Enabling Language Engineering for the Masses

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

ABSTRACT

While language workbenches—tools to define software languages together with their IDEs—are yet to become ubiquitous in industry, a noticeable amount of domain modeling is still done using word processors and spreadsheet calculators. We suggest an approach to use a word processor to define (modeling) 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 be used to validate similar documents and to generate an API for processing models, or it can serve as a front-end and later be transformed to an (equivalent) definition in a language workbench. We discuss how a similar approach for language definition can be implemented in a form of a language workbench.

CCS CONCEPTS

• **Software and its engineering** → **Context specific languages**; **Programming by example**; **Integrated and visual development environments**; *Syntax*; • **Applied computing** → *Annotation*; *Document management and text processing*.

KEYWORDS

example-driven, implicit modeling, Microsoft Word, workbench

ACM Reference Format:
Mikhail Barash. 2020. Enabling Language Engineering for the Masses. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3417990.3420051

1 INTRODUCTION

Implementing a software language—be it a modeling language or a programming language—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 tool support for the language—most commonly, users expect an Integrated Development Environment (IDE) to be shipped together with a language, which is vital for adoption of a language [10].

Recently, an array of tools designed to define software languages together with their IDEs have appeared under the name of *language workbenches* [3, 4, 10]. These tools allow specifying syntax, typing rules, and code generators for a language, and they output a tailored IDE with standard services such as syntax-aware editor, code completion, automatic code corrections, and others. Despite this fact, many of current language workbenches have a very steep learning curve even for experienced software professionals [8], and their adoption rate still leaves much to be desired. At the same time, a noticeable amount of domain modeling is done using "office software", such as word processors [9] and spreadsheet calculators [1].

Our hypothesis is that *meta*-definitions—that are prevalent in language workbenches—is a key factor that complicates language engineering for beginner language engineers. Consider the following *definition* of construct *variable declaration*: `var x = 10`. This definition looks like an *instance* of a variable declaration, and no meta-definition along the lines of `VarDecl` ⟶ "`var`" `ident` "`=`" `expr` is explicitly mentioned.

We suggest an approach where a language is defined by giving examples of code written in it, which are then *annotated* to specify different concerns of language definition—such as abstract syntax, typing rules, validation rules, formatting rules, and dynamic semantics. We consider three possible implementation strategies of this approach: within an existing *rich text processor* (such as Microsoft Word), within a *notebook* (similar to Jupyter notebooks [6]), and within a dedicated *example-driven language workbench*.

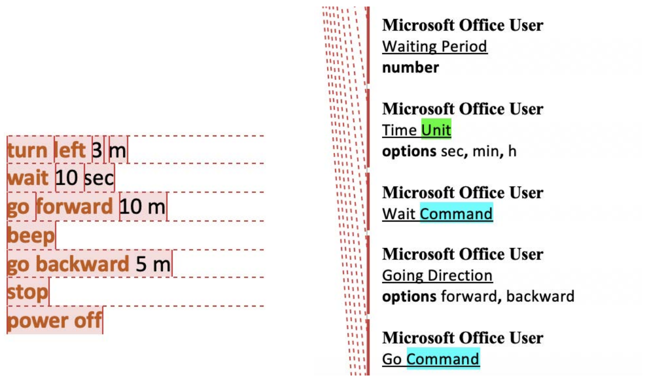
2 DEFINING LANGUAGES IN MICROSOFT WORD

Using numerous formatting styles of Microsoft Word and a possibility to add review comments to text fragments, one can communicate various aspects of language definition in an example-driven way.

Consider an assumed robot control language, and its construct *wait statement*, as shown in Fig. 1. After typing `wait 10 sec`, the user would select this line and add a review comment stating that it constitutes `Wait Command`. The user can then assign style *keyword* to `wait`, and annotate `10` as a feature `Waiting Period` which is required to be of type `number`. The remaining text `sec` can be annotated as `Time Unit`, and possible values of this feature can be listed.

We imagine that from such a language definition, an API for processing the model will be generated; candidate target languages are Visual Basic for Applications and JavaScript[1]. The language definition can also be used to validate other Microsoft Word documents, and to provide domain-level error messages to the users.

[1] *Microsoft Word JavaScript API overview.* Available at: https://docs.microsoft.com/en-us/office/dev/add-ins/reference/overview/word-add-ins-reference-overview.



turn left 3 m
wait 10 sec
go forward 10 m
beep
go backward 5 m
stop
power off

Microsoft Office User
Waiting Period
number

Microsoft Office User
Time Unit
options sec, min, h

Microsoft Office User
Wait Command

Microsoft Office User
Going Direction
options forward, backward

Microsoft Office User
Go Command

Figure 1: Example-driven definition of a language in Microsoft Word: *(left)* rich text with review comments defining various annotations; *(right)* comments that define constructs `Wait` and `Go` of the language.

Alternatively, the definition can be converted to a language definition in a language workbench [3], and thus only serve as a front-end accessible to domain experts not familiar with modeling tools.

We imagine that implementing a dedicated Microsoft Word plug-in would simplify and guide the language definition process for end-users; for example, inconsistent or otherwise invalid specifications of language constructs can be reported.

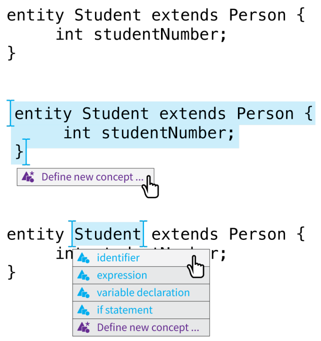
It is worth noting that support of textual, tabular, mathematical, and graphical notations, as well as other rich text-processing features, make Microsoft Word apparently one of the most powerful *projectional (structured) editors* [5]. Voelter et al. [10] discuss benefits of projectional editing in software language engineering; state-of-the-art language workbench JetBrains MPS [3] uses projectional editing.

3 A DEDICATED TOOL

Developing a stand-alone tool (a language workbench) that would allow defining languages in an example-driven way is another possibility to implement our approach. Consider how construct *entity* of an assumed *Entity Language* could be defined in such a tool. First, an instance of an entity is input as simple text, as shown in Fig. 2*(top)*. To communicate that this piece of code represents an *entity*, the user selects the code and, using a context menu, creates a new construct *entity*, as shown in Fig. 2*(center)*. One must then define the abstract syntax of this construct; this is done by further selecting the respective pieces of code and assigning constructs to them, as shown in Fig. 2*(bottom)*. We imagine similar, annotations-based, ways to introduce constraints and other requirements.

Another alternative implementation of our approach can have form of notebooks [7]; mixing formatted comments with examples of code in the language being defined could serve as a way to document languages.

In all the three implementation approaches, it should be possible to specify dynamic semantics—also using annotations—that is, code in target language to be generated from the code examples in the language being defined.



```
entity Student extends Person {
     int studentNumber;
}
```

```
entity Student extends Person {
     int studentNumber;
}
```

```
entity Student extends Person {
     in
}
```

Figure 2: Example-driven definition of concrete and abstract syntax.

4 RELATED WORK

Example-driven modeling [2] uses both explicit examples and abstractions to model complex business knowledge. Abstractions are synthesized from a set of examples [2], while in our approach, the language engineer annotates the examples of constructs and thus defines the abstract syntax only implicitly.

REFERENCES

[1] S. Adam and U. P. Schultz. 2015. Towards Tool Support for Spreadsheet-Based Domain-Specific Languages. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*. Association for Computing Machinery, New York, NY, USA, 95–98. https://doi.org/10.1145/2814204.2814215

[2] K. Bak, D. Zayan, K. Czarnecki, M. Antkiewicz, Z. Diskin, A. Wasowski, and D. Rayside. 2013. Example-driven modeling: model = abstractions + examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl (Eds.). IEEE Computer Society, 1273–1276. https://doi.org/10.1109/ICSE.2013.6606696

[3] 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, and J. van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Comput. Lang. Syst. Struct.* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[4] M. Eysholdt and J. Rupprecht. 2010. Migrating a large modeling environment from XML/UML to Xtext/GMF. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, W. R. Cook, S. Clarke, and M. C. Rinard (Eds.). ACM, 97–104. https://doi.org/10.1145/1869542.1869559

[5] M. Fowler. [n. d.]. Projectional Editing. https://martinfowler.com/bliki/ProjectionalEditing.html

[6] Th. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and et al. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

[7] M. V. Merino, J. J. Vinju, and T. van der Storm. 2020. Bacatá: Notebooks for DSLs, Almost for Free. *Art Sci. Eng. Program.* 4, 3 (2020), 11. https://doi.org/10.22152/programming-journal.org/2020/4/11

[8] D. Ratiu, V. Pech, and K. Dummann. 2017. Experiences with Teaching MPS in Industry: Towards Bringing Domain Specific Languages Closer to Practitioners. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*. IEEE Computer Society, 83–92. https://doi.org/10.1109/MODELS.2017.15

[9] M. Voelter. [n. d.]. The Business DSL: Zurich Insurance. https://blogs.itemis.com/en/the-business-dsl-zurich-insurance

[10] M. Voelter, S. Benz, Ch. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org. 1–558 pages.