Electronic Communications of the EASST
Volume 49 (2012)

Proceedings of the First International Workshop on Bidirectional Transformations (BX 2012)

Language Evolution, Metasyntactically

Vadim Zaytsev

17 pages

Guest Editors: Frank Hermann, Janis Voigtländer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Language Evolution, Metasyntactically

Vadim Zaytsev

vadim@grammarware.net, http://grammarware.net
SWAT, CWI, The Netherlands

Abstract: Currently existing syntactic definitions employ many different notations (usually dialects of EBNF) with slight deviations among them, which prevent efficient automated processing. When changes in such notation are required either due to maintenance activities such as correction or evolution, or because a grammar collection is written in a different notation than the one required by the grammarware toolkit, we speak of metalanguage evolution: i.e., a special language evolution scenario when the language itself does not necessarily evolve, but the notation in which it is written, does. Notational changes need to be propagated to different levels, such as to parsers that used to work with the old notation, to grammars of those notations that served as explanation material, and to the existing grammarbase.

The solution proposed in this paper, relies on composing a notation specification and expressing notation changes as transformations of that specification. These transformation steps are coupled to changes in the notation grammar (i.e., grammar for grammars) and to changes in other grammars written in the original notation. This paper explains the general setup of such an infrastructure, with links to the prototypical implementation of the solution.

Keywords: language evolution; bidirectional transformation; coupled transformation; syntactic notation; grammar convergence.

1 Introduction

The unnecessary diversity of notation for syntactic definitions stems from the current practice of almost every language documentation artefact employing its own notation, usually a dialect of EBNF [Wir77, Zay12a, ZL11]. When changes in such notation are required, we speak of **metalanguage evolution**: i.e., a special language evolution scenario when the language itself does not necessarily evolve, but the notation in which it is written, does. Scenarios when the need for such changes arise, include:

Notation correction/enforcement. Most of the grammars found in the language documentation, have never been formally validated and are known to contain many types of errors. One specific category of such errors is misused notation. For example, in Java Language Specification [GJSB05] a grouping metasymbol (i.e., a possibility to group symbols with parenthesis) is never specified in the notation description, yet still used on several occasions. Changing such grammar to fit into the intended notation is in fact a notation change from the actual notation to the intended one.

Notation evolution. Syntactic notations can be considered software languages themselves, and as their design and development commence, they become a target to change. For example, the BNF-like notation used by the Grammar Deployment Kit (a framework for grammar maintenance and manipulation), considerably evolved since the first publication [KLV02] to the current version [Kor03]. However, these changes are not immediately noticeable, and decorational changes (e.g., renamed nonterminals in the grammar for grammars) and conceptual changes (e.g., adding a notation for separator lists) are indistinguishable.

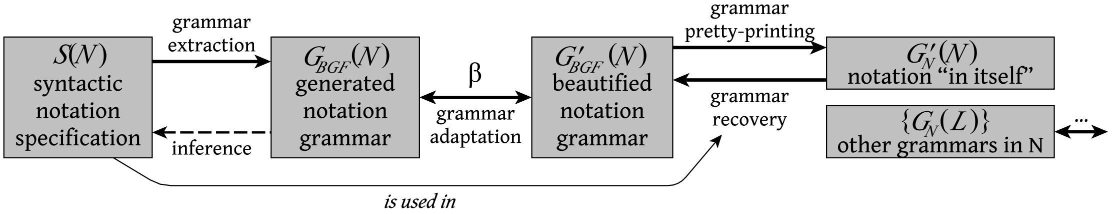
Mapping between notations. When a language engineer possesses a number of grammars (a grammarbase) in a particular notation, they may need to be mutated if there is an intention to use a particular grammarware framework (say, GDK [KLV02], TXL [Cor06], Rascal [KSV11], SLPS [ZLS+12]) that works with a different notation (which is perhaps just as expressive). Bidirectionality [CFH+09] plays an especially important role here because if the grammarware framework changed the grammar, such changes will need to be propagated back to the original notation.

Notational changes always need to be propagated to different levels: parsers that used to work with the original notation; grammars of those notations that served as explanation material; the existing grammarbase. The solution proposed in this paper, relies on composition of a notation specification and expressing notation changes as transformations of that specification. These transformation steps are coupled to changes in the notation grammar (i.e., grammar for grammars) and to changes in other grammars written in the original notation. Although the general theory of metamodel evolution and coupled metamodel/metametamodel transformation is not at all limited to the grammarware technical space, as we know from [Wac07, CCLP11], we confine ourselves to grammar specifics and only briefly discuss similar approaches from other fields.

The rest of the paper is organised as follows. §2 introduces the notation specification and other artefacts related to it. §3 considers a scenario with two notations involved in notation evolution. §4 presents a real notation evolution case study and explains the prototypical application of the proposed megamodel to it. §5 references and discusses issues related to ours and touches on possible future explorations. §6 concludes the paper by listing contributions and achievements.

2 Notation life cycle megamodel

Following Bézivin et al [BJV04], we present the general setup for notation life cycle in a “megamodel”. In our case, we will use boxes for entities and arrows for actions. Consider the following artefacts and relationships between them (will be explained from left to right):



If N is a notation for syntactic definition, we can compose a **notation specification** $S(N)$ (the leftmost box on the figure). Such a specification consists of a set of indications that have previously been proposed in [Zay12a]:

Confix constructs (bracketing start & end metasymbols):
grammar, comment, label, nonterminal, terminal, special, group, optionality, star repetition, plus repetition, star separator list, plus separator list

Infix metasymbols:
terminator, possible terminator, defining, multiple defining, definition separator, concatenation, inner choice, exception

Postfix metasymbols:
optionality, star repetition, plus repetition

Prefix metasymbols:
start one line comment

Other metasymbols:
line continuation, tabulation, empty sequence

Conventions:
whitespace reliability, indentation, definition direction, nonterminal if defined, nonterminal if contains, glue consecutive terminals, decomposition of symbols, uppercase nonterminals, lowercase nonterminals, camelcase nonterminals, mixed case nonterminals, uppercase terminals, lowercase terminals, camelcase terminals, mixed case terminals

Predefined sets:
ignored line indicators, masked terminals, nonterminals may contain, built-in nonterminals

Together, these are powerful enough to define any EBNF dialect. Its representation in our toolset is called EDD (stands for **E**BNF **D**ialect **D**efinition) and, being a list of metasymbol name-value tuples, is not technically interesting and is publicly available at the repository of Software Language Processing Suite (SLPS) [ZLS$^+$12] as `shared/xsd/edd.xsd` as a schema, with `shared/edd` directory containing specifications of several notations we have encountered.

Constructing a notation specification is technically equivalent (yet better maintainable, as we will argue later) to making a grammar for grammars (a parser specification that will allow to parse grammars written in N): e.g., $G_{Rascal}(N)$. The parser generated from it is useful for getting IDE support for various grammarware engineering activities such as (semi-)automatic grammar recovery [Zay12b], but is not an essential part of this paper's solution. However, it can serve as a source for grammar extraction, and provides us a **notation grammar** $G_{BGF}(N)$ for the given notation, where BGF is an internal representation for grammars[1]. Being derived within an "abstraction by extraction" paradigm [LZ09], it contains slightly less information than the more detailed parser specification, making bidirectionalisation of this step somewhat problematic. For instance, lexical syntax is ignored by the extractor; hence, all metasymbols specified there (most notably the start and the end terminal metasymbols) are lost if the parser specification $G'_{Rascal}(N)$ is re-exported upwards again. Note that we did not develop a tool for inferring the notation specification from its parser: such tool would have been either much too restricted, since it is

[1] BGF stands for **B**NF-like **G**rammar **F**ormat, its logic programming-based specification can be found in previously published sources [LZ09, Zay10, LZ11, ...], and its schema is available as `shared/xsd/bgf.xsd` at SLPS. For understanding this paper, it is enough to assume BGF as a term-like internal representation for context-free grammars.

clearly impossible to automatically extract a notation information from *any* voluntarily written parser, unless some extra information is provided in a lens-like [FGM+07] manner.

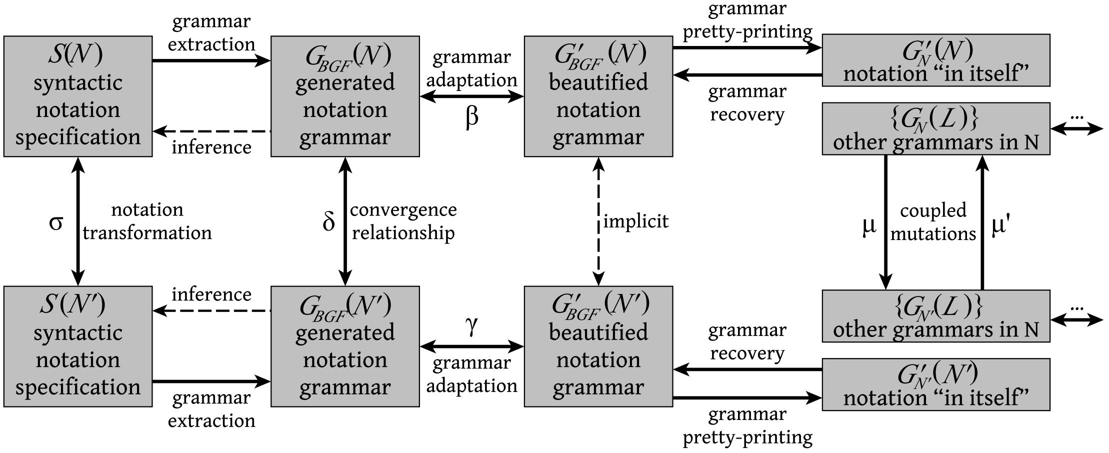
With $G_{BGF}(N)$ being quite a precise definition of N for many purposes, it is not perfect for including it in a documentation, since all nonterminal symbols used in it, would have names that were automatically generated by the grammarware framework. A **beautified notation grammar** $G'_{BGF}(N)$, is linked to $G_{BGF}(N)$ by a bidirectional grammar adaptation relation β, so that $\overrightarrow{\beta}(G_{BGF}(N)) = G'_{BGF}(N)$ and $\overleftarrow{\beta}(G'_{BGF}(N)) = G_{BGF}(N)$. Such a readable grammar can then be pretty-printed in the desired notation, to result in $G_N(N)$, a definition **"in itself"**.

Defining a syntactic notation "in itself" is the current practice in grammar engineering and language documentation. We argue that it is suboptimal and unsuitable for automatic machine processing, because all the notational details that make up the notation specification $S(N)$, are present there only in an indirect way, and it takes effort even for a human reader to extract them on the fly in order to, for example, compare two different notations. However, the reverse of formatting a grammar according to a notation specification, is a technique known as grammar stealing [LV01a] or grammar recovery [LV01b], which is reliable enough to deliver the grammar in precisely the same form that it was stored in, especially if a notation-parametric grammar recovery approach is taken [Zay12b]. Thus, the presence of the notation specification $S(N)$ makes this last step bidirectional and bijective.

We assume a possible presence of **other grammars** $\{G_N(L)\}$ that are also written in the notation N. These grammars can be used for parsing [ALSU06], analysis [PM00], convergence [LZ09], computing differences based on models [Era11], schemata [RB01], graphs [SM96], trees [SZ97] and views [AAN+06], in grammar-based black box testing [FLZ12], for documentation (re)generation [ZL11] and in many other activities. If such an activity expects another syntactic notation, it is useful to provide automated aid in migrating the existing grammarbase.

3 Notation evolution

Consider a similar set up with two related notations. What exactly the relationships between different entities and actions in this megamodel will be, if we agree to approach this solution with maximal automation as the key objective?


$S(N)$ syntactic notation specification
grammar extraction
inference
$G_{BGF}(N)$ generated notation grammar
grammar adaptation
β
$G'_{BGF}(N)$ beautified notation grammar
grammar pretty-printing
grammar recovery
$G'_N(N)$ notation "in itself"
$\{G_N(L)\}$ other grammars in N
...
σ
notation transformation
δ
convergence relationship
implicit
μ
coupled mutations
μ'
$S(N')$ syntactic notation specification
inference
grammar extraction
$G_{BGF}(N')$ generated notation grammar
γ
grammar adaptation
$G'_{BGF}(N')$ beautified notation grammar
grammar recovery
grammar pretty-printing
$\{G_{N'}(L)\}$ other grammars in N
...
$G'_{N'}(N')$ notation "in itself"

Here we see that a notation evolution step Δ consists of the following coupled components:

- σ, a bidirectional **notation transformation** that changes the notation itself;
- δ, a **convergence relationship** that can transform the notation grammars;
- γ, a bidirectional **grammar adaptation** that prepares a beautified readable version of N'.
- μ, an unidirectional **coupled grammar mutation** that migrates the grammarbase according to notation changes;
- possibly μ', an unidirectional **coupled grammar mutation** that migrates the grammarbase according to the inverse of the intended notation changes;

Let us look into these components in more detail.

3.1 Notation transformation

Since we can specify a syntactic notation $S(N)$ and store it as a standalone entity, we can also define a language for transforming it. The bidirectional notation transformation σ describes a relation between $S(N_1)$ and $S(N_2)$ if and only if all differences between N_1 and N_2 are intended and $\overrightarrow{\sigma}(S(N_1)) = S(N_2)$ and $\overleftarrow{\sigma}(S(N_2)) = S(N_1)$. The corresponding transformation language aptly called XEDD is meant to represent notation evolution (see `shared/xsd/xedd.xsd` for the schema and `topics/transformation/xedd/xedd.py` for the XEDD processor). The transformation suite consists of only three operators:

rename-metasymbol(s, v_1, v_2) where s is the metasymbol and values v_1 and v_2 are strings
For example, we can decide to update the notation specification from using "`:`" as a defining metasymbol to using "`::=`". This is the most trivial transformation, but also bidirectional by nature.

introduce-metasymbol(s, v) where s is the metasymbol and v is its desired string value
For example, a syntactic notation can exist without terminator metasymbol, and we may want to introduce one.

eliminate-metasymbol(s, v) where s is the metasymbol and v is its current string value
Naturally, eliminate and introduce together form a bidirectional pair. Specifying the current value of a metasymbol is not necessary, but enables extra validation, as well as trivial bidirectionalisation.

The behaviour of the XEDD processor, however, heavily depends on the particular metasymbol to be removed, introduced or changed, especially when taking all the coupled transformations, mutations and relationships, into consideration. It is also sensible for confix metasymbols that always come in pairs, to have a double introduce and eliminate that deals with start and end metasymbols in one step.

3.2 Convergence relationship

A relationship between two grammars can be expressed within the grammar convergence approach [LZ09] as a sequence of grammar transformation steps. XBGF, an operator suite for programming such grammar transformation steps, was proposed earlier [Zay10, ZLS+12]. It is used for programmable grammar transformations: every step in such a transformation plan is a properly parametrised operator — the semantics of the step is that of the operator, while the applicability and the outcome depend on the parametres. The superiority of XBGF both in expressiveness and attention to details with respect to alternative operator sets, has been demonstrated [LZ11]. However, XBGF is not completely bidirectional by design, so we defined a language for bidirectional grammar transformation on top of it, and called it ΞBGF[2]. A subset of ΞBGF, sufficient for understanding this paper, is presented below:

- **ξbgf:add-removeH**(p_m)
  → xbgf:addH(p_m)
  ← xbgf:removeH(p_m)
- **ξbgf:add-removeV**(p)
  → xbgf:addV(p)
  ← xbgf:removeV(p)
- **ξbgf:designate-unlabel**(p)
  → xbgf:designate(p)
  ← xbgf:unlabel($p.l$)
- **ξbgf:downgrade-upgrade**(p_1, p_2)
  → xbgf:downgrade(p_1, p_2)
  ← xbgf:upgrade(p_1, p_2)
- **ξbgf:extract-inline**(p)
  → xbgf:extract(p)
  ← xbgf:inline($p.n$)
- **ξbgf:factor-factor**(e_1, e_2)
  → xbgf:factor(e_1, e_2)
  ← xbgf:factor(e_2, e_1)
- **ξbgf:fold-unfold**(n)
  → xbgf:fold(n)
  ← xbgf:unfold(n)
- **ξbgf:horizontal-vertical**(n)
  → xbgf:horizontal(n)
  ← xbgf:vertical(n)
- **ξbgf:inline-extract**(p)
  → xbgf:inline($p.n$)
  ← xbgf:extract(p)
- **ξbgf:massage-massage**(e_1, e_2)
  → xbgf:massage(e_1, e_2)
  ← xbgf:massage(e_2, e_1)
- **ξbgf:narrow-widen**(e_1, e_2)
  → xbgf:narrow(e_1, e_2)
  ← xbgf:widen(e_2, e_1)
- **ξbgf:remove-addH**(p_m)
  → xbgf:removeH(p_m)
  ← xbgf:addH(p_m)
- **ξbgf:remove-addV**(p)
  → xbgf:removeV(p)
  ← xbgf:addV(p)
- **ξbgf:rename-renameN**(n_1, n_2)
  → xbgf:renameN(n_1, n_2)
  ← xbgf:renameN(n_2, n_1)
- **ξbgf:rename-renameT**(t_1, t_2)
  → xbgf:renameT(t_1, t_2)
  ← xbgf:renameT(t_2, t_1)
- **ξbgf:replace-replace**(e_1, e_2)
  → xbgf:replace(e_1, e_2)
  ← xbgf:replace(e_2, e_1)
- **ξbgf:reroot-reroot**(n_1^*, n_2^*)
  → xbgf:reroot(n_2^*)
  ← xbgf:reroot(n_1^*)
- **ξbgf:unlabel-designate**(p)
  → xbgf:unlabel($p.l$)
  ← xbgf:designate(p)
- **ξbgf:upgrade-downgrade**(p_1, p_2)
  → xbgf:upgrade(p_1, p_2)
  ← xbgf:downgrade(p_1, p_2)
- **ξbgf:unfold-fold**(n)
  → xbgf:unfold(n)
  ← xbgf:fold(n)
- **ξbgf:vertical-horizontal**(n)
  → xbgf:vertical(n)
  ← xbgf:horizontal(n)
- **ξbgf:widen-narrow**(e_1, e_2)
  → xbgf:widen(e_1, e_2)
  ← xbgf:narrow(e_2, e_1)

Most of the operator names should be self-explanatory: **add-removeH** adds an alternative to any symbol or removes an alternative from an existing choice; **designate-unlabel** assigns a unique label to any production rule or strips an existing production from it; **downgrade-upgrade** replaces a nonterminal with one of its definitions or replaces an expression by a nonterminal that

[2] ΞBGF is read as "ksee bee gee eff", to emphasize its relation to XBGF, "iks bee gee eff".

can be evaluated to it; etc. For more information on the original XBGF commands, an interested reader is redirected to the XBGF manual [ZLS+12].

Most of the operators of XBGF are naturally bidirectional — such are, for example, **renameN** or **factor**: their arguments need only to be swapped in order to form an inverted transformation. Some others form pairs, such as **addV** and **removeV**, or **narrow** and **widen**: if the arguments are identical, one operator is always an inverted form of the other. For defining a purely bidirectional language based on XBGF, we had to address the remaining issues: for example, the XBGF operator **extract** (introduction of a new nonterminal with its subsequent folding) requires a production, but its counterpart **inline** expects just the name of the nonterminal, because its definition (which is about to be unfolded and removed from the grammar) can be observed from the grammar. In general, bidirectionalisation required us to disregard some of XBGF's operators that involved more automation, such as **distribute** (aggressive factoring), since results of **distribute** application can be achieved by using **factor** explicitly and without any loss of generality. We also had to assume non-triviality of operators' parameters and their uniqueness within the given scope, otherwise **rename-renameN**(*a*,*b*) would work incorrectly on *ab* because its reverse application will not be able to distinguish between *b* that needs to be replaced and *b* that needs to stay. In order to simplify this paper somewhat, we reserve a comprehensive investigation into bidirectionalising grammar transformation scripts for future work. ΞBGF is available through SLPS as a schema definition `shared/xsd/ξbgf.xsd`. A processor of ΞBGF maps ΞBGF commands to XBGF ones for forward execution and for reverse one. Two equivalent implementations are available: in XSLT (`shared/tools/ξbgf2xbgf`) and in Rascal (`transform::ΞBGF`).

Classic grammar transformation is used to represent language evolution, correction, adaptation, etc [Pep99, LW01, Läm01, Läm04, LZ11]. Bidirectional grammar transformation is a slightly more stable way to represent a relationship between two languages (or variants of the same language). Imagine for instance a relationship between an abstract syntax and a concrete syntax of the same software language: they are structurally similar, but even in the simplest case the former lacks all the terminals found in the latter and may have different order of arguments for some constructs. Another example that we will see later is a relationship between an automatically derived grammar and the one prepared for publication (such preparation may entail renaming, refactoring for improved readability and hiding uninteresting implementation details). It is fairly straightforward to extend the relationship if one of the involved entities is transformed, which means that we can have the grammar relationship coevolve when the grammars evolve.

3.3 Notation grammar adaptation

The bidirectional grammar adaptation chain β usually consists of two parts: renaming β_n and restructuring β_r. We have emphasized the difference between nominal and structural changes before [LZ11], and in this setup it is even more apparent. Nominal adaptations β_n can always be propagated through the grammar evolution coupled to notation evolution. Structural adaptations are considerably harder to propagate, but they are not that crucial, if we limit the form of the adaptation chain to prevent the use of patterns that rely on the a priori unknown parts of the structure. Thus, if $\delta = \delta_n \circ \delta_r$, $\beta = \beta_n \circ \beta_r$, $\gamma = \gamma_n \circ \beta_r$, then $\overrightarrow{\gamma}_n = \overleftarrow{\delta}_n \circ \overrightarrow{\beta}_n$ and $\overleftarrow{\gamma}_n = \overleftarrow{\beta}_n \circ \overrightarrow{\delta}_n$.

By pushing the nominal adjustments of δ directly to β, we can improve automation by yet another degree and avoid having γ as a manually programmed part of notation transformation framework. In general, γ can always be completely inferred if σ does not introduce any new metaconstructs, and can still be partially inferred otherwise.

In some cases it can be deemed appropriate to let β_r (the restructuring part of the beautifying grammar adaptation) contain transformation steps that are not grammar refactorings. It is common for "readable" grammars to be more liberal than their implementable counterparts, because the learning process of human readers deals with false positives better than an automated language recogniser or a parser. For this purpose, both XBGF and ΞBGF contain language-increasing and language-decreasing grammar transformation operators.

3.4 Grammar mutations

In order to fully comprehend coupled grammar mutations and limits on their bidirectionalisation, let us first formally introduce what we mean by them.

We inherit the term *"grammar transformation"* from existing scientific literature [Pep99, Läm04, ...]. Usually a transformation operator is not completely context independent and can be instantiated with one of more parameters: for example, a **renameN** operator from [LZ11, ZLS+12] needs a source nonterminal name and a target name; only then it can check if the source name is taken and the target one free, and finally perform substitution of all occurrences of one with the other. However, there is a very specific kind of transformations that virtually take the whole source grammar as a parameter: examples from [Zay10] include commands like "strip the grammar of all terminals" (impossible to know all terminals that need to be projected before looking at the grammar) or "reroot to top" (in order to turn all top nonterminals into starting symbols, one needs to calculate the set of top nonterminals). We will call such transformations *"grammar mutations"* to avoid confusion and reach clarity. Mutations were called "automated actions" in the language convergence infrastructure [Zay11] and "transformation generators" elsewhere [Zay10], because they worked by analysing a grammar, generating needed transformations and applying them to the source grammar. However, this is not the only way of implementing grammar mutations, and we abstract from those implementation details here. Mutations are almost unavoidable in practical grammar convergence endeavours with grammars of industrial size, since they save a lot of effort and are easily reusable.

A *grammar transformation operator* τ can be formalised as a triplet $\tau = \langle c_{pre}, t, c_{post} \rangle$, where c_{pre} is a precondition, c_{post} is a postcondition, and t is a transformation operator name. A *grammar transformation* then is $\tau_{a_i}(G)$, where a_i are its parameters of use (of different types and quantity for each operator) and G is the input grammar. When applying a transformation, we can reach different outcomes:

- if a_i are of incorrect types and quantity than expected by t, then τ is *incorrectly called*;
- if the constraint c_{pre} does not hold on G, then τ_{a_i} is *inapplicable* to G;
- if the constraint c_{post} holds on G, then τ_{a_i} is *vacuous* on G;
- if the constraint c_{pre} holds on G, $G' = \tau_{a_i}(G)$ is the transformed grammar, and c_{post} does not hold on G', then t is *incorrectly implemented*;
- if c_{pre} holds on G, $G' = \tau_{a_i}(G)$ is the transformed grammar, and c_{post} holds on G', then τ has been *applied* correctly with arguments a_i to grammar G resulting in grammar G'.

In the scope of XBGF [LZ11, ZLS+12] and grammar convergence [LZ09, Zay10], we were considering all incorrect, inapplicable and vacuous transformations as *unsuccessful*.

As an running example, consider a nonterminal renaming transformation ($t =$ **renameN**). It is incorrectly called unless it is given two nonterminal names as arguments: $a_1, a_2 \in \mathbb{N}$. It is inapplicable to G if a_1 is not defined and not referenced in G. It is also inapplicable to G if a_2 is already defined or referenced in G. It is vacuous if $a_1 = a_2$. Let $G' = \tau_{a_1,a_2}(G)$. If a_1 is still present in G' or if the new definitions of a_2 are not equivalent to the old definitions of a_1 modulo renaming, then t is incorrectly implemented. Otherwise G' is the result of correct application of τ to G with arguments a_1 and a_2.

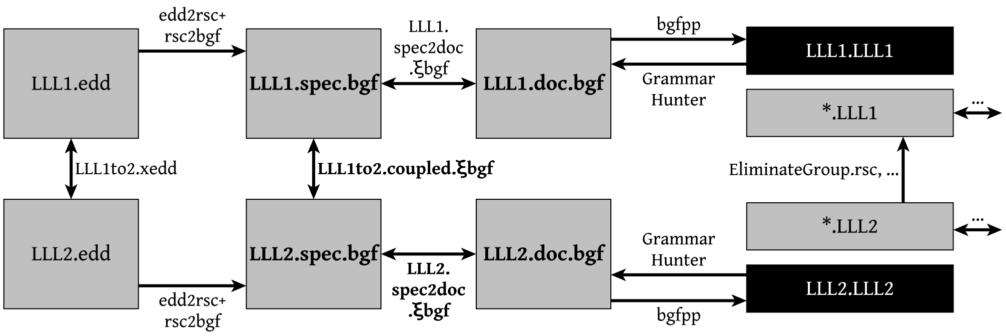
Unlike a grammar transformation, a grammar mutation does not have a single precondition: instead, it has a set of preconditions that serve as triggers for transformations, which we denote as $\mu = \langle\{c_i\},\{t_i\},c_{post}\rangle$. Consider a mutation that makes all nonterminal names uppercase. It has a precondition that holds if a nonterminal name is not uppercase, and triggers a renaming. The mutation terminates once no trigger c_i holds and the postcondition c_{post} is met. Even if no transformations are triggered (i.e., c_{post} holds for G), the application of μ can be considered successful since the goal of enforcing the c_{post} constraint is reached (all nonterminal names are uppercase). If we implement mutations as transformation generators as in [Zay10], we can define mutation failure differently based on applicability and vacuousness of the transformations they generate. In this paper we intentionally disregard such knowledge about implementation details.

Due to this asymmetry in our definition of a grammar mutation, it is purely unidirectional by nature, since it takes a grammar in an unknown state and transforms it into a grammar in a known state. The only way to make it bidirectional is then to only allow mutations between consistent states. Such a *bidirectional grammar mutation* $\mu_{bx} = \langle c_{pre},\{c_i\},\{t_i\},c_{post}\rangle$ will be an instantiation of a grammar mutation, i.e., one grammar mutation spawns forth a whole family of bidirectional grammar mutations. For example, consider the abovementioned example of a mutation that enforces uppercase naming convention for nonterminals. It spawns forth bidirectional mutations that turn lowercase into uppercase, camelcase into uppercase, etc. With this example it also becomes easy to see that the family of spawned bidirectional mutations does not define the original mutation: i.e., $\forall\mu\ \exists G \exists G'\ \nexists\mu_{bx},\ G' = \mu(G) \wedge G' = \overrightarrow{\mu_{bx}}(G) \wedge G = \overleftarrow{\mu_{bx}}(G')$. If G does not follow one naming convention consistently, any possible μ_{bx} should have knowledge about partitioning of $\mathbb{N}$, which does not let us specify μ_{bx} in a general non-parametric way.

This again calls for a lens-like [FGM+07] setup which we try to avoid in this paper. We reserve detailed research of bidirectionalising grammar mutation for future work and focus on more generally applicable unidirectional grammar mutation in this paper instead. The last thing that we want to emphasize is that although the grammar mutation μ is neither naturally bidirectional, nor easily bidirectionalised, the notation specification transformation σ is bidirectional, hence, one can infer the coupled μ' from σ^{-1}. Then, $\overrightarrow{\mu'}$ will not necessarily be equivalent to $\overleftarrow{\mu}$, if no assumptions are made about the grammars in the grammarbase.

4 Evaluation

LLL is an EBNF-like grammar notation used inside Grammar Deployment Kit. There exist at least two variants of it: with a syntax for separator lists and without. They are published in the form of grammars of notations defined "in themselves" in [KLV02] and [Kor03]. Let us recall the megamodel from §3 and see if the proposed solution indeed makes a difference:



Previously existing entities are presented in **dark boxes**. Let us look at them closer here. The LLL1 syntactic notation presented "in itself" looks exactly like this [KLV02, p.2]:

```
grammar    : rule+;
rule       : sort ":" alts ";";
alts       : alt alts-tail*;
alts-tail  : "|" alt;
alt        : term*;
term       : basis repetition?;
basis      : literal | sort;
repetition : "*" | "+" | "?";
```

Note how some nonterminals are left undefined in the original paper, presumably because their definitions were deemed to be trivial by the authors. One of the direct consequences is that this grammar is not immediately useful for parsing, unlike the notation specification that we construct.

We also take the definition of LLL2 from the GDK reference manual [Kor03, p.3][3]:

```
specification : rule+;
rule          : ident ":" disjunction ";";
disjunction   : {conjunction "|"}+;
conjunction   : term*;
term          : basis repetition?;
basis         : ident
              | literal
              | alternation
              | group
              ;
repetition    : "+" | "*" | "?";
alternation   : "{" basis basis "}" repetition;
group         : "(" disjunction ")" ;
```

Since both grammars are extremely small, a human reader can spot differences after some cursory examination, but most of them are not related to language evolution as such: it is purely coincidental whether to call the starting nonterminal symbol "grammar" or "specification"

[3] The original LLL2 grammar contains an error that was noted and fixed in Grammar Tank [ZLS+12]. Here we consider the corrected version. We also remove the special rule for ε for the sake of simplicity of this paper.

and whether to call nonterminal symbols themselves "`sort`"s or "`ident`"(ifier)s. By analysing these grammars, we can manually construct the notation specification of LLL1 in terms of EDD [Zay12a]:

defining metasymbol	`:`	definition separator metasymbol	`|`
terminator metasymbol	`;`	postfix optional metasymbol	?
postfix star metasymbol	⋆	postfix plus metasymbol	+
start terminal metasymbol	`"`	end terminal metasymbol	`"`

Features new to LLL2 with respect to LLL1 are grouping of symbols and separator lists:

start group metasymbol	(	end group metasymbol	)
start separator list star metasymbol	{	end separator list star metasymbol	} ⋆
start separator list plus metasymbol	{	end separator list plus metasymbol	} +

From these tables, we compose and store two notation specifications (the leftmost boxes): `LLL1.edd` and `LLL2.edd`. Since both of them are known to us, the bidirectional evolution σ which is stored as an XEDD sequence, will be used for validating their convergence, not for propagating the changes. In this case, σ, expressed in XEDD, looks like this (see `LLL1to2.xedd`):

introduce-metasymbol(*group*, '(', ')');
introduce-metasymbol(*seplist-star*, '{', '} ⋆');
introduce-metasymbol(*seplist-plus*, '{', '} +');

Now let us try to move to the right in the megamodel. To process notation specifications, we use a Rascal tool called `topics/recovery/edd2rsc` that automatically produces corresponding parser specifications in Rascal. These can be used for IDE support of both notations, but here we view them just as sources for grammar extraction. The extractor, written in Rascal itself, `extract::RascalSyntax2BGF`, automatically produces BGF grammars for both LLL1 and LLL2. To validate correctness of our actions so far, these grammars need to converge. The coupled δ generated by the `topics/transformation/xedd` processor produces the following ΞBGF (see `LLL1to2.coupled.ξbgf`):

rename-rename(*LLL1Grammar*, *LLL2genGrammar*);
rename-rename(*LLL1Production*, *LLL2genProduction*);
rename-rename(*LLL1Definition*, *LLL2genDefinition*);
rename-rename(*LLL1Symbol*, *LLL2genSymbol*);
rename-rename(*LLL1Nonterminal*, *LLL2genNonterminal*);
rename-rename(*LLL1Terminal*, *LLL2genTerminal*);
add-remove(**p**(**l**(*group*), *LLL2genSymbol*, ','(**t**('('),**slp**(*LLL2genDefinition*,'|'),**t**(')'))));
add-remove(**p**(**l**(*sepliststar*), *LLL2genSymbol*, ','(**t**('{'),**n**(*LLL2genSymbol*),**n**(*LLL2genSymbol*),**t**('}∗'))));
add-remove(**p**(**l**(*seplistplus*), *LLL2genSymbol*, ','(**t**('{'),**n**(*LLL2genSymbol*),**n**(*LLL2genSymbol*),**t**('}+'))));

Thus, both notation grammars on this layer, as well as the convergence relationship between them, is derived automatically (presented **in bold** on the megamodel) from the existing entities. If we make another step to the right, both beautified notation grammars, `LLL1.doc.bgf` and `LLL2.doc.bgf`, can be derived from the notations defined "in themselves" (listings we have shown earlier). Since currently we have no instrument to approach fully automated convergence, both the notation grammar `LLL1.spec.bgf` and the beautified notation grammar

ada-kellogg	108	csharp-iso-23270-2003	0	java-1-jls-read	0
ada-kempe	89	csharp-iso-23270-2006	0	java-2-jls-impl	36
ada-laemmel-verhoef	79	csharp-msft-ls-1.0	0	java-2-jls-read	0
ada-lncs-2219	89	csharp-msft-ls-1.2	0	java-5-habelitz	65
ada-lncs-4348	109	csharp-msft-ls-3.0	0	java-5-jls-impl	60
c-iso-9899-1999	0	csharp-msft-ls-4.0	0	java-5-jls-read	1
c-iso-9899-tc2	0	csharp-zaytsev	23	java-5-parr	95
c-iso-9899-tc3	0	dart-google	58	java-5-stahl	92
cpp-iso-14882-1998	0	dart-spec-0.01	56	java-5-studman	91
cpp-iso-n2723	0	dart-spec-0.05	62	mediawiki-bnf	32
csharp-ecma-334-1	0	eiffel-bezault	45	mediawiki-ebnf	30
csharp-ecma-334-2	0	eiffel-iso-25436-2006	345	modula-sdf	50
csharp-ecma-334-3	0	fortran-derricks	101	modula-src-052	65
csharp-ecma-334-4	0	java-1-jls-impl	0	w3c-xpath1	3

Table 1: Applying coupled mutation to **eliminate-metasymbol**$(group)$ to Grammar Zoo. Values mean the number of times the triggers of the grammar mutation fired.

LLL1.doc.bgf, should be used by a grammar engineer as guidance for convergence, resulting in the bidirectional grammar adaptation β, LLL1.spec2doc.ξbgf.

Propagation of nominal refactorings from δ (LLL1to2.coupled.ξbgf) to β in order to form γ (LLL2.spec2doc.ξbgf) is performed by an XSLT script ξbgf$_2$. In general, propagating structural changes is hard and sometimes impossible (for some transformations, there is no easy way to express their permutation in XBGF), and in this particular scenario is even undesirable. We save space in the paper by reserving detailed investigation for future work. What is important here, is that the beautifying grammar adaptation of the generated LLL2 grammar to its desired form, is performed automatically. However, as discussed earlier, the part that beautifies the newly introduced metaconstructs, need to be prepared manually and provided as a part of notation evolution step. Beautified grammars do not need to be converged separately, because they are already converged by the composition of three bidirectional grammar transformation sequences φ such that $\overrightarrow{\varphi} = \overleftarrow{\beta} \circ \overrightarrow{\delta} \circ \overrightarrow{\gamma}$ and $\overleftarrow{\varphi} = \overleftarrow{\gamma} \circ \overleftarrow{\delta} \circ \overrightarrow{\beta}$.

Since all transformations only add new notational features, minimal unidirectional grammar mutations μ that correspond to them, do not change the grammars at all: the postcondition of being able to express the grammar in the given notation holds immediately. In the Table 1 we present results of applying an inverted coupled mutation μ', EliminateGroup.rsc, that corresponds to removing start and end group metasymbols from the notation specification ($\overleftarrow{\sigma}$), to Grammar Zoo [ZLS$^+$12]. Zeros mean the absence of group metasymbols in the original notation that was used as an extraction source — since no groups were found there, there are also no groups in the extracted grammar. Low numbers (like 1 for java-5-jls-read) are observed when the language engineers were planned to go without group metasymbols, but "forgot" about it. High numbers (up to 345 for eiffel-iso-25436-2006) indicate that the functionality we are retiring with this mutation was heavily and intentionally used. The mutations corresponding to the other steps produce similar results, and can be found implemented in Rascal as `EliminateSLS.rsc` for eliminating the star-kind of separator lists and `EliminateSLP.rsc` for treating the plus-kind.

This evaluation has shown us that once the notation specifications are constructed and the changes between them are represented as notation specification transformation steps, the application of grammar recovery tools and bidirectional grammar transformations, either provides significant help (in the case of constructing grammar adaptation β) or completely automates change propagation and verification (all other cases presented in bold on the megamodel).

5 Related and future work

Cicchetti et al [CCLP11] have illustrated that many difficulties arise when two levels of models (models and metamodels in UML/OOP technical space for them; syntax and metasyntax for us) evolve at the same time, and evolution steps not only need to be propagated from one level to the other, but also be combined with transformations already happening there. Since we practically transform the grammars in their internal representation, such conflicts will never arise, because the extraction and exporting steps will naturally take care of any pending metasyntactic evolution. In that respect our approach is closer to the one taken by Wachsmuth in [Wac07], which is only to be expected since he borrows heavily from grammarware engineering. However, Wachsmuth also studies metamodel relation classes (semantics-preserving, introducing, instance-preserving, etc) which can be studied for XEDD as well. For example, introducing a new metasymbol will always be an introducing transformation. However, new classes will need to be defined such as abstract instance preservation: many strictly semantics-preserving metasyntactic transformations such as renaming a terminator metasymbol will not preserve pretty-printed instances, but will essentially preserve their structure, which is different from not preserving instances at all. This issue requires more research into these overlapping technical spaces, as well as in the ways coevolution is addressed in the database domain [CH06] and in evolution of data formats [LL01].

Other formal properties of bidirectional grammar transformations, such as correctness and hippocraticness [Ste07], need further investigation. There are a lot of open questions in bidirectionalising existing grammar transformation, which we mostly solved but need considerably more space for related explanations. Thus, the results of this investigation will be published separately.

Given two parsers of presumably different versions of the same language, one can hardly tell the linguistic difference just from analysing them. In §3, we have stated that the only way to compare parsers directly and automatically was grammar-based differential testing, which is not completely true. In a very lucky yet not impossible scenario, metasyntactic formulae are spotted directly in the source code [LV01a]. This enables very reliable grammar extraction, which produces $G_{BGF}(N)$ in a form very close to $G_{Rascal}(N)$ (or any other $G_{parser}(N)$). Such extracted grammars can be used for direct comparison or for making testing results more reliable.

In §4, we have seen two completely differently looking grammars of LLL1 and LLL2, taken from their respective documentation. In the approach we propose to use in this paper, in order to change the definition of a notation "in itself", we would need to change (or develop, if it does not exist yet) a grammar adaptation chain β. However, $G_N(N)$ can be edited inline, with the readable notation grammar $G'_{BGF}(N)$ extracted from it automatically: given that the edits are purely decorative, the notation itself will stay the same, hence enabling automated reliable recovery. The only problem that stays in the way of implementing this evolution scenario is the (current) inability of inferring bidirectional grammar transformation by looking at two supposedly related grammars. Since this issue is definitely to be addressed in future grammar-related research, this room for improvement can eventually be filled.

6 Conclusion

We have extended XBGF, the grammar transformation operator suite, to bidirectionality. This resulted in ΞBGF, which can be used to formulate grammar convergence, evolution and adaptation scenarios in a more robust and flexible way.

We have also formulated a way to specify a syntactic notation in EDD and a notation transformation in XEDD. The notation specification was designed after extensive analysis of dozens syntactic notations from currently existing language manuals, specifications and standards. In this paper, we have presented a case study taken from real life, when a notation LLL was changed during development of Grammar Deployment Kit. We have represented both the source and the target notation in EDD, and formulated the evolution as XEDD steps.

We have generalised the transformers and generators from prior work to mutations of grammars, which are conceptually deeply different from grammar transformations. A grammar transformation becomes executable when provided with arguments, and can turn out to be inapplicable or vacuous depending on the input grammar. A grammar mutation is always applicable, but not easily bidirectionalisable. We avoid the issue of bidirectionalisation of grammar mutation in this paper by providing automated coupling of grammar mutation to notation evolution.

We have implemented an XEDD processor that evolves the notation specification, automatically infers and delivers a coupled convergence relationship between the source grammar and the target one, propagates the naming changes to the bidirectional adaptation chain, and also delivers a mutation that can migrate the existing grammarbase from the old notation to the new one. All actions performed by the XEDD processor need to be properly parametrised by the notation specification and its transformation steps, but after that are fully automatic.

All schemata, notation specifications, grammars, transformations, mutations, languages and their processors, involved in this research project, are made freely available at the GitHub repository of the Software Language Processing Suite.

Bibliography

[AAN$^+$06] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, D. Garlan. Differencing and Merging of Architectural Views. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. Pp. 47–58. IEEE Computer Society, 2006.

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, second edition, 2006.

[BJV04] J. Bézivin, F. Jouault, P. Valduriez. On the Need for Megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2004.

[CCLP11] A. Cicchetti, F. Ciccozzi, T. Leveque, A. Pierantonio. On the Concurrent Versioning of Metamodels and Models: Challenges and Possible Solutions. In Di Ruscio and Kolovos (eds.), *Proceedings of the Second International Workshop on Model Comparison in Practice (IWMCP'11)*. ACM SIGSOFT, June 2011.

[CFH$^+$09] K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Paige (ed.), *Theory and Practice of Model Transformations*. LNCS 5563, pp. 260–283. Springer Berlin / Heidelberg, 2009.

[CH06] A. Cleve, J.-L. Hainaut. Co-transformations in Database Applications Evolution. In *Revised Papers of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05), Braga, Portugal*. LNCS 4143, pp. 409–421. Springer, 2006.

[Cor06] J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming* 61(3):190–210, 2006.

[Era11] R. Eramo. *Bidirectional and Change Propagating Model Transformations in MDE*. Lambert Academic Publishing, 2011.

[FGM$^+$07] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, May 2007.

[FLZ12] B. Fischer, R. Lämmel, V. Zaytsev. Comparison of Context-free Grammars Based on Parsing Generated Test Data. In Aßmann and Sloane (eds.), *Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*. LNCS 6940, pp. 324–343. Springer, Heidelberg, 2012.

[GJSB05] J. Gosling, B. Joy, G. L. Steele, G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005. Available at http://java.sun.com/docs/books/jls.

[KLV02] J. Kort, R. Lämmel, C. Verhoef. The Grammar Deployment Kit: System Demonstration. In van den Brand and Lämmel (eds.), *Electronic Notes in Theoretical Computer Science*. Volume 65. Elsevier Science Publishers, 2002.

[Kor03] J. Kort. Grammar Deployment Kit: Reference Manual. Universiteit Amsterdam, May 2003. http://gdk.sourceforge.net/gdkref.pdf.

[KSV11] P. Klint, T. van der Storm, J. Vinju. EASY Meta-programming with Rascal. In Fernandes, Lämmel, Visser and Saraiva (eds.), *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering* (GTTSE'09). LNCS 6491, pp. 222–289. Springer-Verlag, Berlin, Heidelberg, January 2011.

[Läm01] R. Lämmel. Grammar Adaptation. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*. LNCS 2021, pp. 550–570. Springer-Verlag, 2001.

[Läm04] R. Lämmel. Transformations Everywhere. *Science of Computer Programming. Special Issue on Program Transformation* 52(1–3):1–8, August 2004. Editorial.

[LL01] R. Lämmel, W. Lohmann. Format Evolution. In *Proceedings of the Seventh International Conference on Reverse Engineering for Information Systems (RETIS'01)*. books@ocg.at 155, pp. 113–134. OCG, 2001.

[LV01a] R. Lämmel, C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pp. 78–88, Nov./Dec. 2001.

[LV01b] R. Lämmel, C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience* 31(15):1395–1438, December 2001.

[LW01] R. Lämmel, G. Wachsmuth. Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment. In *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01)*. ENTCS 44. Elsevier Science, 2001.

[LZ09] R. Lämmel, V. Zaytsev. An Introduction to Grammar Convergence. In Leuschel and Wehrheim (eds.), *Proceedings of 7th International Conference on Integrated Formal Methods (iFM'09)*. LNCS 5423, pp. 246–260. Springer-Verlag, Berlin, Heidelberg, February 2009.

[LZ11] R. Lämmel, V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal* 19(2):333–378, June 2011.

[Pep99] P. Pepper. LR Parsing = Grammar Transformation + LL Parsing. Technical report CS-99-05, TU Berlin, 1999.

[PM00] J. F. Power, B. A. Malloy. Metric-Based Analysis of Context-Free Grammars. In *Proceedings of the Eighth International Workshop on Program Comprehension (IWPC'00)*. Pp. 171–178. IEEE Computer Society, Washington, DC, USA, 2000.

[RB01] E. Rahm, P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal* 10:334–350, December 2001.

[SM96] E. Salvat, M.-L. Mugnier. Sound and Complete Forward and Backward Chainings of Graph Rules. In Eklund et al. (eds.), *Conceptual Structures: Knowledge Representation as Interlingua*. LNCS 1115, pp. 248–262. Springer, 1996.

[Ste07] P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *MODELS'07*. LNCS 4735, pp. 1–15. Springer, 2007.

[SZ97] D. Shasha, K. Zhang. Approximate Tree Pattern Matching. In Apostolico and Galil (eds.), *Pattern Matching Algorithms*. Pp. 341–369. Oxford University Press, 1997.

[Wac07] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Ernst (ed.), *ECOOP'07*. LNCS 4609, pp. 600–624. Springer, July 2007.

[Wir77] N. Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM* 20(11):822–823, 1977.

[Zay10] V. Zaytsev. *Recovery, Convergence and Documentation of Languages*. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, October 2010.

[Zay11] V. Zaytsev. Language Convergence Infrastructure. In Fernandes, Lämmel, Visser and Saraiva (eds.), *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering* (GTTSE'09). LNCS 6491, pp. 481–497. Springer-Verlag, Berlin, Heidelberg, January 2011.

[Zay12a] V. Zaytsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In Mernik and Bryant (eds.), *Programming Languages Track, Volume II of the Proceedings of the 27th ACM Symposium on Applied Computing (SAC 2012)*. Pp. 1910–1915. ACM, March 2012.

[Zay12b] V. Zaytsev. Notation-Parametric Grammar Recovery. In Sloane and Andova (eds.), *Pre-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012)*. Pp. 105–118. Institute of Cybernetics at Tallinn University of Technology, March 2012.

[ZL11] V. Zaytsev, R. Lämmel. A Unified Format for Language Documents. In Malloy, Staab and van den Brand (eds.), *Post-proceedings of the Third International Conference on Software Language Engineering (SLE 2010)*. LNCS 6563, pp. 206–225. Springer-Verlag, Berlin, Heidelberg, January 2011.

[ZLS$^+$12] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, G. Wachsmuth. Software Language Processing Suite[4] 2008–2012. http://grammarware.github.com. Contains, among other works: *XBGF Manual: BGF Transformation Operator Suite v.1.0* (V. Zaytsev, August 2010), http://grammarware.github.com/xbgf; Grammar Zoo (V. Zaytsev, 2009–2012), http://grammarware.github.com/zoo; Grammar Tank (V. Zaytsev, 2011–2012), http://grammarware.github.com/tank.

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