Understanding Metalanguage Integration by Renarrating a Technical Space Megamodel

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. There are many software languages which are not exposed protocols, exchange formats, interfaces and storage formats, and are only used for intermediate representation, runtime data manipulation and tool-specific serialisation. Yet, they can be important for technology comprehension, since such internal implementation details may have indirect impact on some aspects of the externally observed behaviour of the system. In this paper, we show a concrete example of how various tools and their technological differences can be explained based on one abstract megamodel and its different renarrations.

1 Introduction

Megamodels are used for modelling complex systems involving many artefacts, each of which is also in turn a model or a transformation [3,4]. For instance, they can help represent an entire technological space or a technical space in order to expose its components and to explain them to previously unaware audience (such as students) [6,10]. The main focus of megamodelling is usually on externally observable (meta)languages: communication protocols, data interchange formats, application programming interfaces, algebraic data types, public library interfaces, serialisation formats, etc. Yet there are a lot of (meta)language used behind the scenes for internal presentation of data structures — and we all know very well how much of an impact can a different data structure have on performance of an algorithmically nontrivial application. As it turns out, megamodelling can be very helpful here as well.

Megamodels (also called linguistic architecture models [6,10], macromodels [15], technology models, etc) come in a great variety of forms and approaches and are theoretically useful for solving many problems of different stakeholders. However, one of the main showstoppers is their overwhelming complexity: not only a typical megamodel requires considerable investment in deep domain analysis, exploratory experimentation, modelling and metamodelling; but also the result thereof is a towering monolith easily intimidating any possible users. At the same time, simplification is possible yet often undesirable, for the devil lurks in the details. One of the existing solutions is investing in packaging the megamodel as well as in its development. We can slice the megamodel into digestible parts and navigate stakeholders through them, possibly through various itineraries depending on the priorities — this process is referred to as *megamodel renarration* [18].

A renarration of a megamodel is a story that traverses the elements of this megamodel in order to guide the users through it and to gradually introduce them to the model elements and thus to domain concepts. Formally, a renarration relies on operators for addition/removal, restriction/generalisation, zoom in/zoom out, instantiation/parametrisation, connection/disconnection and can make use of backtracking [11]. In prior work we have shown renarrations as annotated megamodel transformations, but have not used them in multiple scenarios based on one original model.

The approach we propose in this paper involves investing in a global megamodel of an entire technical space, and then using renarrations of it to demonstrate each existing technology. Thus, the contribution of the paper is mainly the focus on using one baseline white box megamodel for establishing a common ground for explaining various subtly different technologies of the same domain by renarrating it repeatedly.

Specifically in the context of the GEMOC initiative, megamodelling addresses the second focus (integration of heterogeneous model elements), while renarration treats the first issue (catering various stakeholder concerns). So far this material has been (in a more volatile form) used in teaching courses on software language engineering [2], software evolution, software construction and in supervision of Master students.

2 Parsing with many faces

For a demonstration of the proposed approach let us consider a megamodel for parsing in the broad sense that we presented in earlier work [21]. The model includes twelve kinds of artefacts commonly found in software language enginering (as well as commonly encountered mappings among them, see Figure 1):

- ◊ Str — a string, a file, a byte stream.
- ◊ Tok — a finite sequence of strings (called *tokens*) which, when concatenated, yields Str. Includes spaces, line breaks, comments, etc — collectively, *layout*.
- ◊ TTk — a finite sequence of typed tokens, possibly with layout removed, some classified as numbers, strings, etc.
- ◊ Lex — a lexical source model [19] that addes grouping to typing; in fact a possibly incomplete tree connecting most tokens together in one structure.
- ◊ For — a forest of parse trees, a parse graph or an ambiguous parse tree with sharing; a tree-like structure that models Str according to a syntactic definition.
- ◊ Ptr — an unambiguous parse tree where the leaves can be concatenated to form Str.
- ◊ Cst — a parse tree with concrete syntax information. Structurally similar to Ptr, but abstracted from layout and other minor details. Comments could still be a part of the Cst model, depending on the use case.
- ◊ Ast — a tree which contains only abstract syntax information.
- ◊ Pic — a picture, which can be an ad hoc model, a "natural model" or a rendering of a formal model.

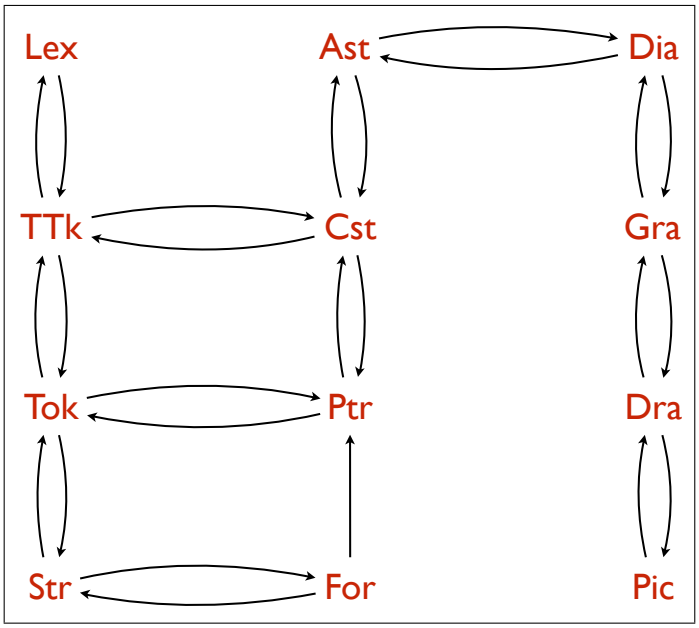
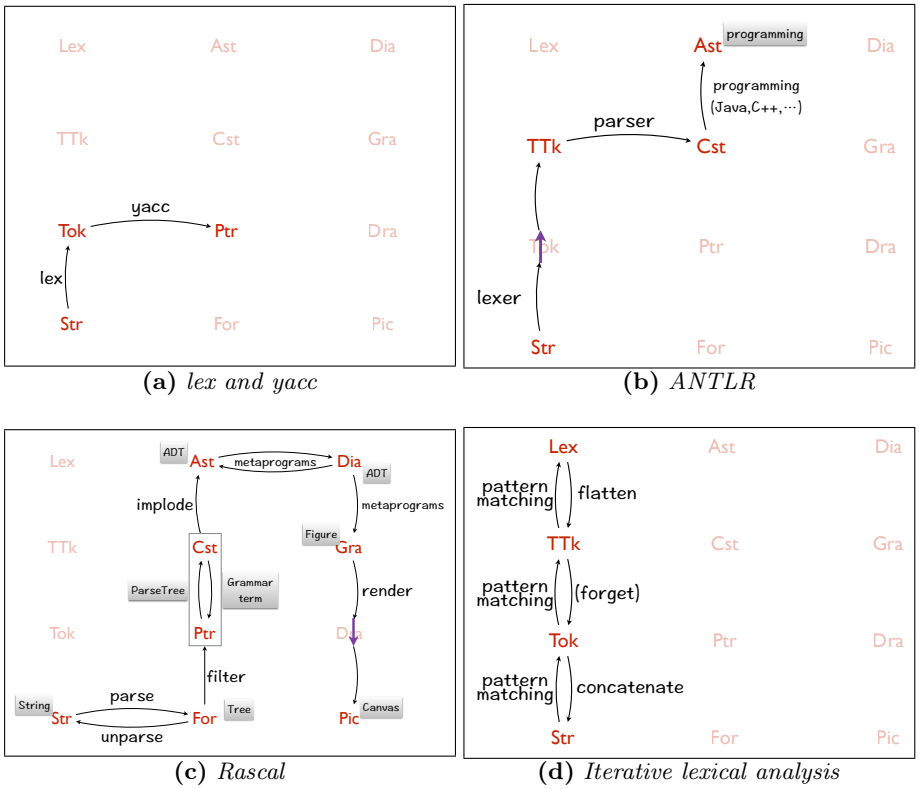


Fig. 1. A megamodel of parsing in a broad sense — see [21] for detailed definitions and descriptions of these kinds of software artefacts and mappings.

- ◊ Dra — a graphical representation of a model (not necessarily a tree), a drawing in the sense of GraphML or SVG, or a metamodel-indepenent syntax but metametamodel-specific syntax like OMG HUTN.
- ◊ Gra — an entity-relationship graph, a categorical diagram or any other primitive "boxes and arrows" level model.
- ◊ Dia — a diagram, a graphical model in the sense of EMF or UML, a model with an explicit advanced metamodel.

The megamodel from Figure 1 provides a unique uniform view on parsing, unparsing, formatting, pretty-printing, disambiguation, visualisation and related activities — it is a big step from heterogeneous discordant papers originating from relevant technical spaces toward general understanding of the field. Yet, as we have claimed before [18,11], a monolithic megamodel can play a role of a knowledge container, but cannot be used directly as the deployed artefact. (As a side remark, this corresponds to the claim by Bézivin et al that a megamodel as a model of models should not be used as a reference model [3]). Hence, we need a renarration of a megamodel to successfully deliver the knowledge behind it. A renarration can happen naturally (e.g., as a lecture for students) or be formally inferred with megamodel transformation operators for addition, connection, instantiation, etc [11].



(a) *lex and yacc* **(b)** *ANTLR*

(c) *Rascal* **(d)** *Iterative lexical analysis*

Fig. 2. Four illustrated renarrations of the (slices of the) megamodel from Figure 1

In this paper, we use English for the narrative, and the models themselves are available at ReMoDD: `http://www.cs.colostate.edu/remodd/v1/content/renarrating-metalanguage-integration`. In the following sections, we demonstrate several renarrations of the megamodel from Figure 1.

2.1 Parsing in a narrow sense: lex + yacc

One of the textbook approaches to parsing is using two tools to obtain a parse tree from the input string: one for lexical analysis and one for syntactic analysis. In many classic compiler construction courses lexical analysis is done with `lex` [12] or one of its successors. The tokens that are obtained by lexical analysis, are in fact typed, but the type information is not necessarily used for anything, so we can model the result of the lexical analysis with Tok. The next step is handled by a compiler compiler like `yacc` [7] or its more modern counterparts (but not too innovative — we want to stick to the classic DragonBook-like view [1]). This syntax analysis tool consumes Tok and produces a parse tree — Ptr. This can be seen on a rather simple Figure 2**(a)**.

2.2 Advanced parsing technology: ANTLR

Consider ANTLR [14], a state of the art compiler compiler that can be used for the same purpose as lex+yacc, but incorporates the results of several decades of research on parsing, compiler construction and interactive programming environments. Both a lexer and a parser are generated from the uniform syntactic definition (grammar). Lexical nonterminals, usually written in CAPSLOCK, define a grammar used for lexical analysis. Most of them are preterminals — their definitions contain only terminals, combined sequentially, with disjunction, Kleene closure and other combinators typical for regular expressions [8]. As shown on Figure 2**(b)**, the output of the lexer is TTk, a stream of strongly typed tokens — each token has to either belong to one of the lexical categories (be parsed as a lexical nonterminal) or match one of the terminal symbols used in the rest of the grammar — they are turned into preterminals automatically by ANTLR. The untyped version of the same representation (Tok) is not available directly: if needed, one could possibly either disregard the typing information (e.g., by using code duplicates inside semantic actions) or plug in into the internals of the generated lexer.

A typed token stream is processed by a parser which ANTLR generates from the input grammar. The result is Cst, a parse tree that abstracts from some details like layout and comments. It is important to note that ANTLR generates the definition of the Cst and provides means to traverse them. However, if one still desires to use an abstract syntax tree, both Ast itself and the mapping from Cst to Ast need to be programmed explicitly in the base language of ANTLR (Java, C++, C#, Python, etc). The mapping can be scattered among the nonterminal definitions directly in the grammar (as semantic actions), or it can be written as a separate program that traverses the ANTLR Cst with the ANTLR visitor and constructs a specific Ast. The class structure of the Ast itself always needs to be defined and processed independently from ANTLR.

2.3 Rascal metaprogramming language

Rascal [9] is another state of the art piece of grammarware — however, an important difference from ANTLR is that Rascal is advertised as a "one-stop-shop" for software analysis, transformation and visualisation. Let us try to understand this difference from Figure 2**(c)**.

Rascal uses generalised parsing (more specifically, GLL), which yields a parse forest instead of a parse tree, if the grammar is ambiguous. Such parse forests (For) are represented internally with the same structure — a term representation that is allowed to explicitly contain ambiguity node. Thus, in order to decide if a given tree is For or Ptr, we need to perform a deep match on an `amb(_,_)` pattern (since pattern matching is one of the basic constructions in Rascal, this operation is trivial to express, even though it might become a performance bottleneck).

By Rascal design, there is no observable distinction between Ptr and Cst. All trees are stored internally as Ptr, but all pattern matching behaves as if both the pattern and term is Cst (with the pattern allowed to be incomplete). Each

unambiguous tree conforms to the grammar (a syntax specification) that was used to parse it. A grammar is defined in Rascal within the same module or imported as a separate one. Relying on such grammatical structure can simplify pattern matching immensely: instead of checking for a term which is an application of a particular production rules with certain arguments, we write the same intent down with a term on the left hand side, typed to a particular nonterminal and thus fully conforming to its structure (modulo intended gaps to be skipped during unification).

A Cst can be mapped to Ast explicitly by writing a pattern-matching visitor, which is done in some cases that require sophisticated compulations as a part of the mapping. However, an easier way is to use an `implode()` library function that has a set of stable heuristic rules for finding bidirectional correspondence between a given syntax definition and a given algebraic data type. The ADT itself (the structure of Ast) must still be programmed manually, which is traditionally not considered to be a burden since one wants to have full control about the way abstract syntax is defined. (When this is not the case, it can be inferred from the grammar by grammar mutations [20] of GrammarLab, a Rascal library for manipulating grammars in a broad sense[1]). `implode()` is not shipped with a reverse function, so any derivation from Ast to Cst/Ptr, if needed, must be programmed manually.

High level abstract diagrams (Dia) are also modelled in Rascal by algebraic data types managed by the (meta)programmer. A universal yet still a high level visual model (Gra) is provided in the standard Rascal library and contains elements like boxes, grids, graphs, trees, plots. A `render()` function, however, positions all these elements automatically and only outputs the final picture (Pic) on screen or to a file, effectively skipping over Dra — for a Rascal programmer it means having no control over the exact positioning of most elements on canvas, except for general constraints which are a part of the metamodel of Gra.

2.4 Semiparsing: building lexical models with ILA

Semiparsing [19] is an umbrella term for techniques of imprecise manipulation of source code (its variations are known as agile modeling, robust parsing, lightweight processing, error repair, etc). They are inherently very different because usually come into existence for solving a very particular practical problem — we have claimed recently that Boolean grammars [13] and parsing schemata [16] can be helpful in modelling all possible variations of semiparsing. However, as useful as these two formalisations could be in deep understanding of the methods, relating them and positioning among themselves, they are not always as effective for their implementation-driven comprehension, especially by software engineering practitioners without background in formal methods.

Consider Figure 2**(d)**, which demonstrates a semiparsing technique called iterative lexical analysis [5] (a similar technique has recently emerged in a more modern framework called TEBA for analysis of tokenised syntax trees [17]). The

[1] GrammarLab: `http://grammarware.github.io/lab`.

technique relies mainly on patterns which are classified in levels: the higher the level, the more unstable and the less desirable the pattern is. So, on the first level there are strict matches for terminals such as keywords, and on the last level there are "desperate" heuristics that are meant more to ensure that the process produces some kind of result than to actually claim any correctness. Hence, we only work with the left column of our megamodel: the higher we are in the model, the more abstract and imprecise patterns are applied. There is no direct correspondence between pattern levels and layers of the megamodel, but for each concrete pattern we can easily find a place. For example, a pattern that detects strings and demotes the role of tokens inside a string from possible metasymbols (e.g., so that a curly bracket in `a = "b{";` is never used for block identification) clearly works on TTk, while a pattern that matches an identifier followed by a bracketed comma-separated list of identifiers followed by a block of statements and promotes it to a function definition, naturally produces Lex.

Operations for descending from Lex to TTk to Tok to Str are not explicitly described in the paper about iterative lexical analysis, but are certainly available in any sensible framework: we need to flatten (unfold) all hierarchical constructs to get down to TTk, disregard type information to get down to Tok and concatenate all tokens to get all the way to Str.

3 Conclusion

Megamodels are used as an understanding aid in complex scenarios involving various technologies, software languages, methodologies, approaches and transformations [3,4]. Renarrations of megamodels improve their usefulness by guiding megamodel consumers through the forest of immanently complicated artefacts and mappings [11,18]. Megamodels, whether ad hoc (a sentence "model M conforms to a metamodel MM" is in fact a tiny megamodel) or formal (AMMA, MEGAF, SPEM, MCAST, MegaL, CT), perform undeniably well for teaching purposes when introducing students to a new technology and explaining subtle differences between two almost identical technologies. In this paper, we have claimed that the same approach can be used for internal "languages", the ones that are hiding behind the scenes inside our tools. For this purpose, we propose to have one baseline megamodel of the domain — in a formal sense, it will include a lot of abstract entities, unbounded elements, constraints based on roles, etc — and use its refined renarrations for each of the concrete technologies that need to be explained and understood. We have demonstrated this approach with our megamodel for parsing in a broad sense [21], which we have used as a baseline model for four renarrations: classic lex+yacc parsing [1,7,12], ANTLR language workbench [14], Rascal one-stop-shop [9] and iterative semiparsing [5,17,19].

Beside the obvious future work claims such as promises of (mega)modelling different domains and perhaps even megamodelling relations among such domains, some other open questions remain. For example, some megamodels require explicit distinction between kinds of mappings they express (injective, bijective, monomorphic, isomorphic, asymmetric bidirectional, symmetric bidirec-

tional, etc), and such distinctions would also have to be properly specified and renarrated. In other cases, the modelling framework may already have a metamodel suitable for expressing typical renarrations, and the megamodel navigating arsenal would need to be adjusted with respect to the language it must be expressed in (instead of the opposite situation which we always assume).

As any other modelling method which introduces unification and heterogenuity, (mega)modelling different technologies with renarrations of the same baseline megamodel can help not only in explaining the actual state of the art, but also in spotting singularities. Anything irregular could be a signal of a bug, a not yet implemented feature or a comprehension mistake. Why is there a mapping from Cst to Ast in Rascal but no universtal mapping from Ast to Cst? Perhaps we should include one! Is there a good reason for Dra to not be accessible in Rascal? Having it explicitly as a (possibly optional) first class entity could allow us to do things we otherwise cannot! Would it help organising patterns for ILA/TEBA based not on their "desperation", but on the kind of artefacts they are actually dealing with (untyped tokens, typed tokens, grouped tokens)? Exploration of the extent of usefulness of such conclusions remains future work.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. A. H. Bagge, R. Lämmel, and V. Zaytsev. Reflections on Courses for Software Language Engineering. In *Tenth Educators Symposium (EduSymp 2014)*, 2014.
3. J. Bézivin, S. Gérard, P.-A. Muller, and L. Rioux. MDA components: Challenges and Opportunities. In *Proceedings of the First International Workshop on Metamodelling for MDA*, pages 23–41, Nov. 2003.
4. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.
5. A. Cox and C. Clarke. Syntactic Approximation Using Iterative Lexical Analysis. In *Proceedings of the International Workshop on Program Comprehension (IWPC 2003)*, pages 154–163, 2003.
6. J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *MoDELS*, pages 151–167, 2012.
7. S. C. Johnson. YACC—Yet Another Compiler Compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, 1975.
8. S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, pages 3–42, 1956.
9. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *GTTSE 2009*, volume 6491 of *LNCS*, pages 222–289. Springer, Jan. 2011.
10. R. Lämmel and A. Varanovich. Interpretation of Linguistic Architecture. In *ECMFA*, pages 67–82, 2014.
11. R. Lämmel and V. Zaytsev. Language Support for Megamodel Renarration. In *XM 2013*, volume 1089 of *CEUR*, pages 36–45. CEUR-WS.org, Oct. 2013.
12. M. E. Lesk. LEX—A Lexical Analyzer Generator. Computer Science Technical Report 39, AT&T Bell Laboratories, 1975.
13. A. Okhotin. Conjunctive and Boolean Grammars: The True General Case of the Context-Free Grammars. *Computer Science Review*, 9:27–59, 2013.

14. T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.
15. R. Salay, J. Mylopoulos, and S. Easterbrook. Using Macromodels to Manage Collections of Related Models. In *CAiSE*, pages 141–155. Springer, 2009.
16. K. Sikkel. *Parsing Schemata — a Framework for Specification and Analysis of Parsing Algorithms*. Springer, 1997.
17. A. Yoshida and Y. Hachisu. A Pattern Search Method for Unpreprocessed C Programs based on Tokenized Syntax Trees. In *SCAM*, 2014.
18. V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. In *MPM 2012*, pages 61–66. ACM, Nov. 2012.
19. V. Zaytsev. Formal Foundations for Semi-parsing. In *CSMR-WCRE 2014 ERA*, pages 313–317. IEEE, Feb. 2014.
20. V. Zaytsev. Software Language Engineering by Intentional Rewriting. *EC-EASST; Software Quality and Maintainability*, 65, Mar. 2014.
21. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *MoDELS*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.