Megamodelling with NGA Multimodels

Vadim Zaytsev
Raincode Labs, Kazernestraat 45, Brussels 1000, Belgium
vadim@grammarware.net

Abstract

One of the contemporary methods of tackling complexity in information systems is megamodelling: creating explicit models to express relations among artefacts, languages and transformations. Such models can encapsulate architectural knowledge of a system while retaining the ability to "zoom in" and provide implementation details whenever needed, up to providing links to concrete assets like files and tools. There are many approaches to megamodelling yet none as widely accepted as UML in software modelling or as (E)BNF in the grammarware technological space. In this paper, we propose a methodology to model the zooming feature of megamodels explicitly, without fixing the depth up front, and explain why a behavioural aspect is required in many circumstances. The three aspects we propose are Nodes, Graphs and Automata (NGA for short), representing the abstract view on an architectural entity, a more structurally detailed view and a dynamically behaving executable model, respectively. There were none prior (mega)modelling approaches to cover all three such aspects. The NGA approach to megamodelling allows to add behavioural properties to the specifications of information systems while keeping all the functionality of a usual megamodelling methodology (abstraction, navigation, traceability and resolution). We provide a range of case studies to justify bearing with the added complexity.

CCS Concepts • **Computing methodologies** → **Model development and analysis**; • **Theory of computation** → *Semantics and reasoning*;

Keywords Megamodelling, multimodeling, automata

ACM Reference Format:
Vadim Zaytsev. 2017. Megamodelling with NGA Multimodels. In *Proceedings of ACM SIGPLAN International Workshop on Comprehension of Complex Systems (CoCos'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3141842.3141843

1 Motivation and Background

Modelling as the methodology of creating structural entities truthfully representing a selection of interesting qualities of an information system while abstracting from others, has existed long enough to be widely accepted as one of the most productive tools in the software engineering arsenal. A typical software development project contains some forms of requirements models, interaction models, performance models, data format models, communication protocol models, design models, domain models, user experience models, etc. To understand the roles these models play in the process of creating and, even more importantly, maintaining software systems, people have come up with *megamodels* as models of complex modelling systems. An example of the simplest nontrivial megamodel can be a statement that any model must conform to a metamodel, or, $\forall m \ \exists M \quad m \xrightarrow{\chi} M$.

To lower the bar for the acceptance of megamodelling, which is already pretty high, we are usually expected to use simpler graphic notations instead of mathematical formulae. The most colourful of them is MegaL by Favre et al. [14]:

Megamodelling languages have reasonably well-defined meaning assigned to shapes, colours and other visual dimensions of such models and try to reuse standing traditions (so the existential nature of the conformance relation is drawn with a dashed line in MegaL). Seidewitz warns us [40] about distinguishing meaning as *interpretation* (when we establish what our models mean, in terms of the knowledge they manage to capture about the actual artefacts they model) and meaning as *theory* (when the model is viewed as a specification that should have some properties that hold on the thing it models). A good example of interpretation would be checking that all source artefacts are indeed present in the deployment package and all intermediate ones actually exist as files after the generation takes place; an example of a theory would be calculation of the dependency graph and deriving which models depend on a given one, based on it. The struggle of megamodelling languages to cover both aspects, will be described in further subsections.

Megamodelling as the methodology of modelling what models model, or, in other words, of expressing how models relate to one another, is an important part of the overall endeavour to let models help developers and other stakeholders to increase their understanding of the information system. Such a megamodel usually manipulates classical MDA layers: M0 as real world entities, M1 as models, M2 as metamodels and M3 and metametamodels [34], — in a volatile manner, grabbing them whenever they are needed and sticking them

CoCos'17, October 23, 2017, Vancouver, Canada

into the complete picture. A more systematic way of manipulating these levels and having an arbitrary number of them, is called multilevel modelling.

Independently of the megamodelling movement, there is a tendency to move away from individual models that are perfectly modular to *multimodels* which represent sets of interrelated models. Multimodels express not only collections of atomic models, but also properties that arise from the very fact of their collaborative use and are not inferable from the individual items [25]. For example, one can have global uniqueness constraints that are obviously invisible when looking at each model fragment individually or in small groups; or ordering constraints that make it impossible to claim with certainty whether a communication trace conforms to the protocol, just by looking at individual messages without considering their sequential composition. The main challenge currently standing in the way of multimodelling hides behind the fact that the natural definition of them (as a set of global constraints attached to a collection of models to be merged) contradicts with the usual implementation of them, which is different simply because merging all individual models is impractical for truly complex systems. Still, there are ways to infer global consistency claims from the results of locally performed checks.

Obviously, instead of attempting to model several levels simultaneously, it is possible to define machinery with operators or other means to add more elements for a megamodel as means of refinement [30] and/or renarration [49, 50]. However, that process is fairly straightforward and is specific to neither megamodels nor multimodels, so we leave it out of the scope of this paper. The only significant research advances happening in that area, concern partiality and uncertainty [16, 37].

The remainder of the paper is structured as follows. The rest of this section explains megamodelling in much more detail and in particularly identify three schools of thought concerning the use of megamodels: they can be viewed as models of how smaller models fit together; as models of the entire domain of modelling; or as means to map smaller models to resources and other concrete artefacts. Then, § 2 will introduce the main proposed contribution of this paper: namely, the way to model different artefacts as three-level multimodels that can be combined and used as N-views, G-views and A-views. The theoretical introduction was deliberately made as brief as possible, as we promptly switch to case studies of increasing sophistication, that demonstrate the need for our approach and pinpoint shortcomings of competing single-level approaches. We cover three such examples: legacy software migration, compilation and parsing, and coupled transformation. Finally, § 3 concludes the paper by summarising its contributions.

Megamodels, occasionally called megamodules or macromodels, are used at least since 1992 to describe architectures of complex information systems [44]. Processes associated with them are *megaprogramming* [6], when they are encapsulated as components and used to build actual systems, and *megamodelling* [5] when they are used as stepping stones in complex system comprehension, allowing to cope with accidental complexity — the two processes conceptually realise the meaning-as-interpretation and meaning-as-theory that we mentioned in the introductory section [40], but each theory always tries to provide at least some means to address the need for interpretation. There are three schools of thought in megamodelling.

1.1 Three Uses of Megamodels

The first school of thought [2–5, 14, 47, et al.] focuses mostly on the abstract aspect and defines a megamodel as *a model some elements of which represent other models*. This is the most general approach, it enforces very little limitations and is therefore the most widely applicable, but at the same time the hardest to demonstrate benefits of. MEGAF is an example of a practically geared system implemented within this paradigm, capable of expressing viewpoints, stakeholders, models, model kinds, views, rules, concerns, etc, and linking them with hypergraphs [22]. More limited and focused approaches exist: for instance, languages can be represented by nodes and mappings by arcs in a megamodel graph [45].

The second use of megamodelling [13, 15, 17] concentrates on building a model of main model-driven concepts, tying them together in something resembling a sketchy domain model (of the domain of modelling) or an ontology of model-driven engineering. Similar reasoning is followed while building a megamodel of all possible parsing and unparsing technologies [51]. The main difference between these megamodels and the ones from the previous paragraph lies in the fact that megamodel elements here refer to concepts and not necessarily to artefacts, even abstract ones: some elements might be concrete (e.g., MOF refers to [35]), but they are freely mixed with families of artefacts (e.g., BNF, explained as a product line by [46]) and with artefact roles ("a model", "a program", "a grammar", etc).

The third research direction is about resolvable megamodels [14, 21, 26, 27], models where each element not just represents another model, but refers to an actual existing software artefact or resource: typically a model, a metamodel, a model transformation, a tool, a file, a data type. Favre et al. call them *linked megamodels* [14] and Salay et al. consider all megamodels to fall into this category and call unresolved megamodels "mgraphs" [38]. Vignaga et al. do not insist on resolving megamodel elements to concrete artefacts and show how information about their roles and types could already be leveraged to prevent certain runtime errors [43]. Lämmel uses megamodels to show how software languages relate to one another within a repository [26] — the notable difference from the examples above being that all megamodel elements are linked to concrete files within the repository. Previous stages of the same line of work, such as MegaL by

[14], were meant to be used like this as well but were successfully applied to model abstract situations without linking to artefacts (e.g., [48, 49]). In that sense, (meta)syntactic differences aside, MegaL [14] could be seen as a domain-specific version of MEGAF [22] for languages and transformations.

Coming back to the very first example at the start of this paper ("all models must conform to a metamodel"), it can be interpreted in one of these three ways by the schools described above:

- For any model found in our system there should be a metamodel shipped with it which it conforms to.
- All models are universally demanded to have a metamodel to conform to, and anything else by definition is not a model at all.
- The complete list of all models and metamodels within the information system is provided, showing which files are they stored at, and each of those models is linked to at least one of those metamodels.

These three ways of thinking of megamodels (as models of modelling ecosystems, as modelling ontologies and as mappings between models and resources) are not mutually exclusive. For instance, *renarration* is a technique of knowledge representation and dissemination that can instantiate elements of a megamodel gradually, resolve them to artefacts or abstract from details [30, 49, 50].

1.2 Megamodel Definition

Hebig et al. [20] propose an appealingly simple formalisation of what a megamodel is, by redefining that each megamodel element should be a "model" (which can in turn be defined and refined further). Essentially, the core metamodel for megamodels then states nothing more than that a model contains zero or more models and zero or more relations, and each relation connects one or more models together. "Terminal" kinds of models such as the source model, the target model or the transformations, are defined as subclasses of a "model". This is a nicely natural extension of the general understanding that "everything is a model".

Salay et al. [38] propose to formalise megamodels as pairs so that each megamodel has an *mgraph* of model elements and an mgraph homomorphism called a *dereferencing mapping*. Nodes of an mgraph are either models linked by relationships or megamodels linked by *megarels* (i.e., relations between megamodels).

In this paper, we extend these views and introduce several degrees of flexibility by adding ideas from multimodelling. In our eyes, each megamodelling entity is simultaneously a node in some mgraph connecting it to others, an mgraph by itself defining its own internal structure, and a dynamic system that can be executed to behave in a certain way.

2 Node–Graph–Automaton

Since we would like to concentrate on preserving alignment between model elements, we will consider linked pairs of megamodel elements instead of individual elements. The approach is easy to generalise for more than two elements (for multiary relations that are not simplifiable to a set of binary ones) and becomes trivial in contexts where the traces are not important. This approach is inspired by categorical thinking [10] but we do not claim any level of correctness in the naïve category-theoretical interpretation of the definitions provided here. Developing a proper theory is reserved for future work.

The basic idea behind the NGA approach is to have each of the megamodel elements being able to masquerade as one of the three elements: Node (N), Graph (G) and Automaton (A) — hence, the NGA name. The **N**ode view represents the most abstract view of the element as possible: quite often it is just a node in a megamodel/mgraph, even if it represents an entire language. Alternatively, the Node view can contain two nodes linked by a mapping relation if that is where the modelling focus lies. The **G**raph view shows more details: it refines the single node to a number of somehow interrelated ones or refines a relationship from one arc to some static representation of the process and/or transformation sequence. The **A**utomaton view is all about the behaviour, it models what happens if the system is "run". Typical automata used in software engineering are activity diagrams, statecharts, Moore machines, Mealy machines, Petri nets, Büchi automata, Kripke structures as well as variations of finite state machines. In some practical implementations of the NGA paradigm we were also using functions as first class citizens to model automata: in this case a functional language is certainly required, but it helps even more to have a homoiconic language capable of freely manipulating code as data, decomposing and composing function definitions on the fly, such as Clojure, Rebol, Scheme or Racket.

2.1 Software Migration Example

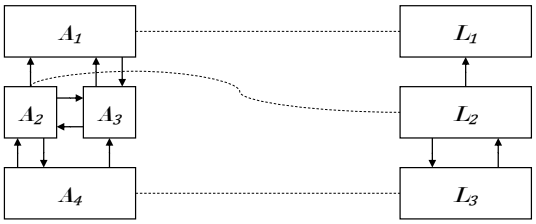
Software systems evolve, grow and rot. Solutions that have been introduced decades ago for performance reasons or for the sake of compatibility with other systems which got replaced in the meantime, are not relevant any more and weigh heavy on maintenance activities performed by modern engineers. Choices made in architectural design due to the lack of expertise and evidence, become apparent in their suboptimality after being re-evaluated after the hype has lost its power. Languages that were once thought to replace older counterparts, become legacy themselves and turn into business liabilities when it gets impossible to hire new developers with any knowledge of them.

The domain of software migration is well-researched. There are many contradicting definitions of "legacy systems" and "legacy code" among practitioners, all assigning different

technical meanings to the intuitive concept of a software asset that is not sufficiently understood and currently in need of adjustment [23]. There are even more different families of approaches, such as model transformations applied in particular to architectural modelling, program transformation and direct source code manipulation, concept analysis, requirements recovery and analysis, especially various knowledge elicitation approaches. Verification of software migration processes is usually out of reach, but there is considerable effort dedicated to consistency claims through testing.In general, completed software migration endeavours are quite successful, there exist several companies whose business is solely dedicated to performing such projects (the author of the paper works at such a company), and empirical research shows that in practice, software migration of legacy systems into modern environments is not only capable of reaching the initially set goals, but also tends to cause unforeseen additional benefits [24]

The N-view of a software migration megamodel is conveniently simple: it states that there is a system which gets translated to another system: $S_1 \xrightarrow{\tau} S_2$.

We can combine this NGA element with others to specify more transformations or express that these systems are elements of (ε) or representations of (μ) other elements — all of which would also be mere nodes in the mgraph. On the G-view, we refine each node to a graph and each arrow to a relation, which in our case could look like this:



Conceptually, A_i and L_j represent architectural layers of S_1 and S_2 respectively, and relations *within* the model show which ones are allowed to interact. Each of these elements can be seen as an N-view, which can in turn be "zoomed into" and unfolded into a G-view, and so on. In this example, we can expand, say, $A_2 \xrightarrow{calls} A_1$, into subsystems of A_1 and A_2 and how they call one another; and then further one to packages, assemblies, namespaces, programs, classes, methods, statements, expressions, instructions.

The relation represented by dashed lines, is a refinement of the *application* of τ, not of τ itself ([14] elaborate well on the distinction between a function and a function application in the context of modelling). In the literature it is called model alignment [11], or horizontal sameness or correspondence relations [12] or "correspondsTo" [14]. This refined relation shows how τ propagates model elements from one side to the other. Depending on the modelling scenario, this relation can be purely structural or contain homomorphic mappings, and not necessarily show how the actual mapping take place (i.e., if there is a data dependency between two components during migration). So in this particular case the G-view does not show what should happen with A_3: will it be removed, subsumed by L_2 or merged into L_3.

The A-view is capable of modelling that detail and much more. For example, one can complete the NGA multimodel with a system dependence graph expressing which parts of the system have data-based or call-based dependences on other parts. This will complement G-level constructs like $A_i \xrightarrow{calls} A_j$ which essentially express the *policy* of having dependences, with an actual model of system behaviour so that one can check one against the other, and do so both in the original system to ensure that the convention is indeed in place, as well as in the migrated system to verify that the migration did not violate it.

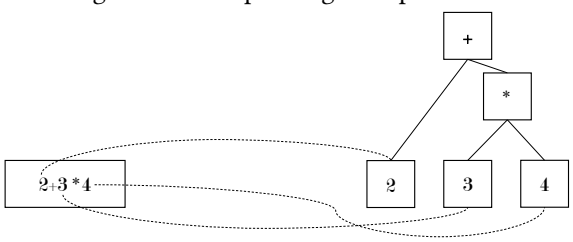
If we keep "zooming in" into all levels, then in the context of software migration the ultimate point of the A-level view would express the system under interest in terms of hardware architecture, and the ultimate point of the G-view would the syntax graph of the code of the system.

2.2 Parsing and Compilation Example

In the next scenario the A-view will only be needed on one side of the transformation. Many approaches tend to abstract from that fact and pretend to assign what they call semantics to the entire model while in fact assigning it to one side of the model.

There are many models of parsing such as parsing schemata by [41], that has the indisputable advantage over the alternatives by being independent from the actual algorithm used for the parsing process. The megamodel of parsing [51] is way too complex for us to reproduce here fully, but as admitted previously [50], it is possible to apply it partly to model simpler scenarios and possibly position them with respect to alternatives.

Let *parsing* be defined as an automated process of recognising structure in textual inputs and expressing it explicitly. So, on the left hand we have an input stream consisting of symbols of the chosen alphabet, and on the right we have an output graph which terminal elements (leaves) correspond to the relevant fragments of the input. The N-view and G-view are trivial for us after the previous section: the N-view shows that there is a parsing function transforming a string into a tree: $S \xrightarrow{parse} T$; and the G-view shows a concrete example of such a string and a corresponding example of a tree:



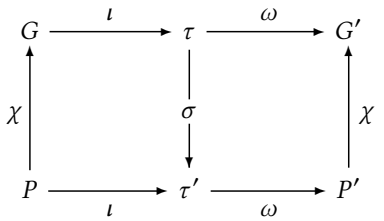
However, what is the A-view of them? How do we know what does it mean for these artefacts (a string and a tree) to be executed, to behave? There are several approaches to context handling and code generation and optimisation [19], that all more or less revolve around the classic concept of syntax-driven translation [1], which means that each element of the model that we have on the right, is mapped to a well-known and unit-testable executable pattern. This by definition means that the model on the left cannot possibly have semantics assigned to it, because its internal structure has not been recognised yet.

Hence, in these circumstances, the entity on the right is a full proper NGA multimodel which is simultaneously valid as a node, a graph and an automaton, but the entity on the left lacks the A-part. Depending on the purpose of modelling, one can assume that the A-level part of the left part of the megamodel is by definition the same as the A-level part on the right, but this agreement is only valid for valid mappings.

There is currently a very promising movement towards fully verified compilers: not just parsing, but also code generation, optimisation, memory allocation, internal representation soundness and completeness — everything can be formally specified in a language consumable by an automated theorem prover and verified mechanically. Great examples of such endeavours are CompCert [32], Vellvm [52], vanHelsing [33], RESOLVE [8], VerifiCard [42], Verisoft [31], Verifix [18]. For older projects (1967–2003), [9] kept a fairly comprehensive bibliography.

2.3 Cotransformation Example

The megamodels seen so far, were distilled to the simplest possible form, which on the N-level is obviously two nodes linked with a relation. Before we wrap up, let us consider a slightly more sophisticated scenario. This time it is borrowed from the domain of so-called cotransformations, or coupled transformations [7, 26]. In general, cotransformations are transformations that restore consistency to a model after some other transformations bring in changes that break it. Cotransformations happen when changes in a data schema need to be propagated to the data and to the query programs; or when the codebase needs to be updated to stay in sync when the language it is written in, evolves; or when files saved in an old format, need to be resaved in a the new one.



The diagram below prents an N-view of a nontrivial cotransformation setup. Let us renarrate it: a program P conforms to a grammar G, which is being transformed into a new grammar G'. The transformation τ from G to G', is being used by a higher order function σ that infers a deeply related cotransformation τ' which transforms P to P' in such a way that P' conforms to the new G'.

One may have already noticed two peculiarities of this megamodel with respect to previously shown ones. First, we "promote" τ from an annotation on an edge, to a proper vertex in the mgraph. This is done to explicitly emphasize that τ and τ' are different artefacts, as well to enable a non-cringy way to draw more edges incident to and from τ. This also means that the megamodel above can be seen as a G-level refinement of an N-level square with P, P', G and G', and the shown relations "input of" (ι) and "has output" (ω) contribute to the refinement of a τ from such a square. Second, each element of this megamodel, be it a node or an edge, is possible to refine deeper following the NGA paradigm. The main focus is on the actual cotransformation σ, but it is almost impossible to understand what it represents, without the other elements.

There are languages developed specifically to express transformations of grammars [28, 29, 47], so a G-view of τ could include its conformance to that language's definition and its element-of relation to the language itself. This is also a new way to refine transformations that we have not observed before: instead of writing out the correspondence relation, we make a proper high level model of it, and assign semantics to elements of that model — in the case of [28, 29] it can sentially language embedding, which is a perfect scenario for mega- and multimodelling.

The instance-level artefacts (P and P') are modelled similarly to S_i from § 2.1, and transformational artefacts (τ' and σ) are handled in a way similar to τ. The A-view of χ in practice is usually represented as a validator/checker tool that ensures conformance, but in theory it can reuse semantics (meaning-as-theory [40]) of the language in which P was written — it is a well-known fact in constructive mathematics that the choice of such a language determines the set of provable statements expressed in it.

3 Conclusion

In this paper, we have considered research crossroads between multimodelling [25] and megamodelling [5] and focused on tasks of constructing, using and maintaining models of complex information systems. For such models it is not uncommon to be represented hierarchically, denoting complex structural entities as mere dots in a bird's-eye view of a system. Our proposal is to explicitly acknowledge that elements of such megamodels are simultaneously nodes and graphs, and to add the third level representing behaviour. As extra motivation, we have provided proof of concept models of three concise case studies: legacy software migration, parsing in a compiler, and coupled transformations.

There are many directions to be investigated in the future: the faith of N-G models (such as the left one in § 2.2) and N-A models (executable artefacts with nontrivial semantics and unknown syntax); refinement of A-views (automata theory is insufficient: e.g., Mealy machines are obvious conceptual refinement of Moore machines since they can express inputs and outputs while Moore machines deal only with input, but they are equally expressive for automata theory since technically both are just edge annotations of finite automata); formalising $N \longrightarrow G$ refinement with air grammars [36] or triple grammars [39]; and so forth.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, Techniques and Tools.* Addison-Wesley.

[2] F. Allilaire, J. Bézivin, H. Brunelière, and F. Jouault. 2006. Global Model Management in Eclipse GMT/AM3. In *eTX at ECOOP'06.*

[3] M. Barbero, Frédéric Jouault, and J. Bézivin. 2008. Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In *ECBS.* 277–286.

[4] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. 2004. Modeling in the Large and Modeling in the Small. In *MDAFA'03/'04.* 33–46.

[5] J. Bézivin, F. Jouault, and P. Valduriez. 2004. On the Need for Megamodels. *MDSD at OOPSLA & GPCE* (2004).

[6] B. Boehm and B. Scherlis. 1992. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference.* Meridien Corp.

[7] A. Cleve, J. Henrard, and J.-L. Hainaut. 2005. Co-transformations in Information System Reengineering. *ENTCS* 137, 3 (2005), 5–15.

[8] C. T. Cook, H. K. Harton, H. Smith, and M. Sitaraman. 2012. Specification Engineering and Modular Verification using a Web-integrated Verifying Compiler. In *ICSE.* IEEE, 1379–1382.

[9] M. A. Dave. 2003. Compiler Verification: A Bibliography. *ACM SIGSOFT SE Notes* 28, 6 (2003), 2.

[10] Z. Diskin, R. Salay, B. Schätz, and V. Zaytsev. 2015. MMMDE: Workshop on Mathematical Models for MDE. *MoDELS* (2015).

[11] Z. Diskin, Y. Xiong, and K. Czarnecki. 2011. From State- to Delta-Based Bidirectional Model Transformations: The Asymmetric Case. *JOT* 10 (2011), 137–161.

[12] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. 2011. From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In *MoDELS (LNCS)*, Vol. 6981. Springer, 304–318.

[13] J.-M. Favre. 2004. Towards a Basic Theory to Model Model Driven Engineering. In *Third Workshop in Software Model Engineering (WiSME).*

[14] J.-M. Favre, R. Lämmel, and A. Varanovich. 2012. Modeling the Linguistic Architecture of Software Products. In *MoDELS (LNCS)*, Vol. 7590.

[15] J.-M. Favre and T. NGuyen. 2004. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS* 127, 3 (2004).

[16] D. Fischbein, N. D'Ippolito, G. Brunet, M. Chechik, and S. Uchitel. 2012. Weak Alphabet Merging of Partial Behavior Models. *ACM ToSEM* 21, 2 (2012), 9:1–9:47.

[17] D. Gašević, N. Kaviani, and M. Hatala. 2007. On Metamodeling in Megamodels. In *MoDELS (LNCS)*, Vol. 4735. Springer, 91–105.

[18] G. Goos and W. Zimmermann. 1999. Verification of Compilers. In *Correct System Design (LNCS)*, Vol. 1710. Springer, 201–230.

[19] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. 2012. *Modern Compiler Design.* Springer.

[20] R. Hebig, A. Seibel, and H. Giese. 2011. On the Unification of Megamodels. *EC-EASST* 42 (2011).

[21] M. Heinz, R. Lämmel, and A. Varanovich. 2017. Axioms of Linguistic Architecture. In *MODELSWARD'17.* SciTePress, 478–486.

[22] R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. 2010. Realizing Architecture Frameworks through Megamodeling Techniques. In *ASE.*

[23] R. Khadka, B. V. Batlajery, A. Saeidi, S. Jansen, and J. Hage. 2014. How Do Professionals Perceive Legacy Systems and Software Modernization?. In *ICSE.* ACM, 36–47.

[24] R. Khadka, P. Shrestha, B. Klein, A. Saeidi, J. Hage, S. Jansen, E. van Dis, and M. Bruntink. 2015. Does Software Modernization Deliver What it Aimed for?. In *ICSME.* IEEE, 477–486.

[25] H. König and Z. Diskin. 2016. Advanced Local Checking of Global Consistency in Heterogeneous Multimodeling. In *ECMFA (LNCS 9764).*

[26] R. Lämmel. 2016. Coupled Software Transformations—Revisited. In *SLE.* ACM, 239–252.

[27] R. Lämmel and A. Varanovich. 2014. Interpretation of Linguistic Architecture. In *ECMFA'14 (LNCS)*, Vol. 8569. Springer, 67–82.

[28] R. Lämmel and V. Zaytsev. 2009. An Introduction to Grammar Convergence. In *iFM (LNCS)*, Vol. 5423. Springer, 246–260.

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

[30] R. Lämmel and V. Zaytsev. 2013. Language Support for Megamodel Renarration. In *XM (CEUR)*, Vol. 1089. 36–45.

[31] D. Leinenbach and E. Petrova. 2008. Pervasive Compiler Verification — From Verified Programs to Verified Systems. *ENTCS* 217 (2008), 23–40.

[32] X. Leroy. 2009. Formal Verification of a Realistic Compiler. *CACM* 52, 7 (2009), 107–115.

[33] R. Lezuo, I. Dragan, G. Barany, and A. Krall. 2015. vanHelsing: A Fast Proof Checker for Debuggable Compiler Verification. In *SYNASC.*

[34] S. J. Mellor, K. Scott, A. Uhl, D. Weise, and R. M. Soley. 2004. *MDA Distilled: Principles of Model-Driven Architecture.* Addison-Wesley.

[35] OMG. 2006. *Meta-Object Facility (MOF^{TM}) Core Specification* (2.0 ed.). Object Management Group.

[36] T. W. Pratt. 1971. Pair Grammars, Graph Languages and String-to-graph Translations. *J. Comput. System Sci.* 5, 6 (1971), 560–595.

[37] R. Salay, M. Chechik, M. Famelis, and J. Gorzny. 2015. A Methodology for Verifying Refinements of Partial Models. 14, 3 (Aug. 2015), 3:1–31.

[38] R. Salay, S. Kokaly, A. DiSandro, and M. Chechik. 2015. Enriching Megamodel Management with Collection-based Operators. In *MoDELS.*

[39] A. Schürr. 1995. Specification of Graph Translators with Triple Graph Grammars. In *IWGT.* Springer, 151–163.

[40] E. Seidewitz. 2003. What Models Mean. *Software* 20, 5 (2003), 26–32.

[41] K. Sikkel. 1997. *Parsing Schemata — a Framework for Specification and Analysis of Parsing Algorithms.* Springer. I–XVI, 1–365 pages.

[42] M. Strecker. 2002. Formal Verification of a Java Compiler in Isabelle. In *CADE (LNCS)*, Vol. 2392. Springer, 63–77.

[43] A. Vignaga, F. Jouault, M. Bastarrica, and H. Brunelière. 2011. Typing Artifacts in Megamodeling. *SoSyM* (2011), 1–15.

[44] G. Wiederhold, P. Wegner, and S. Ceri. 1992. Toward Megaprogramming. *Commun. ACM* 35, 11 (Nov. 1992), 89–99.

[45] V. Zaytsev. 2011. Language Convergence Infrastructure. In *GTTSE'09 (LNCS)*, Vol. 6491. Springer, 481–497.

[46] V. Zaytsev. 2012. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In *SAC.* ACM, 1910–1915.

[47] V. Zaytsev. 2012. Language Evolution, Metasyntactically. *EC-EASST BX* 49 (2012).

[48] V. Zaytsev. 2012. Negotiated Grammar Transformation. In *XM.* ACM.

[49] V. Zaytsev. 2012. Renarrating Linguistic Architecture: A Case Study. In *MPM.* ACM, 61–66.

[50] V. Zaytsev. 2014. Understanding Metalanguage Integration by Renarrating a Technical Space Megamodel. In *GEMOC (CEUR)*, Vol. 1236.

[51] V. Zaytsev and A. H. Bagge. 2014. Parsing in a Broad Sense. In *MoDELS (LNCS)*, Vol. 8767. Springer, 50–67.

[52] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL.* ACM, 427–440.