Specifying Software Languages: Grammars, Projectional Editors, and Unconventional Approaches

Mikhail Barash
University of Bergen, Norway
mikhail.barash@uib.no

Abstract

We discuss several approaches for defining software languages, together with Integrated Development Environments for them. Theoretical foundation is grammar-based models: they can be used where proven correctness of specifications is required. From a practical point of view, we discuss how language specification can be made more accessible by focusing on language workbenches and projectional editing, and discuss how it can be formalized. We also give a brief overview of unconventional ideas to language definition, and outline three open problems connected to the approaches we discuss.

1 Introduction

Modern society depends on software as never before, and errors in software implementation can lead to drastic consequences. While business experts convey to software developers specifications of data, processes, and systems, developers' lack of domain knowledge and a frequent ambiguity of specifications pose a risk of defects in software. A solution to this are domain-specific languages (DSL) that offer concise programming notations inspired by business needs and thus improve communication between developers and domain experts. Implementing a software language—including a DSL—requires specifying its syntax, as well as static and dynamic semantics. While conceptually none of these present a particular challenge, developers often struggle to bring their languages to life. Reasons for obstacles range from the lack of knowledge on parser implementation to absence of an Integrated Development Environment (IDE) that is vital to adoption and success of a language. We discuss in this paper several approaches for defining software languages, together with IDEs for them.

Theoretical foundation is grammar-based models, which are discussed in Section 2. We discuss limitations of context-free grammars in defining syntax of programming languages, and give an overview of several their extensions—*Boolean grammars* [43], *parsing expression grammars* [28], and *grammars with contexts* [8, 9, 7]—that allow overcoming those limitations, while still having a manageable time complexity of parsing algorithms [44, 28, 49, 6, 10]. Section 3

This paper was presented at the NIK-2020 conference; see http://www.nik.no/.

takes a more practical look at how *cross-references*—that are vital for expressing static semantics in grammars—differ "in theory" (grammars with contexts) and "in practice", where a discussion on industrial tools for language definition follows. Such tools take form of *language workbenches* [29, 22, 25, 15], and they allow specifying both syntax, static semantics, and dynamic semantics of a language together with an IDE tailored for that very language. We then discuss *IDE-first* language specification, where the desired properties of an IDE drive the language definition process. In particular, we discuss *projectional editors* that allow overcoming the burden of implementing a parser for a language, and consequently allow for a much greater syntactic freedom than traditional text-based approaches [66, 65, 12, 68]. In Section 4, we discuss how language specification can be made more available for non-experts in software language engineering.

2 Grammars to Specify Syntax

Context-free grammars [16] are a standard model for defining syntax of programming languages [75, 76]: rule

$$\texttt{VarDecl} \rightarrow \text{“}\texttt{var}\text{”}\ \texttt{ident}$$

states that whenever there is a keyword `var` followed by an identifier, this sequence of strings is a variable declaration. Context-free grammars are capable of defining *structure* of programs, but fail to specify context conditions—or, *static semantics*—that every identifier should be declared before use, or that the number of formal parameters and actual arguments in a function call should agree. For example, the following Java block is *structurally* correct, though it contains undeclared identifiers:

```
{ int x; xx:=xxx; }
```

Floyd [27] considered a similar Algol program to show that Algol is not context-free and cannot be defined by a context-free grammar; the same applies to Java among all other languages that require identifiers to be declared before their use.

Subsequent new grammatical models appeared that enabled specifying desired context conditions [1]. *Two-level grammars* by van Wijngaarden were used to formally specify syntax and static semantics of Algol 68 [73, 63], though they, as well as many other models, turned out to be Turing-complete [74] and hence no practical parsing algorithms can exist for them.

Boolean operations in grammars

Drawing upon set-theoretic properties of formal languages, a systematic study of Boolean operations in grammars led to conjunctive and Boolean grammars [43, 59, 40]. Rule

$$S \rightarrow A \,\&\, B$$

defines all strings produced both by A and B, and rule

$$S \rightarrow A \,\&\, \neg B$$

defines all strings produced by A but not produced by B at the same time. Now rule

$$\texttt{ValidIdent} \rightarrow \texttt{ident}\ \ \&\ \ \neg\,\texttt{Keyword}$$

specifies in a most natural way that an identifier cannot coincide with a keyword. Conjunctive grammars can define non-trivial non-context-free languages, including copy language with central marker $\{wcw \mid w \in \Sigma^*\}$, $c \notin \Sigma$ which abstracts the condition of having each identifier declared before use.

Main parsing algorithms for context-free grammars, including cubic-time general parsing algorithm, Earley's algorithm, recursive descent and Generalized LR, have been extended to the case of conjunctive and Boolean grammars [45, 44, 43], have the same time complexity, and are implemented in a prototype parser generator [42]. Okhotin constructed a Boolean grammar to specify syntax and static semantics (including scoping rules) of a programming language [46]. This was apparently the first such specification by an efficiently parsable grammatical model. The programming language only had one data type and no approach of how to implement type checking was suggested.

Using contexts in grammars

Similarly to Boolean grammars, Ford's *parsing expression grammars* [28] provide predicates for conjunction (`&`) and negation (`!`). Those are positive and negative lookahead predicates, respectively. Predicate

`&BulletSymb`

succeeds if the next symbol of the input is a bullet, and predicate

`!BulletSymb`

succeeds if the next symbol is not a bullet. Neither of the predicates consume any input: they are only used to check the lookahead symbols in the input, and those lookahead symbols can be regarded as the right *context* of a string [33]. Drawing upon both parsing expression grammars and Boolean grammars, *grammars with contexts* [8] provide a built-in mechanism to specify what left and right contexts should be. The following informal rule states that whenever an identifier is used in a program, its declaration should appear to its left:

`ValidIdent` $\rightarrow$ `ident` $\;\&\lhd$ *it was declared before*

A symmetrical rule uses right context operator ($\rhd$).

Expressive power and applicability of grammars with contexts

Syntax and static semantics of a typed C-like programming language has been defined by a grammar with contexts [9]. It checks whether expression of `while` loop has type `bool`, types of formal parameters and actual arguments in function calls agree, type of expression in a `return` statement is the same as the returning type of a function, as well as other context conditions. The grammar also specifies scopes of visibility: identifiers are visible in the block where they are defined, and in all its inner blocks.

Several parsing algorithms, including cubic-time general parsing algorithm [8, 49], linear time recursive descent [10], and linear-to-cubic time Generalized LR [6], have been successfully extended to grammars with contexts. Importantly, these algorithms have the same time complexity as their context-free analogues, and have been implemented in a parser generator [11, 6].

Grammars with contexts are a purely syntactic formalism, and after a language is defined by a grammar, no additional code is required to further define its static semantics. Use of formal grammatical models—including grammars with contexts—for small sublanguages of larger programming languages (for example, a sublanguage of arithmetical expressions) is generally motivated by a desire to formally prove correctness of the definitions [37].

Definition of a typed programming language by a grammar with contexts mentioned above is heavily based on the copy language *wcw* that is used to check cross-references. This language is defined in a non-trivial manner, and would require a substantial modification to accommodate possible changes to how identifiers are defined in a language. This brings the following open problem [5].

Open Problem: How can the formalism of grammars with contexts be made more approachable? A solution of this problem is two-folded. One direction is to implement a "grammar development and debugging tool"[1] for grammars with contexts. This is feasible because all necessary parsing algorithms exist [6, 10], they were proven to have decent time complexity, and are implemented in a prototype parser generator [11].

A more general direction is to introduce an adequate and convenient formalism on top of grammars with contexts. Specifications in such simplified "front-end" formalism [63, 31] will be then translated to a grammar with contexts. For example, constructs of a language can be annotated with tags ("suffix numbers" [63, p. 37] [2]) to define cross-references, as shown in the rules below.

```
     NewIdent  :  ident  &  ! DeclaredIdent
DeclaredIdent  :  < 'var' ident.1 |  &  ident.1
```

In the rule for `DeclaredIdent`, both the ordinary (`ident.1`) and contextual (`< ... ident.1 |`) occurrences of `ident` have the same tag to express that these identifiers should be identical. That is, this rule expresses that "a declared identifier is any identifier that was declared before" in a natural way.

3 From Parsing to Projectional Editing

Grammars with contexts can define arbitrary *cross-references* within a string [7]. They have been used to specify an abstract programming language where identifiers can be declared before or after their use (example motivated by function prototypes in C) [7]. Ability to define cross-references is what makes grammars with contexts applicable to software languages. This ability is also distinctive in parser-based *language workbenches* [22], which are software development tools to define, reuse and compose programming languages together with their integrated development environments [20, 29].

Expressing cross-references

In language workbench *Eclipse Xtext* [25, 13], definition of a language starts with writing a grammar in a meta-language that is similar to that of ANTLR [48] and

[1] Such tool can take a form of a *language workbench*, as discussed in the next section.

supports cross-references.

```
  Variable  :  'var' name=ID;
Assignment  :  left=[Variable] '=' right=Expression;
```

The first rule defines construct *variable declaration*: it is keyword `var` followed by an identifier, which is "stored" in feature `name` associated with this rule. From a grammar, Xtext creates an object model in Eclipse Modeling Framework (EMF) [58]. In a simplified setting, rules become classes (instances of EMF Ecore `EClass`) and features of rules become fields of those classes. The model is then populated during the parsing, resulting in an AST that can be further analyzed or transformed by the user, for example, to define and validate static semantics of a language [13, 66], or to define a code generator via a model-to-text or a model-to-model transformation [58, 18].

Term `[Variable]` in the second rule above specifies a reference to an existing `Variable`, in particular, to its feature `name`. The reference is made to an instance of `EClass Variable`; if such an instance does not exist, an error is reported. If the instance exists, the reference is associated with the value of its field `name`. This is conceptually very similar to grammars with contexts [9], where `ValidVariable` would be used to specify the idea of what `[Variable]` expresses in Xtext. An important difference between the two approaches is that to perform static checking, Xtext mainly relies on imperative code that walks the AST, while grammars with contexts are a purely syntactic formalism and their correctness can be established in a formal way.

IDE-first language specification

Language definition task cannot be considered finished even when mechanisms to read, validate and execute programs written in a language are implemented. More and more software developers nowadays are accustomed to powerful integrated development environments (IDE) [66], such as Eclipse, Visual Studio, Apache NetBeans, IntelliJ IDEA, GitHub Atom, and others. Hence, adoption of a custom language in practice can be facilitated by a powerful IDE tailored to that particular language [66], which is expected to have language-specific services such as syntax-aware editing with syntax highlighting, code formatting and folding, code completion, code navigation and hyperlinking, code outlining, name refactoring, code version control, automatic code corrections and incremental syntax checking [26].

Apart from specifying the grammar, language specification in Xtext comprises of imperative code in a JVM language for describing the behaviour of the mentioned IDE services [13]. A full language specification in Xtext results in an Eclipse-based integrated development environment for that language, with the IDE services enabled. Alternatively, a language server for the language compliant to the Language Server Protocol [36] can be output by Xtext.

Another language workbench—*Metaprogramming System MPS*—while being conceptually similar to Eclipse Xtext, provides mechanisms to specify IDE services using the approach of *language-oriented programming* [72, 20, 67]. This approach comprises of dividing a complex task into subtasks, for each of which a dedicated domain-specific language (DSL) is created. The solution to a particular subtask is then expressed in that DSL. Thusly, language definition in MPS is viewed through

a prism of *aspects*—particular viewpoints on a language corresponding to different IDE services, such as editor, type system rules validator, or code generator. For each aspect, MPS provides a DSL which is then used by language engineer to specify the details of a language with respect to that aspect [15]. Similarly to Xtext, a language definition in MPS results in a powerful integrated development environment which is based upon industry-leading IntelliJ-platform with its extensive source-code-related capabilities [26].

Projectional editing in MPS: no parsing, free IDE

A distinctive feature of MPS is its projectional (structured) editor [15]. Unlike text-based editors where program is both stored and represented as text, in a projectional editor, key representation of a program is its abstract syntax tree. This AST can be projected into different representations: textual, tabular, and graphical [65]. As the user edits the abstract syntax tree directly, projectional editing does not require parsing—this enables *composition* of languages with conflicting syntaxes [23, 70]. Projectional editing also makes possible automatic language *migration*, as all existing code in a language is stored as an object model that can be automatically transformed to reflect the changes to the language itself. For the same reason, code generation in MPS is possible via graph-to-graph transformations [15, 18].

Traditional limitations of projectional editors [68, 12, 57, 30], such as linear input of arithmetical expressions instead of subtree-based input, has been addressed in MPS via *substitution actions*, which a language engineer defines on classes of subtrees of an AST [15, 69]. This approach results in an editing setting near in user experience to traditional text-based editors [69]. Many existing systems that support projectional editors [61, 47, 56, 39, 55, 21, 62] address the mentioned issues each anew; our hypothesis is that a formalism of projectional editing paradigm could unify development process of such systems. This suggests the following open problem.

Open Problem: How projectional editing be formalized? In a projectional editor, lines of code can be represented as a table with each token being located in its own *cell*. To combine several cells to form a line of code, a compound cell—horizontal collection—can be defined. Several lines of code will form a vertical collection; other types of cells can be defined, for example, indentation cells and so on [15].

Consider an arithmetic expression `2 + 3`, to be entered in an assumed projectional editor. As soon as the user types in integer constant `2`, the persisted AST is of the form `IntegerConst(2)`. In a projectional editor without substitution actions, the only way for the user to type in the entire desired expression would be to erase the already input expression `2` and to restart the process by typing in the trigger "`+`" of an addition expression. This would result in two placeholders for the left and right subexpressions: □ `+` □, which the user will then fill in with values `2` and `3`, respectively. Using substitution actions, after entering the original subexpression `2`, the user would type in *right transformation* trigger "`+`" which will automatically transform subtree `IntegerConst(2)` into a binary subtree `Addition(·, ·)`, where one placeholder is assigned to already entered subexpression `2`: `Addition(IntegerConst(2),` □`)`. To fill in the second placeholder, the user will continue typing in the right part of the expression. This process can be implemented by a language engineer in Metaprogramming System using a special imperative

DSL [15]; the process, however, has not been defined in formal way.

A possible formalization can be based upon *tree-adjoining grammars* (TAG) [32], which can be thought of a version of context-free grammars adopted for tree manipulation. Tree-adjoining grammars can be parsed in polynomial time using a generalization of Cocke-Younger-Kasami [32, 53], Earley [54] and LR [41] algorithms. Other formalisms, such as *tree grammars* [17], *graph grammars* [35, 52] or grammar systems based on *first-order logic* [34] can be considered as potential foundations to formally reason about the behaviour of projectional editors with the goal of proving their correctness. While an existing formalization of projectional editing [71] targets specifying the *behaviour of implementations* of projectional editors, our approach would be concerned with specifying the *structure* of the editors.

4 Low-Code Language Engineering

Engineering a programming language requires very versatile knowledge and software skills from a language developer, as it includes syntax specification, type system definition, code generation implementation, in addition to establishing behavior of IDE services. Some of the existing language workbenches have a very steep learning curve even for experienced software professionals [50].

Low-code development approach applied in software industry [14, 60] allows creating applications using a visual user interface in combination with model-driven logic, without or with a minimum of coding. This approach enables a wider range of persons to contribute to application development and can lower the cost of setup, training, deployment and maintenance of software [14]. A question arises of whether a similar approach can be used in language engineering, thusly making the latter more accessible. Our hypothesis [4] is that one of the key factors that inherently makes language engineering non-trivial is the prevalence of *meta*-definitions, with language definition tools (such as grammars and language workbenches) themselves being on the *meta-meta*-level.

We suggest an approach [4] to use a word processor to define languages in *example-driven way*: a language is defined by giving examples of code written in it, which are then annotated to specify abstract syntax, formatting rules, dynamic semantics, and so on. Such a definition can then be used to validate similar documents and to generate an API for processing models. Alternatively, it can serve as a front-end and later be transformed to an (equivalent) definition in a language workbench. We imagine a similar approach for language definition using of a dedicated language definition tool [3]. In such a tool, a language is defined by giving examples of code written in it using illustrative syntax definition [3]. These examples are then annotated to specify different concerns of language definition—abstract syntax, typing rules, validation rules, formatting rules, and dynamic semantics.

With the goal of further significantly simplifying language specification and implementation, one can draw ideas from other fields, for example, natural language processing or human-computer interaction. We can thus formulate the following generic open problem.

Open problem: How can unconventional approaches to language definition make it more accessible? We give a brief overview of possible research directions.

Utilizing chat bots for language definition can be based on preliminary results in [51] that discuss creation of UML diagrams via command-like imperative messages given by a user. Building on these ideas, using a controlled natural language [24] for software language definition presents an interesting question. This approach can be applied to different aspects of language specification, including annotating examples of code (as discussed in above). Work by Cabot et al. [19], who give an example of a DSL to extract variable data from textual fragments, seems a possible starting point.

Recognizing hand-drawn UML diagrams [38] can be extended to recognizing specifications of projectional editors. Visual representation of such specifications can be further enhanced using augmented reality [64].

Yet another possible approach to specify (syntax of) software languages is by using tabular notation in a spreadsheet calculator. This could significantly lower the entry barrier for language engineering, due to popularity of spreadsheet software.

5 Conclusion

Specifying syntax of software languages presents both theoretical and practical interest, and the ultimate goals are to understand better the fundamental principles behind programming language definition in a broad sense, and to make language definition process approachable for non-experts in language engineering. Giving a more practical flavour to grammatical models, and trying to find a formalization of practice-driven approaches to language definition might be an enabler of a sought common ground between the two worlds.

Acknowledgments

The author is thankful to Magne Haveraaen for discussions on earlier versions of the paper.

References

[1] A. V. Aho, *Indexed Grammars – An Extension of Context-Free Grammars*, J. ACM 15(4). 647–671. 1968.

[2] *American National Standard COBOL*. American National Standards Institute. 1975.

[3] M. Barash, *Example-Driven Software Language Engineering*, In Proceedings of the 13th International Conference on Software Language Engineering (SLE), to appear. 2020.

[4] M. Barash, *Enabling Language Engineering for the Masses*, ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion). 2020.

[5] M. Barash, *A New Life for Legacy Language Definition Approaches?*, In STAF Workshop Proceedings (STAF), to appear. 2020.

[6] M. Barash, A. Okhotin, *Generalized LR Parsing Algorithm for Grammars with One-Sided Contexts*, Theory Comput. Syst. 61(2). 581–605. 2017.

[7] M. Barash, A. Okhotin, *Two-sided context specifications in formal grammars*, Theor. Comput. Sci. 591. 134–153. 2015.

[8] M. Barash, A. Okhotin, *An extension of context-free grammars with one-sided context specifications*, Inf. Comput. 237. 268–293. 2014.

[9] M. Barash, *Programming language specification by a grammar with contexts*, NCMA 2013. 51–67.

[10] M. Barash, *Recursive descent parsing for grammars with contexts*, SOFSEM 2013. Local Proceedings II. 10–21. 2013.

[11] M. Barash, *Parser generator with grammars with right contexts*. 2012. Available at: `http://users.utu.fi/mikbar/gwrc/`.

[12] T. Berger, M. Voelter, H. P. Jensen, T. Dangprasert, J. Siegmund, *Efficiency of projectional editing: a controlled experiment*. SIGSOFT FSE 2016: 763–774.

[13] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*, Packt Publishing. 2016.

[14] J. Cabot, *Low-code platforms to build smart apps*, Available at: `https://modeling-languages.com/low-code-platforms-smart-apps-ai/`.

[15] F. Campagne, *The MPS Language Workbench, Vol. 1*. CreateSpace Independent Publishing Platform. 2014.

[16] N. Chomsky, *A Note on Phrase Structure Grammars*, Information and Control 2(4). 393–395. 1959.

[17] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, *Tree Automata Techniques and Applications*. 2007.

[18] K. Czarnecki, U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional. 2000.

[19] G. Daniel, J. Cabot, L. Deruelle, M. Derras, *Xatkit: A Multimodal Low-Code Chatbot Development Framework*, IEEE Access 8. 15332–15346. 2020.

[20] S. Dmitriev, *Language Oriented Programming: The Next Programming Paradigm*. 2004. Available at: `http://www.onboard.jetbrains.com/articles/04/10/lop/`.

[21] O. Dvorak, *Projectional editor for domain-specific languages*, thesis, Charles University in Prague. 2013.

[22] S. Erdweg, T. van der Storm, M. Voelter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, *Evaluating and comparing language workbenches: Existing results and benchmarks for the future*. Comput. Lang. Syst. Struct. 44: 24–47. 2015.

[23] S. Erdweg, P. G. Giarrusso, T. Rendel, *Language composition untangled*, LDTA 2012. 7.

[24] M. D. Ernst, *Natural Language is a Programming Language: Applying Natural Language Processing to Software Development*, SNAPL 2017. 4:1–4:14.

[25] M. Eysholdt, J. Rupprecht, *Migrating a large modeling environment from XML/UML to Xtext/GMF*, SPLASH/OOPSLA 2010. 97–104. 2010.

[26] D. K. Fields, S. Saunders, E. Belyaev, *IntelliJ IDEA in Action*, Manning. 2006.

[27] R. W. Floyd, *On the nonexistence of a phrase structure grammar for ALGOL 60*, Commun. ACM 5(9). 483–484. 1962.

[28] B. Ford, *Parsing expression grammars: a recognition-based syntactic foundation*, POPL 2004. 111–122. 2004.

[29] M. Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?* Available at: `https://www.martinfowler.com/articles/languageWorkbench.html`.

[30] S. M. Guttormsen, A. Prinz, T. Gjøsæter, *Consistent Projectional Text Editors*. MODELSWARD 2017. 515–522.

[31] Q. T. Jackson, *Efficient formalism-only parsing of XML/HTML using the s-calculus*, SIGPLAN Notices 38(2). 29–35. 2003.

[32] A. K. Joshi, Y. Schabes, *Tree-Adjoining Grammars*, Handbook of Formal Languages (3). 69–123. 1997.

[33] S. M. Kearns, *Extending Regular Expressions with Context Operators and Parse Extraction*, Softw., Pract. Exper. 21(8). 787–804. 1991.

[34] R. Kowalski, *Logic for Problem Solving*, North-Holland, Amsterdam. 1979.

[35] H.-J. Kreowski, G. Rozenberg, *On structured graph grammars. I.*, Inf. Sci. 52(2). 185–210. 1990.

[36] Language Server Protocol Specification. Available at: https://microsoft.github.io/language-server-protocol/.

[37] R. Lämmel, *Grammar Testing*, ETAPS 2001. 201–216. 2001.

[38] E. Lank, J. S. Thorley, S. Jy-Shyang Chen, *An interactive system for recognizing hand drawn UML diagrams*, CASCON 2000.

[39] D. H. Lorenz, B. Rosenan, *Cedalion: a language for language oriented programming*, OOPSLA 2011. 733–752.

[40] A. Megacz, *Scannerless Boolean Parsing*, Electr. Notes Theor. Comput. Sci. 164(2). 97–102. 2006.

[41] M.-J. Nederhof, *An Alternative LR Algorithm for TAGs*, COLING-ACL 1998. 946–952.

[42] A. Okhotin, *Whale Calf, a Parser Generator for Conjunctive Grammars*, CIAA 2002. 213–220.

[43] A. Okhotin, *Boolean grammars*, Inf. Comput. 194(1), 19–48. 2004.

[44] A. Okhotin, *Generalized LR Parsing Algorithm for Boolean Grammars*, Int. J. Found. Comput. Sci. 17(3). 629–664. 2006.

[45] A. Okhotin, *Recursive descent parsing for Boolean grammars*, Acta Inf. 44(3-4). 167–189. 2007.

[46] A. Okhotin, *On the existence of a Boolean grammar for a simple programming language*, AFL 2015.

[47] K. Osenkov, *Designing, implementing and integrating a structured C# code editor*, thesis, Brandenburg University of Technology. 2007.

[48] T. Parr, K. Fisher, *LL(*): the foundation of the ANTLR parser generator*, PLDI 2011. 425–436. 2011.

[49] M. Rabkin, *Recognizing Two-Sided Contexts in Cubic Time*, CSR 2014. 314–324. 2014.

[50] D. Ratiu, V. Pech, K. Dummann, *Experiences with Teaching MPS in Industry: Towards Bringing Domain Specific Languages Closer to Practitioners*. MoDELS 2017: 83–92.

[51] R. Ren, J. W. Castro, A. Santos, S. Pérez-Soler, S. Acuna, J. de Lara, *Collaborative modelling: chatbots or on-line tools? An experimental study*, EASE 2020.

[52] B.K. Rosen, *Deriving Graphs from Graphs by Applying a Production*, Acta Informatica 4(4). 337–357. 1975.

[53] G. Satta, *Tree-Adjoining Grammar Parsing and Boolean Matrix Multiplication*, Computational Linguistics 20(2). 173–191. 1994.

[54] Y. Schabes, A. Joshi, *Parsing with Lexicalized Tree Adjoining Grammar*, in: M. Tomita, Current Issues in Parsing Technology, Springer. 25–47. 1991.

[55] C. Simonyi, M. Christerson, S. Clifford, *Intentional software*, OOPSLA 2006. 451–464.

[56] R. Solmi, *Whole Platform*. Ph.D. thesis, Università di Bologna e Padova. 2005.

[57] F. Steimann, M. Frenkel, M. Voelter, *Robust Projectional Editing*, SLE 2017: 79–90.

[58] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, *EMF – Eclipse Modeling Framework*, Addison-Wesley. 2008.

[59] A. Stevenson, J. R. Cordy, *Parse views with Boolean grammars*, Sci. Comput. Program. 97. 59–63. 2015.

[60] M. Henkel, J. Stirna, *Pondering on the Key Functionality of Model Driven Development Tools: The Case of Mendix*, BIR 2010. 146–160.

[61] Structured Editing in Dark Language. 2020. Available at: `https://darklang.github.io/docs/structured-editing`.

[62] F. Tomassetti, M. Torchiano, *PrEdE: a Projectional Editor for the Eclipse Modeling Framework*. 2011.

[63] J. C. Cleaveland, R. C. Uzgalis, *Grammars for Programming Languages*, Elsevier. 1977.

[64] B. Victor, *Humane representation of thought: a trail map for the 21st century*, UIST 2014. 699.

[65] M. Voelter, S. Lisson, *Supporting Diverse Notations in MPS' Projectional Editor*. GEMOC@MoDELS 2014: 7–16.

[66] M. Voelter, S. Benz, Ch. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, G. Wachsmuth, *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages*. 2013.

[67] M. Voelter, *Fusing Modeling and Programming into Language-Oriented Programming – Our Experiences with MPS*. ISoLA (1) 2018: 309–339.

[68] M. Voelter, J. Siegmund, T. Berger, B. Kolb, *Towards User-Friendly Projectional Editors*. SLE 2014: 41–61.

[69] M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, Th. Berger, *Efficient development of consistent projectional editors using grammar cells*, SLE 2016. 28–40.

[70] M. Voelter, *Language and IDE Modularization and Composition with MPS*, GTTSE 2011. 383–430.

[71] E. Walkingshaw, K. Ostermann, *Projectional editing of variational software*, GPCE 2014. 29–38.

[72] M. P. Ward, *Language Oriented Programming*, Software – Concepts and Tools 15. 147–161. 1995.

[73] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisker, *Revised Report on the Algorithmic Language ALGOL 68*, Acta Inf. 5. 1–236. 1975.

[74] A. van Wijngaarden, *The Generative Power of Two-Level Grammars*, ICALP 1974. 9–16.

[75] V. Zaytsev, *Grammar Zoo: A corpus of experimental grammarware*, Sci. Comput. Program. 98. 28–51. 2015.

[76] V. Zaytsev, *BNF Was Here: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions*, SAC 2012.