- **Language Workbench:** All the features of a language workbench.
    - **Language definition:** All the features directly concerning the definition of a language.
        - **Pragmatics**
            - **Define good practices:** Be able to define good language practices, such as design patterns.
        - **Syntax Definition:** Features concerning the definition of a language's syntaxes.
            - **Abstract syntax:** Features concerning the explicit definition of a language's abstract syntax.
                - **AS from examples:** The language designer gives examples from which the abstract syntax definition artifact is inferred. It is useful when the language designer is not a language expert.
                - **AS from scratch:** The language designer explicitly defines the abstract syntax.
            - **Concrete Syntax:** Features concerning the explicit definition of a language's abstract syntax.
                - **CS from examples:** The language designer gives examples from which the concrete syntax definition artifact is inferred. It is useful when the language designer is not a language expert.
                - **CS from scratch:** The language designer explicitly defines the abstract syntax.
            - **AS-CS Mapping:** Properties of the mapping allowing to obtain the concrete syntax from the abstract syntax or vice-versa.
                - **Expressiveness:** Characterize the allowing gap between the abstract and the concrete syntax, *i.e.*, how far concrete syntax can be from abstract syntax.
                    - **1 to 1:** Each abstract syntax element is necessarily represented by one concrete syntax element and vice-versa.
                    - **1 to [0-1]:** Each abstract syntax element is represented by at most one concrete syntax element, *i.e.*, an abstract syntax element may be not represented in the concrete syntax.
                    - **1 to *:** Each abstract syntax element can be represented by zero, one, or several concrete syntax element(s).
                    - *** to *:** Each abstract syntax element can be represented by zero, one, or several concrete syntax element(s). Conversely, a concrete syntax element can represent zero, one, or several abstract syntax element(s).
                - **Mapping specification:** How the mapping is specified in the language workbench.

            - **Implicit:** The mapping is inferred from another artifact such as the concrete syntax definition (e.g., in a grammar artifact).
            - **Explicit:** The language workbench provides a dedicated way to manually define this mapping, through a meta-language for instance.
    - **Semantics def:** Features concerning the definition of a language's semantics.
      - **Sem Type:** The different types of semantics that can be defined in the language workbench.
        - **Dynamic:** Features concerning the dynamic semantics definition, i.e. the semantics of execution.
          - **Approach:** The different dynamic semantics approach types.
            - **Translational:** Compilation to a program expressed in another language.
              - **M2T:** Compilation from a model of the developed language to the text of another language.
              - **M2M:** Compilation from a model of the developed language to the model of another language.
                - **Concrete syntax**
            - **Interpretative:** Direct execution by the host language without prior translation.
          - **Model of communication:** Capability to coordinate the behavior of several languages.
          - **Model of concurrency:** Capability to execute a model in concurrency.
        - **Static:** Features concerning the static semantics definition.
          - **Type system:** Capability to define type systems.
          - **Well-formedness rules:** Capability to define validation rules.
      - **Sem Artefact**
        - **Rules:** The semantics are defined through logical rules, in a declarative way.
        - **Grammar attributes:** The semantics are defined as grammar attributes.
        - **Programmatic def:** The semantics is directly defined in a GPL.
  - **Modeling workbench definition:** Features concerning the definition of a language workbench, *i.e.*, of the different services around the languages to assist the user.
    - **MW Syntactic services:** Services that can be explicitly defined by a language designer and concerning the syntax of the developed language.

    - **MW Highlighting:** Define styles to visually distinguish syntax elements of models in the editor using colors and others to improve readability.
    - **MW folding:** Define what can be collapsed and expanded in models based on structural elements to improve focus and readability.
    - **MW Syntactic completion:** Define possible completions for model elements based on syntax rules.
    - **MW Auto-formatting:** Define style rules to adjust indentation, spacing, layout, and structure.
  - **MW Semantic services**
    - **MW Semantic completion:** Define context-aware suggestions for models.
    - **MW Quick fixes:** Define solutions to suggest and apply for detected issues
  - **Validation:** Features concerning the validation of a model made with the developed language.
    - **Structural:** Validation of the structure of the model (syntaxes).
    - **Semantic:** Validation concerning the semantics.
      - **Naming:** Name binding
      - **Types:** Type systems
      - **Programmatic:** implemented through a GPL
    - **Formal verification:** Capability to prove parts of the language definitions, through a compilation to Coq for instance.
    - **Model test case generation:** Capability to generate models from a language definition, allowing the language designer to validate the resulting models.
  - **Custom GUI:** Capability to explicitly customize parts of GUI such as the palette, adding new buttons to a menu, etc.
  - **MW testing:** Capability to define test tools for the defined language.
    - **MW debugger:** Capability to define a debugger.
      - **Omniscient debugger:** Capability to define an omniscient debugger, *i.e.*, a debugger allowing the user to go backward in addition to forward.
- **Editor:** Available language workbench's editor features.
  - **Editing mode:** How the metamodels are edited.
    - **Free-form:** The language designer freely edits the persisted model.
    - **Representation:** The language designer edits a representation of the model and both are persisted. The representation does not necessarily have a fixed layout.
    - **Projectional:** The language designer edits a projection of the persisted model in a fixed layout.
  - **Syntactic services:** Language workbenches syntactic services.
    - **Highlighting:** Visually distinguishes syntax elements of metamodels in the editor using colors and styles to improve readability.

            - **Outline:** Displays a structured, hierarchical view of a metamodel's components to aid navigation.
            - **Folding:** Allows collapsing and expanding sections of metamodels based on structural elements to improve focus and readability.
            - **Syntactic completion:** Suggests possible completions for metamodel elements based on syntax rules.
            - **Diff:** Compares different versions of a metamodel, highlighting added, removed, or modified parts.
            - **Auto-formatting:** Automatically adjusts indentation, spacing, layout, and structure according to predefined style rules.
        - **Semantic services:** Language workbenches semantic services.
            - **Reference resolution:** Identifies and links metamodel elements to their declarations or definitions.
            - **Semantic completion:** Provides context-aware suggestions by analyzing the meaning of metamodel parts.
            - **Refactoring:** Supports automated metamodel transformations (e.g., renaming, extracting parts) to improve maintainability without altering functionality.
            - **Error marking:** Detects and highlights syntactic or semantic issues in the metamodel, often with tooltips explaining the problem.
            - **Quick fixes:** Suggests and applies automated solutions for detected issues.
            - **Origin tracking:** Keep track of metamodel's elements during the different transformation steps. Useful for error displays.
            - **Live translation:** Capability to use the designed language during its development.
        - **Viewpoint management:** How the different meta-languages are presented to the language designer.
            - **Multi-views:** Capability to propose different viewpoints over the whole defined language/modeling workbench.
            - **Blended modeling:** Capability to propose different notations for a single model.
    - **Meta-languages:** Features of the available meta-language in the language workbench.
        - **Notation:** The notation of the meta-language.
            - **Textual:** The language designer edits the metamodel through text.
                - **Symbols:** The language designer can use symbols such as mathematical notations.
            - **Graphical:** The language designer edits the metamodel through different graphical representations such as diagrams.
            - **Tabular:** The language designer edits the metamodel through a table.
            - **Tree:** The language designer edits the metamodel through a tree editor.

                - **Block:** The language designer edits the metamodel through a block representation like Blockly.
                - **Form:** The language designer edits the metamodel through a form representation.
            - **Type:** Type of the meta-language.
                - **Internal:** The meta-language is presented as an internal language of another language such as a GPL. "using a host language to give the host language the feel of a particular language." [1]
                    - **Fluent API:** Method of designing an API to allow method chaining, making code more readable and expressive.
                    - **Shadow embedding:** Capability to embed in the host language custom syntaxes. For instance, TSX from React embedding HTML concepts.
                        - **Specialization:** Capability to specialize some concepts of the host language for the internal DSL.
                - **External:** The meta-language is presented as an external language, with its own syntax uncoupled from the host language.
            - **Paradigm:** Fundamental approach of the meta-language.
                - **Imperative:** The language designer defines *how* a task should be performed, i.e., on the control flow. Examples: Java, C.
                - **Declarative:** The language designer defines *what* should be done, i.e., focus on the desired outcomes. Examples: Haskell, SQL, CSS
            - **Artifact:** Which form does the artifact made with the language workbench take?
                - **Model-based:** The artifact is a (meta-)model, possibly connected to other models of other concerns.
                - **Grammar-based:** The artifact is a grammar (mostly for concrete syntax or concrete syntax services).
                - **Template-based:** The artifact takes the form of raw data with the insertion of structured data inside. Example: Jinja (Python template engine).
        - **Testing:** Features to help the language design verify their modeling workbench/language definition.
            - **DSL debugging:** The language workbench provides a debugger to debug some modeling workbench/language definition concerns.
            - **DSL testing:** The language workbench provides ways to unit-test the different language concerns.
            - **Editor testing:** The language workbench provides ways to test editor concerns such as interaction with the representation, auto-completion, …
        - **Composability:** Features concerning the capacity of the language workbench to produce languages by reusing (parts of) existing languages.
            - **Feature:** Which concerns can be composed?

            - **Syntax/view**
            - **Validation**
            - **Semantics**
            - **Editor services**
        - **Operator:** Language composition operators, according to Erdweg et al. [2].
            - **Extension:** A language designer combines a base language with a language extension. The extension depends on the base language and its implementation may imply the modification of the base language implementation.
            - **Unification:** A language designer combines two independent languages by unifying them with glue code.
            - **Extension composition:** the capability of language extensions to work together. That is, whether language extensions can be composed, either through the incremental extension of a language or by the union of independent extensions.
            - **Restriction:** A language designer can remove parts of an existing language.
        - **Language component repository:** The language workbench provides a shared registry containing languages or language slices that can be reused to build a new language. A language designer can contribute to this registry.
    - **Collaboration:** Features specific to the collaboration between different language designers.
        - **Live collaboration:** Collaboration at the same moment in time.
            - **Strategy:** How the live collaboration is done inside the language workbench.
                - **Optimistic:** Language designers can edit the same model or even the same element at the same time. Requires modification merge strategy.
                - **Pessimistic:** Editing an element causes it and possibly its related items to be locked.
            - **Collaboration architecture:** In technical terms, how the collaboration is done among the different language designer's clients.
                - **Distributed:** Each client is independent and can work offline. The data are exchanged among the different clients (*e.g.*, Git or CRDT).
                - **Centralized:** A central server is required to manage and control the collaboration (*e.g.*, SVN).
        - **Versioning:** The language workbench proposes an integrated way to version developed languages.
    - **Architecture:** Features concerning the architecture of the language workbench.
        - **Platform:** On which kind of platform does the language workbench run?

            - **Desktop:** The backend and frontend of the language workbench cannot be uncoupled and are both directly executed on the language designer device.
            - **Cloud-native:** The backend and frontend of the language workbench are uncoupled and may be executed on different devices.
        - **Modular:** The language workbench is thought to be extended, proposing APIs.
    - **MW production:** Features concerning the way the modeling workbench and associated languages are produced by the language workbench.
        - **Method:** How does the language workbench produce the modeling workbench and associated languages?
            - **Generation:** The language workbench produces artifacts that must be run separately from the language workbench.
            - **Interpretation:** The language workbench directly interprets the modeling workbench/languages inside its own environment.
        - **Continuous production:** The language workbench can continually produce the modeling workbench/languages.
    - **Co-evolution:** Features proposed to the language designer to facilitate the co-evolution between the language they build and the existing models based on previous versions of the language.
        - **Depreciation:** The language designer can annotate parts of their language to notify users that these parts will be removed in future versions.
        - **Model migration customization:** The language designer can define how to migrate a model from the previous version to the next one.
        - **Automatic model migration:** The language workbench provides default rules to automatically migrate models to a new version of the language.
        - **Version check:** The language workbench provides a way to check if a model is compatible with the new language version.

[1] https://martinfowler.com/dsl.html
[2] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language composition untangled. In Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA '12). Association for Computing Machinery, New York, NY, USA, Article 7, 1–8. https://doi.org/10.1145/2427048.2427055