Negotiated Grammar Transformation

Vadim Zaytsev
Software Analysis & Transformation Team, Centrum Wiskunde & Informatica,
Amsterdam, The Netherlands
vadim@grammarware.net

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 and explains two kinds of different adaptability of transformation (through tolerance and through adjustment) and contains examples of possible usage of this negotiated grammar transformation process.

Categories and Subject Descriptors

F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Grammarware*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program Transformation*; C.4 [**Performance of Systems**]: Fault Tolerance—*Adaptability*

Keywords

Tolerance, grammar transformation, extreme modelling.

1. INTRODUCTION

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 **rename** is an operator that changes the name of one nonterminal symbol, then **rename**(a, b) is a valid transformation command. However, suppose that the nonterminal a disappears from the original grammar (due to some evolution happening concurrently: renaming, unfolding, 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. However, such "fragment" is not always sequential, it is in fact more of a slice — for instance, in the above-mentioned example with renaming a to b, the applicability condition concerns presence of any production rules defining or referring to a).

Existing research on adaptability in grammarware mostly concerns adaptation of grammars towards a specific cause [4, 9, 12, 14, 16, 17, 18, 25]; while adaptation and co-adaptation of grammar transformation scripts remains a much less popular topic [15, 20].

This problem is not at all specific to the XBGF/ΞBGF operator suite that we use as the backend of our prototypes [18, 25]. Coarse grammar transformations redefining nonterminals entirely or adding new production rules to them, which is common for frameworks like TXL [4] or GDK [12], are robust to a greater extent, but there is no control over the kind of adaptation we will experience (tolerance or adjustment, see next section). Finer grammar transformations that can, for example, fold a symbol sequence as a definition of a new nonterminal or change one particular repetition from the "one or more" kind to the "zero or more", that are possible with frameworks like GRK [16] or FST [17], are extremely 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.

In the next sections we will 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.

The rest of the paper discusses two different kinds of transformation adaptability, that can be desirable in different situations (§2), explains the negotiated grammar transformation in detail (§3) and provides some concrete examples taken from a prototype implementation (§4). For the figures on the pages of this paper, we will use MegaL/yEd, a domain-specific (mega)modelling language for specifying and discussion linguistic architecture [5]. The entity types will be distinguished by the colour and an associated icon: an **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; a **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. a **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. The types of relationships occurring on the figures, will be either classic or intuitive, and hopefully will not need any explanation (e.g., inputOf, hasOutput, elementOf, realisationOf). MegaL models are *megamodels* [1, 6] — models of linguistic architecture that specifically address relationships between complex entities such as software languages and (meta)model transformations in order to comprehend software technologies and relate technological spaces [13]

2. TRANSFORMATION ADAPTABILITY

We can think of two kinds of adaptability that we may desire in metamodel transformation: through tolerance and through adjustment.

2.1 Adaptation through tolerance

Figure 1 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, will yield Grammar2. If we assume that some evolution (which is also technically a transformation) happens with the original grammar (the group on the left), then 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 Xformation12 is f and XformationE is e, then XformationT is $t = e^{-1} \circ f$.

In conventional unidirectional programmable grammar transformation, most destructive operators inherently exhibit this property (with Script1 being equal to Script2T): 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.

2.2 Adaptation through adjustment

Figure 2 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 in the sense of $f \circ e' = e \circ a$, where a is XformationA.

Adaptation through adjustment is common in many scenarios when the changes brought in by the original transformation and by metamodel evolution, are independent (i.e., in terms of grammarware, they concern different nonterminals). Apparently, in case of complete independence their superposition is commutative, so $e' = e$, 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, needs adjustment.

3. 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.

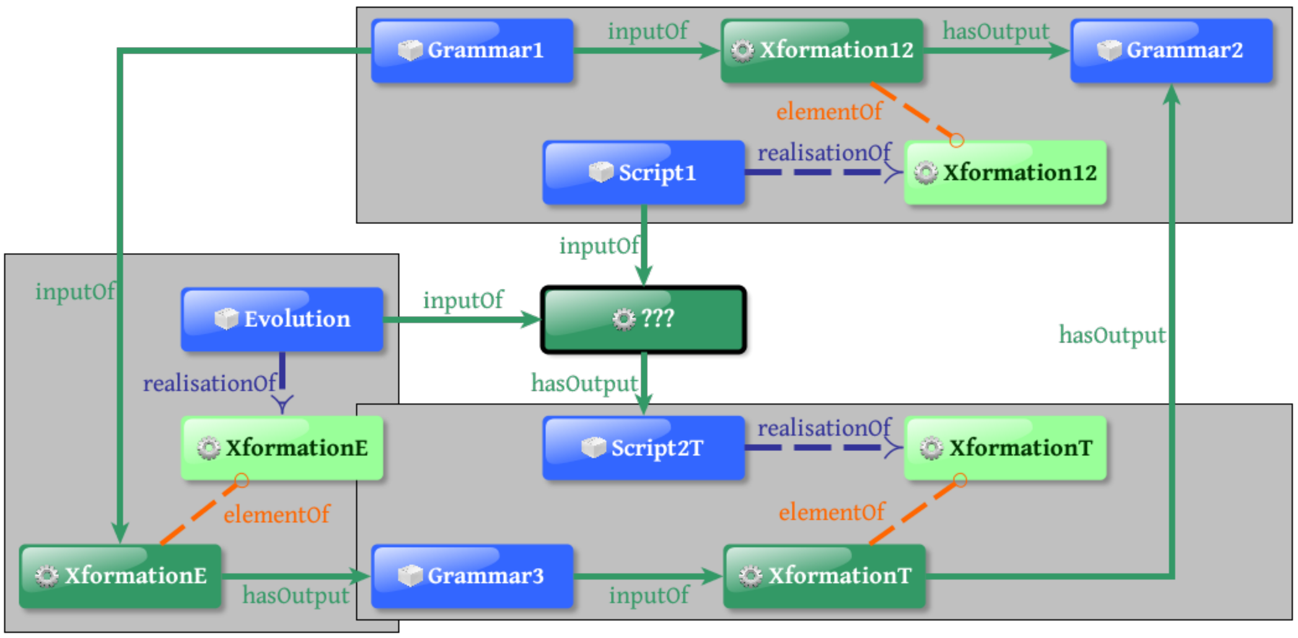

Figure 1: Adaptation through tolerance.

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 [24]. Isolating this aspect to a separate component that systematically implements the desired level of adaptability, allows us to encode any desired behaviour between those two known approaches and beyond them.

4. EXAMPLES

4.1 renameN

Consider one of the easiest to understand grammar transformation operator: renameN, which renames a nonterminal. Its implementations are available in the GitHub repository of the Software Language Processing Suite [25] in Prolog[1] and Rascal[2], and conceptually **renameN**(x, y) follows

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

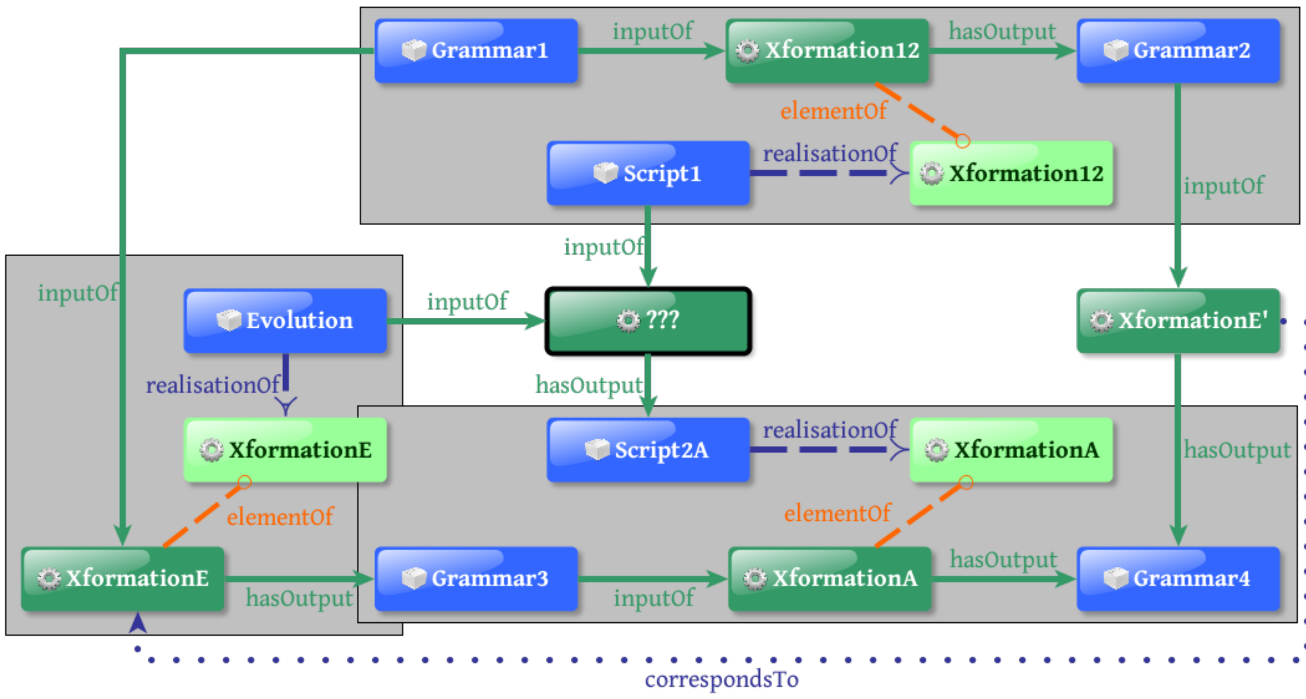

Figure 2: Adaptation through adjustment.

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.

For the remaking of this transformation operator in the negotiated grammar transformation paradigm, we isolate the steps (3.) though (5.) as the core transformation code and reformulate the first two constraints as recommenders:

1. 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 with the lowest score.

2. If the target name y for renaming is not fresh, we make three proposals: one of the form of $y1$, $y2$, etc (whatever is the lowest number that is not taken yet); one is obtained by concatenating underscores to y; and one randomly generated with the same length and letter cases as y (i.e., “AbcDef” can lead to “FooBar”).

Some design decisions explained here are mere implementation details, but they are still included for the sake of providing examples of how suggestions can be formed in a concrete scenario for negotiated grammar transformations.

4.2 vertical

The **vertical** operator consumes production rules of one nonterminal, that contain a top-level choice, and replaces them with an equivalent definition consisting of multiple production rules [25, XBGF Manual]. (The latter style of engineering grammars is called “non-flat” [17] or “vertical” [18], 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.

These two applicability preconditions are easily realised within the negotiated grammar transformation paradigm. The search for a different nonterminal name is not unlike the search for x in the previous example, but it also filters out flat/horizontal nonterminals. The vacuousness, however, does not pose any additional challenge at all: the intended semantics of the operator is to ensure that a particular nonterminal is defined vertically — and if the transformation command is vacuous, then it is already the case, so the postcondition is satisfied. Hence, a negotiated version of the **vertical** operator disregards the assertion of non-vacuousness.

4.3 extract

Extracting a production rule means adding a new production rule of a fresh nonterminal to the grammar, and folding it — i.e., replacing all occurrences of its right hand side with the newly introduced nonterminal [25, XBGF Manual]. Its implementation can be found in the same place we referenced above, but conceptually **extract**(n : *rhs*) works as follows:

1. Left hand side n is expected to be fresh (i.e., it must not be present in the input grammar before renaming).
2. The transformation is expected to be useful (i.e., *rhs* should occur at least once in the input grammar before adding the production rule).
3. All occurrences of *rhs* are replaced with n.
4. The production rule (n : *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 will 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 implement a fuzzy search algorithm that tries to identify fragments in the input grammar that could possibly be modifications of the right hand side that was provided as an argument.

5. RELATED WORK

Herrmannsdörfer et al [8, 10], Wachsmuth [22], Cicchetti et al [2] 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 transformation scripts are traditionally written in a more adaptable way, so they suffer less from the problem we are solving here.

In §4.3, 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 [19] is not the perfect metric in this scenario: ultimately, we would want one that puts “expr” closer to “expression” than to “abcd” [26]. To the best of our knowledge, such metric does not exist yet, since the need for it has apparently not previously risen. 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 [23] or parser-based matching [7], could be applied here as well.

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 [3] and incorporated directly to the resulting model [11], which enables efficient handling of inconsistency detection and resolutions as graph transformation rules [21]. These approaches can be used together with negotiated grammar transformation or as an alternative to it.

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 (grammars). However, the simplicity of the metametamodel (EBNF 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.

6. CONCLUSION

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

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 more efficient manipulation of existing grammar transformation scripts and their controlled adaptability.

Fragments of a prototype were shown and discussed in the paper as well, all of them available publicly in the GitHub repository of the Software Language Processing Suite [25]. Reimplementing all 50+ operators of XBGF within the negotiated grammar transformation paradigm, is still work in progress.

7. REFERENCES

[1] J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.

[2] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[3] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, Oct. 2007. TOOLS EUROPE 2007 — Objects, Models, Components, Patterns.

[4] T. R. Dean, J. R. Cordy, A. J. Malton, and K. 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.

[5] J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proceedings of MoDELS 2012*, LNCS. Springer, 2012. 17 pages. To appear.

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

[7] B. Fischer, R. Lämmel, and V. Zaytsev. Comparison of Context-free Grammars Based on Parsing Generated Test Data. In U. Aßmann and A. 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.

[8] M. Herrmannsdörfer, S. Benz, and E. 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.

[9] M. Herrmannsdörfer, D. Ratiu, and M. Kögel. Metamodel Usage Analysis for Identifying Metamodel Improvements. In B. A. Malloy, S. Staab, and M. 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.

[10] M. Herrmannsdörfer, S. Vermolen, and G. Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In B. A. Malloy, S. Staab, and M. 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.

[11] M. Kögel, H. Naughton, J. Helming, and M. 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.

[12] J. Kort, R. Lämmel, and C. 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.

[13] I. Kurtev, J. Bézivin, and M. Akşit. Technological Spaces: an Initial Appraisal. In *Proceedings of CoopIS, DOA'2002, Industrial track*, 2002.

[14] R. 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.

[15] R. Lämmel. Coupled Software Transformations. In *First International Workshop on Software Evolution Transformations (SET'04)*, Nov. 2004.

[16] R. 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).

[17] R. Lämmel and G. 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.

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

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

[20] W. Lohmann and G. 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.

[21] T. Mens, R. Van Der Straeten, and M. DâĂŹHondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer Berlin / Heidelberg, 2006.

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

[23] V. Zaytsev. Guided Grammar Convergence. Full Case Study Report. Generated by converge::Guided. *Computing Research Repository (CoRR)*, abs/1207.6541:1–44, July 2012.

[24] V. Zaytsev. Language Evolution, Metasyntactically. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST)*, 49, 2012. Post-proceedings of the First International Workshop on Bidirectional Transformation (BX'12).

[25] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, and G. Wachsmuth. Software Language Processing Suite[3], 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.

[26] V. 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.

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