Example-Driven Software Language Engineering

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

Abstract

Language workbenches—tools to define software languages together with their IDEs—are designed to simplify language engineering and implementation: they free language engineers from many meticulous tasks, but oftentimes have a very steep learning curve even for experienced software professionals. With the assumption that meta-definitions are one of the key factors that hinder language engineering, we introduce an example-driven approach to language definition. We describe in this paper our vision of a web-based tool aimed at beginner language engineers, and list possible requirements for such a tool. A language is defined by giving examples of code written in it using illustrative syntax definition. These examples are then annotated to specify different concerns of language definition—abstract syntax, typing rules, validation rules, formatting rules, and dynamic semantics.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; **Syntax**; **Context specific languages**; **Programming by example**.

Keywords: example-driven, illustrative syntax definition, transformations, language engineering, implicit modeling

ACM Reference Format:
Mikhail Barash. 2020. Example-Driven Software Language Engineering. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3426425.3426945

1 Introduction

Engineering a software language, be it a programming language or a modeling language, requires versatile knowledge and software development skills from a language developer, as they have to define its syntax, static and dynamic semantics, as well as implement a tailored Integrated Development Environment (IDE), which is vital for successful adoption of the language [43]. Over a decade ago, tools to define software languages together with their IDEs have appeared under the name of *language workbenches* [13, 15]; many of them still have a very steep learning curve even for experienced software professionals [34]. Our hypothesis [4] is that one of the key factors that inherently makes language engineering intricate is the prevalence of *meta*-definitions, with language definition tools (such as grammars and language workbenches) themselves being on the *meta-meta*-level.

We describe in this paper our vision of a web-based tool—*Language Wheel*—aimed at beginner language engineers, and list possible requirements for such a tool. A language is defined by giving examples of code written in it using *illustrative syntax definition*. These examples are then *annotated* to specify different concerns of language definition—abstract syntax, typing rules, validation rules, formatting rules, and dynamic semantics.

Such a definition mechanism can serve as a *front-end language workbench*, whose output is a language definition in another language workbench [13], such as, for example, Eclipse Xtext [5, 14], Monticore [21], Spoofax [23], or JetBrains MPS [8]. An alternative possibility is to output a tailored editor—be it textual (e.g., Monaco[1] or Ace[2]) or projectional [18, 46])—and a code generator for a language, or, depending on the language, a CRUD application. Yet another possibility is to output a language server protocol (LSP) instance [10] for the language defined in Language Wheel.

2 Defining Languages in Language Wheel

We describe below how we imagine the language definition process in Language Wheel.

2.1 Illustrative Syntax Definition

Unlike other tools that require a language engineer to define a metamodel first, Language Wheel takes an example-driven approach. Our hypothesis is that it will be easier for beginner language engineers to start prototyping a language by writing down several samples of code written in it, rather than by directly devising a metamodel.

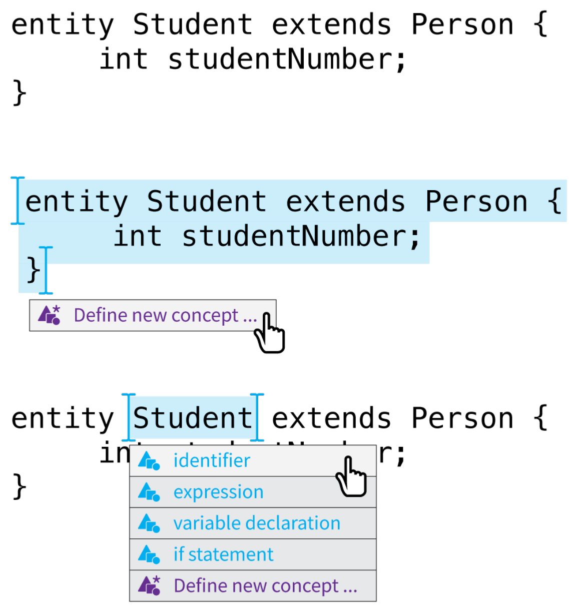
Definition of a language construct in an *illustrative* way is comprised of the following three concerns:

SLE '20, November 16–17, 2020, Virtual, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8176-5/20/11...$15.00
https://doi.org/10.1145/3426425.3426945

[1]`https://microsoft.github.io/monaco-editor/`
[2]`https://ace.c9.io`

- *example* (*instance*), defining concrete syntax;
- *annotation*, that defines several *perspectives*—abstract syntax, formatting, validation rules, type rules; and
- *implementation*, that specifies dynamic semantics.

Consider how concept *entity* in an assumed Entity Language [5] can be defined. First, an instance of an entity is input by a language engineer as simple text, as shown in Fig. 1*(top)*. To communicate that this piece of code represents an *entity*, the user selects the code and, using a context menu, creates a new concept *entity*, as shown in Fig. 1*(center)*. One must then define the abstract syntax of this concept, namely, features *name*, *parent*, and *fields*. This is done by further selecting the respective pieces of code and assigning concepts to them: features *name* and *parent* are *identifiers*[3], and feature *fields* is a *field declaration*, as shown in Fig. 1*(bottom)* and 2.



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

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

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

Figure 1. Illustrative definition of concrete and abstract syntax: *(top)* code sample without annotations; *(center)* annotating the example as a new concept *entity declaration*; *(bottom)* further annotating the example by specifying feature *name* of type *identifier*.

Using further annotation perspectives, one can require that the identifier in feature *name* should not have been declared before, while identifier in feature *parent* should form a cross-reference to an already defined *entity*[4]. Yet another annotation perspective is used to require cardinality of feature *fields* to be $0..9$, as shown in Fig. 2. We discuss such *constraints* on features [38] in more details in Sect. 2.2.

Different annotation perspectives are used to define other properties of concepts, such as syntax highlighting and formatting rules. Fig. 2 showcases a formatting annotation for feature *fields* of concept *entity*: each element should be prepended with an indentation, and elements are separated using a newline character.

Similarly to concept *entity*, one can define concept *field declaration* by giving its instance `var myNumber = 10 + 20;`, as shown in Fig. 6*(top, center)*. It is important to note that the language engineer is not expected to operate with metadefinitions along the lines of `FieldDecl` → "`var`" `ident` "`=`" `expr` "`;`" to define concepts. Our hypothesis is that this might enable non-language-engineers—perhaps even domain experts—to define (simple) languages.

One can define several examples of code for a concept. In that case, annotations shown for each of the examples shall reflect properties contained in annotations defined in all other examples of the concept, thus giving a holistic picture of a concept in each of its instances. One can also define multiple syntaxes (called *projections*) for a concept [8, 37]: for example, one can be verbose and aimed at beginner-users of the language, while another one can be succinctly designed with experienced users in mind [43]. The end-user of the language should be able to switch between different notations, in a way similar to existing projectional editors [8, 37, 43, 44].

A language engineer will be issued a warning if the examples for one concept are *conflicting* each other. For example, in one example of concept *entity*, the concrete syntax might use keyword `extends` before feature *parent* (see Fig. 2), while in another example, a colon might be used. This will be perceived by Language Wheel as either two possible syntaxes for concept *entity*, or a conflict in the examples. By default, the first option will be assumed, but a warning still will be issued.

To keep language definition *consistent* across concerns—be it concrete or abstract syntax, formatting rules, type system definition, validation rules, and so on—a *flavor* can be assigned to a language. Flavor is then used by Language Wheel to guide a language engineer and verify their language definitions against "best practices" specified in the flavor. One can imagine the following practices to be enforced in a Java-like flavor: in terms of abstract syntax—there should be no unconventional restrictions on cardinality of features (e.g., $3..17$); in terms of concrete syntax—keywords should not contain special characters, concept *identifier* should be defined as in Java, statements should have an ending semicolon,

[3]Concepts *identifier* and *expression* are built-in.

[4]We imagine similar, annotations-based, ways to define scopes of visibility, qualified names, and so on, by giving a language engineer a possibility to follow the patterns met in standard general-purpose languages. To help a language engineer navigate in using different annotation perspectives, we imagine Language Wheel guiding the language engineer via context menus ("intentions") with possible actions that might be taken (e.g., "specify cardinality", "specify whether this identifier should refer to an existing identifier", etc.) It should also be possible to toggle the visibility of the annotation perspectives (similarly to how layers in graphical editing tools can be shown and hidden), to allow a language engineer focus on a particular aspect of the language definition.

qualified names should use dot as a separator; in terms of formatting—indentations should be used inside blocks; in terms of validations—identifiers should follow camel case; in terms of type systems—there should be rules for typecasting; and so on. We imagine that Language Wheel will have a set of predefined flavors, and users will be able to define their own flavors to be assigned to their languages. One can also imagine *orthogonality* checking of a language definition.

Besides specifying "complete" code examples, a language engineer can mark a particular *fragment* of an example to be a definition of a concept (rather than the "complete" example itself). For instance, one can define concept *assignment statement* within an *entity* declaration, and mark only the code corresponding to that assignment statement. The parts of code representing an *entity* would then be disregarded as they only provide a *context* for convenience of the language engineer. The same idea applies for a definition of concept *entity*, where feature *parent* should be an existing entity: for this, it suffices to define a "phantom" parent *entity* (`entity Person {}`) within the example in Fig. 2, but only mark the code for *entity* `Student` as the definition[5]. Example fragments as described here are similar to *template fragments* of code generators in JetBrains MPS [8].

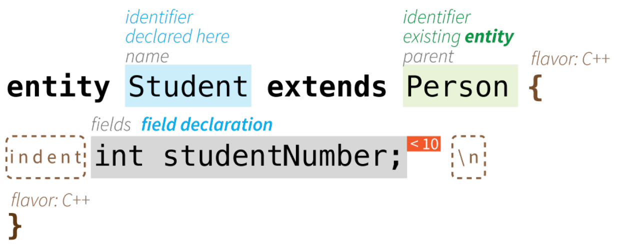
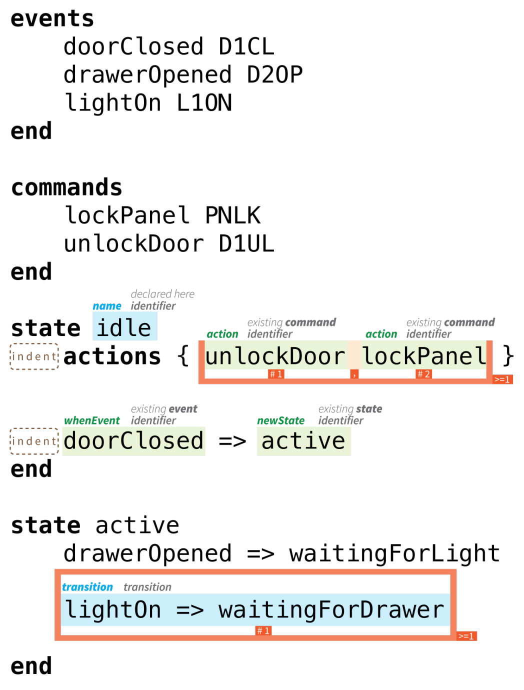

```
entity Student extends Person {
    int studentNumber;
}
```

Figure 2. Several annotation perspectives for concept *entity*: feature *name* of an entity is an identifier that was not declared before; feature *parent* is a cross-reference to an existing entity; the cardinality of feature *fields* is defined as 0..9; formatting of each *field declaration* in *fields* requires an indentation; field declarations are separated by a newline character; curly braces are defined by language flavor and can be changed should a different flavor be chosen.

2.2 Defining Validations

Constraint checks allow users of the language to get notifications about inconsistencies in the code and prevent illegal code from being executed or processed further. It is a task of a language engineer to define such constraint checks and anticipate possible fixes of issues in the code. In the definition of concept *entity*, an example of a constraint is that the identifier in feature *parent* should refer to an already existing

[5]Without using example fragments, the parent *entity* `Person` would be defined as a separate example of concept *entity*.


```
events
    doorClosed D1CL
    drawerOpened D2OP
    lightOn L1ON
end

commands
    lockPanel PNLK
    unlockDoor D1UL
end

state idle
    actions { unlockDoor lockPanel }
    doorClosed => active
end

state active
    drawerOpened => waitingForLight
    lightOn => waitingForDrawer
end
```

Figure 3. An example code for Miss Grant's Controller [17].

entity, and not just be any identifier; the definition of this cross-reference constraint is shown in Fig. 2.

A language engineer can define methods that perform arbitrary custom checks on the abstract syntax of the language; such methods are called *validations* [5]. To facilitate their definition, we envision identifying possible frequent validations and allowing language engineers to use them in their own languages "out-of-the-box". An example of such "pluggable" validation is restricting a token by an enumeration whose values are either fixed, or are taken from a database or via REST API, as shown in Fig. 4. Another example is giving language engineers freedom to define arbitrary cardinality of features, rather than restricting to standard options (1, 0..1, 0..∗, 1..∗); this is showcased in Fig. 2.

We imagine that Language Wheel supports an additional way to specify constraints—by giving avoidable (erroneous) examples and specifying the reason for a badness in the form of a Boolean condition, as shown in Fig. 5. Based on such avoidable examples, a set of validation rules can be formed automatically.

For each of the validations, either predefined or user-defined, it is possible to define a *quickfix*, which is a proposal to solve an issue in the code; quickfixes are typically implemented as a context menu that appears near the error marker [5]. As in the case of validations, we envision providing a language engineer with a set of "pluggable" quickfixes

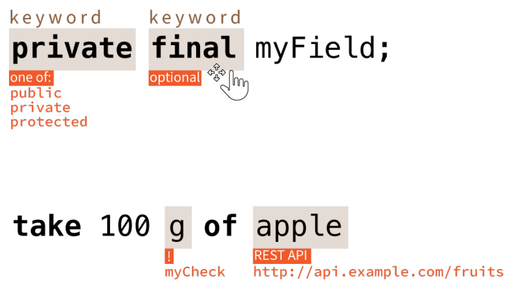


Figure 4. Defining validations: *(top)* a value for the first token is restricted by an enumeration; keyword `final` is made optional; *(bottom)* a custom validation method `myCheck` is associated with token `g`; the token after `of` should be an entry in a JSON object, which is returned by a REST API at a specified address. Note that in the editor, features can be moved around within the concept or to a different concept using drag-and-drop, as discussed in Sect. 3.

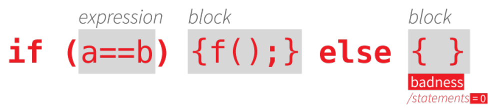


Figure 5. Specifying an avoidable example: badness assigned to the else-block because its feature `statements` has cardinality 0.

that can be used in their languages. This set of quickfixes can be based on the flavor of the language, as discussed above.

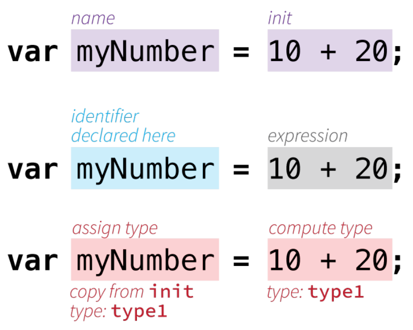


Figure 6. Different annotation perspectives for concept *field declaration*: *(top)* names of features; *(center)* concepts of features; *(bottom)* typing rules. The latter specify that one first needs to compute the type of feature *init*, store it as `type1`, and then assign to feature *name* type `type1`, which is copied from sibling feature *init*.

Yet another annotation perspective shall be used to define *typing rules* for concepts. Our vision is to have a predefined set of possible type meanings, with the language engineer assigning those meanings to types used in their language. We distinguish between inference rules and checking rules, similarly to how type systems are defined in JetBrains MPS [8].

An *inference rule* ascribes a type to a feature; it can either be an exact fixed type or be copied from a sibling feature of the concept. Fig. 6*(bottom)* shows an example of a *field declaration*, where first the type of feature *init* is computed and stored in type variable `type1`, and then ascribed to feature *name*. A *checking rule* only checks whether certain types are compatible. An example of *integer field declaration*, shown in Fig. 7, requires that the type of feature *init* is `number`. A similar annotation could check, for example, that feature *condition* of concept *if statement* is of type `bool`.

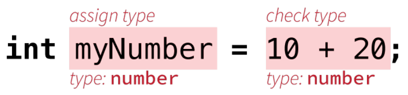


Figure 7. Type system annotation for concept *integer field declaration*. Both features *name* and *init* have type `number`; it is *ascribed* to *name*, but should be *validated* for *init*.

We see a special case for concept *expression*, where a language engineer has to define types of primitive expressions and typing templates for operators (such as `number` + `number` = `number`, or `number` + `text` = `text`).

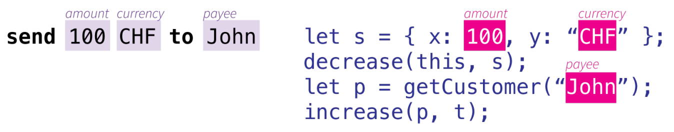
2.3 Defining Transformations

For each code example that the language engineer has defined, an *implementation*—i.e., an output associated with that example—must be specified. *Code to be generated* is represented in a textual form, with annotations that refer to features mentioned in the instance of the concept.

Consider concept *send statement* of an assumed financial DSL, defined illustratively as `send 100 CHF to John`, as shown in Fig. 8(*left*). This concept has features *amount*, *currency*, and *payee* (represented illustratively as `100`, `CHF`, and `John`, respectively), and is transpiled into JavaScript in a way that these features are directly emitted into the output code. In this trivial case, language engineer would only need to provide the desired output string—the annotation of the features can be done automatically, based on the textual equivalence to feature values in the example.

In a general case, we imagine defining transformation rules that provide a mechanism to annotate output code with template conditions and loops, in spirit of template expressions of Eclipse Xpand [19], Eclipse Xtend [5], rewriting rules of Stratego/XT [40] or macros of JetBrains MPS [8]. Again, based on the flavor of the language, "standard" outputs can be suggested by Language Wheel for certain concepts.

Besides model-to-text transformations, one can imagine that model-to-model transformations could be defined in Language Wheel. An alternative implementation of a concept is *defining an interpreter*: a language engineer would define code (in a standard general-purpose language) that will be executed for the concept.



```
send 100 CHF to John
```

```
let s = { x: 100, y: "CHF" };
decrease(this, s);
let p = getCustomer("John");
increase(p, t);
```

Figure 8. Defining a code generator for concept *send statement* in an assumed financial DSL. An example of generated code is presented as a string annotated with the features of the concept.

It should be possible to define several implementations for a code example, that is, having multiple generation targets for a single source.

3 A Web-Based Tool

We imagine Language Wheel as a web-based tool, whose interface could resemble a standard IDE, as shown in Fig. 9. The left pane contains a list of examples of code written in the language being defined, as well as examples imported from other languages (see Sect. 5). The main pane is split into two parts: *language definition* (**M2**) and *sandbox* (**M1**). Language definition part is split into an illustrative definition of a concept and its code generator (cf. Fig. 8). The sandbox part is used to preview language definitions—it has a working editor for the language and can run the code generator to output generated code. It should be possible to enable *end-user mode*, where the language definition part is not visible to the user.

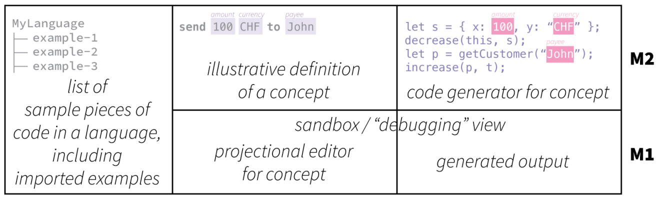

MyLanguage
example-1
example-2
example-3
list of sample pieces of code in a language, including imported examples
send 100 CHF to John
illustrative definition of a concept
let s = { x: 100, y: "CHF" };
decrease(this, s);
let p = getCustomer("John");
increase(p, t);
code generator for concept
M2
sandbox / "debugging" view
projectional editor for concept
generated output
M1

Figure 9. Envisioned user interface of Language Wheel.

Illustrative language definition uses a *"semi-projectional" editor* with an edit-mode and a view-mode. When a language engineer inputs and edits code examples, the edit-mode with an ordinary text editor is activated, and when there is no editing going on, a view-mode with richer notation is enabled. Edit-mode provides a freedom of textual editing, including a possibility to paste code samples from other editors (which can be an issue in modern projectional editors [45]). View-mode, on the other hand, allows showing code annotations in a way similar to figures in the present paper, without complications connected to projectional editing [18, 30]. Annotating code is possible in both modes, though with different flexibility. We imagine some code annotations shown in the ordinary editor, similar to how "code minings" [48] are shown in modern IDEs. In addition, view-mode allows a user to drag-and-drop features and annotations both within and among examples. This is showcased in Fig. 4, where token `final` can be, for example, either moved before token `private`, or altogether into concept *entity*. Drag-and-drop should also apply to annotations and parts of annotations of tokens. To further facilitate editing experience, we envision adopting rich text editors, similar to the ones used in text processors. This shall enable using non-textual notations, such as tabular or positional [43].

4 From Workbench to Platform

One can envision Language Wheel not only as a language or modeling workbench, but as a web-based multi-user platform for defining and using languages.

When a language engineer registers a language as publicly available, a *language space* is assigned to the language. This language space is then made available to end-users who are assigned *workspaces*, where they can create programs in the language. One can imagine that an end-user workspace supports writing programs in languages from different language spaces.

We believe it is important to implement an API so that users of the system could programmatically interact with existing models (i.e., languages and programs in those languages), serialize them, and create new ones.

We imagine collaborative editing available both for language engineers and end-users [22, 41], as well as an integration with version control systems.

We also envision a social-media component for both language engineers and end-users.

5 Language Composition and Migration

Language composition is vital for an efficient modeling experience [42]. As we envision Language Wheel as a multi-user platform, we imagine that when a language is being defined in Language Wheel, *all* other languages that are defined and registered as public by other users, become implicitly imported. A language engineer will then be able to choose which languages or concepts are to be used in their language. This essentially creates a market of languages and concepts [12, Sect. 11.6.2.]—to further facilitate prototyping of new software languages based on the ones already defined. We imagine a standard library—a palette—of languages and concepts that will be predefined in Language Wheel.

With language extension, a language engineer can append an existing concept to their language. This concept can be either cloned (thus behaving as if it were defined by the language engineer who copied it), or be a reference to the original concept (in that case, changes to the original concept, possibly introduced by another user at a later point, will be propagated to the copied one). Language embedding, referencing, and reuse [42, 43] depend on the output generated by Language Wheel, as discussed in the beginning of Sect. 2.

A Language Wheel-specific case of language composition deals with language flavors: if those *are* defined within Language Wheel, then assigning a flavor to a language can be considered as language composition.

We imagine that *migrating a language* to a newer version will propagate the changes in *all* existing models of *all* users; implementation of this will heavily depend on whether the language is textual or projectional; the latter case seems to be easier to handle[6].

6 Related Work

Assembling languages from predefined concepts. *CBS framework* for component-based specification of programming languages [31, 33] provides a library of reusable components. The semantics of each language construct is defined by translation to a collection of predefined constructs.

Miksilo language workbench enables creating languages "by mixing existing languages and building on top of them" [47]. Language transformations are represented as *deltas*; a delta applies a small change to a language, such as adding or removing a language feature, or adding an optimization. Languages are thus defined by composing deltas.

Déjà Vu [36] is an application building platform where a developer imports concepts form the catalogue and adjusts them to fit the particular needs of an application being developed.

Live language development. Language workbench *Spoofax* [23, 27] supports live language development: the user can see a language specification and an editor for the language within the same window, and after changes to the specification are made, the editor gets immediately updated. Thus, language engineers get fast feedback when they change their language, enabling experimentation with language design and development [27]. A similar approach is utilized in language workbenches *JetBrains MPS* [8, 45], *MetaEdit+* [24], and *Racket* [39], among others.

Microsoft Oslo allowed defining an input example and a grammar, and showed a representation of parsing the input with the grammar [16].

Example-driven modeling. *Example-driven modeling* [3, 9, 28] is a modeling approach that uses both explicit examples and abstractions to model complex business knowledge [1]. In this approach, abstractions are synthesized from a set of examples [3], whereas in Language Wheel, the language engineer him- or herself annotates the examples of concepts and thus defines the abstract syntax only implicitly.

Another modus operandi of example-driven modeling is *example derivation* [3], that is, generating examples from abstractions. In connection to Language Wheel, we could imagine the following process: a language engineer defines a collection of annotated examples of a concept (implicitly specifying the abstract syntax), and then annotated examples that conform to the abstract syntax are derived automatically. Our hypothesis is that this could help a beginner language engineer in spotting issues and inconsistencies in their original examples. The language engineer can modify the derived examples thusly *refining* [2] the specification of abstract syntax—which will be propagated back to the original examples of the concept.

Projectional editing. Similar to the "semi-projectional" editing as explained in Section 3, *hybrid editors* augment text-based programs with additional information [20].

MacGnome environment [29] had a special *editing mode* that allowed converting sections of code into plain text to perform editing; after that the sections were converted back to a structural representation.

In *Greenfoot* [6], a program is represented as frames, which are created using text- and mouse-based operations [26]; expressions can be entered in a textual mode and it is possible to convert them on the fly into structured expressions [6].

Barista [26] supports structure views that enable representing structural items in code in a visual way instead of textual. *Graphite* [32] allows incorporating custom highly-specialized interactive code generation interfaces directly into textual editors (e.g., a visual color picker associated with a textual representation of an instance of class `Color` in Java).

Many of these ideas can be reused in the implementation of the "semi-projectional" editor of Language Wheel.

7 Discussion

Language Wheel is apparently the first workbench that focuses on purely example-driven definition of languages. We find it particularly important to guide a language engineer by using language flavors, which can be thought of as (meta)schemata for language definition process.

The current state of implementation of Language Wheel is in the very early phase. Many of the details of how various concerns of language definition can be defined using annotations remain to be elaborated. Foremost, this applies to typing rules and scopes of visibility.

Identifying commonly used concepts (together with their associated annotation perspectives) that will constitute the standard library of Language Wheel is an interesting task in itself. Many of those annotation perspectives will require some notion of a generic definition to be instantiated when specifying a particular language; for example, quickfixes can be considered as generic transformations [7, 11, 25, 35].

Acknowledgments

The author is thankful to the anonymous reviewers for many valuable comments on the paper. The author is also thankful to M. Boersma, M. Haveraaen, B. Kolb, C. Omar, F. Tomassetti, M. Voelter, and others for discussions about earlier versions of this paper presented at informal occasions.

[6]`https://www.jetbrains.com/help/mps/migrations.html`

References

[1] M. Antkiewicz, K. Bak, K. Czarnecki, Z. Diskin, D. Zayan, and A. Wasowski. 2013. Example-Driven Modeling using Clafer. In *MoDELS 2013 (CEUR Workshop Proceedings)*, Vol. 1104. CEUR-WS.org, 32–41.

[2] R.-J. Back and J. von Wright. 1998. *Refinement Calculus - A Systematic Introduction.* Springer.

[3] K. Bak, D. Zayan, K. Czarnecki, M. Antkiewicz, Z. Diskin, A. Wasowski, and D. Rayside. 2013. Example-driven modeling: model = abstractions + examples. In *ICSE '13*. IEEE Computer Society, 1273–1276.

[4] M. Barash. 2020. Enabling Language Engineering for the Masses. In *MODELS '20 Companion.*

[5] L. Bettini. 2013. *Implementing domain-specific languages with Xtest and Xtend: learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices.* Packt Pub.

[6] N. C. C. Brown, A. AlTadmri, and M. Kölling. 2016. Frame-Based Editing: Combining the Best of Blocks and Text Programming. In *LaTICE 2016.* IEEE Computer Society, 47–53.

[7] J.-M. Bruel, B. Combemale, E. Guerra, J.-M. Jézéquel, J. Kienzle, J. de Lara, G. Mussbacher, E. Syriani, and H. Vangheluwe. 2020. Comparing and classifying model transformation reuse approaches across metamodels. *Softw. Syst. Model.* 19, 2 (2020), 441–465.

[8] F. Campagne. 2014. *The MPS Language Workbench, Vol. 1.*

[9] H. Cho, Y. Sun, J. Gray, and J. White. 2011. Key challenges for modeling language creation by demonstration. In *ICSE 2011.*

[10] Microsoft Corp. [n. d.]. Language Server Protocol Specification. https://microsoft.github.io/language-server-protocol

[11] J. S. Cuadrado, E. Guerra, and J. de Lara. 2014. A Component Model for Model Transformations. *IEEE Trans. Software Eng.* 40, 11 (2014), 1042–1060.

[12] K. Czarnecki and U. W. Eisenecker. 2005. *Generative programming: methods, tools, and applications.* Addison Wesley.

[13] 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.

[14] M. Eysholdt and J. Rupprecht. 2010. Migrating a large modeling environment from XML/UML to Xtext/GMF. In *SPLASH/OOPSLA Companion*, W. R. Cook, S. Clarke, and M. C. Rinard (Eds.). ACM, 97–104.

[15] M. Fowler. [n. d.]. Language Workbenches: The Killer-App for Domain Specific Languages?

[16] M. Fowler. [n. d.]. Oslo. https://martinfowler.com/bliki/Oslo.html

[17] M. Fowler and R. Parsons. 2011. *Domain-specific languages.* Addison-Wesley.

[18] A. Gomolka and B. Humm. 2011. Structure Editors: Old Hat or Future Vision?. In *ENASE 2011 (Communications in Computer and Information Science)*, Vol. 275. Springer, 82–97.

[19] R.C. Gronback. 2009. *Eclipse modeling project: a domain-specific language toolkit.* Addison-Wesley Professional.

[20] B. Hempel, J. Lubin, G. Lu, and R. Chugh. 2018. Deuce: a lightweight user interface for structured editing. In *ICSE 2018.* ACM, 654–664.

[21] K. Hölldobler and B. Rumpe. 2017. *MontiCore 5 Language Workbench Edition 2017.* Shaker Verlag.

[22] J. L. Cánovas Izquierdo and J. Cabot. 2013. Enabling the Collaborative Definition of DSMLs. In *CAiSE 2013 (Lecture Notes in Computer Science)*, Vol. 7908. Springer, 272–287.

[23] K. C. L. Kats and E. Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA 2010.* ACM, 444–463.

[24] S. Kelly, K. Lyytinen, M. Rossi, and J.-P. Tolvanen. 2013. MetaEdit+ at the Age of 20. In *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE.* Springer, 131–137.

[25] M. Kessentini, H. A. Sahraoui, M. Boukadoum, and O. Benomar. 2012. Search-based model transformation by example. *Softw. Syst. Model.* 11, 2 (2012), 209–226.

[26] A. J. Ko and B. A. Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *CHI 2006.* ACM, 387–396.

[27] G. Konat, S. Erdweg, and E. Visser. 2016. Towards live language development. In *(LIVE) 2016.*

[28] J. J. López-Fernández, Cuadrado J. S, E. Guerra, and J. de Lara. 2015. Example-driven meta-model development. *Softw. Syst. Model.* 14, 4 (2015), 1323–1347.

[29] P. Miller, J. Pane, G. Meter, and S. A. Vorthmann. 1994. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interact. Learn. Environ.* 4, 2 (1994), 140–158.

[30] S. Minör. 1992. Interacting with Structure-Oriented Editors. *Int. J. Man Mach. Stud.* 37, 4 (1992), 399–418.

[31] P. D. Mosses. 2019. A Component-Based Formal Language Workbench. In *F-IDE@FM 2019 (EPTCS)*, Vol. 310. 29–34.

[32] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. 2012. Active code completion. In *ICSE 2012.* IEEE Computer Society, 859–869.

[33] PLanCompS Project. [n. d.]. CBS: A Framework for Component-Based Specification of Programming Languages. https://plancomps.github.io/CBS-beta

[34] D. Ratiu, V. Pech, and K. Dummann. 2017. Experiences with Teaching MPS in Industry: Towards Bringing Domain Specific Languages Closer to Practitioners. In *MODELS 2017.* IEEE Computer Society, 83–92.

[35] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning syntactic program transformations from examples. In *ICSE 2017.* IEEE / ACM, 404–415.

[36] S. P. De Rosso, D. Jackson, M. Archie, C. Lao, and B. A. McNamara III. 2019. Declarative assembly of web applications from predefined concepts. In *Onward! 2019.* ACM, 79–93.

[37] C. Simonyi, M. Christerson, and S. Clifford. 2006. Intentional software. In *OOPSLA 2006.* ACM, 451–464.

[38] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.

[39] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. 2011. Languages as libraries. In *PLDI 2011.* ACM, 132–141.

[40] E. Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *RTA 2001 (Lecture Notes in Computer Science)*, Vol. 2051. Springer, 357–362.

[41] M. Voelter. [n. d.]. An Open Platform For Systems and Business Engineering Tools: Collaborative Modeling and Analysis at Scale. https://voelter.de/data/pub/APlatformForSystemsAndBusinessModeling.pdf

[42] M. Voelter. 2011. Language and IDE Modularization and Composition with MPS. In *GGTTSE 2011 (Lecture Notes in Computer Science)*, Vol. 7680. Springer, 383–430.

[43] M. Voelter and S. Benz. 2013. *DSL engineering: designing, implementing and using domain-specific languages.* Dslbook.org.

[44] M. Voelter and S. Lisson. 2014. Supporting Diverse Notations in MPS' Projectional Editor. In *GEMOC@Models (CEUR Workshop Proceedings)*, Vol. 1236. CEUR-WS.org, 7–16.

[45] M. Völter, J. Siegmund, T. Berger, and B. Kolb. 2014. Towards User-Friendly Projectional Editors. In *SLE 2014 (Lecture Notes in Computer Science)*, Vol. 8706. Springer, 41–61.

[46] J. Warmer. [n. d.]. ProjectIt – A Framework for building projectional web editors. http://www.projectit.org

[47] R. Willems. [n. d.]. Miksilo: A modulatiry first language construction workbench. http://keyboarddrummer.github.io/Miksilo/

[48] A. Zerr. [n. d.]. Inline code annotations in Eclipse Platform with new CodeMining support. https://www.eclipsecon.org/france2018/session/inline-code-annotations-eclipse-platform-new-codemining-support