Multi-Language Modelling with Second Order Intensions

Vadim Zaytsev

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

Abstract. In the last decade, there have been several fundamental advances in the field of ontological and linguistic metamodelling. They proposed the use of megamodels to link abstract, digital and physical systems with a particular set of useful relations; the distinction between ontological and linguistic layers, identification and separation of them; even formalised the act of modelling and the sense and denotation of a language. In this paper, we propose second order intensions and extensions to more closely model linguistical and ontological conformance and mapping.

1 Formal modelling of languages

In the classic theory of formal languages, a *language* L is defined as a set of sequences of alphabet symbols: $L \subseteq \Sigma^*$ [9]. This definition is easily applicable to textual languages (traditionally associated with programming) and visual languages (traditionally associated with modelling). It is also almost trivially generalisable to graph languages by substituting the reflexive transitive closure in the definition by another operation that (usually recursively) constructs all possible valid language instances from symbols of the alphabet. Even then, all manipulations with the language are done as if it were a set of language instances. For example, a *parser* is generally considered a mapping from the textual language to the tree language in that it assigns a valid parse tree to each valid textual input [18]. Hence, the only relation that is needed to formally describe such processes is an *element of* ($\in$ or ε) relation with rare exceptions like generalised parsers [16] that associate one textual input with a set of several possibly valid parse trees. Since in practice such mapping usually gets implemented to output a *representation* of a set instead of the actual set itself, such cases are more sidesteps than real exceptions: the range of a general parser is a set of parse forests, and each output in an element of this set.

Traditional metamodelling abandons the concept of a language in favour of a *modelling layer* [14,15]. The formal arsenal is expanded to a strict hierarchical structure: the lowest layer is too close to real life to formally decompose and study (e.g., raw data, real life objects, concrete systems), the highest layer is so abstract that it is self-descriptive, and the middle layers M_i contain entities that model entities from the layer below (M_{i-1}) and are expressed in languages defined by entities from the layer above (M_{i+1}). Thus, user data is expressed

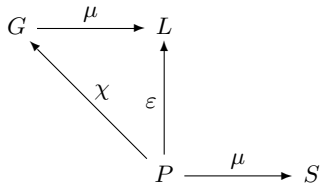

Fig. 1. A grammar G is a model of a language L, it is also a metamodel for the program P to conform to. The said program P is an element of the language L and it models a real system S. (The example demonstrates a megapattern of metamodel conformance [6, Fig.8]).

in user concepts that use UML concepts; UML concepts model user concepts in MOF; and MOF models UML concepts and defines both a language for them and a metalanguage for expressing itself. In this view, a new relation emerges: an instance of (we will denote it as ι), which is more abstract than the ε relation since it works formally even in situations when a set of all valid models cannot be expressed or when it does not make sense conceptually to express it. For example, an object is an instance of its class, and it is much less interesting to consider the set of all possible objects of a class than to investigate the nature of this instantiation and the consequences thereof. Since instantiations is sometimes hard to express universally, we also speak of a conforms to (χ) relation [2]: a model conforms to a metamodel, an object conforms to a class, a program conforms to a programming language, a database conforms to a schema. Since this theory is rooted in the modelling community, relations ι and χ are commonly used together with a *representation of* relation μ used in the conceptual sense: an object Cat models a real cat even though there might be no "language of cats" and the construction of a set of all possible cats is often unnecessary. Relations can be diagrammatically combined to form so called *megamodels* [3,6,5], an example shown on Figure 1.

Formal metamodelling distinguishes between two kinds of instance of relations: the *linguistic* instance of and the *ontological* instance of [1]. This complicates the metamodelling process somewhat but removes most ambiguity associated with the ι relation: the object Fluffy models a very particular cat and it is both an ontological instance of a class Cat (because Fluffy the real cat is a conceptual instance of cats in general) and a linguistic instance of an Object (because it needs to belong to a certain class, to be instantiated in a certain way and obey other constraints typical for all objects but not for all cats). An example is given on Figure 2 and is easily extensible to new ontological levels: cats are ontological instances of species, which are ontological instances of a biological rank [1, Fig.5] — and the formalisation still allows us to distinguish these modelling statements from ones that stay within one ontological level (i.e., that cats are pets, carnivores, mammals, animals, etc — in the object-oriented technological space this is called inheritance). Adding more languages and trans-

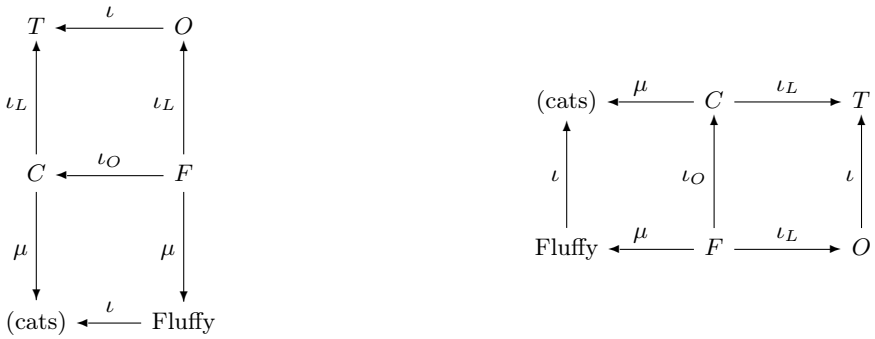

Fig. 2. The linguistic (on the left) and ontological (on the right) metamodelling views on the same megamodel. Fluffy is a cat (instance of a concept of a cat or an element in the set of all cats). Fluffy is modelled (μ) by an object F which is a linguistic instance (ι_L) of an object (O) and an ontological instance (ι_O) of a class Cat (C). The object O is in an instance-type relation to T. Columns on the left and rows on the right roughly correspond to ontological levels; rows in the left and columns on the right roughly correspond to language levels or modelling layers. (The example is a simplified/adapted version from [1, Figs.2,3]).

formations into the megamodel is somewhat more problematic due to the grid nature of the diagrams and to the lack of definitions of ontological instantiation for some languages. Coming back from biology to computer science, this allows us to properly specify that a particular program is a (linguistic) instance of say Java, but an ontological instance of a database application and as such, also obeys a set of structural and behavioural rules.

The next step in refining the theory of metamodelling of megamodels was separating the intensional and the extensional parts of the language [7]. The extensional part is argued to be the set of models allowed in the language. The intensional part models constrains and properties that are characteristic to the instances of the language. If such a distinction is introduced, the meaning of being the "instance of" something becomes ultimately apparent: the model in question must conform (χ) to the intensional part and it is an element (ε) of the extensional part. Since the extension of an abstract system always resides on the lower ontological level, the diagram is also nicely composed of tiles of the meta-level entry and its intentional part on top and its extensional part (the set of valid instances) and the model-level entry that conforms to the intensional part, is an element of the extensional part and is at the same time an instance of the abstract system. An example migrated from the technological space of cats, dogs, breeds and animals, to programming languages, can be seen on Figure 3 (read $\delta_\wedge$ as "has intension" and $\delta_\vee$ as "has extension", as δ was often used for "decomposed in"). In general, tiles like these can always substitute the megapattern from Figure 1, it is in fact its ontology-aware refinement.

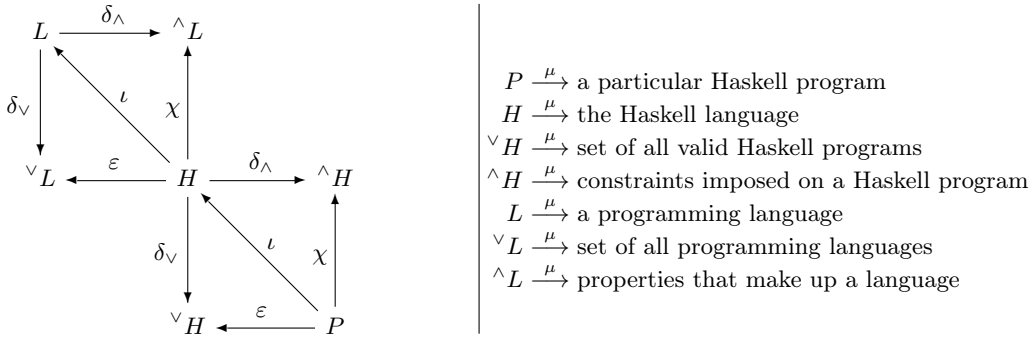

Fig. 3. Two tiles on the left diagram show how an abstract system (a language) H or L is decomposed into an intensional part ${}^{\wedge}X$ and an extensional part ${}^{\vee}X$. Haskell (H) is a programming language (L), so it is an instance (ι) of a language. This means it conforms (χ) to the intension of being a language (${}^{\wedge}L$) and is an element (ε) in the set of all programming languages (${}^{\vee}L$). Similarly, P is a program in Haskell, so it is an instance of H, it conforms to ${}^{\wedge}H$ and is an element of ${}^{\vee}H$. (The example is ported to a more fitting technological space from [7, Fig.7]).

2 The role of a grammar

In the domain of programming languages, one often speaks of a conformance of a program to the grammar of the language in which the program was written in. Is such a grammar the same as the intensional part of the language? ($G = {}^{\wedge}L$?)

The answer given by the formal language theory is *yes* — however, this theory has a slightly different megamodel of the situation: since a language is equated with its extensional part, the "instance of" relation is equated with the "element of" relation. Furthermore, instead of the language being decomposed into two parts, its intension is treated as a *model* of its extension (which is typically infinite, so it helps to have a finite model of it). The result is depicted on Figure 4 in the same style we have used so far. As an example we can consider the technological space of XML: then P is an XML document, G is a DTD or an XML Schema definition, the validator uses G to check $P \xrightarrow{\chi} G$ and programs in XSLT, XQuery, JavaScript and other languages can be written to work on elements of L.

In a more general case, the role of a grammar is the $G \xrightarrow{\mu} L$ chain is called generative or derivational and is used in most proofs in the theory of formal languages and automata. Its role in the $P \xrightarrow{\chi} G$ chain is called analytical and is often utilised by using it prescriptively [8] and generating a parser out of it. What does such a parser do? In the simplest case, it analyses the text of the program and constructs a term that aligns tokens (lexemes) of the input with its understanding of how the structure of any program should look like. This already does not fit our picture at all, and we lack means to express that not only the grammar serves in at least two different roles, but also the language apparently at the same time is a language of strings (that are acceptable inputs

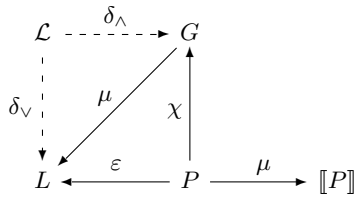

Fig. 4. The formal language theory view on relating languages and instances: the program P is an encoding of a solution $[\![P]\!]$, which is an element of the formal language L and conforms to the grammar G. The grammar G is also a finite model of a (typically infinite) language instance set L. The conceptual language $\mathcal{L}$ is implicit and utterly intangible, and its instantiation is never discussed explicitly.

for a parser) and a language of terms or trees (that are outputs of a parser). An attempt to fit this transformational megapattern into our view is shown on Figure 5: a grammar G serves a model of both two languages (textual and tree language) and the mapping between two representations of the same program (the text and the parse tree). However, since $\mathcal{L}_C$ and $\mathcal{L}_T$ are implicit, we cannot make any statement about the relation between L_T and L_C, which makes the megamodel a bit less useful.

This multipurposefulness of grammars in a broad sense is unfortunate from the modelling point of view, but it explains their omnipresence in software engineering [10]. In terms of modelling modelling[1], $G \overset{\mu}{\rightsquigarrow} X$ for all megamodel elements X related to G: they share some intention and can be partially represented by one another [13]. We will explore this intersection in the next section and make it explicit.

In practical software language processing, grammars try to balance in between all these roles, with a varying degree of success. In certain cases, people separate some constraints (traditionally called static semantic rules) that are too hard to express in the chosen grammar formalism and are purely related to $G \xrightarrow{\mu} L$ and $P \xrightarrow{\chi} G$; in other cases (in particular related to mapping between already structured concrete syntax and an improved abstract syntax) there could be several grammars defining separate languages, with $G \xrightarrow{\mu} \tau$ included in one of them or shipped as a third separate artefact.

Interestingly, the role of a language in modelware engineering is slightly different yet also not perfect. Consider the following statement [13]:

$$F \xrightarrow{\mu} L \xrightarrow{\mu} \left\{ M \mid M \xrightarrow{\mu} S \right\}$$

What is stated here is that the formal system F truthfully models a language L which in turn models a set of models M such that they all model the system S.

[1] NB: the original MoDELS 2009 paper used "$\overset{\mu}{\rightarrowtail}$" for shared intention instead, we use a much more fitting notation from the extended journal version.

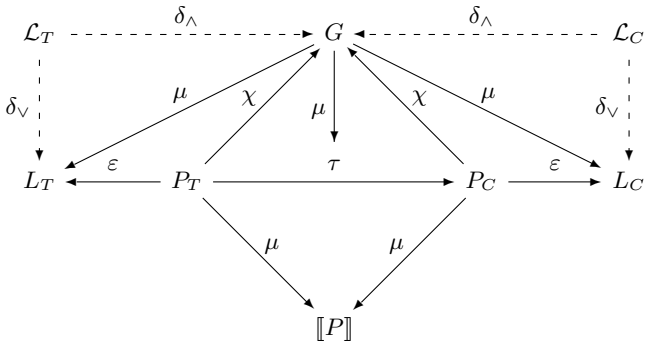

Fig. 5. The classic grammar theory view on relating languages and instances: the text of the program P_T and a concrete parse tree P_C both model the conceptual program (algorithm) $[\![P]\!]$ and both conform (χ) to the grammar G which acts as a model (μ) of both the string language L_T and the tree/term language L_C viewed as the sets of their corresponding instances. The same grammar G simultaneously models the transformation between text instances and tree instances. Conceptual languages $\mathcal{L}_T$ and $\mathcal{L}_C$ remain intangible, denotation $[\![P]\!]$ might exist in a form of flow diagrams.

If represented diagrammatically on Figure 6, we see the main differences between our approach and the method of Muller et al: instead of being decomposed into an intension and an extension, the language is considered to be a model of its extension, and its intension (grammar in a broad sense, some kind of formal model by requirements) is considered to be its model. Furthermore, there is no explicit consideration for the conformance between the models and this formal definition.

3 Second order to the rescue

As we have seen, the grammar of a software language is its intensional part (or approximates it very closely). Let us deconstruct it further. For ${}^\wedge\mathit{Collie}$, Gašević et al claimed it was a model of the real world intension of the concept of a collie, combining properties such as "has long hair", "has bushy tail" and "can herd sheep" [7]. For a programming language, the intension is a model of two kinds of properties: essential ("supports parametric polymorphism", "uses lazy evaluation", "contains a conditional statement which must contain a condition and a statement", "variable names should start with a letter") circumstantial ("functions have comma-separated arguments", "statement blocks are defined by indentation", "one statement per line of code"). The essential properties refer to the way the language concepts are constructed and manipulated — therefore, their model is the intension of the intension of the language (${}^\wedge{}^\wedge L$). The circumstantial properties refer to the way the concepts of the language are represented in the instances — in other words, how language instances (elements of ${}^\vee L$) are

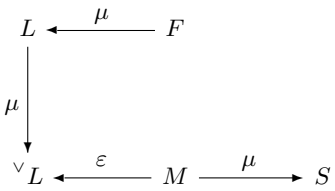

Fig. 6. Modelling modelling modelling: all models M that model a system S, are collected in a set ${}^{\vee}L$. The language L is considered to be a model of its extension, and is in turn modelled by a formal system F which is conceptually its intension which remains disconnected from the model M. (This example is a diagrammatic representation of [13, Fig.17]).

constructed; hence, it is the intension of the extension of the language $({}^{\wedge\vee}L)$. The extension of the intension is even more interesting because it is supposed to collect examples of how the intensions can be expressed: it is a collection of all possible syntaxes for a language. To the best of our knowledge such an entity has not yet been formally investigated, but if it also represented as a set, the intension of the extension is an element of that set; otherwise it is still bound to be an instance of ${}^{\vee\wedge}L$. The resulting diagram is depicted on Figure 7 with the decomposition, conformance and instantiation relations.

Since now we have ${}^{\wedge\wedge}L$ to denote the constraints essential to the language L, it should be the same for all related languages. Indeed, if the modelling intent [17] is limited to this essence, and a grammar that respects it, it also models all variants of languages closely without any regard to the choice of ${}^{\wedge\vee}L$. In other words, for $\mu' = \mu/_{{}^{\wedge\wedge}L}$,

$$ {}^{\vee}L_T \xleftarrow{\mu'} G \xrightarrow{\mu'} {}^{\vee}L_C $$

This result agrees with the "shift in linguistical conformance" by Muller et al [13] (when talking about mapping among models with the same intent) and with the "constant functions all the way down" by Dowty et al [4] (when talking about intensions of intensions). The final diagram is presented on Figure 8: a "real" program can be encoded by a programmer as either text (conforming to ${}^{\wedge\vee}L_T$) or a tree (conforming to ${}^{\wedge\vee}L_C$), and as long as the intension of the intension is preserved ($P_T \xrightarrow{\chi} {}^{\wedge\wedge}L$), the languages stay *same but different* and valid instances can be freely mapped in any direction. The same holds for mappings between abstract syntax and concrete syntax, up to a homomorphism (such mappings often permute arguments and perform other component rearrangements).

4 Concluding remarks and related work

In short, we have proposed to consider four components of software languages: intension-intensions (conceptual constraints to always conform to); extension-

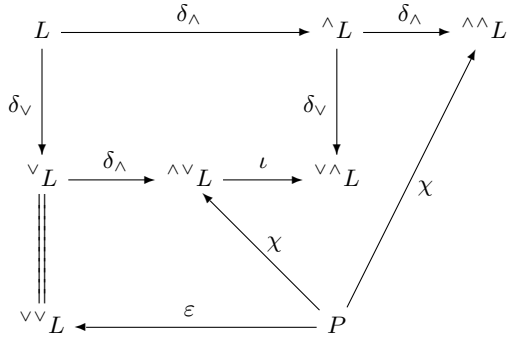

Fig. 7. The language L is decomposed (δ) into its intensional ($^\wedge L$) and extensional ($^\vee L$) parts. They are, in turn, decomposed in the same fashion. The extensional part $^\vee L$ is a set, so the extension of the extension $^\vee{}^\vee L$ is the same set. However, the intension of the extension $^\wedge{}^\vee L$ represents structural constraints of the elements of the set, while the intension of the intension $^\wedge{}^\wedge L$ represents linguistic constraints that refer to the intensional language and not to its accidental representation. The extension of the intension $^\vee{}^\wedge L$ represents a model of possible syntactic representations of programs in the language: it is modelled by a simple set, the particular syntax $^\wedge{}^\vee L$ is an element of that set, otherwise it is still bound there in an instance of (ι) relation.

extensions (sets of valid language instances in a given notation); intentions of extensions (circumstantial constraints specific to the chosen syntax but not the the language as such); extensions of intensions (models of possible syntaxes compatible with the language core), in a hope that it brings us closer to understanding the nature of modelling languages and various artefacts related to them. In previously existing work, languages are mostly considered either with a fixed syntax or with two or three of them which are also fixed and claimed to be related to different aspects of language processing. This work can serve as a foundation for formal manipulation of languages with multiple syntaxes, or systems where "the same" language is claimed to be used across technical spaces (ORM, parsing, convergence, etc).

For the sake of clarity and conciseness of notation, we have opted for the use of commutative diagrams to represent megamodels instead of using any of the existing megamodelling languages. We have adopted Favre's symbols for relations: μ as "models" or "representation of"; ε as "element of", χ as "conforms to". Atkinson and Kühne did not have a shorthand notation so we used ι_O for "ontological instance of", ι_L for "linguistic instance of" and just ι for "instance of" where the meaning is unknown or universal. We have adopted Montague notation: $^\wedge L$ for intensions and $^\vee L$ for extensions. We also took the liberty of using $[\![P]\!]$ for denotations of single instances — in formal semantic theory denotation is equated to extension but in the current state of multi-level metamodelling we do not yet need to differentiate between $^\wedge P$ and $^\vee P$. However, in the future research on multi-language modelling, we recommend to consider substituting

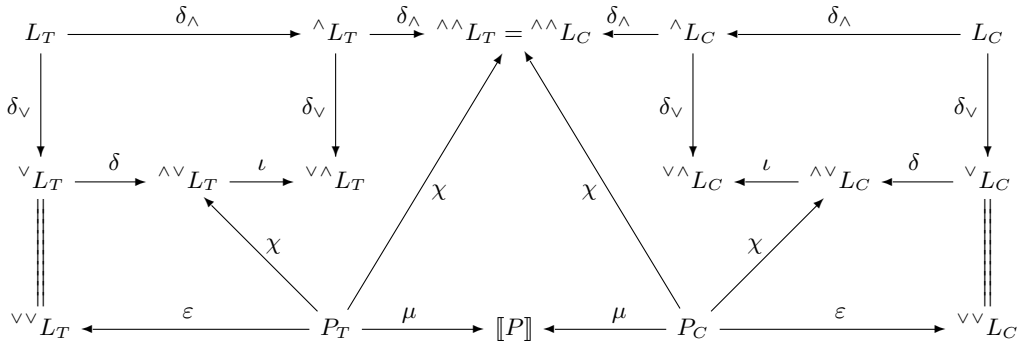

Fig. 8. Each language shown — the textual language L_T and the concrete tree language L_C — are decomposed (δ) into their intensions ($^\wedge L_x$) and extensions ($^\vee L_x$), which are in turn also decomposed into their intensions and extensions. The denotation of a program ($[\![P]\!]$) is modelled by two entities: the program text (P_T) and its concrete parse tree (P_C). They follow the same megapatterns: each P_x conforms (χ) to both the intension of the extension of the corresponding language ($^{\wedge\vee} L_x$) and the intension of the intension of the language ($^{\wedge\wedge} L_x$). Each P_x is also an element (ε) of the extension of the language ($^\vee L_x$). The intension of an extension ($^{\wedge\vee} L_x$) is an instance (ι) of the extension of the intension ($^{\vee\wedge} L_x$), and in the simplest case also just its element.

$[\![P]\!]$ with $^\wedge P$ and using $^\vee P$ for the hypothetical set of all possible representations of a program P.

The notion of commitment to grammatical structure was discussed by Klint, Lämmel and Verhoef [10], and recently was elaborated by a megamodel of various software language engineering artefact kinds such as parse trees, visual models, lexical templates, etc, in the context of (un)parsing in a broad sense [18]. The contribution of this paper to that trend was showing that concrete syntaxes are ontological instances of the abstract syntax (more precisely, $^{\wedge\vee} L \xrightarrow{\iota} {}^{\vee\wedge} L$). The idea that a software language should be allowed to have different syntaxes while staying essentially the same language, is not new but has always been rejected by formalisations.

Atkinson [1], Bézivin [3], Favre [6], Gašević [7], Kühne [11], Muller [13] and many others have made significant contributions to comprehension and refinement of the processes of modelling, metamodelling and megamodelling. This paper is an endeavour to contribute to that trend by making a yet another step in improving the formalisations, as well as by providing concrete examples from software and grammarware engineering, even when they seemed less suitable to discuss ontological matters than the classic Lassie/Fido – Collie – Breed example. We hope such results will both help to identify problems we can solve in the future and bring less meta-minded modelling practitioners and language engineers closer.

Apart from the Fregean perspective on intensional logic, Montague considered a Russelian variant where the extension of the intension (called the "sense denotation") is not a fundamental concept [12]. It has proven to be less useful to him, but in software language engineering this idea has never even been tried yet.

References

1. C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.
2. J. Bézivin. In search of a basic principle for model-driven engineering. *Novatica Journal*, 2004.
3. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.
4. D. R. Dowty, R. E. Wall, and S. Peters. *Introduction to Montague Semantics*. Kluwer Academic Publishers, 1981.
5. J.-M. Favre. Towards a Basic Theory to Model Driven Engineering. In *Proceedings of the Third Workshop on Software Model Engineering*, 2004.
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. D. Gašević, N. Kaviani, and M. Hatala. On Metamodeling in Megamodels. In *MODELS'07*, pages 91–105, 2007.
8. W. Hesse. More Matters on (Meta-)Modeling: Remarks on Kühne's "matters". *SoSyM*, 5(4):387–394, 2006.
9. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2007.
10. P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM ToSEM*, 14(3):331–380, 2005.
11. T. Kühne and D. Schreiber. Can Programming be Liberated from the Two-Level Style: Multi-Level Programming with DeepJava. In *OOPSLA*, pages 229–244. ACM, 2007.
12. R. Montague. Universal Grammar. *Theoria*, 36(3):373–398, 1970.
13. P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling Modeling Modeling. *SoSyM*, 11(3):347–359, 2012.
14. Object Management Group. *Unified Modeling Language*, 2.1.1 edition, 2007. Available at `http://schema.omg.org/spec/UML/2.1.1`.
15. Object Management Group. *Meta-Object Facility (MOF™) Core Specification*, 2.0 edition, January 2006. Available at `http://www.omg.org/spec/MOF/2.0`.
16. M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
17. E. S. K. Yu and J. Mylopoulos. Understanding "Why" in Software Process Modelling, Analysis, and Design. In *ICSE*, pages 159–168. IEEE Computer Society / ACM Press, 1994.
18. V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*, volume 8767 of *LNCS*, pages 50–67, Switzerland, Oct. 2014. Springer International Publishing.