factorings based on associativity, distributivity and commutativity;
- **introduce**$(n ::= \mathit{rhs})$ — defines a previously unused nonterminal and adds it to the grammar;
- **eliminate**(n) — removes existing definitions of a nonterminal symbol which is unused in the rest of the grammar;
- **widen**(x, y) — generalises a metaproperty (e.g., turns a one-or-more repetition to a zero-or-more);
- **define**$(n ::= \mathit{rhs})$ — defines a used but previously undefined nonterminal;
- **renameN**(a, b) — globally changes the name of a nonterminal symbol;
- **fold**(n) — for a previously defined nonterminal, traverses the grammar and replaces any occurrences of the right hand side of its definition with a reference to it;
- **extract**$(n ::= \mathit{rhs})$ — same as **fold**, but first introduces a previously unknown definition to the grammar;
- **unfold**(n) — the opposite of **fold**: replaces all occurrences of a nonterminal by its definition;
- **disappear**(p, m) — prohibits the use of a previously optional element;
- **permute**(p, q) — changes the order in a sequence;

[1] `http://slps.github.io/xbgf`
[2] `http://github.com/grammarware/slps/blob/master/shared/xsd/xbgf.xsd`
[3] `http://github.com/grammarware/slps/blob/master/shared/prolog/xbgf1.pro`
[4] `http://github.com/grammarware/slps/blob/master/shared/rascal/src/transform/XBGF.rsc`

Operator [LZ09, LZ11, ...]	**Group [LZ11]**	**Class [CREP08]**	**Delta [GKP07]**
bypass()	preserving	updative	not breaking
factor(x, y)	preserving	updative	not breaking
introduce$(n ::= rhs)$	preserving	additive	not breaking
eliminate(n)	preserving	subtractive	not breaking
widen(x, y)	increasing	additive	not breaking
define$(n ::= rhs)$	revising	additive	not breaking
renameN(a, b)	preserving	updative	resolvable
fold(n)	preserving	additive	resolvable
extract$(n ::= rhs)$	preserving	additive	resolvable
unfold(n)	preserving	subtractive	resolvable
disappear(p, m)	decreasing	subtractive	resolvable
permute(p, q)	revising	updative	resolvable
concretize(p, m)	revising	additive	resolvable
abstractize(p, m)	revising	subtractive	resolvable
project(p, m)	revising	subtractive	resolvable
narrow(x, y)	decreasing	subtractive	unresolvable
inject(p, m)	revising	additive	unresolvable
replace(x, y)	revising	—	unresolvable

Table 1 – A representative excerpt from the XBGF operator suite. Among operands, a, b, n are nonterminals, p, q are production rules, x, y are grammatical expressions, m are markers. A “preserving” transformation preserves the language defined by the grammar, an “increasing” or “decreasing” one makes it larger or smaller, and a “revising” operator can have other outcomes [LZ11]. An “updative” transformation preserves the size of the modelling space, an “additive” or “subtractive” transformation makes it larger or smaller [CREP08]. A “not breaking” metamodel transformation does not invalidate language instances, a “resolvable” one implies an algorithm for automatic coevolution, and an “unresolvable” requires manual work or extra data [GKP07].

- **concretize**(p, m) — injects a bit of concrete syntax;
- **abstractize**(p, m) — projects a bit of concrete syntax;
- **project**(p, m) — projects an arbitrary element (removes it from a sequence);
- **narrow**(x, y) — the inverse of **widen**, restricts a metaproperty;
- **inject**(p, m) — adds a new element at an arbitrary place;
- **replace**(x, y) — globally replaces any expression by any other expression.

XBGF has many more operators, and finds its uses in grammar recovery (for correcting mistakes in the original grammar-containing artefacts or ones introduced by the automated extraction process), grammar specialisation (for automatically deriving a tool-specific grammar from a baseline grammar), grammar convergence (for validating claims about language or grammar equivalence), grammar beautification (for increasing readability of a grammar), technological space travel (transforming a grammar in a narrow sense to a database schema or a class diagram to an algebraic data type), etc.

Many authors — in particular, Herrmannsdörfer et al [HBJ09, HVW11], Cicchetti et al [CREP08], Wachsmuth [Wac07] — have previously considered, proposed or analysed metamodel transformation operators that are somewhat similar to grammar transformation operators. In this paper, we have limited ourselves to grammar evolution not only because grammars are considered somewhat simpler than arbitrary metamodels, but also because metamodel evolution scripts are traditionally written in a more adaptable way, so they suffer less from the problem we are solving here. On the other hand, they are too coarse to replace state-of-the-art grammar transformation operators: generally considered metamodel transformation operators rarely go beyond renaming, moving and folding/unfolding. However, in Table 1 we try to summarise the operators introduced above, from the metamodeling point of view.

In previous work on XBGF [LZ09, LZ11, ...] we have differentiated between the following groups of transformation operators:

- *language preserving*: the software language defined by the grammar, is not influenced by the change expressed by such an operator with any operands;
- *language increasing*: the software language defined by the grammar, becomes bigger if this operator is used to express the change;
- *language decreasing*: the software language defined by the grammar, becomes smaller if this operator is used to express the change;
- *language revising*: the change expressed by this operator, can increase, decrease or redefine the language in a different way, depending on the operands.

We also reuse the terms "updative", "additive" and "subtractive" from the work of Cicchetti et al. [CREP08]. These metamodel transformation classes are defined based on the modelling space, which is preserved, increased or decreased respectively. We extend Table 1 with operators that show that increasing the modelling space is not the same as increasing the language defined by a grammar. Grushko at al. [GKP07] propose to differentiate between metamodel deltas by looking at their impact on the conforming models: according to them, there are "not breaking" metamodel deltas that do not require any adjustment of the language instances, and two kinds of "breaking" deltas: a "resolvable" one that can be resolved by a generally applicable algorithm, and a "unresolvable" one where at least some cases exist that prevent the existence of such an algorithm. Typically unresolvable deltas involve adding an entity without a default value or removing an entity that was in use elsewhere. Herrmannsdörfer et al. [HBJ09] use a similar classification, geared more towards the reuse of such coevolution algorithms: in their eyes, all "not breaking" operators from Table 1 are "metamodel-only" changes, all "resolvable" are either "metamodel-specific" if operands are considered or "metamodel-independent" if operators are considered independently from them, and the rest are "model-specific" since they require grammar engineer's input during the migration process.

Kniesel and Koch [KK04] describe an idea of a "refactoring editor" that is capable of composing conditional transformations statically in OR-sequences in the style of Opdyke [Opd92] or AND-sequences in the style of Roberts [Rob99]. Such an extensive framework does not exist yet for grammar transformation: a transformation script for grammarware is a sequential list of transformation commands that only fails or succeeds at run time (cf. section 4). There is some evidence that static and safe composition of transformations is desirable and achievable for adjacent topics such as software evolution [Men99] and model evolution [HKA11].

Meyers and Vangheluwe [MV11] show four scenarios of coevolution: "model evolution" when a model is changed without any metamodel changes, "domain evolution" when a source metamodel changes, "image evolution" when the change concerns the target metamodel and "transformation evolution" when the transformation script changes, leading to necessary changes in the target metamodel. In fact, this definition generalises views found in coevolution research about data, code, grammars, schemata, databases and scripts [HTJC94, CH06, BCPV07, VV08]. On Figure 1 we can see how easy the case of image evolution is for grammarware, if the coupled model-level transformations are either not considered or left implicit — grammar transformation scripts can be just concatenated. The following sections of the paper will mostly concern domain evolution.

As we will see in detail in subsection 4.3, each operator is described as an applicability condition and a grammar rewriting algorithm (and possibly a postcondition). They are all parametrised: a sequence of operator calls with all operands specified, is referred to as a transformation script. Software language preserving properties of such a script are determined strictly by the operators used within it (which define dependencies among operands) and by the chosen semantics (string-based, tree-based, generative, analytic, etc), since "grammars in a broad sense" are pure structural definitions that can assume different semantics under various circumstances [KLV05]. In the presence of an input grammar, a transformation script can be applied to obtain the resulting grammar or fail while trying.

Another family of extreme modelling methods of inconsistency management of concurrent transformations, allows conflicts to not be resolved on the spot. Such inconsistencies can be represented as separate first-class entities [CRP07] and incorporated directly to the resulting model [KNHH10], which enables efficient handling of inconsistency detection and resolutions as graph transformation rules [MSD06]. These approaches can be used together with negotiated grammar transformation, as an alternative to it, or as implementation of the advanced negotiations impact propagation.

Besides a small Prolog example, in the next sections we will mostly see software language processing code in Rascal [KvdSV09]. This is the metaprogramming language of our choice and the one where XBGF grammar manipulation frameworks are implemented. Rascal's pattern driven dispatch would possibly have to be replaced with some other multiconditional (switch/case) operator, if another workbench is used; set comprehensions would have to be written out in a perhaps somewhat more verbose form; and lists or arrays would have to be used instead of sets. Beside these tiny implementation details, there are no implicit limitations that make our proposed method specific to Rascal only, and replications of the presented implementations are feasible in ANTLR [Par07], ASF+SDF Meta-Environment [BDH$^+$01], Bison [Lev09], GDK [KLV02], JavaCC [Cop07], Kiama [SKV11], Stratego [BKVV08], TXL [DCMS02], YACC [Joh75] and other grammarware frameworks and workbenches.

If we do not make any assumptions about the order of transformations (thus shifting the execution paradigm from the functional one to the declarative one), and drop the limitation on the number of times each "step" can be executed, then such a generalisation becomes a term rewriting [BN98, BKdV03] or a graph rewriting [Hof13] system. Investigating dead rules in such systems, that are never executed, and adapting them according to their applicability conditions and controlled levels of variability, is also a valid open problem for future research, partly covered by the *pending evolution* paradigm [Zay13].

$$\begin{tikzcd}
G_1 \arrow[r, "f"] \arrow[dr, "f_i"'] & G_2 \arrow[d, "e"'] \\
& G_4
\end{tikzcd}$$

G_1 is the original source grammar
G_2 is the intended target grammar
G_4 is the evolved target grammar
f is the original transformation
$f_i = e \circ f$

Figure 1 – Image evolution for grammar transformation.

$$\begin{tikzcd}
G_1 \arrow[r, "f"] \arrow[d, "e"'] & G_2 \arrow[d, equal] \\
G_3 \arrow[r, "f_t"] & G_4
\end{tikzcd}$$

G_1 is the original source grammar
G_2 is the intended target grammar
G_3 is the evolved source grammar
f is the transformation being adapted
$f_t = f \circ e^{-1}$ or $f_t = f$

Figure 2 – Adaptation through tolerance.

$$\begin{tikzcd}
G_1 \arrow[r, "f"] \arrow[d, "e"', ""{name=L}] & G_2 \arrow[d, "\tilde{e}", ""'{name=R}] \\
G_3 \arrow[r, "f_a"] & G_4
\arrow[from=L, to=R, "E_a"]
\end{tikzcd}$$

G_1 is the original source grammar
G_2 is the original target grammar
G_3 is the evolved source grammar
G_4 is the evolved target grammar
f is the transformation being adapted
$f_a \circ e = \tilde{e} \circ f$
$\tilde{e} = E_a(e)$

Figure 3 – Adaptation through adjustment.

3 Transformation adaptability

We can think of two kinds of adaptability that we may desire in metamodel domain transformation: through tolerance and through adjustment. Clearly, when two changes compete, there are two distinct possible conceptual scenarios: when the desired effect includes both and when it prefers only one of them.

Let us consider an example of a grammar G_1 and a grammar transformation f that produces $G_2 = f(G_1)$. Suppose that G_1 undergoes some changes by a transformation e, which produces $G_3 = e(G_1)$, and we still want to apply f to G_3, resulting in G_4. "Adaptation through tolerance", as seen on Figure 2, prefers one change over the other, and an adapted function f_t in fact works as f applied to the reverse of e (where actually reversing a transformation is a nontrivial task by itself). The term uses the word "tolerance", since changes contributed by e are tolerated but effectively disregarded. The other kind is "adaptation through adjustment", since extra adjustments need to be done and propagated further down the transformation chain. In that case, reported on Figure 3, an adapted function f_a, must be constructed in such a way that it results in G_4, which is a result of an applying $\tilde{e} = E_a(e)$, a hypothetic coevolution transformation with some correspondence to e.

3.1 Adaptation through tolerance

Figure 2 presents a megamodel for one kind of adaptation. Suppose we have a grammar G_1 and a transformation script describing a function, which yields $G_2 = f(G_1)$. If we assume that some evolution e (which is also technically a transformation) happens with the original grammar, yielding $G_3 = e(G_1)$, then we need to derive another transformation f_t, which takes the adjusted grammar G_3 and produces exactly the same result as the original transformation: $f_t(G_3) = G_4 = G_2 = f(G_1)$.

Adaptation through tolerance is not uncommon in situations when the evolutional part refers to some backend adjustments of the baseline grammar that we do not want to be affecting the transformation result in any way. In that case, conceptually, f_t can be thought of as undoing e and then applying f. However, undoing a grammar transformation and in general finding an inverse metamodel manipulation operation has some nontrivial aspects, as has been investigated and resolved before [Zay12b].

Example 1 *Suppose that e contains the use of an operator* **eliminate**(n) *from Table 1. Its reverse is the operator* **introduce**$(n ::= rhs)$*, which arguments contain not just the name of the nonterminal being eliminated/introduced, but also its definition. To compensate for the lacking information, the bidirectionalisation process must rely on the manual feed of additional information or consult the grammar being transformed, for the details [Zay14a].* ■

In conventional unidirectional programmable grammar transformation, most destructive operators inherently exhibit this property, which makes $f_t = f$. For example, a nonterminal definition is successfully **eliminate**d from the grammar just by matching its nonterminal name — hence, if the definition itself was changed by e, it will still be removed. From the operators on Table 1, **disappear**, **abstractize**, **project** and **narrow** can also sometimes have this property, depending on their operands.

Example 2 *Now suppose that f contains* **eliminate**(n)*, while e consists of calls to operators like* **factor**(x, y)*, which change the definition of this nonterminal n. The definition, whether changed by e or not, is absent from the resulting grammar G_4. Situations like this occur in practice when f is a transformation chain that produces a tool-specific grammar, while e changes the baseline grammar in a way that is relevant to some tools but not to all of them [DCMS02] — in this case, our f will abstract from the changed fragment just as it would have abstracted from the original one.* ■

3.2 Adaptation through adjustment

Figure 3 presents a different megamodel for adaptation of grammar transformation scripts. It is similar to the previous megamodel in many aspects, except for the output of the adapted transformation f_a is different from the original intended output grammar. In this case, we preserve the evolutional steps by assuming a hypothetic function $\tilde{e}$ which has some correspondence to the original evolution function e. The exact kind of correspondence (the form of the higher order function E_a) depends on the context and the desired grammar transformation composition semantics.

Adaptation through adjustment is common in many scenarios when the changes brought in by the original transformation and by metamodel evolution, are independent (for example, they may concern different nonterminal symbols). Apparently, in the case of complete independence their composition is commutative, so $E_a = \mathbf{id}$ and $\tilde{e} = e$. However, there are many cases when the transformations are essentially independent, but the scripts that represent them, still need to be adjusted: think of changing different parts of the same production rule — since the access scheme most probably entails including the whole production rule as an argument in both cases, the one that take place latest, requires adjustment.

Example 3 *Operators* $\mathbf{project}(p, m)$ *and* $\mathbf{inject}(p, m)$ *expect two arguments: a production rule and a name of the marker that marks the place for projection or injection within that production rule. If* f *and* e *are based on calls to those operators and concern different production rules, then we can guarantee that their impact will always be limited to only those production rules, and thus they will never interfere with each other:* $f \circ e = e \circ f$. ■

4 Negotiated evolution

In XBGF, any change to a grammar can be expressed as a chain of calls to grammar transformation operators from an extensive operator suite [LZ11]. Since the activity of creating such a chain closely resembles programming, it is commonly referred to as "grammar programming" [DCMS02], "metaprogramming" [KvdSV09] or "grammar engineering" [LV01, KLV05], even though all three terms also cover other activities that go beyond operator-based manipulation of grammars in a broad sense in the style of event sourcing [Fow05]. In order to distinguish the operators themselves from the calls to them, we will refer to the latter as "transformation commands". Thus, any grammar evolution can be expressed as a sequence of transformation commands, or a transformation sequence.

Example 4 *Let grammar* G_1 *be:*

$$e ::= e\ (\ \text{“+”}\ \mid\ \text{“−”}\ )\ e;$$

Let grammar G_2 *be:*

$$e ::= p \mid m; \quad p ::= e\ \text{“+”}\ e; \quad m ::= e\ \text{“−”}\ e;$$

They are different, but express the same expression language. The language equivalence problem is undecidable, but we can still converge G_1 *and* G_2 *by providing a transformation sequence that transforms one into the other [LZ09]. In this case it will be:*

$$\begin{aligned} &\mathbf{factor}(e\ (\ \text{“+”}\ \mid\ \text{“−”}\ )\ e, (e\ \text{“+”}\ e) \mid (e\ \text{“−”}\ e));\\ &\mathbf{extract}(p ::= e\ \text{“+”}\ e;);\\ &\mathbf{extract}(m ::= e\ \text{“−”}\ e;); \end{aligned}$$

In plain English, we factor the original definition of e *to push the choice (expressed by a BNF bar) outwards, and then introduce two new nonterminals while folding them (replacing their definitions by references to them). If the above three steps represent a grammar transformation sequence* f*, then* $f(G_1) = G_2$. ■

The method we propose as a way to address the controlled adaptability problem that was identified in the previous section, changes the model of this process. We first reintroduce the existing process in subsection 4.1, then propose a new model in subsection 4.2, refactor the implementation in subsection 4.3 and demonstrate the advantages of the resulting model with more concrete examples in section 5.

4.1 Grammar transformation

Previously, the transformational model could be described as follows:

1. The transformation command is supplied to the transformation engine that has access to the input grammar.
2. The applicability of the transformation command is assessed.
3. If the transformation command is deemed inapplicable to the input grammar, an error is reported and the transformation sequence halts since the grammar cannot be transformed further.
4. If the transformation command turns out to be vacuous (leads to zero changes) if applied to the input grammar, a different error is reported, and the transformation sequence still halts.
5. If the transformation command is applicable and non-vacuous, it is applied, and the transformation engine proceeds to (1.) with the next command.

4.2 Negotiations about grammars

The new model, that we refer to as "negotiated transformation", can be described like this:

1. The transformation command is supplied to the transformation engine that has access to the input grammar.
2. The applicability of the transformation command is assessed.
3. If the transformation command is applicable and non-vacuous, it is applied, and the transformation engine proceeds to (1.) with the next command.
4. If the transformation command turns out to be vacuous (leads to zero changes) when applied to the input grammar, and such a result is acceptable according to the semantics of the operator, a warning is reported, but the transformation process still continues to (1.) with the next command.
5. If the transformation command is deemed inapplicable to the input grammar or unacceptably vacuous, alternatives are explored and reported back in the form of a collection of possible operands that make the transformation applicable.
6. Based on the report received from the transformation engine, we can decide whether to report an error and halt the transformation process or proceed to (1.) with the *same* operator with *alternative* operands.

The last two items beg for more detailed explanation. By "reported back" we can mean *one* of the following:

- The alternative operand values are compared with the variability limits that are specified explicitly as a part of the transformation script (e.g., lists of allowed values, pattern matching, mini-grammar). In this case the role of the actual argument is somewhat diminished to the preferred one.

Example 5 *All three steps from Example 4 can possibly be expressed without fixing the name of the focus nonterminal as "e". We could also allow names like "exp", "expr", "expression" or "x", without adjusting the* intent *of the change. However, in this case we have a risk of a false positive, when a production rule is pattern matched with an argument, but should not.*

- The alternatives are literally reported back to the user who runs the transformation scripts, and the choice among them, with the always present option to fail, is up to this user.

Example 6 *The operator* **disappear**(p, m) *expects a production rule and a marker that marks an optional symbol, which is then deleted from the production rule as the effect of the transformation. A negotiable version of* **disappear** *can check for the marker to be present in the production rule and display all other markers to the user otherwise. If no markers are present at all, another negotiation strategy would be to let the user choose any optional element of the target production rule — in fact, any element that can lead to successful application of the operator, which is the main idea behind negotiations anyway.*

- The transformation sequence is halted as usual, but the suggestions are displayed to the user as recommendations.

Example 7 *Imagine a semi-repeatable grammar manipulation activity such as bulk recovery of grammars from error-prone sources like language documentation. Typically, one grammar is extracted, analysed and then corrected to account for all mistakes introduced in the source, by the nature of the source (e.g., text recognition), by the extraction process, etc. Then, when a similar grammar is encountered, we like to reuse these correcting transformation sequences, but there is no guarantee that exactly the same set of patches will be applicable — the practice shows that usually some of them can be shared*[5]*. The process of reuse would become more friendly and efficient, if suggestions about applicability and usefulness are displayed automatically about each step, instead of an iterative process.*

- A message about violating the contract is displayed, but the transformation sequence proceeds by choosing one option randomly or according to some minimality considerations.

Example 8 *Consider a situation when we need to* **introduce** *a new nonterminal to a grammar as a part of the transformation sequence, then use it within the following steps (e.g., for folding purposes) and then remove it again by performing* **eliminate***. This situation is actually encountered quite often in existing grammar transformation scripts [ZLvdS+ 14]. Within this scenario we only care that the name of this temporary nonterminal is unique so that it does not clash with any existing definitions, and it seems reasonable to just let the transformation engine adjust the name randomly instead of involving the grammar engineer in such a mundane task.*

[5]The evidence comes from recovering more than 500 grammars in a broad sense from various sources, they are available as the Grammar Zoo at `http://slps.github.io/zoo` [Zay14b]. Typically correction steps are at least partially shared within a group that can be "all Ada grammars", "all grammars from ISO standards", "all grammars created with ANTLR", etc.

- One alternative is chosen, but the other ones are stored in order to enable falling back to them if the transformation sequence gets stuck later on.

  Example 9 *Guided grammar convergence [Zay14c], a search-based method of converging two arbitrary grammars of the same intended software language automatically, could be seen as a fairly complicated variation of negotiated grammar transformations. For instance, one of the phases of this process is the nominal resolution — the process of establishing bidirectional mapping between nonterminal sets of two input grammars: technically we are constructing possible* **renameN** *chains and trying to explore the result; if the automated convergence algorithm gets stuck later on, it falls back to another possible mapping between nonterminals and tries to match their definitions again.*

Any other useful utilisation of the set of alternative operands by an additional system component can also be added to this list. One of the trivial ways to implement such a component is to let the transformation sequence fail anyway — this is equivalent to the traditional grammar transformation (with somewhat better error reporting, if the alternatives are displayed). On the other side of the spectrum, we can hypothetically think of encoding very large or infinite sets of allowed alternatives, or specifying the variability limits by constraints, which is in fact equivalent to grammar mutation [Zay12b]. Isolating this aspect to a separate component that systematically implements the desired level of adaptability, allows us to encode any desired behaviour between those two known approaches and beyond them.

4.3 Reengineering the implementation

In this section we will demonstrate that introducing the negotiable aspect to the existing transformation engine is a very simple and straightforward procedure. Consider the **renameN** operator that changes the name of a nonterminal symbol. The original implementation from 2009 [LZ09] uses Prolog and looks like this[6]:

```
renameN((N1,N2),G1,G2)
 :-
   allNs(G1,Ns),
   require(
      member(N1,Ns),
      'Source␣name␣~q␣for␣renaming␣must␣not␣be␣fresh.',
      [N1]),
   require(
      (\+ member(N2,Ns)),
      'Target␣name␣~q␣for␣renaming␣must␣be␣fresh.',
      [N2]),
   transform(try(xbgf1:renameN_rules(N1,N2)),G1,G2).
```

It has been straightforwardly reformulated in Rascal [KvdSV09] in 2012 [Zay12d, §3.2] to look as follows[7]:

[6]http://github.com/grammarware/slps/blob/master/shared/prolog/xbgf1.pro
[7]http://github.com/grammarware/slps/blob/master/shared/rascal/src/transform/XBGF.rsc

```
BGFGrammar transform(renameN(str x, str y), BGFGrammar g) {
       ns = allNs(g.prods);
       if (x notin ns)
              throw "Source␣name␣<x>␣for␣renaming␣must␣not␣be␣fresh.";
       if (y in ns)
              throw "Target␣name␣<y>␣for␣renaming␣must␣be␣fresh.";
       return
              performRenameN(x,y,g);
}
```

In these implementations, `renameN_rules` and `performRenameN` are helping predicates/functions of lesser interest that perform the actual traversal and rewriting. Conceptually **renameN**(x, y) follows this plan:

1. Source name x for renaming is expected to not be fresh (i.e., it *must* be present in the input grammar before renaming).
2. Target name y for renaming is expected to be fresh (i.e., it *must not* be present in the input grammar before renaming).
3. If x is listed among the root (starting) nonterminals, it is replaced there by y.
4. All production rules for nonterminals other than x, have their right hand sides altered such that every occurrence of x is replaced by y.
5. All production rules defining x, if they are present, undergo the same transformation, plus their left hand sides are changed to define y instead.

In order to enable negotiated computation of this operator without compromising the existing functionality, we keep the core transformation code of steps (3.) through (5.) isolated and unchanged and refactor the rest of the code to return structured errors instead of throwing exceptions[8]. The changes are made deliberately local and minimal:

```
XBGFResult transform(renameN(str x, str y), BGFGrammar g)
{
       ns = allNs(g.prods);
       if (x notin ns)
              return <problemStr("Source␣name␣must␣not␣be␣fresh",x),g>;
       if (y in ns)
              return <problemStr("Target␣name␣must␣be␣fresh",y),g>;
       return
              <ok(),performRenameN(x,y,g)>;
}
```

Where the data type of the return result is defined as follows[9]:

[8]http://github.com/grammarware/slps/blob/master/shared/rascal/src/transform/library/Nonterminals.rsc

[9]http://github.com/grammarware/slps/blob/master/shared/rascal/src/transform/Results.rsc

```
alias XBGFResult = tuple[XBGFOutcome r,BGFGrammar g];
data XBGFOutcome
       = ok()
       | problem(str msg)
       | problemXBGF(str msg, XBGFCommand xbgf)
       | problemStr(str msg, str x)
       | problemStrs(str msg, list[str] xs)
       | problemExpr(str msg, BGFExpression e)
       | problemProd(str msg, BGFProduction p)
       | problemProds(str msg, list[BGFProduction] ps)
       | ...
       ;
```

The code for grammar evolution operators in the GrammarLab library[10] is identical to the negotiated version we have seen above, modulo names for types and functions.

5 Advantages and uses

In the previous sections we have introduced a novel approach to express grammar evolution in terms of actions that we expect to be executed on a grammar at hand, and the limits up to which the results of the actual evolution can be negotiated. With this, we can specify the following forms of grammar programming:

Traditional grammar transformation. After each step, its outcome is examined: with a sign of any kind of problems, an exception is thrown and the execution is stopped. This corresponds exactly to the old policy of XBGF [LZ09] or any other grammar programming frameworks [DCMS02, KLV02, Läm01].

Interactive negotiated grammar transformation. Same as above, but the outcome of each step is pattern matched: the process continues if no problems are encountered, and advice is attempted to be generated otherwise. Only if no known problem patterns match, the process is halted, otherwise the list of possible suggestions is displayed to the user, who chooses the suitable one. This corresponds to the iterative process used in grammarware engineering [LV01, Zay11], but makes it more efficient by saving time.

Automatic negotiated grammar transformation. Same as above, but the choice is made automatically, based on a weighting algorithm or even at random, so the process continues as long as there is at least one successful alternative for each step. This option corresponds to heuristic-based automation methods, in particular for error recovery and robust processing [LZ11].

Automatic negotiated grammar transformation with backtracking. Same as above, plus all the non-explored choices made before are collected and used to roll back to if the process is halted several steps later. This is essentially a search-based method, and can be used as a lightweight substitution to more specific methods like guided grammar convergence [Zay14c].

[10]GrammarLab, `http://grammarware.github.io/lab`.

One more important detail for this reengineering initiative is propagating the *negotiations impact*. Any grammar transformation step usually exists in the context of the steps that follow and thus may rely on particular properties of the grammar being rewritten. Hence, purely negotiating the outcome of one transformation step independently of the subsequent ones, is insufficient, and negotiations need to be propagated until the last step is completed and the result is obtained. To abstract from the most troublesome details and to make the solution realistic, we follow [Zay12b] and commit to propagating naming adjustments only, which is an easily solvable problem that covers most of the basic needs of the rest of our approach.

If we continue considering the case of **renameN** explored before, we can write[11]:

```
set[XBGFCommand] negotiate(BGFGrammar g, XBGFCommand _, ok()) = {};
set[XBGFCommand] negotiate(BGFGrammar g,
       renameN(str x, str y),
       problemStr("Nonterminal␣must␣not␣be␣fresh", x))
 = {renameN(n,y) | str n <- adviseUsedNonterminal(x,allNs(g.prods))};
set[XBGFCommand] negotiate(BGFGrammar g,
       renameN(str x, str y),
       problemStr("Nonterminal␣must␣be␣fresh", y))
 = {renameN(x,n) | str n <- adviseFreshNonterminal(y,allNs(g.prods))};
default set[XBGFCommand] negotiate(BGFGrammar _,
       XBGFCommand _, XBGFOutcome _) = {};

set[str] adviseUsedNonterminal(str x, set[str] nts)
 = {z | z<-nts, distance(z,x)==min([distance(s,x) | s<-nts])};
```

In other words, if the source name x for renaming is fresh, we compute distances between x and all nonterminals that actually occur anywhere in the grammar, and recommend the one(s) with the lowest score. The computation of such distances can be as simple as the classic Levenshtein algorithm [Lev66] or involve something more advanced like longest common subsequence [GZ05]: ultimately, we want one that puts "expr" closer to "expression" than to "abcd", but even (modifications of) much more advanced techniques such as those relying on signature-based equivalence [Zay14c] or parser-based matching [FLZ12], could be applied here as well. The `adviseFreshNonterminal` function is somewhat more bulky and would not add much value to the paper — an interested reader is welcome to have a look at it, as well at other programmed negotiations, in the repository of SLPS [ZLvdS$^+$14]. In short, if the target name y for renaming is not fresh, it recommends three alternative fresh names for renaming: one of the form "*expr1*" (whatever the lowest number is that is not taken yet), one of the form "*expr_*" (obtained by concatenating underscores to the original target nonterminal name) and one made of random letters while preserving the original length and capitalisation (i.e., "AbcDef" can lead to "FooBar") — all three are guaranteed to be fresh.

As another example, consider the **abstractize**(p, m) operator from Table 1. It expects a production rule and a name of the marker that marks a bit of concrete syntax that will be projected by the operator application. Technically its semantics is a slightly limited subcase of **project**(p, m), but conceptually projection (removing one element from a sequence) does not necessarily preserve the language defined

[11] `http://github.com/grammarware/slps/blob/master/shared/rascal/src/transform/NegotiatedXBGF.rsc`

by the grammar, while this "abstractising" always preserves the abstract language (the term algebra) of the language defined by the grammar. Removing all concrete syntax elements is usually a part of transformation sequence from concrete syntax to abstract syntax, which is a well-known problem by itself with several known solutions [Wil97, JS10, vdSCL14, ZB14]. By using the negotiated transformation paradigm, we can write a transformation sequence that removes particular concrete syntax elements, and it will still be applicable to similarly structured abstract grammars (i.e., adaptation through tolerance) and to similarly structured grammars with the same terminal symbols used for their concrete syntax (i.e., adaptation through adjustment).

Example 10 *Consider the following grammar of function definitions:*

$$\textit{fdef} ::= \textit{fname}\ \text{"("}\ \textit{farg}^{+}\ \text{")"}\ \text{"="}\ \textit{fbody};$$

Also consider the following transformation sequence (where by "$\langle \mathrm{x} \rangle$:*y" we will denote a grammatical expression y marked by the name x):*

$$\mathbf{abstractize}(\textit{fname}\ \langle \mathrm{lp} \rangle\colon \text{"("}\ \textit{farg}^{+}\ \text{")"}\ \text{"="}\ \textit{fbody};, \langle \mathrm{lp} \rangle);$$
$$\mathbf{abstractize}(\textit{fname}\ \textit{farg}^{+}\ \langle \mathrm{rp} \rangle\colon \text{")"}\ \text{"="}\ \textit{fbody};, \langle \mathrm{rp} \rangle);$$
$$\mathbf{abstractize}(\textit{fname}\ \textit{farg}^{+}\ \langle \mathrm{eq} \rangle\colon \text{"="}\ \textit{fbody};, \langle \mathrm{eq} \rangle);$$

If we assume the intent of this transformation sequence to be in getting rid of the concrete syntax elements, then our model can be able to allow **bypass**() *as a negotiable alternative for all cases when the applicability precondition of the operator fails, but its first operand is equal to one of the production rules in the grammar, modulo terminal symbols. With this, adaptation through tolerance is achieved and the transformation sequence will serve as an assertion for the lack of terminals in this production rule, and will become applicable to grammars like this:*

$$\textit{fdef} ::= \textit{fname}\ \textit{farg}^{+}\ \textit{fbody};$$ ■

Example 11 *Consider the same grammar from Example 10 and the same transformation sequence. With the negotiated grammar transformation model, we can explore another direction for negotiations. Suppose that e is a language evolution that extends the grammar as follows:*

$$\textit{fdef} ::= \textit{fmodifier}?\ \textit{ftype}\ \textit{fname}\ \text{"("}\ \textit{farg}^{+}\ \text{")"}\ \text{"="}\ \textit{fbody};$$

We can still cheaply locate the terminal symbols marked in the original transformation operands by traversing the semi-matching definitions of the actual grammar, and successfully **abstractize** *the evolved grammar from them, thus achieving adaptation through adjustment.*

As the last and the most complicated example, consider the **extract** operator. It "extracts" a nonterminal symbol, which entails adding a new production rule of a fresh nonterminal to the grammar, and subsequently folding it — i.e., replacing all occurrences of its right hand side with the newly introduced nonterminal [ZLvdS+14, XBGF Manual]. Its implementation can be found in the same place we referenced above, but conceptually **extract**(n : *rhs*) works as follows:

1. Left hand side n is expected to be fresh (i.e., it must not be present in the input grammar before renaming).

2. The transformation is expected to be useful (i.e., *rhs* should occur at least once in the input grammar before adding the production rule).
3. All occurrences of *rhs* are replaced with n.
4. The production rule defining n as *rhs* is added to the grammar.

The steps (3.) and (4.) belong to the core transformation code and are folded into a separate function that can be called from both the regular and the negotiated grammar transformation functions, just like in the previous example. The step (1.) is also easily reused from the **renameN** example. However, the second step is not easily reused from the **abstractize** example, since **extract** does not make sense when it is vacuous: its base objective is to fold an existing symbol sequence into a new nonterminal, not to introduce a nonterminal unrelated to the rest of the grammar. Hence, we must implement a search-based strategy that attempts to identify fragments in the input grammar that could possibly be modifications of the right hand side that was provided as an argument.

In this paper, we only address propagation of changes in the names of nonterminals: negotiated renamings are remembered and used to preprocess the following steps that use the "old" names of nonterminals, selectors and production labels to access the "new" ones. Propagating other kinds of impact of negotiations though the subsequent transformation steps, is not a trivial task. A generalisation of that problem entails calculating the mutual impact of two transformation steps and the conditions that enable the change of execution order, which is a big open problem on its own: how to infer such f' from f and g' from g, that $f \circ g = g' \circ f'$?

6 Conclusion

Some metamodel transformation paradigms, like unidirectional programmable grammar transformation, are rather rigid. They are written to work with one input grammar, and are not easily adapted if the grammar changes. However, such adaptations are often desirable: in fact, we have presented megamodels of two scenarios when different kinds of adaptability can be useful (Figure 2 and Figure 3).

Our proposed solution entails isolation of the applicability assertions into a component separate from the rest of the transformation engine, and enhancing the simple accept-and-proceed vs. reject-and-halt scheme into one that proposes a list of valid alternative arguments and allows the other transformation participant (the oracle, the script, the end user running it, etc) to choose from it and negotiate the intended level of adaptability and robustness. This solution enables efficient manipulation of existing grammar transformation scripts and their controlled adaptability.

Fragments of a prototype were shown and discussed in the paper, and all of them available publicly in the GitHub repositories of the Software Language Processing Suite [ZLvdS+14] and of GrammarLab[12]. The places of most interest there are the core backend code at `shared/rascal/src/transform/NegotiatedXBGF.rsc`, a demonstration at `shared/rascal/src/demo/Negotiated.rsc` and the directory with textual outputs of sample runs for the conventional grammar transformation, both successful and failing, and the negotiated variation, at `topics/transformation/negotiated`.

In general, it is not outrageous to assume that the concept of negotiating the outcome of a transformation step instead of failing it, is applicable beyond the level

[12]GrammarLab, `http://grammarware.github.io/lab`.

of metamodels. However, the simplicity of the metametamodel (EBNF [Zay12a] in grammarware terms: terminals, nonterminals, symbol repetition, etc) is one of the key factors for the approach to be successful, since it is often feasible to come up with useful alternative suggestions.

References

[BCPV07] Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled Schema Transformation and Data Conversion for XML and SQL. In *Proceedings of the Ninth International Symposium on Practical Aspects of Declarative Languages (PADL 2007)*, volume 4354 of *LNCS*, pages 290–304. Springer, 2007. doi:10.1007/978-3-540-69611-7_19.

[BDH$^+$01] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of the 10th International Conference on Compiler Construction (CC 2001)*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001. doi:10.1007/3-540-45306-7_26.

[BKdV03] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. doi:10.1016/j.scico.2007.11.003.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[CH06] Anthony Cleve and Jean-Luc Hainaut. Co-transformations in Database Applications Evolution. In *Revised papers of the First International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, volume 4143 of *LNCS*, pages 409–421. Springer, 2006. doi:10.1007/11877028_17.

[Cop07] Tom Copeland. *Generating Parsers with JavaCC: An Easy to Use Guide for Programmers*. Centennial Books, 2007.

[CREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC 2008, pages 222–231. IEEE Computer Society, 2008. doi:10.1109/EDOC.2008.44.

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, October 2007. TOOLS EUROPE 2007 — Objects, Models, Components, Patterns. doi:10.5381/jot.2007.6.9.a9.

[DCMS02] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Grammar Programming in TXL. In *Proceedings of the Second IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2002)*, pages 93–102. IEEE, 2002. doi:10.1109/SCAM.2002.1134109.

[FLZ12] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. Comparison of Context-free Grammars Based on Parsing Generated Test Data. In Uwe Aßmann and Anthony Sloane, editors, *Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, volume 6940 of *LNCS*, pages 324–343. Springer, 2012. doi:10.1007/978-3-642-28830-2_18.

[Fow05] Martin Fowler. Event Sourcing, 2005. URL: http://martinfowler.com/eaaDev/EventSourcing.html.

[GKP07] Boris Gruschko, Dimitrios S. Kolovos, and Richard F. Paige. Towards Synchronizing Models with Evolving Metamodels. In *International Workshop on Model-Driven Software Evolution (MODSE)*, 2007.

[GZ05] Michael W. Godfrey and Lijie Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005. doi:10.1109/TSE.2005.28.

[HBJ09] Markus Herrmannsdörfer, Sebastian Benz, and Elmar Juergens. COPE — Automating Coupled Evolution of Metamodels and Models. In *Proceedings of the 23rd European*

Conference on Object-Oriented Programming (ECOOP 2009), pages 52–76. Springer, 2009. doi:10.1007/978-3-642-03013-0_4.

[HKA11] Florian Heidenreich, Jan Kopcsek, and Uwe Aßmann. Safe Composition of Transformations. *Journal of Object Technology*, 10:7:1–20, 2011. doi:10.5381/jot.2011.10.1.a7.

[Hof13] Berthold Hoffmann. Graph rewriting with contextual refinement. *Electronic Communications of the European Association of Software Science and Technology (ECEASST)*, 61, 2013. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/828.

[HRK11] Markus Herrmannsdörfer, Daniel Ratiu, and Maximilian Kögel. Metamodel Usage Analysis for Identifying Metamodel Improvements. In Brian A. Malloy, Steffen Staab, and Mark G. J. van den Brand, editors, *Post-proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *LNCS*, pages 62–81. Springer, January 2011. doi:10.1007/978-3-642-19440-5_5.

[HTJC94] Jean-Luc Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Schema Transformation Techniques for Database Reverse Engineering. In *Proceedings of the 12th International Conference on the Entity-Relationship Approach (ER 1993)*, volume 823 of *LNCS*, pages 364–375. Springer, 1994. doi:10.1007/BFb0024380.

[HVW11] Markus Herrmannsdörfer, Sander Vermolen, and Guido Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In Brian A. Malloy, Steffen Staab, and Mark G. J. van den Brand, editors, *Post-proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *LNCS*, pages 163–182. Springer, January 2011. doi:10.1007/978-3-642-19440-5_10.

[Joh75] S. C. Johnson. *YACC—Yet Another Compiler Compiler.* Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[JS10] Adrian Johnstone and Elizabeth Scott. Tear-Insert-Fold grammars. In *LDTA*, pages 6:1–6:8. ACM, 2010. doi:10.1145/1868281.1868287.

[KK04] Günter Kniesel and Helge Koch. Static Composition of Refactorings. *Science of Computer Programming*, 52(1–3):9–51, 2004. Special Issue on Program Transformation. doi:10.1016/j.scico.2004.03.002.

[KLV02] Jan Kort, Ralf Lämmel, and Chris Verhoef. The Grammar Deployment Kit. In M. G. J. van den Brand and R. Lämmel, editors, *Proceedings of the Second Workshop on Language Descriptions, Tools and Applications (LDTA 2002)*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002. doi:10.1016/S1571-0661(04)80430-4.

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an Engineering Discipline for Grammarware. *ACM TOSEM*, 14(3):331–380, 2005. doi:10.1145/1072997.1073000.

[KNHH10] Maximilian Kögel, Helmut Naughton, Jonas Helming, and Markus Herrmannsdörfer. Collaborative Model Merging. In *Companion of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, SPLASH 2010, pages 27–34. ACM, 2010. doi:10.1145/1869542.1869547.

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009)*, pages 168–177. IEEE, 2009. doi:10.1109/SCAM.2009.28.

[Läm01] Ralf Lämmel. Grammar Adaptation. In *Proceedings of the International Symposium of Formal Methods Europe*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001. doi:10.1007/3-540-45251-6_32.

[Läm04] Ralf Lämmel. Coupled Software Transformations. In *First International Workshop on Software Evolution Transformations (SET 2004)*, November 2004.

[Läm05] Ralf Lämmel. The Amsterdam Toolkit for Language Archaeology. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 137(3):43–55, 2005. Proceedings of the Second International Workshop on Metamodels, Schemas and Grammars for Reverse Engineering (ATEM 2004). doi:10.1016/j.entcs.2005.07.004.

[Lev66] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[Lev09] John Levine. *flex & bison.* O'Reilly Media, 2009.

[LR03] Wolfgang Lohmann and Günter Riedewald. Towards Automatical Migration of Transformation Rules after Grammar Extension. *Proceedings of the 15th European*

Conference on Software Maintenance and Reengineering (CSMR 2003), page 30, 2003. doi:10.1109/CSMR.2003.1192408.

[LV01] Ralf Lämmel and Chris Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001. doi:10.1002/spe.423.

[LW01] Ralf Lämmel and Guido Wachsmuth. Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *ENTCS*. Elsevier Science, 2001. doi:10.1016/S1571-0661(04)80918-6.

[LZ09] Ralf Lämmel and Vadim Zaytsev. An Introduction to Grammar Convergence. In Michael Leuschel and Heike Wehrheim, editors, *Proceedings of the Seventh International Conference on Integrated Formal Methods (iFM 2009)*, volume 5423 of *LNCS*, pages 246–260. Springer-Verlag, February 2009. doi:10.1007/978-3-642-00255-7_17.

[LZ11] Ralf Lämmel and Vadim Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ)*, 19(2):333–378, March 2011. doi:10.1007/s11219-010-9116-5.

[Men99] Tom Mens. *A Formal Foundation for Object-Oriented Software Evolution.* PhD thesis, Vrije Universiteit Brussel, 1999.

[MSD06] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2006. doi:10.1007/11880240_15.

[MV11] Bart Meyers and Hans Vangheluwe. A Framework for Evolution of Modelling Languages. *Science of Computer Programming*, 76(12):1223–1246, 2011. Special Issue on Software Evolution, Adaptability and Variability. doi:10.1016/j.scico.2011.01.002.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[Rob99] D. B. Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[SD96] S. Doaitse Swierstra and L. Duponcheel. Deterministic, Error-Correcting Combinator Parsers. *Advanced Functional Programming*, 1129:184–207, 1996. doi:10.1007/3-540-61628-4_7.

[SKV11] Anthony M. Sloane, Lennart C.L. Kats, and Eelco Visser. A Pure Embedding of Attribute Grammars. *Science of Computer Programming*, pages 1752–1769, 2011. Special section on Language Descriptions Tools and Applications (LDTA 2008–09). doi:10.1016/j.scico.2011.11.005.

[Swi01] S. Doaitse Swierstra. Combinator Parsers: From Toys to Tools. *Electronic Notes in Theoretical Computer Science*, 41(1):38–59, 2001. Proceedings of the 2000 ACM SIGPLAN Haskell Workshop. doi:10.1016/S1571-0661(05)80545-6.

[vdSCL14] Tijs van der Storm, William R. Cook, and Alex Loh. The Design and Implementation of Object Grammars. *SCP*, 2014. In Press, Corrected Proof. doi:10.1016/j.scico.2014.02.023.

[VV08] Sander Vermolen and Eelco Visser. Heterogeneous Coupled Evolution of Software Languages. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *LNCS*, pages 630–644. Springer, 2008. doi:10.1007/978-3-540-87875-9_44.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, editor, *21st European Conference on Object-Oriented Programming (ECOOP 2007)*, volume 4609 of *LNCS*, pages 600–624. Springer, July 2007. doi:10.1007/978-3-540-73589-2_28.

[Wil97] David S. Wile. Abstract Syntax from Concrete Syntax. In W. Richards Adrion, Alfonso Fuggetta, Richard N. Taylor, and Anthony I. Wasserman, editors, *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 472–480. ACM, 1997. doi:10.1145/253228.253388.

[Zay11] Vadim Zaytsev. Language Convergence Infrastructure. In João Miguel Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software*

Engineering (GTTSE 2009), volume 6491 of *LNCS*, pages 481–497. Springer, January 2011. doi:10.1007/978-3-642-18023-1_16.

[Zay12a] Vadim Zaytsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In Sascha Ossowski and Paola Lecca, editors, *Programming Languages Track, Volume II of the Proceedings of the 27th ACM Symposium on Applied Computing (SAC 2012)*, pages 1910–1915. ACM, March 2012. doi:10.1145/2245276.2232090.

[Zay12b] Vadim Zaytsev. Language Evolution, Metasyntactically. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST)*, 49, 2012. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/708.

[Zay12c] Vadim Zaytsev. Negotiated Grammar Transformation. In Juan De Lara, Davide Di Ruscio, and Alfonso Pierantonio, editors, *Proceedings of the Extreme Modeling Workshop (XM 2012)*. ACM Digital Library, November 2012. doi:10.1145/2467307.2467313.

[Zay12d] Vadim Zaytsev. The Grammar Hammer of 2012. *Computing Research Repository (CoRR)*, 4446:1–32, December 2012. URL: http://arxiv.org/abs/1212.4446.

[Zay13] Vadim Zaytsev. Pending Evolution of Grammars. In Juan De Lara, Davide Di Ruscio, and Alfonso Pierantonio, editors, *Post-proceedings of the Second Workshop on Extreme Modeling (XM 2013)*, volume 1089 of *CEUR Workshop Proceedings*, pages 28–35. CEUR-WS, October 2013. URL: http://ceur-ws.org/Vol-1089/4.pdf.

[Zay14a] Vadim Zaytsev. Case Studies in Bidirectionalisation. In *Pre-proceedings of the 15th International Symposium on Trends in Functional Programming (TFP 2014)*, May 2014. Research Paper Extended Abstract. URL: http://grammarware.net/writes/#Bidirectionalisation2014.

[Zay14b] Vadim Zaytsev. Grammar Zoo: A Repository of Experimental Grammarware. Submitted to the Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5). Pending second revision, 2014. URL: http://grammarware.net/writes/#Zoo2014.

[Zay14c] Vadim Zaytsev. Guided Grammar Convergence. In *Poster Proceedings of the Sixth International Conference on Software Language Engineering (SLE 2013)*. JOT, 2014. In print. URL: http://grammarware.net/writes/#Guided2014.

[ZB14] Vadim Zaytsev and Anya Helene Bagge. Parsing in a Broad Sense. Submitted to the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014). Pending reviews, March 2014. URL: http://grammarware.net/writes/#Parsing2014.

[ZL11] Vadim Zaytsev and Ralf Lämmel. A Unified Format for Language Documents. In Brian A. Malloy, Steffen Staab, and Mark G. J. van den Brand, editors, *Post-proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *LNCS*, pages 206–225. Springer, January 2011. doi:10.1007/978-3-642-19440-5_13.

[ZLvdS$^+$14] V. Zaytsev, R. Lämmel, T. van der Storm, Lukas Renggli, Ruwen Hahn, and Guido Wachsmuth. Software Language Processing Suite[13], 2008–2014. Contains, among other works: *XBGF Reference Manual: BGF Transformation Operator Suite* (V. Zaytsev, 2009), http://slps.github.io/xbgf. URL: http://slps.github.io.

[13]The authors are given according to the list at http://github.com/grammarware/slps/graphs/contributors.

About the author

Vadim Zaytsev also known in the social media as @grammarware, is a lecturer at the University of Amsterdam and an ex-employee of the Centrum Wiskunde & Informatica (CWI) in Amsterdam. He has acquired PhD in 2010 at the Vrije Universiteit Amsterdam in the field of software language engineering, in which his current main research interests lie. Prior to that, he received MSc cum laude degrees from Rostov State University in Russia (applied mathematics, model checking) and from Universiteit Twente in the Netherlands (telematics, grammar-based testing). Besides hardcore software language engineering with grammar(ware) technology, his interests and research activities tend to invade such topics as software quality assessment, source code analysis and transformation, modelling, metamodelling and megamodelling, programming paradigms, declarative and functional programming, dynamic aspects of software languages, maintenance and renovation of legacy systems and others. He is also actively practicing open science and open research, contributing to a range of open data and open source projects, co-organising and presenting at (mostly academic) events. Contact him at `vadim@grammarware.net`, or visit `http://grammarware.net`.