Guided Grammar Convergence

Vadim Zaytsev, vadim@grammarware.net

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

Abstract. Relating formal grammars is a hard problem that balances between language equivalence (which is known to be undecidable) and grammar identity (which is trivial). In this paper, we investigate several milestones between those two extremes and propose a methodology for inconsistency management in grammar engineering. While conventional grammar convergence is a practical approach relying on human experts to encode differences as transformation steps, guided grammar convergence is a more narrowly applicable technique that infers such transformation steps automatically by normalising the grammars and establishing a structural equivalence relation between them. This allows us to perform a case study with automatically inferring bidirectional transformations between 11 grammars (in a broad sense) of the same artificial functional language: parser specifications with different combinator libraries, definite clause grammars, concrete syntax definitions, algebraic data types, metamodels, XML schemata, object models.

1 Introduction

Modern grammar theory has shifted its focus from general purpose programming languages to a broader scope of *software languages* that comprise programming languages, domain specific languages, markup languages, API libraries, interaction protocols, etc [12]. Such software languages are specified by *grammars in a broad sense* that still rely on the familiar infrastructure of terminals, nonterminals and production rules, but specify general commitment to grammatical structure found in software systems. In that sense, a type safe program commits to a particular type system; a program that uses a library, commits to using its exposed interface; an XML document commits to the structure defined by its schema — failure to commit in any of these cases would mean errors in interpretation of the language entity. These, and many other, scenarios can be expressed and resolved in terms of grammar technology, but not all structural commitments profit from grammatical approach (as the most remarkably problematic ones we can note indentation policies and naming conventions).

One of the problems of multiple implementations of the same language, which is known for many years, is having an abstract syntax definition and a concrete syntax definition [25]. Basically, the abstract syntax defines the kind of entities that inhabit the language and must be handled by semantics specification. A concrete syntax shows how to write down language entities and how to read them back. It is not uncommon for a programming language to have several possible concrete syntaxes: for example, any binary operation may use prefix, infix or postfix notation, without any changes to the language semantics. Indeed, we have seen infix dialects of postfix Forth (Forthwrite, InfixForth) and prefix dialects of infix REBOL (Boron). For software languages, the problem is broader: we can speak of one *intended language* specification and a variety of abstract and concrete syntaxes, data models, class diagrams, metamodels and similar contracts that conform to it.

Our definition of the intended language relies on bidirectional transformations [1,17,22,28] and in particular on their notation by Meertens [17], which we redefine here for the sake of completeness and clarity:

Definition 1. *For a relation $R \subseteq S \times T$, a* ***semi-maintainer*** *is a function $\rhd : S \times T \to T$, such that $\forall x \in S, \forall y \in T, \langle x, x \rhd y \rangle \in R$, and $\forall x \in S, \forall y \in T, \langle x, y \rangle \in R \Rightarrow x \rhd y = y$.*

The first property is called *correctness* and ensures that the update caused by the semi-maintainer restores the relation. The second property is *hippocraticness* and states that an update has no effect ("does no harm"), if the original pair is already in the relation [22]. Other properties of bidirectional transformations such as *undoability* are often unachievable. A *maintainer* is a pair of semi-maintainers $\rhd$ and $\lhd$. A *bidirectional mapping* is a relation and its maintainer.

Definition 2. *A grammar G conforms to the language* ***intended*** *by the master grammar M, if there exists a bidirectional mapping between instances of their languages.*

$$\begin{aligned} G \models L(M) \iff & \exists R \subseteq L(G) \times L(M) \\ & \exists \rhd : L(G) \times L(M) \to L(M) \\ & \exists \lhd : L(G) \times L(M) \to L(G) \end{aligned}$$

Naturally, for any grammar holds $G \models L(G)$.

For example, consider a concrete syntax G_c of a programming language used by programmers and an abstract syntax $M = G_a$ used by a software reengineering tool. We would need $\rhd$ to produce abstract syntax trees from parse trees and $\lhd$ to propagate changes done by a reengineering tool, back to parse trees. If those can be constructed — examples of algorithms can be seen in [10,23,25], — then G_c conforms to the language intended by G_a. As another example, consider an object model used in a tool that stores its objects in an external database (XML or relational): the existence of a bidirectional mapping between entries (trees or tables) in the database and the objects in memory, means that they represent the same intended language, even though they use very different ways to describe it and one may be a superlanguage of the other.

Roadmap. In the following sections, we will briefly present the following milestones of relationships between languages:

§2. *Grammar identity*: structural equality of grammars

§3. *Nominal equivalence*: name-based equivalence of grammars

§4. *Structural equivalence*: name-agnostic footprint-matching equivalence

§5. *Abstract normalisation*: structural equivalence of normalised grammars

Then, §6 summarises the proposed method and discusses its evaluation.

Finally, §7 concludes the paper by establishing context and contributions.

2 Grammar identity

Let us assume that grammars are traditionally defined as quadruples $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S} \rangle$ where their elements are respectively the sets of nonterminal symbols, terminal symbols, production rules and starting nonterminal symbols.

Definition 3. *Grammars G and G' are **identical**, if and only if all their components are identical: $G = G' \iff \mathcal{N} = \mathcal{N}' \wedge \mathcal{T} = \mathcal{T}' \wedge \mathcal{P} = \mathcal{P}' \wedge \mathcal{S} = \mathcal{S}'$.*

The definition is trivial, and in practice is commonly weakened somehow. For example, many metalanguages allow the right hand sides of rules from $\mathcal{P}$ to contain disjunction (inner choice), which is known to be commutative, so it is natural to disregard the order of disjunctive clauses when comparing grammars — e.g., `gdt`, the "grammar diff tool" used in [15,29] implements that. However, many grammar manipulation technologies such as PEG [8] or TXL [2], use ordered choices, so this optimisation can be perceived as premature. For this reason, we will explicitly abandon disjunction in later sections.

3 Nominal equivalence

Since identity can be seen as a trivial bijection, a disciplined weakening of Def. 3 that works across all grammars in a broad sense, is this:

Definition 4. *Grammars G and G' are **nominally equivalent**, if there is a bijection β between their production rules:*

$$G \approxeq G' \iff \mathcal{N} = \mathcal{N}' \wedge \mathcal{T} = \mathcal{T}' \wedge \mathcal{S} = \mathcal{S}' \wedge \exists \beta : \mathcal{P} \to \mathcal{P}',$$
$$\forall q \in \mathcal{P}', \exists p \in \mathcal{P}, q = \beta(p); \qquad \forall p_1, p_2 \in \mathcal{P}, \beta(p_1) = \beta(p_2) \Rightarrow p_1 = p_2$$

Algorithms that are used to construct β can be different. For example, in Popart the metalanguage is designed in such a way that it contains enough information to generate both abstract and concrete syntaxes [25]. In TIF-grammars, a concrete syntax specification is annotated with directions on which nodes need to be folded/unfolded or removed when constructing an abstract syntax tree (AST) [10]. In Rascal, the `implode` function that maps parse trees to ASTs uses names of nonterminals and subexpressions to direct the automatic construction of β: for example, an optional nonterminal occurrence can be mapped to a string, a list or a Boolean; a Kleene star can be imploded to either a list or a set, etc [13]. Similar techniques can be spotted in OOP [4], in MDE [6], in data binding frameworks [7], etc.

Once the bijective β is agreed on the level of grammars, we need to construct a coupled maintainer on the language instance level. If it cannot be constructed, then β is useless for us. However, there are many cases when the maintainer can be constructed to be bidirectional (R from Def. 2 can be partial and $\rhd$ and/or $\lhd$ can be only injective but not surjective). An example of that is matching different representations of lists/sets: "one or more" and "zero or more" Kleene repetitions are commonly used in syntactic notations, but one can always be bidirectionally matched to the other. In our prototype implementation, we disregard unreachable nonterminals, treat built-in and user-defined nonterminals equally, allow sequence element permutations and desugar metasyntactic constructs like "separator lists", as well as match different kinds of lists/sets. Such strategy set was chosen to be valid for nominal matching techniques, but also useful for the following sections.

4 Structural equivalence

In order to reuse the methods from the nominal equivalence approach in the case when nonterminal names do not match, we need to construct an additional mapping between the nonterminals, and use that instead of nominal identity. We shall refer to this mapping as *nominal resolution.* To construct it, we generalise permutations and construct *signatures* that express general structure of a production rule. Such signatures depend heavily on the expressiveness of the chosen metalanguage (in particular, when the formal grammar is defined with just terminals and nonterminals, their use is severely limited), but most common metalanguages contain enough functionality to allow signatures to work.

Definition 5. *A* ***footprint*** $\pi_n(x)$ *of a nonterminal* n *in an expression* x *is defined as a multiset of presence indicators.*

$$\pi_n(x) \overset{def}{=} \begin{cases} \{1\} & \text{if } x = n \\ \{?\} & \text{if } x = n? \\ \{+\} & \text{if } x = n^+ \\ \{*\} & \text{if } x = n^* \\ \pi_n(y) & \text{if } x = {}_{name:}y \\ \pi_n(e_1) \cup \pi_n(e_2) & \text{if } x = e_1 e_2 \\ \varnothing & \text{otherwise} \end{cases}$$

Definition 6. *Two footprints are* ***equivalent****, if they are equal modulo repetition kinds:* $\pi \approx \xi \iff \pi = \xi \vee \pi' = \xi'$*, where* ζ' *is* ζ *with all* $+$ *elements replaced by* $*$ *elements.*

Note how disjunction is missing from the definition of a footprint. We will see later how it can be removed from any grammar by factoring and folding. The identity of footprints follows the standard definition of identity of multisets (which naturally subsumes abstraction over permutations). We also define equivalence of them that generalises the treatment of lists discussed in the previous section. Footprints together form a signature.

Definition 7. *A prodsig, or a* ***signature*** *of a production rule* $p = (m ::= x)$ *is defined as a set of tuples with nonterminals used in its right hand side and their footprints:* $\sigma(m ::= x) = \{\langle n, \pi_n(x)\rangle \mid n \in \mathcal{N}, \pi_n(x) \neq \varnothing\}$.

For example, the prodsig of a production rule $P ::= F^+$ is $\{\langle F, \{+\}\rangle\}$ and the prodsig of $F ::= SS^*E$ is $\{\langle E, \{1\}\rangle, \langle S, \{1, *\}\rangle\}$.

Definition 8. *Two production rules are* ***prodsig-equivalent****, if and only if there is an equivalent match between tuple ranges of their signatures:*

$$p \Leftrightarrow q \iff \forall\langle n, \pi\rangle \in \sigma(p),\ \exists\langle m, \xi\rangle \in \sigma(q),\ \pi \approx \xi$$

Consider a simple case of exactly one production rule taken from each of the grammars: p_m from the master grammar and p_s from the servant grammar. Suppose that the left hand sides of them are assumed to match, and we want to see if the right hand sides are matched nominally as well, and whether they deliver any new information with respect to nominal resolution. When prodsigs $\sigma(e_m)$ and $\sigma(e_s)$ are constructed, we have effectively built relations that bind nonterminal names to their occurrences. By subsequently matching the *ranges* of them with either strong or weak prodsig-equivalence, we can infer nominal matching of the nonterminals by matching the *domains* of the relations.

Definition 9. *For any two prodsig-equivalent production rules p and q, $p \bowtie q$, there is (at least one) nominal resolution relationship $p \diamond q$ that satisfies:*

$$\forall \langle a,b\rangle \in p \diamond q : \quad a = \omega \vee b = \omega \ \vee \exists \pi, \exists \xi, \pi \approx \xi, \langle a,\pi\rangle \in \sigma(p), \langle b,\xi\rangle \in \sigma(q)$$
$$\forall \langle c,d\rangle \in p \diamond q : \quad a = c \neq \omega \Rightarrow b = d, \text{where } \omega \text{ denotes unmatched nonterminals.}$$

For two arbitrarily provided grammars (presumably of the same intended software language, but not necessarily admitting any kind of equivalence), we cannot claim the existence of only one nominal resolution that works across all their production rules, but we can attempt to construct a minimum possible one:

Definition 10. *Given two grammars G_1 and G_2, a nominal resolution between them is a relation between their nonterminals $\Diamond \in \mathcal{N}_1 \times \mathcal{N}_2$ such that $\forall p_i \in \mathcal{P}_1$, if $\exists q_j \in \mathcal{P}_2, p_i \bowtie q_j$, then $\exists \diamond_{ij} \subset \Diamond$, such that $p_i \diamond_{ij} q_j$.*

In our case study, we have used various definitions of the same toy functional language FL taken from [15]. For instance, some grammars were extracted from object models by analysing Java code. Since the Java implementation of FL used `List<Expr>` to represent lists, the production rules for function declaration and function call assumed zero or more arguments, while the master grammar assumed one or more. Hence, production rules $F_1 ::= S_1 S_1^* E_1$ and $E_1 ::= S_1 E_1^*$ were matched with their prodsig-equivalent counterparts $F_2 ::= S_2 S_2^+ E_2$ and $E_2 ::= S_2 E_2^+$. All the various grammars of FL, their prodsigs and nominal matching reports are exposed for inspection in the full report on the case study [27].

5 Abstract normalisation

In order to apply the methodology based on nonterminal footprints, production signatures and their equivalence relations, we need the input grammars to comply with some assumptions that have been left informal so far. In particular, we can foresee possible problems with names/labels for production rules and subexpressions, terminal symbols (often not a part of the abstract syntax), disjunction (inner choices, also non-factored), separator lists and other metasyntactic sugar, non-connected nonterminal call graph, inconsistent style of production rules, etc. If by $\mathcal{P}_n \subset \mathcal{P}$ we denote the subset of production rules concerning one particular nonterminal: $\mathcal{P}_n = \{p \in \mathcal{P} \mid p = n ::= \alpha,\ \alpha \in (\mathcal{N} \cup \mathcal{T})^*\}$, then we can define the Abstract Normal Form as follows:

Definition 11. *A grammar $G = \langle \mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S}\rangle$, where $\mathcal{T} = \varnothing$ and $\mathcal{S} = \{s\}$, is said to be in Abstract Normal Form, if and only if:*

- *$\mathcal{N}$ is decomposable to disjoint sets, such that $\mathcal{N} = \mathcal{N}_+ \cup \mathcal{N}_- \cup \mathcal{N}_\perp$*
- *One of them is not empty and includes the root: $s \in \mathcal{N}_+ \cup \mathcal{N}_-$*

- *Nonterminals from one subset are undefined:* $n \in \mathcal{N}_\perp \Rightarrow \mathcal{P}_n = \varnothing$
- *Nonterminals from one other subset are defined with exactly one rule:* $n \in \mathcal{N}_- \Rightarrow |\mathcal{P}_n| = 1,\ \mathcal{P}_n = \{n ::= \alpha\},\ \alpha \in \mathcal{N}^+$
- *Nonterminals from the other subset are defined with chain rules:* $n \in \mathcal{N}_+ \Rightarrow \forall p \in \mathcal{P}_n,\ p = (n ::= x),\ x \in \mathcal{N}$

In fact, any grammar can be rewritten to assume this form: in our prototype implementation, this is done by programming a *grammar mutation* [28]. (A grammar mutation is a general intentional grammar change that cannot be expressed independently of the grammar to which it will be applied. Thus, if "rename" is a parametric grammar transformation operator, then "rename A to B" is a transformation, but "rename all nonterminals to uppercase" is a mutation that is equivalent to transformations like "rename a to A" or "rename b to B" depending on the input grammar). Our prototype, `normal::ANF`, is a metaprogram in Rascal [13] that is available for inspection as open source [29]. It is in fact a superposition of mutations that address the items from the definition individually: remove labels, desugar separator lists, fold/unfold chain production rules, etc.

All the rewritings performed by transforming a grammar to its ANF, are assumed to be monadic in the sense of not only normalising the grammar, but also yielding a bidirectional grammar transformation chain which execution would normalise the grammar. (In our implementation, these steps are specified in the ΞBGF language, primarily because no other bidirectional grammar transformation operator suite exists). The bidirectional grammar transformation chain can then be coupled to the bidirectional mapping between language instances from Def. 2, with the methodology described by [28]. This is required for traceability: the conversion to ANF is one of the steps to achieve automated convergence, not a one-way preprocessing.

6 Discussion

To summarise, grammar convergence is a technique of relating different grammars in a broad sense of the same intended software language [15]. It relies on the transformations being programmed by an experienced grammar engineer: even beside the required expertise, the process is not incremental — the transformation steps need to be considered carefully and constructed for each new grammar added to the mix. With the definitions from previous sections, we have described the process of *guided grammar convergence*, where the master grammar of the intended language is constructed once, and the transformations are inferred for any directly available grammars as well for the ones possibly added in the future. The process works as follows.

- Extract pure grammatical knowledge from the grammar source.
- Use grammar mutations to preprocess your grammars, if necessary.
- Normalise the grammar by removing all problematic/ambiguous constructs.
- Start by matching the roots of the connected normalised grammar.

- Match multiple production rules by prodsig-equivalence; infer new nominal matches by matching equivalent prodsigs. Repeat for all nonterminals.
- If several matches are possible, explore all and fallback in case of failure. If global nominal resolution scheme was impossible to infer, fail.
- Resolve structural differences in the production rules that matched nominally.

To evaluate the method of guided grammar convergence, we have applied it to a case study of **11** different grammars of the same intended functional language that was defined and used earlier in order to demonstrate the original grammar convergence method that converged **5** of these grammars. The following grammar sources were used (all of them are available in the repository of SLPS [29] together with their evolution history and authorship attribution):

adt: an algebraic data type[1] in Rascal [13];
antlr: a parser description in the input language of ANTLR [19], with semantic actions (in Java) intertwined with EBNF-like productions;
dcg: a logic program written in the style of definite clause grammars [20];
emf: an Ecore model, automatically generated by Eclipse [5] from the XML Schema of the **xsd** source;
jaxb: an object model obtained by a data binding framework, generated automatically by JAXB [7] from the XML schema for FL;
om: a hand-crafted object model (Java classes) for the abstract syntax of FL;
python: a parser specification in a scripting language, using the PyParsing library [16];
rascal: a concrete syntax specification in the metaprogramming language of Rascal language workbench [13];
sdf: a concrete syntax definition in the notation of SDF [11] with scannerless generalized LR parsing as parsing model.
txl: a concrete syntax definition in the notation of TXL (Turing eXtender Language) transformational framework [2], which, unlike SDF, uses a combination of pattern matching and term rewriting).
xsd: an XML schema [9] for the abstract syntax of FL.

The complete case study is too big to be presented here, interested readers are redirected to a 40+ page long report [27], containing all production rules, signatures, matchings and transformations. The case study was successful: on average ANF was achieved after 20–30 transformation steps, nominal resolution took up to 9 (proportional to the number of nonterminals) and structural resolution needed 0–5 more steps, after which all 11 grammars were converged. ANTLR, DCG and PyParsing used layered definitions and therefore were the only three grammars to require mutation (another 2–6 steps). The case study is available for investigation and replication both in the form of Rascal metaprograms at `http://github.com/grammarware/slps` [29] (the main algorithm is

[1] `http://tutor.rascal-mpl.org/Courses/Rascal/Declarations/AlgebraicDataType/AlgebraicDataType.html`.

located in the `converge::Guided` module which can be observed and modified at `shared/rascal/src/converge/Guided.rsc`) and as a PDF report with all grammars and transformations pretty-printed automatically [27].

Guided grammar convergence is a methodology stemming from the grammarware "technological space" [14]. When looking for similar techniques in other spaces (engaging in "space travel"), the obvious candidates are schema matching and data integration in the field of data modeling and databases [21]; comparison of UML models or metamodels in the context of model-driven engineering [6,26]; model weaving for product line software development [3,24]; computation of refactorings from different OO program versions [4,18]; etc. For example, [3] utilised metamodel properties for automatically producing weaving models. The core difference is that (meta)model weaving ultimately aims at incorporating all the changes into the resulting (meta)model, while guided grammar convergence also makes complete sense when some changes in the details are disregarded. The lowest limit in this process is needed (otherwise additional claims on the minimality of inferred transformations are required), and we specify this lowest limit as the master grammar. Another difference is that model weaving rarely involves a number of models bigger than two, and even our case study of guided grammar convergence had 10+ grammars in it. In general, prodsig-based matching is more lightweight than those methods, since it in fact compares straightforwardly structured prodsigs and thus easily wins in performance and implementability but loses in applicability to complex scenarios.

7 Conclusion

We knew that language equivalence is undecidable and that grammar identity is trivial. In this paper, we have attempted to reach a useful level of reasoning about language relationships by departing from grammar identity as the "easy" side of the spectrum. This was done in the scope of grammar convergence, when several implementations of the same software language are inspected for compatibility.

A definition of an *intended language* was provided (Def. 2) based on a bidirectional transformation between language entities. Then we revisited existing and possible techniques of structural matching that assumed nominal identity (Def. 3). In order to automatically infer nominal matching, we introduced nonterminal footprints (Def. 5), production signatures (Def. 7) and various degrees of equivalence among them. An extensive normalisation scheme (Def. 11) was proposed to transform any given grammar into the form most suitable for nominal and then structural matching. It has been explained that when such normalisation is not enough, a more targeted yet still automated approach is needed with grammar mutation strategies making the method robust with respect to different grammar design decisions, such as the use of layers instead of priorities or recursion instead of iteration. Just as all other parts of the proposed process, these mutations operate automatically and do not require human intervention.

A case study was used to evaluate the proposed method of guided grammar convergence. The experiment concerned several implementations of a simple functional language in ANTLR, DCG, Ecore, Java, Python, Rascal, SDF, TXL, XML Schema. The diversity in language processing frameworks — metaprogramming languages, declarative specifications, syntax definitions, algebraic data types, parsing libraries, transformation frameworks, software models, parser definitions — was intentional and aimed at stressing the definition of the intended language and the guided convergence method. Casting all grammars from our case study to ANF allowed us to make inference quicker and with less obstacles, as well as to explain the process more clearly.

All artifacts discussed on the pages of this paper, are transparently available to the public through a GitHub repository [29]. For each of the sources of the case study, one could inspect the original file, the extracted grammar, the extractor itself, the mutations that have been derived and applied, the normalisations to ANF, the normalised grammar, the nominal resolution and reasons for each match, as well as the structural resolution steps. One could also investigate the implementation of the method of guided grammar convergence, the algorithm for calculating prodsigs and the process of convergence. Supplementary material contains 40+ pages of the full report, also generated by our prototype [27].

On the practical side, guided grammar convergence provides a balanced method of grammar manipulation, positioned right between unstructured inline editing (which makes grammar development very much like software development but lacks important properties such as traceability and reproducibility) and strictly exogenous functional transformation (which requires substantially more effort but is robust, repeatable and exposes the semantics). Its future role can be seen as a support for **grammar product lines** that allows both steady adaptation plans for deriving secondary artifacts from the reference grammar, and occasional inline editing of the derived artifacts with subsequent automated restoration of the adaptation scripts. This is a contribution to the field of engineering discipline for grammarware [12].

References

1. K. Czarnecki, J. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *TPMT*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.
2. T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Grammar Programming in TXL. In *SCAM*. IEEE, 2002.
3. M. D. Del Fabro and P. Valduriez. Semi-Automatic Model Integration Using Matching Transformations and Weaving Models. In *SAC*, pages 963–970, 2007.
4. D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated Detection of Refactorings in Evolving Components. In *ECOOP*, volume 4067 of *LNCS*, pages 404–428. Springer, 2006.
5. Eclipse. Eclipse Modeling Framework Project (EMF 2.4), 2008. `http://www.eclipse.org/modeling/emf`.
6. J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In *MoDELS*, volume 5301 of *LNCS*, pages 326–340. Springer, 2008.

7. J. Fialli and S. Vajjhala. *Java Specification Request 31: XML Data Binding Specification*, 1999.
8. B. Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In *POPL*, January 2004.
9. S. Gao, C. M. Sperberg-McQueen, and H. S. Thompson. W3C XML Schema Definition Language (XSD) 1.1. *W3C Recommendation*, Apr. 2012.
10. A. Johnstone and E. Scott. Tear-Insert-Fold grammars. In *LDTA*, pages 6:1–6:8. ACM, 2010.
11. P. Klint. Meta-Environment for Generating Program- ming Environments. *TOSEM*, 2(2):176–201, 1993.
12. P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *TOSEM*, 14(3):331–380, 2005.
13. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *GTTSE*, volume 6491 of *LNCS*, pages 222–289. Springer, January 2011.
14. I. Kurtev, J. Bézivin, and M. Akşit. Technological Spaces: an Initial Appraisal. In *CoopIS, DOA'2002, Industrial track*, 2002.
15. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *iFM*, volume 5423 of *LNCS*, pages 246–260. Springer, February 2009.
16. P. McGuire. *Getting Started with Pyparsing*. O'Reilly, first edition, 2007.
17. L. Meertens. Designing Constraint Maintainers for User Interaction. Manuscript, June 1998.
18. M. O'Keeffe and M. O. Cinnéide. Search-Based Refactoring: an Empirical Study. *JSME*, 20(5):345–364, 2008.
19. T. Parr. ANTLR—ANother Tool for Language Recognition, 2008.
20. F. Pereira and D. Warren. Definite Clause Grammars for Language Analysis. In *Readings in Natural Language Processing*, pages 101–124. Morgan Kaufmann Publishers Inc., 1986.
21. E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10:334–350, December 2001.
22. P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In *MODELS*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007.
23. T. v. d. Storm, W. R. Cook, and A. Loh. Object Grammars: Compositional & Bidirectional Mapping Between Text and Graphs. In *SLE*, volume 7745 of *LNCS*, pages 4–23. Springer, July 2013.
24. M. Völter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC*, pages 233–242. IEEE, 2007.
25. D. S. Wile. Abstract Syntax from Concrete Syntax. In *ICSE*, pages 472–480, 1997.
26. Z. Xing and E. Stroulia. Refactoring Detection based on UMLDiff Change-Facts Queries. In *WCRE*, pages 263–274. IEEE, 2006.
27. V. Zaytsev. Guided Grammar Convergence. Full Case Study Report. Generated by `converge::Guided`. *ACM CoRR*, 6541:1–44, July 2012.
28. V. Zaytsev. Language Evolution, Metasyntactically. *EC-EASST*, 49, 2012.
29. V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, and G. Wachsmuth. Software Language Processing Suite[2], 2008–2013. http://slps.github.io.

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