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

Negotiated Grammar Transformation

Vadim Zaytsev[a]

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

Abstract In this paper, we study controlled adaptability of metamodel transformations. We consider one of the most rigid metamodel transformation 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 different 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 lists 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,

Vadim Zaytsev. *Negotiated Grammar Transformation*. In Journal of Object Technology, vol. V, no. N, 2013, pages M:1–18. doi:10.5381/jot.201Y.VV.N.aN

slicing, etc) — this makes the command of renaming it, irreparably inapplicable. 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).

Prior research on adaptability in grammarware mostly concerns adaptation of grammars towards a specific cause [DCMS02, HRK11, KLV02, Läm01, Läm05, LW01, LZ11, ZLvdS$^+$13]; 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, Zay09, LZ11, Zay12b] considered grammar transformation adaptability explicitly.

In the next section we revisit two significant background topics: grammar programming with the XBGF operator suite (subsection 2.1) and megamodelling with the MegaL/yEd notation (subsection 2.2). Readers familiar with either of them are welcome to skip as they see fit. We still note that the problem being solved is not at all specific to the XBGF which was used as the backend for our prototypes [ZLvdS$^+$13]. 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 the way that allows to send suggestions back to the user instead of simply refusing to work. Section 5 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$^+$13]. The paper is concluded with section 6 discussing research topics directly linked, relevant or conceptually close to the presented work, and section 7 briefly revisiting all main contributions[1].

[1] Sections 3, 4 and 6 extend the material previously presented at the Extreme Modelling Workshop [Zay12c].

2 Background

2.1 Grammar programming with XBGF

BGF, or BNF-like Grammar Format, is being used within various SLPS-hosted projects at least since 2009 [LZ09], as a common internal format for storing grammars. Its expressiveness 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, Zay11a] and received multiple applications since [LZ11, ZL11, Zay11b, Zay12b, Zay12d]. Its complete description is available as a reference manual[2] [Zay09], an XML Schema definition[3], a Prolog interpreter[4] and a Rascal interpreter[5] [ZLvdS+13]. 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;
- **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 : rhs) — same as **fold**, but first introduces a previously unknown definition to the grammar;
- **renameN**(n, m) — globally changes the name of a nonterminal symbol;
- **vertical**(n) — for a given nonterminal symbol, converts its definition by a top level choice, to an equivalent one by multiple alternative production rules;
- **undefine**(n) — removes existing definitions of a nonterminal symbol.

As we will see in detail in section 5, each operator is described as an applicability condition and a grammar rewriting algorithm (and possibly a postcondition). The 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 (not by 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.

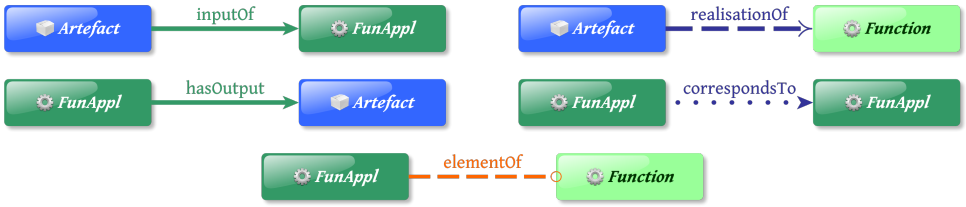

Figure 1 – Possible relationships between core entities in MegaL models in this paper: artefacts, functions and function applications. Italicised labels denote variables, normal font labels always refer to concrete entities.

2.2 Megamodelling with MegaL

For the figures on the pages of this paper, we will use MegaL/yEd, a domain-specific (mega)modelling language for specifying and discussing linguistic architecture proposed by Favre, Lämmel and Varanovich in [FLV12]. The current subsection briefly reintroduces a subset of it for the sake of self-sufficiency of this paper — any reader interested in the whole language is referred to the original publications on MegaL.

The entity types will be distinguished by the colour and an associated icon:

Artefact (blue, dark grey in greyscale, box icon) is a tangible software artefact — i.e., a file, a file fragment, a language definition, a language instance, a library;

Function (light green, light grey in greyscale, cogwheel) is a function in the (meta)model transformation sense; for the sake of brevity, partially evaluated functions are also depicted as functions;

Function application (dark green, dark grey in greyscale, cogwheel) is a concrete transformation, usually conforming to some function definition, but also having all arguments at its disposal.

Figure 1 lists all MegaL entity types (on top) and those relationship types that will be used within this paper:

- We say that *an artefact is an input of function application*, when this artefact (a file, a program, another tangible entity) is necessary for the function to run and serves to instantiate the function.
- Similarly, we say that a *function application has an artefact as an output*, when this artefact is expected to be generated when a valid input is provided.
- *An artefact is a realisation of a function or a function application*, when it represents a tangible serialisation of this function or a function application.
- Any entity *corresponds to* another entity, when there is some (possibly unspecified) correspondence relation between them.
- *Function application is an element of a function*, when the function application truly conforms to the function definition and is supplied with appropriate parameters.

MegaL models are *megamodels* [BJV04, FN04] — models of linguistic architecture that specifically address relationships between complex entities such as software lan-

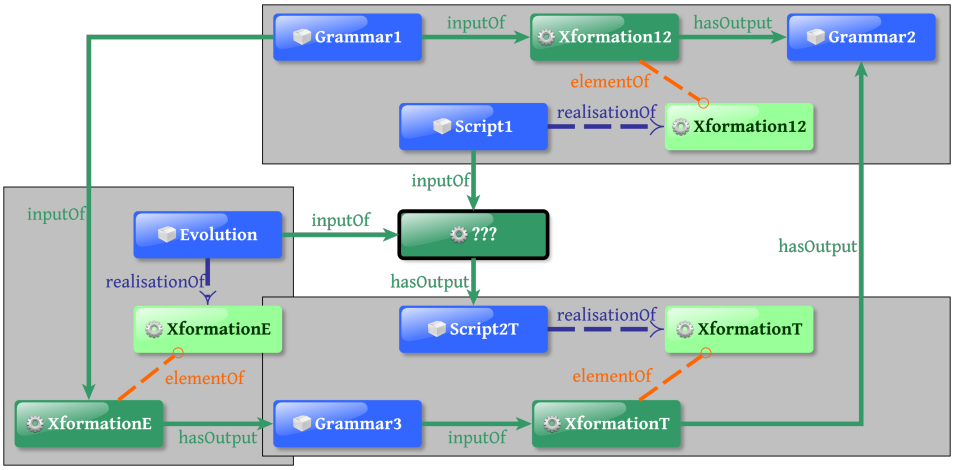

Figure 2 – Adaptation through tolerance.

guages and (meta)model transformations in order to comprehend software technologies and relate technological spaces [KBA02].

3 Transformation adaptability

We can think of two kinds of adaptability that we may desire in metamodel transformation: through tolerance and through adjustment. 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. Clearly, there are two distinct possible conceptual scenarios: with $G_4 = f_t(G_3) = G_2$, where $f_t = f \circ e^{-1}$; and with $G_4 = f_a(G_3) \neq G_2$, where $f_a \circ e = e' \circ f$ and $e' = E_a(e)$ is a hypothetic coevolution transformation with some correspondence to e. In other words, in the first scenario we expect the outcome to persistently stay the same even if the source grammar G_1 evolves (we call this "adaptation through tolerance", since changes contributed by e are tolerated but effectively disregarded). In the second scenario we expect the transformation result to combine the impact of both f and e, which we call "adaptation through adjustment", since extra adjustments need to be done and propagated further down the transformation chain.

3.1 Adaptation through tolerance

Figure 2 presents a megamodel for one kind of adaptation. We start with the group in the right top corner. It contains three artefacts: two grammars and a transformation script that describes a function, which, if applied to Grammar1 (G_1), will yield Grammar2 (G_2). If we assume that some evolution (which is also technically a transformation) happens with the original grammar (the group on the left), then

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

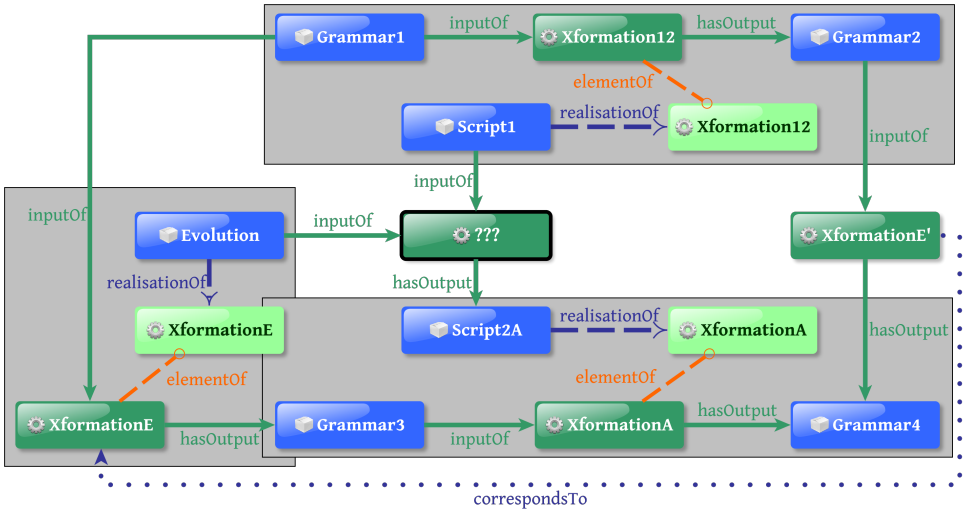

Figure 3 – Adaptation through adjustment.

we might need to derive another transformation script Script2T, which will take the adjusted grammar and produce exactly the same result as the original Script1 (the exact way to derive this script, is unknown, non-automated or irrelevant, which we depict as the "???" box on the megamodel).

Adaptation through tolerance is quite common 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, if XformationT (f_t) can be thought of as undoing e and then applying f.

In conventional unidirectional programmable grammar transformation, most destructive operators inherently exhibit this property (with Script1 being equal to Script2T, $f_t = f$): for example, when a nonterminal is **undefine**d, it means that all its production rules are disregarded and removed from the grammar — so, even if they are adjusted by XformationE, they are still removed, and the adjustments disappear from the output grammar.

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 transformation function application in the right bottom group. In this case, we preserve the evolutional steps by assuming a hypothetic function XformationE' (e') which has some correspondence to the original evolution function. 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 $e' = e$, but there are many "grey" 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.

4 Negotiated transformation

The method we propose as one of the ways to address the controlled adaptability problem that was identified in the previous section, changes the model of the process. The current model is 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.
4. If the transformation command turns out to be vacuous (lead 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.

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 (lead to zero changes) if 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 arguments that make the transformation applicable.
6. The side that supplied the transformation command, decides by itself whether to report an error and halt the transformation process or proceed to (1.) with the *same* command and one of the *alternative* sets of arguments.

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

- The alternatives are compared with the variability limits that are specified explicitly as a part of the transformation script. In this case the role of the actual argument is somewhat diminished to the preferred one.
- 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.
- A message about violating the contract is displayed, but the transformation sequence proceeds by choosing one option randomly or according to some minimality considerations.
- 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.
- The transformation sequence is halted as usual, but the suggestions are displayed to the user as recommendations.
- Any other useful utilisation by the set of alternatives by an additional component.

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.

5 Reengineering the implementation

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 was straightforwardly reformulated in Rascal [KSV11] 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 runRenameN(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
		transform::library::Core::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]:

```
XBGFResult runRenameN(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)
       | problemScope(str msg, XBGFScope w)
       | problemExpr(str msg, BGFExpression e)
       | problemProd(str msg, BGFProduction p)
       | problemProds(str msg, list[BGFProduction] ps)
       | ...
       ;
```

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 (or any all other grammar programming frameworks).

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.

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.

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.

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** that was explored before, we can have[10]:

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

```
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, levenshtein(z,x)==min([levenshtein(s,x) | s<-nts])};
```

In other words, if the source name x for renaming is fresh, we compute Levenshtein distances between x and all nonterminals that actually occur anywhere in the grammar, and recommend the one(s) with the lowest score. 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+13]. 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 **vertical** operator: it consumes production rules of one nonterminal symbol, that contain a top-level choice, and replaces them with an equivalent definition consisting of multiple production rules [Zay09]. (The latter style of engineering grammars is called "non-flat" [LW01] or "vertical" [LZ11], hence the operator name). Obviously, it fails to operate when the nonterminal is not present or when it is already vertically defined — i.e., if there is not a single production rule with a top-level choice. Besides the obvious, there are some more subtle constraints maintaining the uniqueness of the production labels (subexpressions are required to have a unique name within a production rule, while production labels must be universally unique — and a top selector can become a label as an effect of this operator).

We have already considered ways to negotiate naming of entities in the context of **renameN**. A new property for **vertical** is the possible vacuousness of the transformation: while verticalising a vertically defined nonterminal should raise an error in the traditional rigid grammar transformation model, a negotiated approach could recommend a special **bypass** transformation which implementation always trivially succeeds by returning `<ok(),g>`, where `g` is the input grammar. Hence, a negotiated version of the **vertical** operator disregards the assertion of non-vacuousness. Some other operators with preconditions being exact negations of the postconditions, follow the same pattern and become like assertions: whatever the form of the definitions of the nonterminal was, it will be vertically defined after the transformation step is completed, whether any actual changes need to happen or not. Thus, *adaptation*

through tolerance is achieved.

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 [Zay09]. 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 **vertical** 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. Propagation of this kind of adaptations is not a trivial task, and we leave it beyond the text of this paper.

6 Related work

Herrmannsdörfer et al [HBJ09, HVW11], Wachsmuth [Wac07], Cicchetti et al [CREP08] and many others have considered, proposed or analysed metamodel transformation operators that are strikingly similar to grammar transformation operators. In this paper, we have limited ourselves to grammar transformation 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.

In section 5, we have explained that alternative renaming candidates are proposed according to the minimal Levenshtein distances between x and any other nonterminal. In fact, the Levenshtein's edit distance [Lev66] is not the perfect metric in this scenario: ultimately, we would want one that puts "expr" closer to "expression" than to "abcd". To the best of the author's knowledge, such metric does not exist yet, since the need for it has apparently not previously risen[11]. The (adjusted) Levenshtein distance algorithm was only used for the sake of simplicity, but even (modifications of) much more advanced techniques such as those relying on nonterminal equivalence [Zay12d, §3.1] or parser-based matching [FLZ12], could be applied here as well.

[11] Vadim Zaytsev (grammarware). "Which string metric meaningfully and consistently puts 'expr' closer to 'expression' than to 'abcd'?" 10 June 2012, 12:20. Tweet. `http://twitter.com/grammarware/status/211764758968418304`.

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.

Propagating the impact of negotiations though the subsequent transformation steps was solved in this paper only for naming adjustments: 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. 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 $f \circ g = g' \circ f'$?

If we do not make any assumptions on 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 [HJG08] system. Investigating dead rules 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.

If the metaprogramming language of our choice was not Rascal [KSV11], then the pattern driven dispatch would possibly have to be replaced with some other multiconditional (switch/case) operator; comprehensions on page 11 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 hidden limitations that make our proposed method specific to Rascal only.

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 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.

7 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.

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 repository of the Software Language Processing Suite [ZLvdS+13]. 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`.

References

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.

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

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

[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 '08, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society. `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 Workshop on Source Code Analysis and Manipulation*, SCAM'02, pages 93–102, Washington, DC, USA, 2002. IEEE Computer Society.

[FLV12] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proceedings of MoDELS 2012*, LNCS. Springer, 2012. 17 pages.

[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, Heidelberg, 2012. `doi:10.1007/978-3-642-28830-2_18`.

[FN04] Jean-Marie Favre and Tam NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *Electronic Notes in Theoretical Computer Science, Proceedings of the SETra Workshop*, 127(3), 2004.

[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'09)*, Genoa, pages 52–76, Berlin, Heidelberg, 2009. Springer-Verlag.

[HJG08] Berthold Hoffmann, Edgar Jakumeit, and Rubino Geiß. Graph Rewrite Rules with Structural Recursion. In Mohamed Mosbah and Annegret Habel, editors, *Proceedings of the Second International Workshop on Graph Computation Models (GCM 2008)*, pages 5–16, 2008. URL: `http://www.informatik.uni-bremen.de/~hof/papers/08-GCM.pdf`.

[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'10)*, volume 6563 of *LNCS*, pages 62–81, Berlin, Heidelberg, January 2011. Springer-Verlag.

[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'10)*, volume 6563 of *LNCS*, pages 163–182, Berlin, Heidelberg, January 2011. Springer-Verlag.

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Akşit. Technological Spaces: an Initial Appraisal. In *Proceedings of CoopIS, DOA'2002, Industrial track*, 2002.

[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'02)*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

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

[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 '10, pages 27–34, New York, NY, USA, 2010. ACM. `doi:10.1145/1869542.1869547`.

[KSV11] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Metaprogramming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009)*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289, Berlin, Heidelberg, January 2011. Springer-Verlag. URL: `http://rascal-mpl.org`.

[Läm01] Ralf Lämmel. Grammar Adaptation. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for*

Increasing Software Productivity, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[Läm04] Ralf Lämmel. Coupled Software Transformations. In *First International Workshop on Software Evolution Transformations (SET'04)*, 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'04).

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

[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'03)*, 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. URL: http://www.cs.vu.nl/grammarware/ge/.

[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'01)*, volume 44 of *ENTCS*. Elsevier Science, 2001.

[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, Berlin, Heidelberg, February 2009. Springer-Verlag. 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.

[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 Berlin / Heidelberg, 2006.

[SD96] S. Swierstra and L. Duponcheel. Deterministic, Error-Correcting Combinator Parsers. *Advanced Functional Programming*, 1129:184–207, 1996.

[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.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, editor, *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 600–624. Springer, July 2007.

[Zay09] Vadim Zaytsev. *XBGF Reference Manual: BGF Transformation Operator Suite*. Universität Koblenz-Landau, 1.0 edition, August 2009. URL: `http://slps.github.com/xbgf`.

[Zay11a] 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, Berlin, Heidelberg, January 2011. Springer-Verlag. `doi:10.1007/978-3-642-18023-1_16`.

[Zay11b] Vadim Zaytsev. MediaWiki Grammar Recovery. *Computing Research Repository (CoRR)*, 4661:1–47, July 2011. URL: `http://arxiv.org/abs/1107.4661`.

[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, Riva del Garda, Trento, Italy, March 2012. ACM. `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.

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

[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, Berlin, Heidelberg, January 2011. Springer-Verlag. `doi:10.1007/978-3-642-19440-5_13`.

[ZLvdS+13] Vadim Zaytsev, Ralf Lämmel, Tijs van der Storm, Lukas Renggli, and Guido Wachsmuth. Software Language Processing Suite[12], 2008–2013. URL: `http://slps.github.com`.

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

About the author

Vadim Zaytsev also known in the social media as @grammarware, is an 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`.