Vision: The Next 700 Language Workbenches*

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

Abstract

Language workbenches (LWBs) are tools to define software languages together with tailored Integrated Development Environments for them. A comprehensive review of language workbenches by Erdweg et al. (Comput. Lang. Syst. Struct. 44, 2015) presented a feature model of functionality of LWBs from the point of view of "languages that can be defined with a LWB, and not the definition mechanism of the LWB itself". This vision paper discusses possible functionality of LWBs with regard to language definition mechanisms. We have identified five groups of such functionality, related to: *metadefinitions*, *metamodifications*, *metaprocess*, *LWB itself*, and *programs* written in languages defined in a LWB. We design one of the features ("*ability to define dependencies between language concerns*") based on our vision.

CCS Concepts • **Software and its engineering → Development frameworks and environments**; **Context specific languages**;

Keywords Language workbenches, software languages, algebraic specifications, metaprogramming

ACM Reference Format (Version of Record):
Mikhail Barash. 2021. Vision: The Next 700 Language Workbenches. In *Proceedings of Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3486608.3486907

1 Introduction

Implementing software languages is impossible without implementing tooling for them: having a powerful tailored Integrated Development Environment (IDE) is crucial for languages' adoption in practice [83]. Language workbenches [29] (LWBs) help automatize tool creation: based on definitions of concerns [82], they output a tailored IDE with editor services, such as syntax-aware editing with syntax highlighting, code formatting, folding, completion, navigation, and so on [29]. Representatives of LWBs include Melange [24], MPS [14], Monticore [40], Racket [81], Rascal [73], Spoofax [50], SugarJ [29], Xtext [83], to name a few.

A comprehensive overview of language workbenches by Erdweg et al. [29] presented a feature model of functionality of LWBs, which is based on input from both academic and industrial research. This feature model focuses primarily on the functionality from the point of view of the languages that can be defined with a language workbench, while features related to the language definition mechanisms of LWBs are—as noted by the survey's authors—only discussed to a rather limited extent. Recent surveys on language development tools [9, 47, 70] also focus on a desirable functionality of user languages rather than the language definition mechanisms.

In this paper, we present a vision of the conceptual framework that can be used to explore possible functionality of LWBs with regard to the language definition mechanisms. Key to this are the notion of a *tool-first language*, where language's properties related to tooling are first-class, and the *Language-Aspects-Concepts* idiom (terminology inspired by MPS [83]), where a language is treated as a set of language constructs (concepts), for each of which several concerns (aspects) can be specified. We give a non-exhaustive list of possible functionality of LWBs, and illustrate how this list—which constitutes the core of the conceptual framework—can be used to design one of the suggested features, namely, the ability to specify dependencies between aspects. We present a research agenda whose goal is to facilitate the use of the envisioned conceptual framework.

2 Definitions

Recall [53] that a language workbench can be used in three different modes: a language user uses a workbench "just" as an IDE to develop application software—*mograms*[1]; a language engineer uses a workbench to design and implement languages that language users then use; and a meta-language engineer uses a workbench to bootstrap it [56, 71, 72]. More formally, a language workbench operates with base meta-languages $B_1, \ldots, B_k$ that are used to define user languages $L_1, \ldots, L_n$, in which user mograms $M_1, \ldots, M_m$ are written. Each language L_i is defined by means of $B_1, \ldots, B_k$, other $L_{j_1}, \ldots, L_{j_\ell}$, and partially constructed itself $\widehat{L_i}$. Bootstrapping of LWBs can be expressed as a requirement on base languages $B_1, \ldots, B_k$ to be definable by means of themselves. For all the mentioned languages, the functionality of a language

*This is a preprint. The definitive Version of Record is available at: https://doi.org/10.1145/3486608.3486907

SLE '21, October 17–18, 2021, Chicago, IL, USA
2021.

[1] As defined by Kleppe [53, p. 4], a mogram, which is a portmanteau of the words "model" and "program", can be either a model, a program, an XML file, or "any other thing written in a software language".

workbench with respect to the language definition mechanisms (e.g., mechanisms to specify syntax, behaviour of editor services, etc.) should be uniform.

To address the fact that tool support is an integral part of a language definition in LWBs [29, 35], we introduce the notion of a *tool-first language*. Formally, a tool-first language is a set of *concepts*[2]: $L = \{C_1, \ldots, C_m\}$, where each concept is a tuple of *aspect instances*: $C_i = \langle C_i|_{A_1}, \ldots, C_i|_{A_\ell} \rangle$. An *aspect* A_j is akin to a *concern* [82], related either to the specification of syntax, semantics, the type system, or to a specification of behaviour of tooling (i.e., an IDE) associated with the language[3]. In a most general setting, an aspect can be considered as a tuple $A = \langle \mathcal{S}, \mathcal{F} \rangle$ of sets $S_1, \ldots, S_k \in \mathcal{S}$ and (effectful) functions $f_1, \ldots, f_\ell \in \mathcal{F}$ on elements of these sets $S_1, \ldots, S_k$. An instance $C_i|_{A_j}$ of an aspect A_j can be then considered as an evaluation of these functions.

Algebraically, this skeleton definition of a tool-first language focuses on *specifying signatures*; we refer to this definition as the Language-Aspects-Concepts idiom. It can be refined as necessary based on a task at hand (e.g., MPS allows defining aspects not bound to concepts); later in Sect. 4 we show an example of a refinement of the notion of an aspect.

3 Functionality of Language Workbenches

We present now the core of the conceptual framework—a list of possible LWB features with regard to language definition mechanisms. We have identified 5 groups of features, related to: (I) defining the behaviour of the LWB itself, (II) defining (meta)languages, (III) introducing modifications to (meta)definitions, (IV) assisting users in the process of language definition, and (V) specifying the behaviour of a LWB while mograms are edited or run. We present below each of these groups, their representative functionality "classes" and sample features. These features should have an algebraic specification based on the Language-Aspects-Concepts idiom or its appropriate refinement. This also dictates how the envisioned framework is applied when designing a new feature: given an appropriate signature specification for it, possible related features are explored based on the groups we present below[4]. A feature model diagram of the features discussed in this Section is presented in Fig. 1.

We start with a classification of LWBs according to the functionality that allows customizing their behaviour.

I: WORKBENCH-LEVEL

Bootstrapped LWBs provide facilities to define metalanguages in terms of those metalanguages. A generalized notion is **meta-LWBs**; those can be used to develop (other) language workbenches. An example of this is MPS which is used to define the metalanguages of LanguageLab [72]. Major LWBs are bootstrapped [29, 56, 71] and are thus meta-LWBs. **Customizable LWBs** allow specifying and modifying their behaviour and appearance (e.g., menu commands) by means of designated metalanguages. Modifications can be performed at *workbench-runtime* (i.e., independently of the languages being defined), *language-runtime* (i.e., during the language definition process), and *mogram-runtime* (i.e., during the use of the LWB by a language user). We note **standalone LWBs**, which allow hiding their meta-functionality (e.g., Rich Client Platforms for Eclipse- and IntelliJ-based LWBs [57, 83]). **Restricted LWBs** are tailored at developing a particular language(s), including general-purpose ones. Oftentimes, the metalanguages will be fixed and new ones cannot be defined. Such a LWB provides, e.g., certain design templates or restrictions for the languages being defined; it could check the consistency of the language constructs, etc. Examples include: a workbench to implement extensions of AspectJ [41]; MPS Lightweight DSLs [14] to define families of Java classes. **Programmable LWBs** provide APIs for programmatically manipulating (meta)languages and mograms; we note here **scriptable** and **REPL-supporting** LWBs. One can also imagine a **LWB Server Protocol**, in the style of Microsoft LSP.

Hosted LWBs are "embedded" into an existing IDE or another tool (e.g., as a plugin). ***Ad hoc* LWBs** emerge when a software tool, not necessarily tailored at software development, is used to produce an artifact that conforms to the definition of a LWB[5]. **Front-end workbenches** output a language definition in another workbench's metalanguages.

II: METADEFINITIONS

In groups II–IV, we present directions of possible functionality and give examples as appropriate. Group II discusses the expected expressive power of LWB's metalanguages.

Definition of dependencies and relationships. Based on the Language-Aspects-Concepts idiom, a LWB has mechanisms[6] to define dependencies (cf. [22]) between languages, aspects, and concepts (L/A/C). We give examples of such functionality: *dependencies between languages*—ability to define and adequately support language composition [31, 46, 66]; *between aspects*—ability to define hierarchies of aspects; *between concepts*—ability to define inheritance between concepts; *cross-dependencies between L/A/C*—ability to define aggregation relationships between languages and concepts.

[2] A concept corresponds to a language construct, e.g., "if statement", etc.

[3] Examples of aspects: Structure, that defines the abstract syntax of a concept; View, that defines its concrete syntax; Highlighter, that defines syntax highlighting; Constraints, that defines aggregation relations between the concept and other concepts; Generator, that defines model-to-model/text transformations applicable to the concept; and so on [29].

[4] A feature model diagram is available at: https://github.com/dsl-course/next-700-LWBs.

[5] An example of this is a sheet developed in a spreadsheet calculator, where certain cell blocks are used to define—using the tabular notation—the abstract syntax of language, and other cell blocks are used to define mograms in that language—also in tabular notation [7].

[6] By "mechanism" we mean "a dedicated metalanguage".

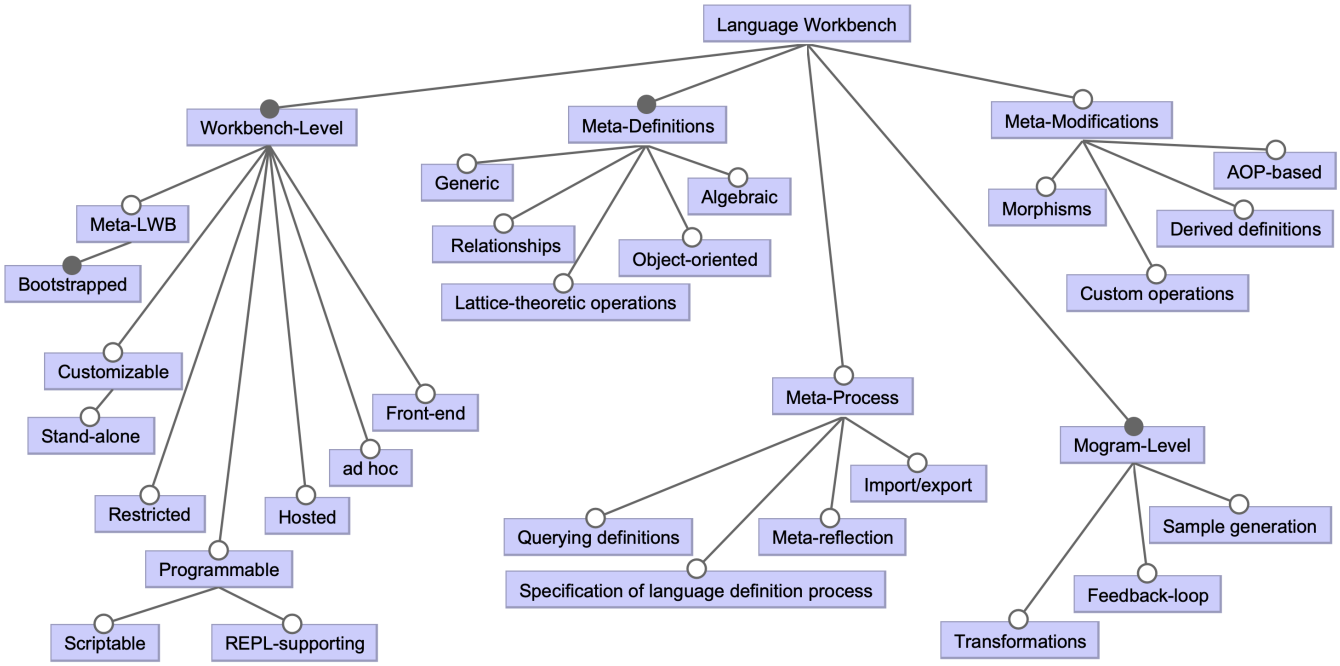

Figure 1. A feature model diagram of the features in groups I–V. Several features are mandatory: a language workbench is expected to: be ***bootstrapped***, have mechanisms to ***specify metadefinitions***, have mechanisms to ***specify mograms***.

Parametrization in definitions. Parametrization refers to ability to define *generic L/A/C and/or their constituents*, with the goal of fostering reuse in language specifications. A workbench is expected to support powerful parametrization mechanisms such as *renaming* [12, 50] in specifications. In order to fully support reuse [19, 25], a mechanism to define and use *libraries of L/A/C* [19, 66] is expected in a LWB.

Lattice-theoretic operations. A LWB provides a mechanism to define *set-theoretic operations on L/A/C and their constituents*, e.g., it should be possible to define operations such as union or set difference [1, 3] both on entire languages ($L_1 \cup L_2$) and on their constituents, e.g., on aspects only ($L_1 \big|_{A_i} \cup L_2 \big|_{A_i}$). A workbench is also expected to support defining and analyzing *lattices of L/A/C* (cf. [26]).

Object-oriented definitions in the style of MPS can be extended to provide mechanisms to define *interfaces, traits, mixins of L/A/C* (cf. [25, 44, 59, 60]); *prototype-based definition of L/A/C*; *(sub)typing relations on L/A/C* (cf. [78]); *visibility modifiers for L/A/C*; and so on. Another direction is the support of the design-by-contract approach, with *pre- and postconditions in (meta)definitions*.

Algebraic-style definitions.[7] A LWB is expected to provide mechanisms to *define signatures of L/A/C*. It should be possible to follow the *à la carte* approach [80] to define constituents of L/A/C. One could expect support for: *modular definitions* [64]; *requires- and provides- interfaces*; *L/A/C invariants*; *holes in definitions*; *canonical, normalized definitions of L/A/C* [34, 48]; and so on. A mechanism to *specify isomorphisms, equivalence-like relations on L/A/C* could be used, e.g., for clone detection in metadefinitions.

[7] Informally, this is where our framework "gets bootstrapped": it should be possible to express the definitions from Sect. 2 in LWB's metalanguage(s).

III: METAMODIFICATIONS

This group includes functionality that deals with (automatic) modifications of (meta)definitions.

Custom operations. One can expect a mechanism to define and execute transformations (semantics-preserving or not) on L/A/C and their signatures. Examples include: making a concept abstract; restoring an aspect instance given a set of other aspect instances [43, 74, 87]; slicing L/A/C and modifying their constituents; casting a concept definition to another one; "extracting" an aspect from a concept and then "applying" it to another concept (e.g., "eyedropping" the Formatter from `IfStmt` to `WhileStmt`).

Morphisms. Extending on this, a workbench could provide mechanisms to express morphisms. The simplest case are *homomorphisms*, for which a LWB provides mechanisms to define, compose, and analyze their properties (e.g., a homomorphism can be defined on aspect View to support having keywords in multiple natural languages). One can imagine a comprehensive support of *transforming (meta)definitions* at workbench-, language-, or mogram-runtime. A theoretical framework for this are *signature morphisms* [37] and their extensions (e.g., Syntactic Theory Functors [42]). These would

enable creating syntactic structuring mechanisms for LWB's specification formalism, in order to, e.g., extend signatures, change notation or LWB behaviour in arbitrary ways, thus unleashing the full power of LWB customization.

Derived (meta)definitions. A LWB can have a mechanism to support *derived (meta)definitions* from existing ones. Examples include: constructing aspects or concepts from parts of other aspects or concepts in a specified way[8]; specifying how to derive *language dialects* from existing languages (cf. [75]); mechanisms to manipulate *parts of L/A/C*.

One can also expect a workbench to support metamodifications inspired by the **AOP paradigm** [36, Sect. 6.1.].

IV: METAPROCESS

This group discusses functionality to support a language engineer during the language definition process.

Meta- metaprogramming/reflection. A language workbench can provide a mechanism to *introspect itself and the (meta)definitions* (cf. [18]), which would be available for (meta)language engineers and language users.

Querying definitions. Along similar lines, mechanisms for *querying (meta)definitions* can be provided (cf. [58]); they can also have a *predicate-logic style* (cf. [15, 39]). One can imagine mechanisms for, e.g., creating and processing ontologies [5] and knowledge graphs [45] of (meta)definitions.

Specification of language definition process. A LWB can have mechanisms to *specify the very process of language specification*, e.g., which concepts and aspects need to be defined and in what order in certain classes of languages, the "style" of the concrete syntax, requirements for code generation, etc. This will enable validating the language definition process and guiding a language engineer along the way.

Importing and exporting. Mechanisms that *specify import and export* of (meta)definitions can be supported, most probably of parts of or of simplified (meta)definitions. A workbench can have mechanisms to *specify and perform exporting meta-artifacts*, such as documentation (cf. [90]), metrics reports in the style of [84, Sect. 6], various diagrams representing (meta)definitions (e.g., UML-like), and so on.

V: MOGRAM-LEVEL

This group focuses on functionality to specify the behaviour of an LWB at mogram-runtime. It could include mechanisms for: metalanguage-agnostic **generation of sample mograms**; **operations on and transformations of mograms** (e.g., slicing a mogram, mutation operations); defining a **feedback-loop from mograms** to change the language definition and/or workbench behaviour; and so on.

[8] E.g.: "default" concrete syntax from abstract syntax (cf. *reflective editors* in MPS [83]); "default" non-conflicting concrete syntaxes for all concepts in a given set of languages that respects how these languages are composed.

4 Example: Dependencies Between Aspects

We present an example of how a feature suggested in group II in the previous section—namely, ability to specify *dependencies between aspects*—can be designed based on our vision. Our conceptual framework is applied in two passes: first, to design mechanisms for defining aspect dependencies, and second, to explore possible related functionality.

The initial step is to express the notion of aspect dependencies within the formal definitions presented in Sect. 2. For that, we refine the definition of an aspect to reflect the dataflow between aspects. Consider a function $f_i \in \mathcal{F}$. When invoked on elements of sets $S_{i_1}, \ldots, S_{i_n}$, it can *observe*, *update*, or *output* [6] elements of these sets. To reflect this, we partition $\mathcal{S}$ into $\mathcal{S}_{\text{obs}}$, $\mathcal{S}_{\text{upd}}$, and $\mathcal{S}_{\text{out}}$. We denote by $\sigma(f_i)$ the signature and by $\delta(f_i)$ the semantics of the function f_i. We can now refine the definition of an aspect by taking into account this partition of $\mathcal{S}$ and the semantics of functions f_i as follows: $A = \langle \mathcal{S}_{\text{obs}}, \mathcal{S}_{\text{upd}}, \mathcal{S}_{\text{out}}, \{\sigma(f_1), \ldots, \sigma(f_\ell)\}, \{\delta(f_1), \ldots, \delta(f_\ell)\}\rangle$.

Based on this definition, we can design a metalanguage to express aspect dependencies. We showcase this metalanguage on several (trivial) examples of aspect definitions[9].

```
aspect Structure
  obs Cardinalities, Concepts
  out Children, Names
  sig addChild: Names * Concepts * Cardinalities
                -> Children

aspect View
  obs Children
  out Keywords, Boxes
  sig project: (Children U Keywords)* -> Boxes

aspect Highlighter
  upd Boxes
  out Styles
  sig stylize: Boxes * Styles -> Boxes
```

An invocation `addChild(n, c, m)` adds into the current concept a new child with name `n`, which has cardinality `m` and is an instance of concept `c`. Depending on a particular definition of the semantics δ(`addChild`), the behaviour (the effect) of this function can be represented, e.g., as an abstract syntax specification using a context-free grammar, a UML-like specification, etc. Note that in this function, the sets Cardinalities and Concepts are *observed*, while Children and Names are *output*: indeed, an invocation of `addChild` spawns a new element in the current concept's children, and the new identifier is only created as a result of the invocation.

Function `project` defines a concept's concrete syntax (e.g., textual, projectional, or graphical). This function *observes* the set Children defined in STRUCTURE, and *outputs* sets Keywords and Boxes: each element of Keywords is spawned

[9] Here Cardinalities is the (global) set of cardinalities, Names is the (global) set of valid identifiers, Concepts is the a of language's concepts, Children is the a of (current) concept's aggregation relationships, Keywords is the (global) set of strings conforming to a definition of a "keyword", Boxes is the a of concrete syntax representations [54] of elements of Children and Keywords, and Styles is a set of style representations of elements of Boxes.

Table 1. Some possible features *related to* "the ability to specify aspect dependencies".

Group	Designed feature: ability to ...
I	... hide LWB's menu commands to create aspects which would be invalid in the current context
II	... define generic (parametrized) aspects
II	... define aspect interfaces
II	... define isomorphisms on aspects
III	... cast aspect instances
III	... derive aspects from other aspects
IV	... query aspect definitions, instances
IV	*... guide a language engineer on the order in which aspects should be defined for a concept*

as a result of invoking `project`, and Boxes [54] are defined in `project`. Depending on the particular definition, `project` may return, for example, a context-free grammar rule $W \longrightarrow w_1 w_2 \cdots w_\ell$, a cell layout of a projectional editor: `horiz-layout`$(w_1, \ldots, w_\ell)$, or a user interface control specification: $\texttt{UIControl}_1(w_1)\texttt{;} \ \ldots \ \texttt{;} \ \texttt{UIControl}_\ell(w_\ell)$.

An invocation `stylize`(b, s) assigns a style $s \in$ Styles to a box $b \in$ Boxes, *updates* the box, and *outputs* the style which has been spawned as a result of the invocation.

These trivial signature specifications show how aspects depend on each other: Highlighter uses information on `Boxes` defined in View, which in its turn uses `Children` defined in Structure, as can be seen in the figure below (only relevant sets shown). We can argue that our refined definition of an aspect adequately expresses aspect dependencies.

We now apply our conceptual framework again—to design an array of LWB features *related to* "the ability to specify aspect dependencies"; Table 1 presents some possible features.

We discuss the last feature mentioned in Table 1. Its implementation could provide validations and quickfixes to inform the language engineer that *a change in a concept's aspect invalidates other aspects* (e.g., if a user modifies `Children` in Structure, then View and Highlighter are invalidated; if `Boxes` is modified in View, then Highlighter is invalidated; etc.), or that *a transformation made to an aspect requires certain transformations to other aspects* (e.g., applying a signature morphism within Structure necessitates corresponding changes first in View, and only after that, in Highlighter). One can also imagine *restricting which aspects can be defined for concepts*: e.g., Highlighter aspect can only be defined after all View aspects for all concepts have been defined. Such a restriction would ideally force a language engineer to reflect on a common styling options first, and only then assign those styles to boxes in concepts' Highlighter aspects. This would mean a lower repetition viscosity [38] of the Highlighter's metalanguage.

5 Research Agenda

Fully benefiting from the envisioned conceptual framework requires *elaborating a meta-theory of language workbenches* and formalizing the notions of a LWB and a tool-first language. This would provide a mathematically sound common ground (cf. [13, 89]) for the *tool*ing aspect of language development *tool*s, which seems to be a blind spot, especially in comparison to an extensive body literature on formalization of language definition mechanisms (e.g., [4, 10, 17, 20, 65, 76, 86, 91]).

Designing and implementing a *featherweight language workbench* would provide a benchmark implementation and a sandbox for experimenting with the LWBs formalization and the conceptual framework envisioned in this paper.

Specifying (parts of) functionality of the major LWBs in the new formalism would be a first step in solving the problem [53] of "transforming language specifications from one LWB to another" (cf. [2, 8, 21, 23, 27, 68]). As recently observed by Ozkaya et al. [70], "importing/exporting ... metamodels is highly important" for practitioners. While solving this problem is overly ambitious, specifying (parts of) major LWBs could facilitate their more formal comparison.

6 Related Work

Like the overview of LWBs by Erdweg et al. [29], similar discussions on language development tools tend to focus primarily on the desirable functionality of user languages, rather than the metamechanisms of the tools. Merkle [63] gives a short introduction to several LWBs that support textual languages. Kelly [52] compares the effort needed to maintain language specifications in graphical modeling tools. Bork et al. [9] give a systematic literature review on visual modeling tools and the specification mechanisms of their abstract syntax. Iung et al. [47] give an extensive overview of tools for defining both textual and visual languages, based on a systematic mapping study of last decade's publications. Ozkaya et al. [70] report on practitioners' preferences on textual and visual meta-tools. While these results describe *existing* tools (and tool functionality of *user languages*), we are only aware of one extended discussion [51, Sect. 14.3] on the *expectations* for some of the *metamechanisms* of language development tools, yet in the context of graphical modeling (unlike our—more general—approach). Combemale et al. [19] describe a unifying framework for language reuse, and overview several LWBs' functionality with respect to it. France et al. [36] present an extensive roadmap of model-driven development, but only mention tooling in passing.

The idea of separation of concerns in language definitions [82] has been extensively explored in the literature. Apart from MPS [14, 83], we note Spoofax [50], which provides various metalanguages for specifying tooling-related language concerns, e.g., debuggers [61]. A definition of a

concern by Combemale et al. [19] considers it as "a configurable unit of reuse that encapsulates a specific ... set of constructs of a language" (e.g., support for expressions, support for exceptions), "which encompasses the definition of ... abstract syntax, concrete syntax and behavioral semantics". This corresponds to "a concept" in our terminology.

An algebraic angle on language engineering is a well-studied topic (cf. ASF+SDF [28], Rascal [11], Spoofax [50]). In the present paper, we merely advocate for a broader use of algebraic methods in specification of behaviour of tooling.

The idea of treating languages as first-class citizens within LWB-like tools has been recently explored. Cimini [16] focuses on languages embedded in an (ML-like) general-purpose language. Languages are treated as expressions: they can be assigned to variables, passed to functions, created and modified at runtime, etc. Mourad et al. [67] describe a tool for transforming such language definitions into each other. Lorenz et al. [62] discuss defining editors for embedded languages. While these approaches aim at embedded languages, our focus is more geared towards custom software languages.

Several results treat IDEs first-class. Fabry et al. [33] present a language to specify commands that are executed on certain events in an IDE. Sousa et al. [77] present a language to customize (standalone—in our terminology) modeling tools. Jeanjean et al. [49] discuss reconfiguring an IDE based on Microsoft LSP-style protocols which are themselves treated first-class. Klint et al. [55] implement a parametrized module manager which coordinates all actions within an IDE.

Finally, we give pointers to formalization of editing tools. Walkingshaw et al. [85] formalize structure editing using choice calculus. Omar et al. [69] formalize a variant of typed structured editing in the Agda proof assistant. Sufrin [79] presents an algebraic formalization of a text-based editor. Yovev [88] discusses theoretical limitations of editing tools.

7 Conclusion

We have outlined a conceptual framework that can be used to *explore* possible features of LWBs with respect to a language definition mechanism, and we have enumerated several examples of such functionality. Fundamentally, the proposed vision is based on a simple idea of having algebraic specifications for all stages of the development journey in a language workbench. We have given a use case of the envisioned conceptual framework, and outlined a research agenda that aims at facilitating its development and use. The focus of this agenda is on the currently lacking meta-theory of LWBs.

References

[1] M. Alanen, I. Porres, *Difference and Union of Models*, TUCS TR 527.

[2] M. Alanen, I. Porres, *A relation between context-free grammars and Meta Object Facility metamodels*, TUCS TR 606.

[3] M. Alanen, I. Porres, *Subset and union properties in modeling languages*, TUCS TR 731.

[4] T. Asikainen, T. Männistö, *Nivel: a metamodelling language with a formal semantics*. Softw. Syst. Model. 8(4). 521–549. 2009.

[5] U. Aßmann et al., *Ontologies, Meta-models, and the Model-Driven Paradigm*. Ontologies for Soft. Engineering and Soft. Technology 2006.

[6] A. H. Bagge et al., *Interfacing Concepts: Why Declaration Style Shouldn't Matter*. Electron. Notes Theor. Comput. Sci. 253(7). 2010.

[7] M. Barash, *Towards a Spreadsheet-Based Language Workbench*. MODELS 2021.

[8] J. Bézivin et al., *Bridging the MS/DSL Tools and the Eclipse Modeling Framework*. OOPSLA 2005.

[9] D. Bork et al., *A survey of modeling language specification techniques*. Inf. Syst. 87. 2020.

[10] A. Boronat, J. Meseguer, *An algebraic semantics for MOF*. Formal Aspects Comput. 22(3-4). 269–296. 2010.

[11] J. van den Bos et al., *Rascal: From Algebraic Specification to Meta-Programming*. AMMSE 2011. 15–32.

[12] A. Brogi et al., *The Use of Renaming in Composing General Programs*. LOPSTR 1998. 124–142.

[13] S. Buro, I. Mastroeni, *On the semantic equivalence of language syntax formalisms*. Theor. Comput. Sci. 840. 234–248. 2020.

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

[15] C. Chambers, *Predicate Classes*. ECOOP 1993. 268–296.

[16] M. Cimini, *Languages as first-class citizens (vision paper)*. SLE 2018.

[17] T. Clark et al., *The Metamodelling Language Calculus: Foundation Semantics for UML*. FASE 2001. 17–31.

[18] T. Clark, J. Gulden, *Model Driven Software Engineering Meta-Workbenches: An XTools Approach*. J. Univers. Comput. Sci. 26(9). 2020.

[19] B. Combemale et al., *Concern-oriented language development (COLD): Fostering reuse in language engineering*. Comput. Lang. Syst. Struct. 54. 139–155. 2018.

[20] B. Combemale, *Towards Language-Oriented Modeling*. 2015.

[21] V. Cosentino et al., *A Model-Driven Approach to Generate External DSLs from Object-Oriented APIs*. SOFSEM 2015. 423–435.

[22] S. Creff et al., *Relationships Formalization for Model-Based Product Lines*. APSEC 2012. 158–163.

[23] M. Dalibor et al., *Mind the gap: lessons learned from translating grammars between MontiCore and Xtext*. DSM@SPLASH 2019. 40–49.

[24] T. Degueule, *Melange: a meta-language for modular and reusable development of DSLs*. SLE 2015. 25–36.

[25] T. Degueule et al., *On Language Interfaces*. Present and Ulterior Software Engineering 2017. 65–75.

[26] U. Dekel, Y. Gil, *Revealing Class Structure with Concept Lattices*. WCRE 2003. 353–365.

[27] J. Denkers et al., *Migrating custom DSL implementations to a language workbench (tool demo)*, SLE 2018. 205–209.

[28] A. van Deursen et al. (Eds), *Language Prototyping: An Algebraic Specification Approach*, World Scientific. 1996.

[29] S. Erdweg et al., *Evaluating and comparing language workbenches: Existing results and benchmarks for the future*. Comput. Lang. Syst. Struct. 44. 24–47. 2015.

[30] S. Erdweg et al., *SugarJ: library-based syntactic language extensibility*. OOPSLA 2011. 391–406.

[31] S. Erdweg et al., *Language composition untangled*. LDTA 2012. 7.

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

[33] J. Fabry et al., *DIE: A Domain Specific Aspect Language for IDE Events*. J. Univers. Comput. Sci. 20(2). 135–168. 2014.

[34] J.-R. Falleri et al., *A Generic Approach for Class Model Normalization*. ASE 2008. 431–434.

[35] M. Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?*

[36] R. B. France, B. Rumpe, *Model-driven Development of Complex Software: A Research Roadmap*. FOSE 2007. 37–54.

[37] J. A. Goguen, R. M. Burstall, *Institutions: Abstract Model Theory for Specification and Programming*. J. ACM 39(1). 95–146. 1992.

[38] T. R. G. Green, M. Petre, *Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework*. J. Vis. Lang. Comput. 7(2). 131–174. 1996.

[39] S. Grewe et al., *Exploration of language specifications by compilation to first-order logic*. Sci. Comput. Program. 155. 146–172. 2018.

[40] H. Grönniger et al., *MontiCore: A Framework for the Development of Textual Domain Specific Languages*, 2014.

[41] A. Hadas, D. H. Lorenz, *A language workbench for implementing your favorite extension to AspectJ*. MODULARITY (Companion) 2015. 19–20.

[42] M. Haveraaen, M. Roggenbach, *Specifying with syntactic theory functors*. J. Log. Algebraic Methods Program. 113. 100543. 2020.

[43] A. S.-B. Herrera et al., *A Domain Specific Transformation Language to Bridge Concrete and Abstract Syntax*. ICMT 2016. 3–18.

[44] A. Hessellund, A. Wasowski, *Interfaces and Metainterfaces for Models and Metamodels*. MoDELS 2008. 401–415.

[45] A. Hogan et al., *Knowledge Graphs*. CoRR abs/2003.02320. 2020.

[46] K. Hölldobler et al., *Software language engineering in the large: towards composing and deriving languages*. Comput. Lang. Syst. Struct. 54. 2018.

[47] A. Iung et al., *Systematic mapping study on domain-specific language development tools*. Empir. Softw. Eng. 25(5). 4205–4249. 2020.

[48] B. Jacobs, *Objects and Classes, Co-Algebraically*. Object Orientation with Parallelism and Persistence 1995. 83–103.

[49] P. Jeanjean et al., *IDE as Code: Reifying Language Protocols as First-Class Citizens*. ISEC 2021. 23:1–23:5.

[50] L. C. L. Kats, E. Visser, *The Spoofax language workbench: rules for declarative specification of languages and IDEs*, OOPSLA 2010.

[51] S. Kelly, J.-P. Tolvanen, *Domain-Specific Modeling*. Wiley. 2008.

[52] S. Kelly, *Empirical comparison of language workbenches*. DSM@SPLASH 2013. 33–38.

[53] A. Kleppe, *Software Language Engineering*. Addison-Wesley. 2008.

[54] A. Kleppe, J. Warmer, *ProjectIT*. Available at: www.projectit.org.

[55] P. Klint et al., *Language Parametric Module Management for IDEs*. Electron. Notes Theor. Comput. Sci. 203(2). 3–19. 2008.

[56] G. Konat et al., *Bootstrapping domain-specific meta-languages in language workbenches*. GPCE 2016. 47–58.

[57] A. Kornstädt, E. Reiswich, *Composing Systems with Eclipse Rich Client Platform Plug-Ins*, IEEE Softw. 27(6). 78–81. 2010.

[58] G. Kotopoulos et al., *Querying MOF Repositories: The Design and Implementation of the Query Metamodel Language (QML)*. IEEE. 2007.

[59] J. de Lara et al., *Facet-oriented Modelling*. ACM Trans. Softw. Eng. Methodol. 30(3). 27:1-27:59. 2021.

[60] J. de Lara, E. Guerra, *Generic Meta-modelling with Concepts, Templates and Mixin Layers*. MoDELS (1) 2010. 16–30.

[61] R. T. Lindeman et al., *Declaratively defining domain-specific language debuggers*. GPCE 2011. 127–136.

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

[63] B. Merkle, *Textual modeling tools: overview and comparison of language workbenches*. SPLASH/OOPSLA Companion 2010. 139–148.

[64] R. Milner et al., *The Definition of Standard ML*. 1997.

[65] M. Monperrus et al., *A Definition of "Abstraction Level" for Metamodels*. ECBS 2009. 315–320.

[66] P. D. Mosses, *A Component-Based Formal Language Workbench*. F-IDE@FM 2019. 29–34.

[67] B. Mourad, M. Cimini, *System Description: Lang-n-Change - A Tool for Transforming Languages*. FLOPS 2020. 198–214.

[68] P. Neubauer et al., *XMLText: from XML Schema to Xtext*. SLE 2015.

[69] C. Omar et al., *Hazelnut: a bidirectionally typed structure editor calculus*. POPL 2017. 86–99.

[70] M. Ozkaya, D. Akdur, *What do practitioners expect from the metamodeling tools? A survey*. J. Comput. Lang. 63. 101030. 2021.

[71] A. Prinz, G. Mezei, *The Art of Bootstrapping*. MODELSWARD 2019.

[72] A. Prinz, A. Shatalin, *How to Bootstrap a Language Workbench*. MODELSWARD 2019. 345–352.

[73] P. Klint et al., *Rascal: A Domain Specific Language for Source Code Analysis and Manipulation*, SCAM 2009. 168–177.

[74] I. Ráth et al., *Synchronization of abstract and concrete syntax in domain-specific modeling languages*. Softw. Syst. Model. 9(4). 453-471. 2010.

[75] L. Renggli et al., *Embedding Languages without Breaking Tools*. ECOOP 2010. 380–404.

[76] J. Smith et al., *Category theoretic approaches of representing precise UML semantics*. ECOOP 2000.

[77] V. Sousa, E. Syriani, *An Expeditious Approach to Modeling IDE Interaction Design*. GEMOC+MPM@MoDELS 2015. 52–61.

[78] J. Steel, J.-M. Jézéquel, *On model typing*. Softw. Syst. Model. 6(4). 2007.

[79] B. Sufrin, *Formal Specification of a Display-Oriented Text Editor*. Sci. Comput. Program. 1(3). 157–202. 1982.

[80] W. Swierstra, *Data types à la carte*. J. Funct. Program. 18(4). 2008.

[81] S. Tobin-Hochstadt et al., *Languages as libraries*, PLDI 2011. 132—141.

[82] E. Visser, *Separation of concerns in language definition*. MODULARITY 2014. 1–2.

[83] M. Voelter et al., *DSL Engineering*. 2013.

[84] M. Voelter et al., *Lessons learned from developing mbeddr: a case study in language engineering with MPS*. Softw. Syst. Model. 18(1). 2019.

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

[86] I. Weisemöller, A. Schürr, *Formal Definition of MOF 2.0 Metamodel Components and Composition*. MoDELS 2008. 386–400.

[87] D. S. Wile, *Abstract Syntax from Concrete Syntax*. ICSE 1997. 472–480.

[88] C. S. Yovev, *Universal editor unattainable*. ACM SIGPLAN Notices 25(12). 89–92. 1990.

[89] V. Zaytsev, *BNF was here: what have we done about the unnecessary diversity of notation for syntactic definitions*. SAC 2012. 1910–1915.

[90] V. Zaytsev, R. Lämmel, *A Unified Format for Language Documents*. SLE 2010. 206–225.

[91] V. Zaytsev, A. H. Bagge, *Parsing in a Broad Sense*. MoDELS 2014. 50–67.