Pending Evolution of Grammars

Vadim Zaytsev

Software Analysis & Transformation Team,
Centrum Wiskunde & Informatica,
Amsterdam, The Netherlands

Abstract. In this paper, we propose an alternative paradigm of software evolution, with a case study on evolution of grammars. While the usual transformational evolution model involves an immediate application of the evolution request, the pending evolution paradigm allows to postpone the actual changes until certain conditions are fulfilled. This kind of extreme evolution can be perceived either as a transformation computation model or as a actionable precondition model.

1 Introduction

A colloquial expression *'consider it done'* means that the subject of the conversation is either indeed already done, or will be done in the very near future — in either case, the receiver of such a message can rest assured that the subject will take place if it has not already, and is expected to act as if it has indeed happened. The technique of *pending evolution* that we introduce in this paper, is similar to that expression, and the benefits of it are not unlike the subtle differences between considering something done and it having been done.

As it turns out, the pending evolution scheme allows us to efficiently model scenarios of grammar evolution, deployment and maintenance that are impossible to model within the traditional grammar transformation paradigm, which is briefly explained in §2. The method is introduced in §3. Since the most profits hide deep in the details, we spend the rest of the paper (§4) by motivating the use of pending evolution for grammars instead of classical evolution scripts, by concrete examples. §5 concludes the paper by summarizing its contributions and discussing related work.

2 GrammarLab

GrammarLab is a codename for a grammar manipulation project that is currently being migrated from the Software Language Processing Suite[1] initiative to its own repository[2]. It is centered around the concept of a grammar in a broad sense [5], which can be *extracted* by abstracting away the idiosyncratic

[1] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, G. Wachsmuth. Software Language Processing Suite, 2008–2013. http://slps.github.io.

[2] V. Zaytsev. GrammarLab, 2013. http://grammarware.github.io/grammarlab.

details that we see on class diagrams, in algebraic data type definitions, object grammars, concrete syntax specifications, database schemata and exposed library interfaces — all these are 'grammars in a broad sense', since they model *commitment to grammatical structure*.

Beside extraction, GrammarLab is good in dealing with programmable grammar *transformations* — a disciplined method of grammar evolution, where every change is expressed as a call to a transformation operator with a well-defined semantics; as well as grammar *mutations* — large scale strategies for changing one simple thing in a priori unknown number of places. GrammarLab also includes library for grammar *analysis* and *metrics*, but they are less relevant for this paper.

3 Pending evolution

In GrammarLab, the evolution of a grammar is specified by a sequence of steps, each referring to a transformation operator or a mutation, with proper parameterization: e.g., first we **rename** a nonterminal, then we **factor** its definition and then **extract** a part of it into a new nonterminal. Each of the possible transformation operators and mutations in the library, have preconditions that determine their applicability and postconditions that demonstrate their successful execution. Whenever a postcondition of a step or a precondition of the next step fails, the transformation sequence is interrupted and an error occurs instead.

When a negotiated transformation paradigm [13] is explored, failure of a pre- or a postcondition means the start of a negotiation: e.g., if a **rename** fails, it can propose alternative name pairs that would enable its execution. A clever strategy for negotiations can drastically increase applicability and reuse of a transformation script, while still allow for full automation.

The *pending evolution* paradigm that we propose here, can hold any transformation step pending until its precondition becomes enabled. As will become apparent from the following examples in §4, it is possible to push the pending steps all the way to the end of the evolution sequence and then disregard them; or leave them forever pending and always ready to be applied any number of times necessary; or relax the constraints about the order of steps; or collect and log the information about all the possibly non-sequential failures in a system; or even intentionally decide that particular steps must be taken by defer their actual execution until later. Only the simplest local cases can be expressed in terms of negotiations. On the other hand, only simplest negotiations can be expressed as pending changes. In short, negotiated transformations enables flexibility with the *outcome* of one step, while pending evolution enables flexibility with the *order* of multiple steps.

4 Scenarios

The paradigm of pending evolution probably has much wider applicability, but here we sketch at least four user stories for it, inspired by the problems in the grammarware technological space that can be addressed and solved there.

4.1 Optional execution

In the classic grammar transformation engine of GrammarLab, any grammar transformation step that changes nothing in the grammar (we call them 'vacuous transformations'), is considered erroneous, since in most reasonable contexts — correction, adaptation, evolution, etc — a *change* that *changes* nothing, is meaningless. However, with some negotiated transformation schemes [13], one could find it sensible to ignore the fact that a transformation step brought no actual changes, if considered in a broader context. In particular, consider the following scenarios:

– Suppose we have a repository of grammars, such as Grammar Zoo[3] or Grammar Tank[4] [15]. The repository is highly heterogeneous and contains 'grammars in a broad sense' extracted from parser specifications, compiler sources, readable documentation, privately created webpages, community contributed wikis, generated and manually built artifacts. However, one of the steps known from grammar research [8] to increase the quality of a grammar, is resolving all 'bottom' nonterminals — the ones that are used within the grammar but never defined (a grammar with no bottom nonterminals is called a 'level 3 grammar' by Lämmel and Verhoef in [8]). While some definitions are simply missing from the grammar due to development mistakes, quite commonly these are lexical, or character-level, definitions, containing the rules about how an identifier name or a numeric literal should look in a language being defined. A big fraction of these, as becomes apparent after mining Grammar Zoo, have meaningful names such as 'string', 'identifier', 'integer', 'id', etc, and can be matched to a small library of predefined production rules such as 'a string is a symmetrically quoted sequence of one or more characters' or 'an integer value is an optional sign followed by one or more digits, the first of which is not zero'. This can be automated and ran over the whole repository, which can of such substantial size that prevents its manual verification[5] — however, it would be desirable for the framework to introduce the missing definitions only if they are truly missing, and allow individual grammars to retain their specific views of what a string or a boolean looks like. Hence, we allow the **introduce** operator[6] to be left pending, and disregard it at the end of the transformation application.

[3] Grammar Zoo, http://slps.github.io/zoo

[4] Grammar Tank, http://slps.github.io/tank

[5] Grammar Zoo and Grammar Tank together contain 564 grammars at the day of paper submission.

[6] **Introduce** and other grammar transformation operators are documented at http://github.com/grammarware/slps/wiki/introduce and similar URIs.

– Consider another scenario. In grammarware technological space, there are two most common styles of production rules, that we will traditionally call *horizontal* and *vertical*. A horizontal definition says that the nonterminal N is ‘either X or Y or Z’, while the vertical one makes three statements that ‘N is X’, as well as ‘N is Y’, as well as ‘N is Z’. These can be formally proven to be equivalent. There are many exceptions, but most language documents prefer horizontal definitions (e.g., Java Language Specification [3]), while language workbenches tend toward vertical ones (e.g., The Meta-Environment [4]) or make no distinction between them (e.g., Rascal [6]). Some transformation operators also expect their arguments to be either horizontal or vertical, which leads to the evolution scenarios specified in such a way where some of the operator calls are preceded by the calls of **horizontal** or **vertical** operators, while others are not. Obviously, this excessive versatility hinders maintainability and changeability of the transformations. It would be better to write these transformation steps as assertions. For instance, we can specify that the definition must by vertical before we **deyaccify** it, and this step would be optional, requiring no action if the original definition is already vertical.

4.2 Error handling

In GrammarLab, transformations are stopped whenever the error occurs, and an error message is displayed. Within the negotiated paradigm [13], it is possible to negotiate for another outcome. One of the rather complex strategies for achieving that, is the one that skips over the failing transformation step and proceeds with the rest of the script, and then displays all the error messages at the end of the computation. To demonstrate the usefulness of this approach, consider the following detailed scenarios.

– In the context of grammar recovery, suppose that we want to extract several grammars in bulk — they are written in the same style, in the same metalanguage, with some a priori unknown differences between them (perhaps they are different versions or dialects of the same language). After carefully considering one of them, a grammar engineer develops a post-extraction transformation script that makes the grammar maximally connected, adds missing definitions, fixes misspelt nonterminal names and corrects other problems. Naturally, we want to reuse the same transformation script for recovering the rest of these grammars. However, in the traditional setup, most of the automated reuse cases will fail because some of the extracted grammars will have some misspellings already fixed, others will lack the part that concerns the fixes, etc. Advanced error handling (or ignoring) can help greatly with scalability in this case, by skipping over inapplicable fixes, applicability classification, etc.
– Imagine another scenario concerning maintenance of grammar transformation scripts. Suppose that we have several grammars that are being converged

together in multiple steps — e.g., the case study converging six Java grammars found in different editions of the Java Language Specification book, consisted of 1611 transformation steps arranged in 70 different scripts [10]. When an error is spotted in one of the existing steps, or when another step needs to be added in the middle of the transformation chain, or when the order of existing steps needs to be adjusted, it becomes a very labor-intensive task since every failure stops the transformation computation — having the luxury of recovering after a failure noticeably increases debugging capabilities.

4.3 Pending fixes

Both the traditional programmed and the negotiated transformation models delegate the decision about the transformation order to the original script: any transformation step takes place after the one that precedes it in the specification and is followed by the one after it. However, there are situations when we can develop certain transformation scenarios and leave them pending so that they can be executed when (if) the times comes and the become applicable. In general this is useful in case of preserving any kind of normal form properties, but we provide two detailed cases taken from practice:

– Recall the difference between horizontal and vertical definitions that we have explained in the previous section. Suppose that our grammar uses vertical definitions exclusively — this is easy to achieve by grammar transformations or mutations, and easy to validate with a metaprogramming formula or by micropatterns. However, if we would like to specify that the dominance of vertical definitions is not incidental and that we would like to preserve it, it is not possible to express this constraint within the straightforward grammar programming approach. With pending evolution, we could leave the verticalizing mutation pending. Then, if someone introduces a new nonterminal to the grammar, and that nonterminal is found to be horizontal, the mutation becomes enabled, is executed and recharged for next use.
– Grammar recovery is a process of extracting a grammar from an existing software artifact that may not be of perfect quality. Automated grammar recovery methodology [14] is based on a collection of heuristics that are partly configurable and partly inferred from the notation specification. One of such heuristics is splitting composite terminals: for instance, if a terminal like ‘`);`’ is found, it can be broken into two consecutive terminals: ‘`)`’ and ‘`;`’ — simply because the resulting atomic terminals are more helpful for other heuristics (like matching parentheses). A grammar mutation that breaks up composite terminals, can be programmed and left pending, such that under any circumstances that would bring such terminals to the grammar (such as importing another grammar, introducing a new nonterminal definition, folding/unfolding, projecting symbols, etc), it becomes enabled and is immediately fired to split such terminals as desired.

These scenarios are sufficiently different from the ones in the previous section not only in motivation, but also in realization, since we speak of pending mutations (which are large scale transformations) and recharging them after transformation.

4.4 Intentional pending

Since we have discussed preserving the grammar already being in the normal form, another scenario deserves mentioning where the grammar is normalized — or rather, when such a normalizing mutation is left pending. Below there are two use cases for this situation, but any normalization could possibly spawn another one.

- Suppose that we have a grammar written in a specific notation (usually a dialect of EBNF). Suppose also that a notation evolves, and the grammar is required to coevolve in order to preserve conformance to the metalanguage. This scenario is called 'metalinguistic evolution' [12] and has been studied sufficiently to be applied in an automated fashion. One of such applications involves a grammar being exported to a particular notation, which it might not perfectly fit. For instance, the grammar may use an explicit repetition (usually denoted with '`*`') or other metaconstructs which are lacking from the notation. Another typical case is that the target metalanguage insists on a particular naming convention for the nonterminal (e.g., all must be written in capitals). In that case, the grammar needs to coevolve with the 'change' of notation from its original one to the one that it is being exported to. However, this coevolution is essentially a part of the exporting process, and as such must always take place after all the other evolution steps. Hence, it can be left pending until the very end of the transformation script, and be executed last, removing the use of excessive metalanguage elements, changing the naming convention and adjusting grammar before the actual export mapping.
- Grammars in a broad sense can be observed in very different environments and extracted from artifacts hailing from different technological spaces: XML schemata, Ecore models, class diagrams, parser specifications, data types, etc. Even when these define one intended language, they are different in many ways. A technique called grammar convergence [9] is used to reverse engineer the real relationships between such grammars: based on expert-written transformation scripts, it can show which grammars define the same language, which define languages that are subsets or supersets of one another, and which are incomparable. It is also possible to automate the creation of such scripts, but the inference algorithm performs best when grammars are in so called 'abstract normal form'. Many constraints of the abstract normal form contradict the practice of grammar engineering, so it would be most desirable to continue working with the non-normalized grammar and then perform the pending normalization right before the guided grammar convergence algorithm is applied. Then, the obtained result can be traced back to the original grammar by reversing the bidirectional transformation chain produced by the normalizer.

5 Concluding remarks

There are some techniques similar to pending evolution in the inconsistency management, most notably with concurrent transformation schemes. Such inconsistencies can be represented as separate first-class entities [2] and incorporated directly to the resulting model [7], which enables efficient handling of inconsistency detection and resolutions as graph transformation rules [11] in a much less extreme way than the one proposed in this paper. The fact that these approaches of inconsistency modeling and resolution are not entirely covered by negotiated grammar transformation, has inspired us to look for common schemes of advanced change impact propagation, importing ideas from modelware to grammarware and adapting them to the domain.

To summarize, we have proposed the following use cases for the technique of pending grammar evolution:

- optional execution (§4.1)
    - optionally complementing the grammar with missing definitions
    - using optional transformations as assertions
- error handling (§4.2)
    - reusing transformations for bulk extraction
    - debugging transformations
- pending fixes (§4.3)
    - persistent commitment to a normal form
    - pending recovery heuristics
- intentional pending (§4.4)
    - pre-export processing
    - pre-convergence normalization

Pending evolution for grammars (either in a broad sense [5] or in the classic sense [1]) has never been considered before. Investigating the impact and opportunities for pending evolution schemes in other fields like program transformation remains future work. In transaction handling domains both of great strictness (such as database management and mainframe job processing) and persistent inconsistency (such as managing wiki contents with a bot) one will be able to find techniques somewhat similar to the one we have proposed here.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. A. Cicchetti, D. Di 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.
3. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.
4. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF—Reference Manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

5. P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 14(3):331–380, 2005.
6. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009)*, volume 6491 of *LNCS*, pages 222–289, Berlin, Heidelberg, Jan. 2011. Springer-Verlag.
7. 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.
8. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, Dec. 2001.
9. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In M. Leuschel and H. Wehrheim, editors, *Proceedings of the Seventh International Conference on Integrated Formal Methods (iFM 2009)*, volume 5423 of *LNCS*, pages 246–260, Berlin, Heidelberg, Feb. 2009. Springer-Verlag.
10. R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ)*, 19(2):333–378, Mar. 2011.
11. T. Mens, R. Van Der Straeten, and M. D'Hondt. 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 (MoDELS'06)*, volume 4199 of *LNCS*, pages 200–214. Springer, 2006.
12. V. Zaytsev. Language Evolution, Metasyntactically. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST)*, 49, 2012.
13. V. Zaytsev. Negotiated Grammar Transformation. In J. De Lara, D. Di Ruscio, and A. Pierantonio, editors, *Post-proceedings of the Extreme Modeling Workshop (XM 2012)*. ACM Digital Library, Nov. 2012. In print, currently available at http://www.di.univaq.it/diruscio/sites/XM2012/xm2012_submission_11.pdf. An extended version is currently under major revision to the Special issue on Extreme Modeling of The Journal of Object Technology (JOT).
14. V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012)*. ACM Digital Library, June 2012.
15. V. Zaytsev. Grammar Zoo: A Repository of Experimental Grammarware. Under major revision for the Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5), 2013.