Languages, Models and Megamodels
A Tutorial

Anya Helene Bagge[1] and Vadim Zaytsev[2]

[1] BLDL, University of Bergen, Norway, anya@ii.uib.no
[2] University of Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. We all use software modelling in some sense, often without using this term. We also tend to use increasingly sophisticated software languages to express our design and implementation intentions towards the machine and towards our peers. We also occasionally engage in metamodelling as a process of shaping the language of interest, and in megamodelling as an activity of positioning models of various kinds with respect to one another.
This paper is an attempt to provide an gentle introduction to modelling the linguistic side of software evolution; some advanced users of modelware will find most of it rather pedestrian. Here we provide a summary of the interactive tutorial, explain the basic terminology and provide enough references to get one started as a software linguist and/or a megamodeller.

1 Introduction

This paper is intended to serve as very introductory material into models, languages and their part in software evolution — in short, it has the same role as the tutorial itself. However, the tutorial was interactive, yet the paper is not: readers familiar with certain subtopics would have to go faster through certain sections or skip them over.

In §2, we talk about languages in general and languages in software engineering. In §3, we move towards models as simplifications of software systems. The subsections of §4 slowly explain megamodelling and different flavours of it. The tutorial paper is concluded by §5.

2 Software Linguistics

Let us start by examining what a *language* is in a software context.

In Wikipedia, the concept is described[3] as follows:

> *Language* is the human ability to acquire and use complex systems of communication, and *a language* is any specific example of such a system. The scientific study of language is called *linguistics*.

[3] http://en.wikipedia.org/wiki/Language.

Even leaving aside the anthropocentricity of this definition, we see that languages are communication systems — spoken, written, symbol, diagrammatic. As communication systems, languages have several properties, including *structure*, *meaning* and *abstraction*.

Structure, often also referred to as *syntax* [7], is about how sentences (programs) of a language are constructed or deconstructed, and in general what components of sentences (programs) can be identified and how the language allows us to put them together. In natural and software languages, the structure is often recursive, allowing to create an infinite number of statements of arbitrary complexity.

Meaning, also called *semantics* [12], assigns sense and value to language constructs — for the sake of simplicity, we mostly assume they are syntactically correct before being concerned with their meaning; in some rare cases like automated error correction we could also contemplate the meaning of incorrect programs. There is usually a tight interplay between structure and meaning, so that by changing the structure of a sentence, you change its meaning — the activity commonly referred to as "programming".

Abstraction is what allows discussion of arbitrary ideas and concepts, that may be displaced in time and space. Abstractions allow engineers to reason about physical systems by focusing on relevant details and ignoring extraneous ones [5]. Of course, the most interesting results are the ones that could not be obtained from the real system — so, *predictions* are preferred to *measurements* [20]. A crucial feature of natural languages as well as many software languages is the ability to define and refine abstractions — for instance, in the way this introduction defines English language abstractions for discussing software languages.

Early written communication (cave paintings) had symbols, but their meaning (if any) remains unknown. Early writing systems used pictures with grammatical structure. Such picture is, in fact, a *model* of a concrete object: a picture of a bull can confer the idea of doing something with it, but cannot feed you; one does not simply smoke an image of a pipe. Once used for abstraction, the symbols can be composed in nontrivial ways. For instance, in hieroglyphics, the word "Pharaoh" is written as a combination of a duck and a circle, because the Pharaoh is the son of Ra, since "son" is pronounced similarly to "duck" and Ra is a god of sun, which is modelled by a circle for its shape [14]. In a similarly nontrivial way, "a butcher's" means "a look" in Cockney rhyming slang, since "look" rhymes with "a butcher's hook" and "butcher's" is a shortened version thereof [19]. Such combinations and combining ways are the main reason new software languages are difficult to learn, if they are paradigmatically far from the already familiar languages: the idioms of C are too different from the idioms of English; and the idioms of Haskell are too different from the idioms of C.

Languages in software engineering as used in multiple ways. There are *natural languages*, which are reused and extended (by jargon) by developers. There are also *formal languages* which are also largely reused after their underlying theories being proposed and developed in fundamental research — essentially they are the same as natural languages, but much easier for automated processing and reasoning. Finally, there are also *artificial languages* which are specifically made by humans — such as C or Esperanto. Usually all kinds of automation-enabling languages that are used in construction and maintenance of software, are referred to as *software languages*: these are programming languages, markup notations, application programming interfaces, modelling languages, query languages, but also ontologies, visual notations with known semantics, convention-bound subsets of natural languages, etc.

For instance, any API (application programming interface) is a software language [1], because it clearly possesses linguistic properties such as:

◇ API has structure (described in the documentation)
◇ API has meaning (defined by implementation)
◇ API has abstractions (contained in its architecture)

However, API does not typically allow definition of new abstractions. For classical programming languages, we would have a similar list, but in domain-specific languages we would have abstractions limited by a particular domain, not by the system design (which means possibly infinite number of them, even if the abstraction mechanism is still lacking), while general purpose programming languages usually leave it to the programmer to define arbitrary abstractions (though not necessarily abstract over arbitrary parts of the language).

3 Moving to models

A *model* is a simplification of a system built with an intended goal in mind: a list of names is a model of a party useful for planning sitting arrangements; “CamelCase” is a model of naming that compresses multiple words into one. Any model should be able to answer some questions in place of the actual system [2]. Models are abstractions that can provide information about the consequences of choosing a specific solution before investing into implementation of the actual software system [29].

◇ Typically, a model represents a system.
◇ Some models represent real systems (programs, configurations, interfaces)
◇ Some models represent abstract systems (languages, technologies, mappings)
◇ Some models are descriptive/illustrative (used for comprehension)
◇ Some models are prescriptive/normative (used for conformance)

A model may be written (communicated) as a diagram or a text or some other representation — possibly even as a piece of software that allows to simulate behaviour. One might draw a model as an ad hoc illustration — similar to a crude cave painting — but for clarity and ease of communication across time and space, one may want to use a modelling language such as UML, BNF, XSD, CMOF, Z, ASN.1, etc.

Systematically discussing, researching and dissecting software languages has inevitably led to a special kind of models — called *metamodels* — that define software languages. For example, a grammar [36] as a definition of a programming language, is a metamodel, and programs written in such a language, are behavioural models conforming to that metamodel. Similarly, a database schema, an protocol description or an algebraic data type definition are examples of metamodels, since they all encapsulate knowledge about allowable (grammatical) structures of a software language, each in their corresponding technical spaces.

Formally speaking, a metamodel models a modelling language [25], in which models are written, and such models are told to conform to this metamodel: an XML file conforms to an XML Schema definition; a Haskell program conforms to the metamodel of Haskell; a program depending on a library uses function calls according to its API.

4 Megamodels explain relations between models

A model of a system of models is called a *megamodel*. For example, the last paragraph of the previous section is a megamodel (in a natural language), since it models the relations between software artefacts (model, metamodel, language). Megamodels are crucial for big-picture understanding of complex systems [4]. In literature they can be called megamodels [4,16,17], macromodels [28], linguistic architecture models [15,34] or technology models [22]. Megamodels can be partial in the sense of not being complete deterministic specifications of underlying systems [13], and they can also be presented in a way that gradually exposes the system in an increasingly detailed way [34,23,35].

A cave painting of a bison may be useful to understand the concept of hunting by abstracting from the personalities of the hunters and the measurements of the animal. However, to surface and understand its implications such as the near extinction and recovery of the species, one must also have models of bison populations, ecology, human society, USA politics, Native American politics, and so on — and be able to see how they relate to each other. In the same way, megamodels can aid in understanding software technologies, comparing them and assessing the implications of design choices in software construction.

4.1 Informal megamodelling

A cave-painting approach to megamodelling could be as minimalistic as follows:

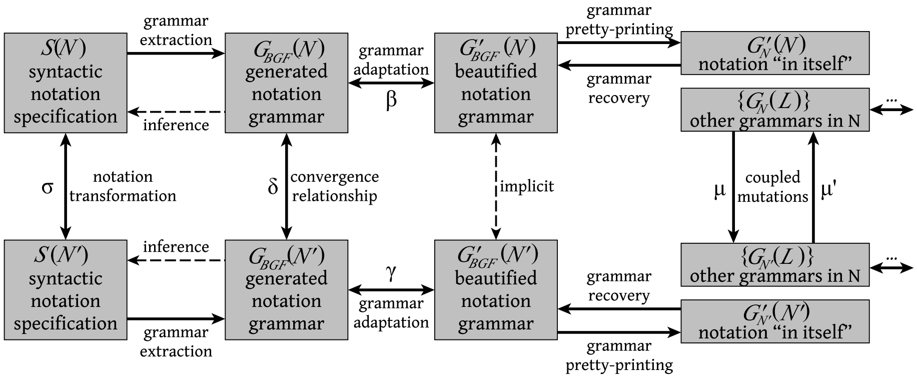
◇ draw a diagram with models as nodes
◇ add relations between them
◇ describe relations in a natural language

The focus of this approach is on understanding and communication [30,4]. For example, many papers, books and specifications in MDE contain an explanation of the stack of M1, M2 and M3 models (models, metamodels and metametamodels correspondingly) which positions them with respect to one another by postulating that models conform to a metamodel and both M2 and M3 conform to a metametamodel. Such an explanation, as well as its visual representation, is a megamodel. We have to draw your attention here to the fact that such a megamodel leaves many questions open and on a certain level of understanding it is incorrect: many models conform to one metamodel, and many metamodels can conform to one metametamodel, and the fact that the metametamodel conforms to itself, is no more than an implementation detail from MDA. That is the reason for various more formal attempts to exist to express the same situation in UML or another universal notation.

There is a big subset of informal megamodelling techniques referred to as "natural" [30] — it happens all the time in unstructured environments, whenever we use conveniently available salt and pepper dispensers as proxies for entities at a conference banquet discussion, or in general whenever we use throwaway abstractions to get to the point in a quick and dirty (volatile) way.

4.2 Ad hoc megamodelling

A slightly more detailed and yet still concrete approach is to explain relations between models and languages by showing mappings between them, without trying to generalise them to relations. Such mappings are easier to define and formalise and may be enough to understand the system. Thus, instead of saying "this model belongs to this language", we show that there is a tool which processes that model and that this tool is a software language processor. Usually such models mix architectural and implementational elements and when it comes to comprehension, almost impenetrable without extensive study of the system at hand. Here is an example [32]:



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

After some frustration we are free to observe here how $S(N)$, whatever it is[4], becomes $G_{BGF}(N)$ after a process called "grammar extraction" [33], and that

[4] In fact, $S(N)$ is a specification of a syntactic notation such as "an Extended Backus-Naur Form dialect that uses dots to separate production rules, same level indentation to list alternatives, ..." [31].

$G_{BGF}(N)$ is linked either bijectively or bidirectionally to $G'_{BGF}(N)$, and all these boxes titled with symbols, subscripts, dashes and parentheses, are linked to their counterparts from a similarly looking chain of transformations that seem to be related to N' rather than to N.

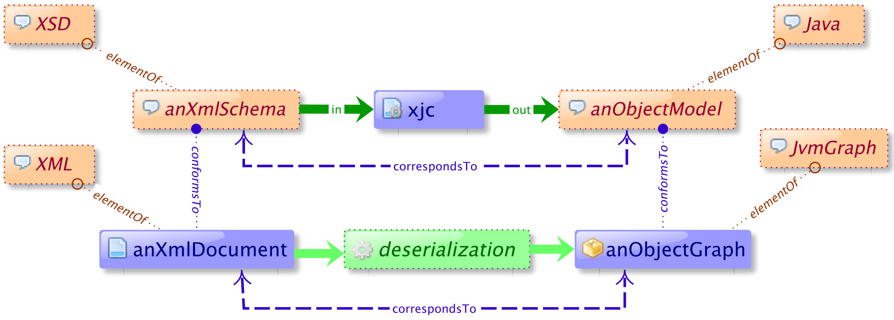
Even with a fair share of guesswork, this megamodel does not immediately bestow its observer with any piece of freshly granted knowledge. This megamodel basically encapsulates everything one could learn from the corresponding paper [32], condensing 17 pages into one diagram. It is more of a visualisation tactic than a comprehension strategy.

Many methods of ad hoc megamodelling are transformational: they use a newly introduced notation, different for each of them, to demonstrate how some software artefacts get turned into other artefacts. Unlike natural megamodelling, some ad hoc megamodelling approaches have very clearly defined semantics for their components instead of a natural language description. Unlike formal megamodelling that we will introduce below, they are typically fairly idiosyncratic and are not expressive enough to unambiguously model a situation sufficiently different from the study showcasing their application.

4.3 Instrumental megamodelling

One of the alternative approaches is to rely on some instrumental support: a tool or a language, perhaps both, that can do what a megamodel should — express relations between models, model transformations and languages. Hence, by using such a tool we can focus on providing such descriptions for a given system, perfecting them, reflecting on their evolution, etc. Committing to a framework means sacrificing at least some of the flexibility that natural and ad hoc megamodelling provide, in exchange of a much more precise understanding and definition of each component. An instrumental megamodel is not a cave painting anymore — it is a Latin text. Latin is a language everyone *kinda* understands, thus enabling its dissemination to a broader public. It might not be the best language to deliver you particular ideas, but once you get a hold on its cases, declensions and conjugations, you can use it again and again for many other tasks.

Here is an example megamodel by Favre, Lämmel and Varanovich [15]:



For a software engineer using such a megamodel “in Latin” means that each of these components is clickable and resolvable to a (fragment of a) real software artefact. In this particular case, the megamodelling language is MegaL[5], it supports entities such as “language”, “function”, “technology”, “program”, etc, and relations such as “subsetOf”, “dependsOn”, “conformsTo”, “definitionOf” and many others. There are other megamodelling languages: AMMA[6], MEGAF[7], SPEM[8], MCAST[9], etc, some people use categorical diagrams, which are closer to the next kind of megamodelling.

The process of navigating a megamodel and assigning a story to it, is called renarration [34]. This technique is needed quite often, since detailed megamodels can get bulky and rather intimidating — yet the same megamodels are supposed to be used to simplify the process of understanding a software system or communicating such an understanding, not to obfuscate it. Indeed, when a megamodel is drawn step by step with increasing level of detail (or vice verse, in increasing level of abstraction), it lets the user treat and comprehend one element at a time while slowly uncovering the intentions behind them. For MegaL, renarration operators include addition/removal of declarations, type restriction/generalisation, zooming in/out, instantiation/parametrisation, connection/disconnection and backtracking [23].

4.4 Formal megamodelling

Relying on tool support can be nice, but it is even better to be backed up by a theory that allows you to prove certain properties and verify your megamodels through solid analysis. Such approaches have rich mathematical foundations and vary greatly in form and taste. The choice is wide, but let us consider two different examples a little closer.

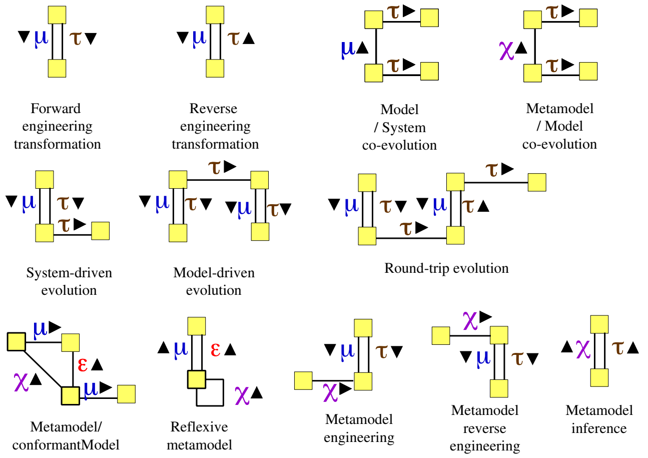
[5] MegaL: Megamodelling Language [15].

[6] AMMA: Atlas Model Management Architecture [3].

[7] MEGAF: Megamodelling Framework [18].

[8] SPEM: Software & Systems Process Engineering Metamodel [27].

[9] MCAST: Macromodel Creation and Solving Tool [28].



Suppose that instead of trying to come up with all kinds of relations that system fragments can have among themselves, we limit the relations to the most essential ones. Such relations can be well-understood and defined with relative ease for any particular technological space. We can refer to Jean-Marie Favre's relations [16,14]: μ — representationOf, ϵ — elementOf, δ — decomposedIn, χ — conformsTo, τ transformedTo. Then, we can on one hand afford to define each of them for our particular domain (e.g., grammarware, XML, Cobol, EMF); and on the other hand see megapatterns in them [16]:

For instance, in the top left corner we see an entity (say, X) that models another entity (say, Y), while X is also being transformed to Y. This is classical forward engineering, as opposed to reverse engineering where X models Y while the system Y is being transformed into the model X [6]. By now you can recognise the diagram of the original taxonomy by Chikofsky and Cross as an ad hoc megamodel, which also contains much more details than such a pattern, which is why the text of the paper is an important renarration of it. A similar pattern is displayed in the right bottom corner where X is being transformed into Y while also conforming to it — this could be grammatical inference, or constructing and XML schema from a selection of documents, or deriving a partial metamodel from a modelbase, or anything of that kind. All megapatterns are simplifications of real scenarios and as such, they are in some sense "wrong" — as are all models.

As another example of formal megamodelling, here is Diskin's definition of complex heterogeneous model mappings [9]:

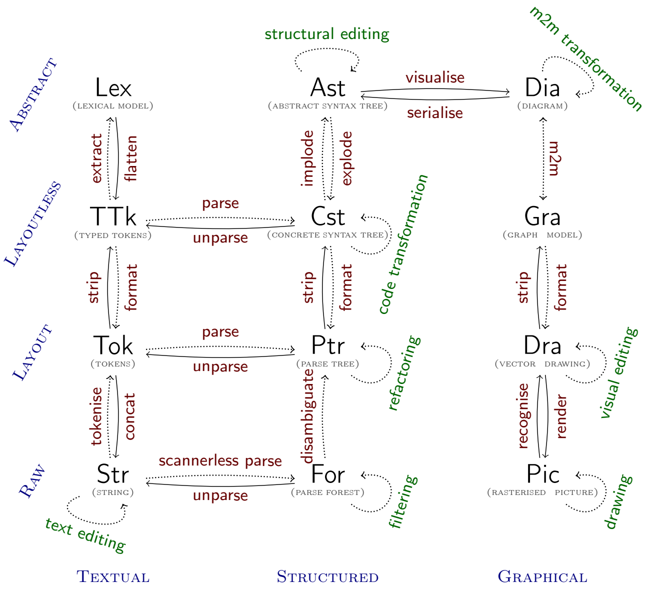
$$\begin{tikzcd}
\boldsymbol{\mathcal{A}} & \boldsymbol{\mathcal{O}} \arrow[l, Leftarrow, "\boldsymbol{v}"'] \arrow[r, Rightarrow, "\boldsymbol{w}"] & \boldsymbol{\mathcal{B}} \\
A{:}\boldsymbol{\mathcal{A}} \arrow[u, "{:}\mu^{\mathbf{QL}}"] & M{:}\boldsymbol{\mathcal{O}} \arrow[u, "{:}\mu^{\mathbf{QL}}"] \arrow[l, Lleftarrow, "f{:}\boldsymbol{v}"'] \arrow[r, Rrightarrow, "g{:}\boldsymbol{w}"] & B{:}\boldsymbol{\mathcal{B}} \arrow[u, "{:}\mu^{\mathbf{QL}}"]
\end{tikzcd}$$

Single-line arrows are links, double arrows are graph mappings, triple arrows are diagrams of graph mappings that encapsulate type mappings inside nodes and metamodel mappings inside arrows. Even this extremely condensed tile diagram is a simplification — since v and w are complex mappings, they should be drawn as triple arrows, while $f : v$ and $g : w$ become quadruple arrows. Still, the diagram itself remains structurally simple while still being unshakably formal. If we provide an accurate definition of our language's syntax, compositionality of metamodel mappings (a routine categorical process of defining Kleisli triples [24]), this graph turns into a (Kleisli) category. By going through some trouble or by limiting ourselves to monotonic queries, we can do the same for model mappings (not just metamodel mappings) [9].

Within this approach, each megapattern — divergence, convergence, revision of match, revision of update, improvement of match, conflict resolution — forms a tile of four involved software artefacts and labelled arrows between some of them. Then, tile algebra provides uniform rules to compose such tiles together [10].

4.5 Space megamodelling

Recall that *a metamodel is a model of a language*. (The previous sentence is a megamodel). Then, *a megamodel is a model of a technology*, since it shows how all involved fragments fit together to facilitate the process. In the previous section we have also made acquaintance with megapatterns — *models of processes within a technology*. Just one more step brings us to a abstract megamodel of an entire *technological space* [21]. For example [38]:



This megamodel models *everything* that can possibly happen when you are doing parsing, unparsing, pretty-printing, formatting, templating, stropping, etc.

Each element here is not resolvable to a concrete artefact, but rather to a subspace with its own stack of models and metamodels. For example, a concrete syntax tree (Cst) element found near the centre of the megamodel, represents concrete syntax trees, their definition as a concrete grammar, and all the techniques and tools that create, transform and validate them. A vector drawing (Dra, think SVG or GraphML), on the other hand, implies having a metamodel defining graphical elements, their coordinates and other attributes, as well as transformations such as a change of colour or realignment.

When renarrated, such a megamodel commits to becoming a representation of one particular technology, hence removing some of the elements that do not exist there and detailing the others so that they become resolvable [35]. We can also use the megamodel as a classificatory tool to look at existing techniques and positioning them with respect to others [37]. For example, what is "model-to-text transformation" commonly used in modelware frameworks and papers and deliberately omitted from being explicitly mentioned on the megamodel? In fact, it is a very particular path through this megamodel starting at Ast or Dia and going to Lex (commonly referred to as a "template" in this particular scenario) and then dropping straight to Str.

One can reasonably claim that such megamodels are in fact ontologies [8].

5 Conclusion

The tutorial was highly interactive and its biggest contribution to SATToSE was the discussion. This paper is a humble attempt to summarise (some of the) issues raised during both lecturing[10] and the hands-on parts, and provide bibliographical pointers for the most interested participants. There are many issues in megamodelling that we did not sufficiently cover — in particular, modelling the very nature of modelling [25,26] and taking both ontological and linguistical aspects into account [20,11,8].

Language is an important instrument of structured and meaningful communication, whether we use natural languages to convey information or create artificial ones tailored to the domain. We model languages with metamodels, since they are models of how software models can be put together. In practice, metamodels take many different forms such as programming language grammars, UML domain models, XML schemata and document types, library API definitions. Megamodels are used to model software technologies as systems of models, aimed first and foremost at understanding software systems, languages, tools and relations between them. Megamodelling makes relations explicit, identifies roles that software artefacts play and thus helps to understand technologies, compare them, validate, debug and deploy in a broad sense.

[10] Slides: http://grammarware.github.io/sattose/slides/Bagge.pdf.

References

1. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. Study of an API Migration for Two XML APIs. In *SLE*, volume 5969 of *LNCS*, pages 42–61. Springer, 2010.
2. J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE*, page 273. IEEE CS, 2001.
3. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *MDAFA*, volume 3599 of *LNCS*, pages 33–46. Springer, 2004.
4. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.
5. A. W. Brown. Model Driven Architecture: Principles and Practice. *SoSyM*, 3(3):314–327, 2004.
6. E. J. Chikofsky and J. H. C. II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.
7. N. Chomsky. *Syntactic Structures*. Mouton, 1957.
8. C. Coral, R. Francisco, and P. Mario. *Ontologies for Software Engineering and Software Technology*. Springer, 2006.
9. Z. Diskin. Model Synchronization: Mappings, Tiles and Categories. In *GTTSE*, volume 6491 of *LNCS*. Springer, 2011.
10. Z. Diskin, K. Czarnecki, and M. Antkiewicz. Model-versioning-in-the-large: Algebraic Foundations and the Tile Notation. In *ICSE CVSM*, pages 7–12. IEEE CS, 2009.
11. D. Djurić, D. Gašević, and V. Devedžić. The Tao of Modeling Spaces. *JOT*, 5(8):125–147, 2006.
12. M. Erwig and E. Walkingshaw. Semantics First! In *SLE'11*, pages 243–262. Springer, 2012.
13. M. Famelis, R. Salay, and M. Chechik. Partial Models: Towards Modeling and Reasoning with Uncertainty. In *ICSE*, pages 573–583. IEEE, 2012.
14. J.-M. Favre. Megamodelling and Etymology. A story of Words: from MED to MDE via MODEL in five millenniums. In *GTTSE*, number 05161 in Dagstuhl, 2006.
15. J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *MoDELS*, LNCS, pages 151–167, 2012.
16. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS*, 127(3), 2004.
17. R. Hebig, A. Seibel, and H. Giese. On the Unification of Megamodels. *EC-EASST*, 42, 2011.
18. R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing Architecture Frameworks through Megamodelling Techniques. In *ASE*, pages 305–308, 2010.
19. J. Jones. *Rhyming Cockney Slang*. Abson Books, 1971.
20. T. Kühne. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.
21. I. Kurtev, J. Bézivin, and M. Akşit. Technological Spaces: an Initial Appraisal. In *CoopIS, DOA*, 2002.
22. R. Lämmel. Programming Techniques and Technologies. http://softlang.wikidot.com/course:ptt13, 2013.
23. R. Lämmel and V. Zaytsev. Language Support for Megamodel Renarration. In *XM*, volume 1089 of *CEUR*, pages 36–45. CEUR-WS.org, Oct. 2013.

24. E. Manes. *Algebraic Theories*. Graduate Text in Mathematics. Springer, 1976.
25. P.-A. Muller, F. Fondement, and B. Baudry. Modeling Modeling. In *MoDELS*, LNCS, pages 2–16, 2009.
26. P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling Modeling Modeling. *Software and Systems Modeling*, 11(3):347–359, 2012.
27. Object Management Group. Software & Systems Process Engineering Metamodel (SPEM). Language Specification, OMG, 2007.
28. R. Salay, J. Mylopoulos, and S. Easterbrook. Using Macromodels to Manage Collections of Related Models. In *CAiSE*, pages 141–155. Springer, 2009.
29. B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.
30. Z. Zarwin, J.-S. Sottet, and J.-M. Favre. Natural Modeling: Retrospective and Perspectives an Anthropological Point of View. In *XM'12*, pages 3–8. ACM, 2012.
31. V. Zaytsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In *SAC PL*, pages 1910–1915. ACM, Mar. 2012.
32. V. Zaytsev. Language Evolution, Metasyntactically. *EC-EASST; Bidirectional Transformations*, 49, 2012.
33. V. Zaytsev. Notation-Parametric Grammar Recovery. In *LDTA*. ACM DL, June 2012.
34. V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. In *MPM*, pages 61–66. ACM DL, Nov. 2012.
35. V. Zaytsev. Understanding Metalanguage Integration by Renarrating a Technical Space Megamodel. In *GEMOC*, volume 1236 of *CEUR*, pages 69–77. CEUR-WS.org, Sept. 2014.
36. V. Zaytsev. Grammar Zoo: A Corpus of Experimental Grammarware. *Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5)*, 98:28–51, Feb. 2015.
37. V. Zaytsev and A. H. Bagge. Modelling Parsing and Unparsing. In *Second Workshop on Parsing at SLE 2014*, Aug. 2014. Extended Abstract.
38. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *MoDELS*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.