The State of the Art in Language Workbenches

Conclusions from the Language Workbench Challenge

Sebastian Erdweg[1], Tijs van der Storm[2,3], Markus Völter[4], Meinte Boersma[5], Remi Bosman[6], William R. Cook[7], Albert Gerritsen[6], Angelo Hulshout[8], Steven Kelly[9], Alex Loh[7], Gabriël Konat[10], Pedro J. Molina[11], Martin Palatnik[6], Risto Pohjonen[9], Eugen Schindler[6], Klemens Schindler[6], Riccardo Solmi[12], Vlad Vergu[10], Eelco Visser[10], Kevin van der Vlist[13], Guido Wachsmuth[10], and Jimi van der Woning[13]

[1] TU Darmstadt, Germany [2] CWI, Amsterdam, The Netherlands [3] INRIA Lille Nord Europe, Lille, France [4] voelter.de, Stuttgart, Germany [5] DSL Consultancy, Leiden, The Netherlands [6] Sioux, Eindhoven, The Netherlands [7] University of Texas, Austin, US [8] Delphino Consultancy, Best, The Netherlands [9] MetaCase, Jyväskylä, Finland [10] TU Delft, The Netherlands [11] Icinetic, Sevilla, Spain [12] Independent, Bologna, Italy [13] Universiteit van Amsterdam

Abstract. Language workbenches are tools that provide high-level mechanisms for the implementation of (domain-specific) languages. Language workbenches are an active area of research that also receives many contributions from industry. To compare and discuss existing language workbenches, the annual Language Workbench Challenge was launched in 2011. Each year, participants are challenged to realize a given domain-specific language with their workbenches as a basis for discussion and comparison. In this paper, we describe the state of the art of language workbenches as observed in the previous editions of the Language Workbench Challenge. In particular, we capture the design space of language workbenches in a feature model and show where in this design space the participants of the 2013 Language Workbench Challenge reside. We compare these workbenches based on a DSL for questionnaires that was realized in all workbenches.

1 Introduction

Language workbenches, a term popularized by Martin Fowler in 2005 [19], are tools that support the efficient definition, reuse and composition of languages and their IDEs. Language workbenches make the development of new languages affordable and, therefore, support a new quality of *language engineering*, where sets of syntactically and semantically integrated languages can be built with comparably little effort. This can lead to multi-paradigm and language-oriented programming environments [8, 61] that can address important software engineering challenges.

Almost as long as programmers have built languages, they have also built tools to make language development easier and language use more productive. The earliest language workbench probably was SEM [52]; other early ones include MetaPlex [7], Metaview [51], QuickSpec [43], and MetaEdit [48]. Graphical workbenches that are still being developed today include MetaEdit+ [28], DOME [24], and GME [38]. On

the other hand, language workbenches that supported textual notations include Centaur [5], the Synthesizer generator [46], the ASF+SDF Meta-Environment [30], GemMex/Montages [2], LRC [36], and Lisa [42]. These systems were originally based on tools for the formal specification of general purpose programming languages [20]. Nonetheless, many of them have been successfully used to build practical domain-specific languages (DSLs) as well [41]. Textual workbenches like JastAdd [49], Rascal [32, 33], Spoofax [27], and Xtext [17] can be seen as successors of these systems, leveraging advances in editor technology of mainstream IDEs. At the same time, projectional language workbenches like MPS [57] and Intentional [47] are reviving and refining the old idea of structure editors [9], opening up the possibility of mixing arbitrary notations.

Throughout their development, language workbenches and domain-specific languages have been used in industry. Examples include:

- Eurofighter Typhoon [1], with IPSYS's HOOD toolset (later ToolBuilder).
- Nokia's feature phones [44], with MetaEdit+.
- RISLA, a DSL for interest-rate products [3], with ASF+SDF.
- Polar's heart rate monitors and sports watches [26], with MetaEdit+.
- WebDSL [56] and Mobl [22] for building Web applications and mobile applications respectively, with Spoofax.
- File format DSL for digital forensics tool construction [53], with Rascal.
- mbeddr [58, 59] a C-based language for embedded software development, including extensions such as units of measure, components, requirements tracing, and variability, based on MPS.

Language workbenches are currently enjoying significant growth in number and diversity, driven by both academia and industry. Existing language workbenches are so different in design, supported features, and used terminology that it is hard for users and developers to understand the underlying principles and design alternatives. To this end, a systematic overview is helpful.

The goal of the Language Workbench Challenge (LWC) is to promote understanding and knowledge exchange on language workbenches: Each year a language engineering challenge is posed and the submissions (often but not exclusively by tool developers) implement the challenge; documentation is required as well, so others can understand the implementation. All contributors then meet to discuss the submitted solutions. By tackling a common challenge, the approaches followed by different workbenches become transparent, and understanding about design decisions, capabilities, and limitations increases. In this paper, we channel the lessons learnt from the previous iterations of the LWC and document this knowledge for the scientific community at large. In particular, we make the following contributions:

- We describe the history of the LWC.
- We establish a feature model that captures the design space of language workbenches as observed in the previous LWCs.
- We present and discuss the 10 language workbenches participating in LWC'13 by classifying them according to our feature model.
- We present empirical data on 10 implementations of the LWC'13 assignment (a questionnaire DSL).
- Based on our investigation, we document the state of the art of language workbenches.

2 Background

The idea of the LWC was born during discussions at the 2010 edition of the Code Generation conference. Since then, LWC has been held three times, each year with a different language to implement as assignment. Below we briefly review the assignments of 2011, 2012, and 2013. Then we describe the methodology we followed in this paper.

2.1 The Challenges of LWC

The LWC'11 assignment[1] consisted of a simple language for defining entities and relations. At the basic level, this involved defining syntax for entities, simple constraint checking (e.g., name uniqueness), and code generation to a general-purpose language. At the more advanced level, the challenge included support for namespaces, a language for defining entity instances, the translation of entity programs to relational database models, and integration with manually written code in some general-purpose language. To demonstrate language modularity and composition, the advanced part of the assignment should be realized without modifying the solution of the basic assignment.

In the LWC'12 assignment[2], two languages had to be implemented. The first language captured piping and instrumentation models which can be used, for instance, to describe heating systems. The elements of this language included pumps, valves, and boilers. The second language consisted of a state machine-like controller language that could be used to describe the dynamic behavior of piping and instrumentation models. Developers were supposed to combine the two languages to enable the simulation of piping and instrumentation systems.

The LWC'13 assignment[3] consisted of a DSL for questionnaires, which should be rendered as an interactive GUI that reacts to user input to present additional questions. The questionnaire definition should be validated, for instance, to detect unresolved names and type errors. In addition to basic editor support, participants should modularly develop a styling DSL that can be used to configure the rendering of a questionnaire. We describe the details of the LWC'13 assignment in Section 5.

2.2 Research Methodology

The main goal of this paper is to document the state of the art of language workbenches in a structured and informative way. We assemble the relevant information based on our experience and involvement in the LWC from 2011 to 2013. Nevertheless, for this paper we focused on the most recent challenge of 2013. We invited all participants of LWC'13 to contribute to the domain analysis and to the language workbench comparison as described below.

Domain analysis. The first part of our methodology addresses the goal of accurately describing the domain of language workbenches. We have asked all participants of LWC'13 to provide a detailed list of features supported by their language workbench.

[1] `http://www.languageworkbenches.net/index.php?title=LWC_2011`
[2] `http://www.languageworkbenches.net/index.php?title=LWC_2012`
[3] `http://www.languageworkbenches.net/index.php?title=LWC_2013`

The first three authors then started to "mine" a feature model [25] to capture the relevant aspects of the language-workbench domain. Since non-functional features have not been in scope of any previous LWC, we solely focused on the functional properties of language workbenches. The extracted feature model was then presented to all participants for feedback. The refined feature model presented in Section 3 provides a way to categorize language workbenches according to which features they support.

Empirical data. In addition to a general overview of language workbenches, we investigated empirical data on the solutions submitted to the LWC'13. We constructed a feature model for the features of the questionnaire DSL and asked the participants to indicate which features they realized in their solution. We present a description of the assignment and the feature model in Section 5.

To get an impression about how different language workbenches achieve various (subsets of) features of the questionnaire DSL, we also asked all participants to answer the following three questions:

- *What is the size of your solution?* The suggested metric for the answer was SLOC (Source Lines of Code)[4].
- *What are the static, compile-time dependencies?* This captures the various libraries, frameworks, and platforms that are needed to run the compiler and IDE of the questionnaire DSL.
- *What are the dynamic, runtime dependencies?* This addresses the additional software components that are needed to run the generated questionnaires GUIs.

We present the answers to these questions and discuss the language workbenches in view of these results in Section 6 and Section 7 respectively.

Generality of the survey. Not all existing language workbenches were represented at LWC'13. Language workbenches that contributed to earlier challenges, but not to LWC'13, include commercial ones, such as the Intentional workbench [47], OOMega[5], and Obeo Designer[6], as well as academic systems such as Atom3 [37], Cedalion [39], and EMFText [21]. As we show in Section 4, the language workbenches covered in our study are very diverse regarding the features they support. To our knowledge, the features of aforementioned language workbenches are covered by our feature model. Hence, even though not all language workbenches are part of this survey, we consider the *domain* of language workbenches sufficiently covered.

3 A Feature Model for Language Workbenches

Language workbenches exist in many different flavors, but they are united by their common goal to facilitate the development of (domain-specific) languages. Based on input provided by the participants of LWC'13, we derived the feature model shown in Fig. 1. It outlines the most important features of language workbenches. We use standard feature-diagram notation and interpretation [4]: The root node (*Language workbench*

[4] SLOC does not count comments or empty lines. Note that SLOC only works for textual languages; we come back to this problem in Section 6.

[5] `http://www.oomega.net/`

[6] `http://www.obeodesigner.com/`

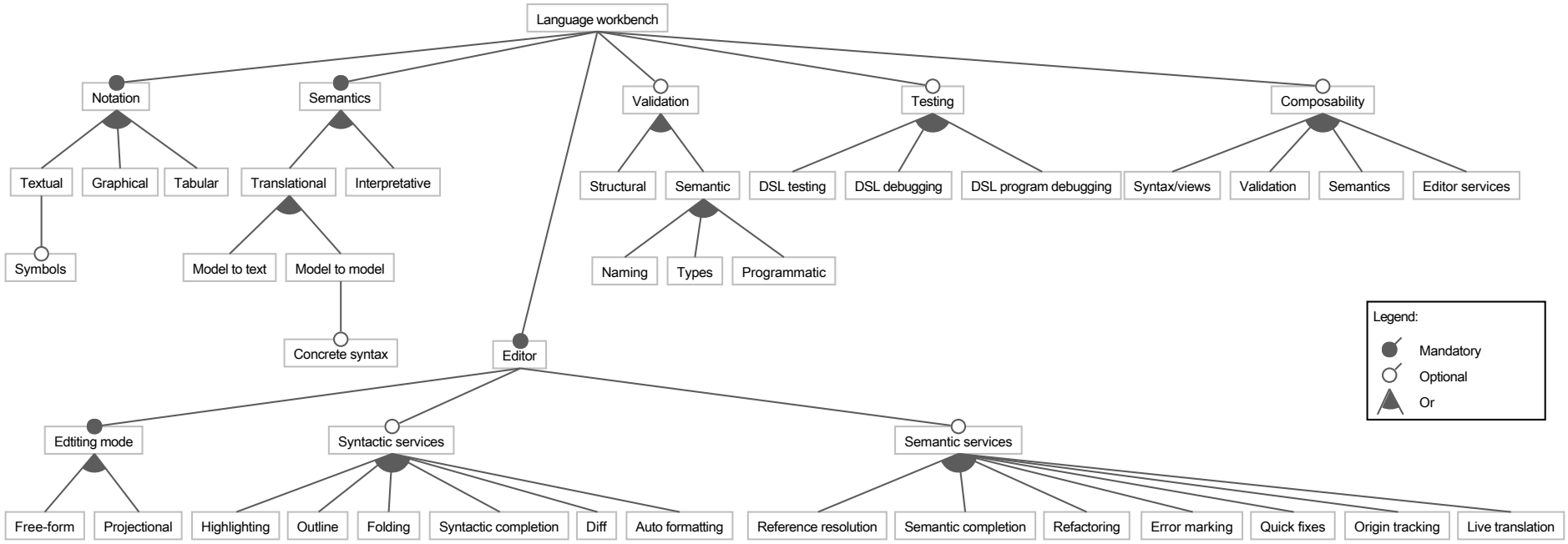

Fig. 1: Feature model for language workbenches. With few exceptions, all features in the feature model apply to the languages that can be defined with a language workbench, and not to the definition mechanism of the language workbench itself.

in Fig. 1) is always selected. A mandatory feature (filled circle) has to be selected if its parent is selected. An optional feature (empty circle) does not have to be selected even if its parent is selected. In a list of *Or* children (filled edge connector), at least one feature has to be selected if the parent is selected.

We separate language workbench features into six subcategories. A language workbench *must* support notation, semantics, and an editor for the defined languages and its models. It *may* support validation of models, testing and debugging of models and the language definition, as well as composition of different aspects of multiple defined languages. In the remainder of this section, we explain the feature model in more detail.

Every language workbench must support the mandatory feature *notation*, which determines how programs or models are presented to users. The notation can be a mix of textual, graphical, and tabular notations, where textual notation may optionally support symbols such as integrals or fraction bars embedded in regular text.

A language workbench must support the definition of language *semantics*. We distinguish translational semantics, which compiles a model into a program expressed in another language, and interpretative semantics, which directly executes a model without prior translation. For translational semantics we distinguish between model-to-text translations, which are based on concatenating strings, and model-to-model translations, which are based on mapping abstract model representations such as trees or graphs. To simplify the handling of abstract model representations, some language workbenches support concrete syntax for source and target languages in transformation rules.

Editor support is a central pillar of language workbenches [19] and we consider user-defined editor support mandatory for language workbenches. The two predominant editing modes are free-form editing, where the user freely edits the persisted model (typically the source code), and projectional editing, where the user edits a projection of the persisted model in a standard, fixed layout. In addition to a plain editor, most language workbenches provide a selection of syntactic and semantic editor services. Syntactic editor services include:

- Customizable visual *highlighting* in models, such as language-specific syntax coloring for textual languages or language-specific node shapes for graphical languages.
- Navigation support via an *outline* view.
- *Folding* to hide part of a model.
- Code assist through *syntactic completion* templates that suggest code, graph, or tabular fragments to the user.
- Comparison of programs via a *diff*-like tool (the basis for version control).
- *Auto formatting*, restructuring, aligning, or layouting of a model's presentation.

Semantic editor services include:

- *Reference resolution* to link different concepts of the defined language such as declarations and usages of variables.
- Code assist through *semantic completion* that incorporates semantic information such as reference resolution or typing into the completion proposal.
- Semantics-preserving *refactorings* of programs or models, ranging from simple renaming to language-specific restructuring.
- In case an error is detected in the model, an *error marker* highlights the involved model element and presents the error message to the user.

– *Quick fixes* may propose ways of fixing such an error. When the user selects any of the proposed fixes, the faulty model is automatically repaired.
– When transforming models, keeping track of a model's *origin* enables linking elements of the transformation result back to the original input model. This is particularly useful for locating the origin of a static or dynamic error in generated code. It is also useful in debugging.
– To better understand the behavior of a model, it can be useful to have a view of the code that a model compiles to. Language workbenches that feature *live translation* can display the model and the generated code side-by-side and update the generated code whenever the original model changes.

In addition to the above services, the language editor provided by most language workbenches can display information about the result of language-specific *validations*. We distinguish validations that are merely structural, such as containment or multiplicity requirements between different concepts, and validations that are more semantic, such as name or type analysis. Language workbenches may facilitate the definition of user-defined type systems or name binding rules. However, many language workbenches do not provide a declarative validation mechanisms and instead allow the definition of validation rules programmatically in a general-purpose programming language.

Another important aspect of building languages is *testing* of the language definition. Testing a language definition may be supported by unit-testing the different language aspects: the syntax (parser or projections), semantics (translation or interpretation), editor (completion, reference resolution, refactoring, etc.), and validation (structure or types). Some language workbenches support debugging. We distinguish between support for debugging the language definition (validation or semantics), and support for constructing debuggers for the defined language. The latter allows, for instance, the definition of domain-specific views to display variable bindings, or specific functionality for setting breakpoints.

Finally, *composability* of language definitions is a key requirement for supporting language-oriented programming [8, 61] where software developers use multiple languages to address different aspects of a software system. Language workbenches may support *incremental extension* (syntactic integration of one language into another) and *language unification* (independent languages can be unified into a single language) [12]. This composition should be achieved for all aspects of a language: syntax, validation, semantics, and editor services.

In summary, our feature model captures most of the design space for language workbenches. In creating this feature model, we ignored *how* the various features can be supported by a language workbench. This is the focus of the subsequent section.

4 Language Workbenches

In this section, we introduce the language workbenches that participated at LWC'13 and show which features of our feature model they support.

4.1 Introduction of the Tools

Ensō (since 2010, `http://www.enso-lang.org`) is a greenfield project to enable a software development paradigm based on interpretation and integration of executable

specification languages. Ensō has its roots in an enterprise application engine developed at Allegis starting in 1998, which included integrated but modular interpreters for semantic data modeling, policy-based security, web user interfaces, and workflows. Between 2003 and 2010 numerous prototypes were produced that sought to refine the vision and establish an academic foundation for the project. The current version (started in 2010) is implemented in Ruby. Rather than integrate with an existing IDE, Ensō seeks to eventually create its own IDE. The goal of the project is to explore new approaches to the model-based software development paradigm.

Más (since 2011, `http://www.mas-wb.com`) is a web-based workbench for the creation of domain-specific languages and models. Más uses projectional editing to provide convenient styling of models and an intuitive editor experience for "non-dev" users, and makes language definition as simple as possible. Language semantics is defined through "activations", consisting, for instance, of declarative code generation templates. Más aims at lowering the entry barrier for language creation far enough to allow adoption and scaling of the model-driven approach across disciplines and industries.

MetaEdit+ (since 1995, `http://www.metacase.com`) is a mature, platform-independent, graphical language workbench for domain-specific modeling [28]. MetaEdit+ aims to be the easiest domain-modeling tool to learn and to use, removing accidental complexity to allow users to concentrate on creating productive languages and good models. MetaEdit+ is commercially successful, used by customers in both industry and academia. Empirical research has consistently shown that MetaEdit+ increases productivity of developers by a factor of 5–10 compared to programming [26, 29, 44].

MPS (since 2003, `http://www.jetbrains.com/mps/`) is an open-source language workbench developed by JetBrains. Its most distinguishing feature is a projectional editor that supports integrated textual, symbolic, and tabular notations, as well as wide-ranging support for composition and extension of languages and editors. MPS realizes the language-oriented programming paradigm introduced by Sergey Dmitriev [8] and has evolved into a mature and well-documented tool. It is used by JetBrains internally to develop various web-based tools such as the Youtrack bugtracker. It has also been used to develop various systems outside of JetBrains, the biggest one probably being the mbeddr tool for embedded software development [58].

Onion (since 2012) is a language workbench and base infrastructure implemented in .NET for assisting in the creation of DSLs. Onion has evolved from Essential (2008), a textual language workbench with a focus on model interpretation and code generation. The main goals of the Onion design is to provide the tools to speed up DSL creation for different notations (text, graphical, projectional) and provide scalability for big models via partitioning and merging capabilities. Onion emphasizes speed of parsing and code generation, enabling real-time synchronization of models and generated code.

Rascal (since 2009, `http://www.rascal-mpl.org`) is an extensible metaprogramming language and IDE for source code analysis and transformation [23, 32, 33, 54]. Rascal combines and unifies features found in other tools for source code manipulation and language workbenches. Rascal provides a simple, programmatic interface to extend the Eclipse IDE with custom IDE support for new languages. Rascal is currently used as a research vehicle for analyzing existing software and the implementation of DSLs.

It provides the implementation platform for a real-life DSL in the domain of digital forensics [53]. The tool is accompanied with interactive online documentation and is regularly released as a self-contained Eclipse plugin.

Spoofax (since 2007, `http://www.spoofax.org`) is an Eclipse-based language workbench for efficient development of textual domain-specific languages with full IDE support [27]. In Spoofax, languages are specified in declarative meta-DSLs for syntax (SDF3 [60]), name binding (NaBL [34]), editor services, and transformations (Stratego [6]). From these specifications, Spoofax generates and dynamically loads an Eclipse-based IDE which allows languages to be developed and used inside the same Eclipse instance. Spoofax is used to implement its own meta-DSLs. Spoofax has been used to develop WebDSL [56] and Mobl [22], and is being used by Oracle for internal projects.

SugarJ (since 2010, `http://www.sugarj.org`) is a Java-based extensible programming language that allows programmers to extend the base language with custom language features [11, 14]. A SugarJ extension is defined with declarative meta-DSLs (SDF, Stratego, and a type-system DSL [40]) as part of the user program and can be activated in the scope of a module through regular import statements. SugarJ also comes with a Spoofax-based IDE [13] that can be customized via library import on a file-by-file basis. A language extension can use arbitrary context-free and layout-sensitive syntax [15] that does not have to align with the syntax or semantics of the base language Java. Therefore, SugarJ is well-suited for the implementation of DSLs that combine the benefits of internal and external DSLs. Variants of SugarJ support other base languages: JavaScript, Prolog, and Haskell [16].

Whole Platform (since 2005, `http://whole.sourceforge.net`) is a mature projectional language workbench supporting language-oriented programming [50]. It is mostly used to engineer software product lines in the financial domain due to its ability to define and manage both data formats and pipelines of model transformations over big data. The Whole Platform aims to minimize the explicit metamodeling efforts, so that users can concentrate on modeling. The Whole Platform aims to reduce the use of monolithic languages and leverages grammar-based data formats for integrating with legacy systems.

Xtext (since 2006, `http://www.eclipse.org/Xtext/`) is a mature open-source framework for development of programming languages and DSLs. It is designed based on proven compiler construction patterns and ships with many commonly used language features, such as a workspace indexer and a reusable expression language [10]. Its flexible architecture allows developers to start by reusing well-established and commonly understood default semantics for many language aspects, but Xtext scales up to full programming language implementations, where every single aspect can be customized in straightforward ways by means of dependency injection. Companies like Google, IBM, BMW and many others have built external and internal products based on Xtext.

4.2 Language Workbench Features

We position the language workbenches above in the design space captured by our feature model as displayed in Table 1. In the remainder of this subsection, we reflect on some of the findings.

		Ensō	Más	MetaEdit+	MPS	Onion	Rascal	Spoofax	SugarJ	Whole	Xtext
Notation	Textual	●	●		●	●	●	●	●	●	●
	Graphical	●	◒	●			◒			●	
	Tabular		●	●	●					●	
	Symbols			●	●					●	
Semantics	Model2Text		●	●	●	●	●	●	●	●	●
	Model2Model			●	●	●	●	●	●	●	●
	Concrete syntax			●	●	●	●	●	●		
	Interpretative	●		●	●		◒	●		●	●
Validation	Structural	●	●	●	●	●	●	●	●	●	●
	Naming	◒	●	●	●	●		●		●	◒
	Types				●				●		●
	Programmatic	●			●	●	●	●	●		●
Testing	DSL testing				●		◒	●		●	●
	DSL debugging	●		●	●		●			●	●
	DSL prog. debugging	●			●					●	●
Composability	Syntax/views	●		●	●	●	●	●	●	●	◒
	Validation			●	●	●	●	●	●	●	●
	Semantics	●		●	●	●	●	●	●		●
	Editor services			●	●	●	●	●	●		●
Editing mode	Free-form	●		●		●	●	●	●		●
	Projectional		●		●	●				●	
Syntactic services	Highlighting		◒	●	●	●	●	●	●	●	●
	Outline			●	●	●	●	●	●	●	●
	Folding		●	●	●	●	●	●	●	●	●
	Syntactic completion			●	●	●		●	●		●
	Diff	●		●	●	●	●	●	●		●
	Auto formatting	●	●	●	●	●	●	●		●	●
Semantic services	Reference resolution		●	●	●	●	●	●	●		●
	Semantic completion		●	●	●	●	●	●	●	●	●
	Refactoring		◒	●	●		●	●		●	
	Error marking		●	●	●	●	●	●	●	●	●
	Quick fixes				●						●
	Origin tracking	●		●	●		●	●	●		●
	Live translation			●		●	◒	●		●	●

Table 1: Language Workbench Features (● = full support, ◒ = partial/limited support)

Notation and editing mode. Most language workbenches provide support for textual notations. Only MetaEdit+ is strictly non-textual. Más, MetaEdit+, MPS, and the Whole Platform provide support for tabular notations. Más, MPS and Onion employ projectional editing, which simplifies the integration of multiple notation styles. Currently, only Ensō combines textual and graphical notations by providing support for custom projections into diagram editors. All other language workbenches only support textual notation,

edited in a free-form text editor. MetaEdit+, MPS, and the Whole Platform also support mathematical symbols, such as integral symbols or fractions.

Semantics. Except for Ensō, all language workbenches follow a generative approach, most of them featuring both model-to-text and model-to-model transformations, and many additionally supporting interpretation of models. In contrast, Ensō eschews generation of code and is solely based on interpreters, following the working hypothesis that interpreters compose better than generators.

Validation. Some language workbenches lack dedicated support for type checking and/or constraints. These concerns are either dealt with programmatically, or assumed to be addressed by the use of semantically rich meta models. MPS, SugarJ [40], and Xtext provide declarative languages for the definition of type systems. Spoofax has a declarative language for describing name binding rules [34].

Testing. MPS, Spoofax, and Xtext feature dedicated sublanguages for testing aspects of a DSL implementations, such as parsing, name binding, and type checking. Rascal partially supports testing for DSLs through a generic unit testing and randomized testing framework. Five language workbenches provide debuggable specification languages. Four language workbenches support the debugging of DSL programs. For example, Xtext automatically supports debugging for programs that build on Xbase and compile to Java. MPS has a debugger API that can be used to build language-specific debuggers. It also defines a DSL for easily defining how debugging of language extension works. Both Xtext and MPS rely on origin tracking of data created during generation. In the Whole Platform both metalanguage and defined language can be debugged using the same infrastructure which has support for conditional breakpoints and variable views.

Composability. Composability allows languages to be built by composing separate, reusable building blocks. Ensō, Rascal, Spoofax, and SugarJ obtain syntactic composability through the use of generalized parsing technology, which is required because only the full class of context-free grammars is closed under union. The composability of Xtext grammars is limited, since it is built on top of ANTLR's LL(*) algorithm [45]. Syntactic composition in Onion is based on composing PEG [18] grammars. The language workbenches MPS and MetaEdit+, which do not use parsing at all, allow arbitrary notations to be combined.

The composability of validation and semantics in Rascal, Spoofax, and SugarJ is based on the principle of composing sets of rewrite rules. In Ensō, composition of semantics is achieved by using the object-oriented principles of inheritance and delegation in interpreter code. In MPS, different language aspects use different means of composition. For example, the type system relies on declarative typing rules which can be simply composed. On the other hand, the composition of transformations relies on the pair-wise specification of relative priorities between transformation rules.

Editor. The free-form textual language workbenches that are built on Eclipse (Rascal, Spoofax, SugarJ, Xtext) all provide roughly the same set of IDE features: syntax coloring, outlining, folding, reference resolution, and semantic completion. Spoofax, SugarJ, and Xtext have support for syntactic completion. Rascal, Spoofax, and Xtext allow the definition of custom formatters to automatically layout DSL programs. Projectional editors such as MPS, Whole Platform or Más always format a program as part of the

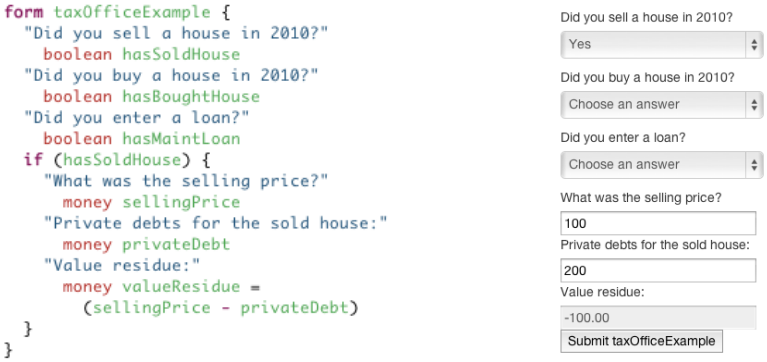

Fig. 2: An example of a textual QL model (left) and its default rendering (right).

projection rules, so this feature is implicit. Textual free-form language workbenches get the Diff feature for free by reusing existing version-control systems. MPS comes with a dedicated three-way diff/merge facility that works at the level of the projected syntax. MetaEdit+ provides a dedicated differencing mechanism so that modelers can inspect recent changes; for version-control a shared repository is used.

5 LWC 2013 Assignment: A DSL for Questionnaires

We use the assignment of LWC'13 for comparing the language workbenches introduced in the previous section. In the present section, we briefly introduce the assignment and its challenges, which was to develop a Questionnaire Language (QL)[7]. The questionnaire language was selected based on the expectation that it could be completed "after-hours" and that it would not be biased towards one particular style of language workbenches (e.g., graphical or textual). We have had no feedback indicating that the assignment was infeasible or unsuitable.

A questionnaire consists of a sequence of questions and derived values. A question may be conditionally visible based on the values of earlier questions. A questionnaire is presented to a user by rendering it as a GUI, as exemplified in Fig. 2. In addition to these mandatory features, we asked participants to realize a number of optional features. All features are shown in the feature model of Fig. 3. Specifically, we asked for a QL language and IDE implementation supporting the following features:

- Syntax: provide concrete and abstract syntax for QL models.
- Rendering: compile to code that executes a questionnaire GUI (or interpret directly).
- Propagation: generate code that ensures that computed questions update their value as soon as any of their (transitive) dependencies changes.
- Saving: generate code that allows questionnaire users to persist the values entered into the questionnaire.
- Names: ensure that no undefined names are used in expressions.
- Types: check that conditions and expressions are well-typed.

[7] Original assignment text: `http://www.languageworkbenches.net/images/5/53/Ql.pdf`.

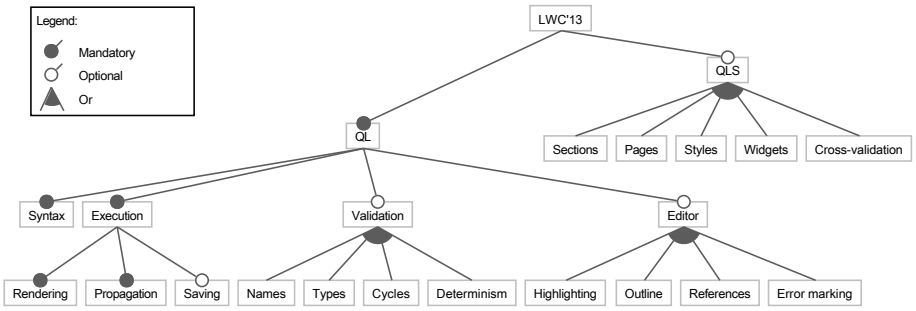

Fig. 3: Feature model of the QL assignment.

– Cycles: detect cyclic dependencies through conditions and expressions.
– Determinism: check that no two versions of equally-named questions are visible simultaneously (requires SAT solving or model checking).
– Highlighting: provide customized visual clues to distinguish language constructs.
– Outline: provide a hierarchical view or projection of QL models.
– References: support go-to-definition for variables used in conditions and expressions.
– Error marking: visually mark offending source-model elements in case of errors.

We also asked participants to develop a second language called QLS for declaring the style and layout of QL questionnaires. QL has the requirement that it should be possible to apply a QLS specification to an existing questionnaire without anticipation in the definition of the questionnaire itself. Specifically, we asked for the following features:

– Sectioning: allow questions to be (re)arranged in sections and subsections.
– Pagination: allow questions to be distributed over multiple pages.
– Styling: allow customization of fonts, colors, and font styles for question labels.
– Widgets: enable the selection of alternative widget styles for answering questions.
– Cross-validation: check that the references within a QLS specification refer to valid entities of the corresponding questionnaire model.

Taken together, there are 17 features of which 3 are mandatory (syntax, rendering and propagation). The next section discusses empirical data on the submitted solutions themselves.

6 Results

The results presented in this section are based on the solutions submitted to LWC'13 (links to the sources of these solutions are listed in Table 2). In Table 3 show for each language workbench which features the corresponding QL/QLS implementation supports. The feature-based categorization of the solutions provides a qualitative frame of reference for interpreting the size and dependency results given in Table 4. To indicate the completeness of a solution, we computed feature coverage as shown in the bottom row of Table 3. The coverage is computed by counting the number of supported features ($\bullet = 1$, $\half = 0.5$), and then dividing by the total number of features (17).

Lang. Workbench	Links to the corresponding QL solutions
Ensō	https://github.com/enso-lang/enso/tree/master/demos/Questionaire
Más	http://www.mas-wb.com/secure/concrete/language?id=120001&securityToken=restricted_public_token http://www.mas-wb.com/languages/inspector?id=120001
MetaEdit+	http://www.metacase.com/support/50/repository/LWC2013.zip
MPS	http://code.google.com/p/mps-lwc13
Onion	https://bitbucket.org/icinetic/lwc2013-icinetic
Rascal	https://github.com/cwi-swat/QL-R-kemi
Spoofax	https://github.com/metaborg/lwc2013
SugarJ	https://github.com/seba--/sugarj/tree/questionnaire/case-studies/questionnaire-language
Whole Platform	https://github.com/wholeplatform/whole-examples/tree/master/org.whole.crossexamples.lwc13
Xtext	http://code.google.com/a/eclipselabs.org/p/lwc13-xtext/

Table 2: Published sources of the QL solutions.

Table 4 summarizes the results on the size of each QL/QLS solution. As a size metric, we use the number of source lines of code (SLOC), excluding empty lines and comments. Because in some language workbenches non-textual notations are used to realize (parts of) the solution, SLOC does not tell the whole story. In these cases, we also count and report the number of model elements (NME). Model elements include any kind of structural entity that is used to define aspects of a language. For example, in MetaEdit+, modeling elements include graphs, objects, relationships, roles, and properties.

For the textual language workbenches Ensō, Onion, Rascal, Spoofax, SugarJ, and Xtext, SLOC were measured using the script `cloc.pl`[8] or by manual count. For Más, MetaEdit+, and the Whole Platform we counted the number of model elements and measured the size additional code artifacts. Since MPS is purely projectional but still provides a textual presentation of languages, we use an approximate SLOC count: We counted modeling elements and computed SLOC of an equivalent Java program by multiplying the number of model elements with different factors for different types of modeling elements [59]. In addition we report the number of SLOC/NME per feature. The number is obtained by dividing the total SLOC/NME by the number of supported features. Finally, the table also shows the compile-time and runtime dependencies of each solution to appreciate the complexity of deploying the resulting QL/QLS IDE and the generated questionnaire applications.

It is important to realize it is not our intention to present the quantitative results of Table 4 as an absolute measure of implementation effort or complexity (as is, e.g., done in [35]). They cannot be used to rank language workbenches. Factors that prevent such ranking include:

- The SLOC count is incomplete in systems where non-textual languages are used, such as in Más, MetaEdit+, MPS and Whole Platform. The NME count only partially makes up for this.
- A single number of SLOC is presented, but in each language workbench (a multiplicity of) different programming, modeling, and specification languages are used.
- The architecture and design may be substantially different across QL/QLS solutions. For instance, choosing a client-server Web architecture over a desktop GUI design may or may not affect SLOC.

[8] `http://cloc.sourceforge.net`

		Ensō	Más	MetaEdit+	MPS	Onion	Rascal	Spoofax	SugarJ	Whole	Xtext
	Syntax	●	●	●	●	●	●	●	●	●	●
Execution	Rendering	●	●	●	●	●	●	●	●	●	●
	Propagation	●	●	●	●	●	●	●	●	●	●
	Saving	●		●	●	●	●	●		●	●
Validation	Names		●	●	●	●	●	●	●	●	●
	Types		◒	●	●	●	●	●	●		●
	Cycles					●		●			●
	Determinism							◒	●		
IDE	Coloring		●	●	●	●	●	●	●	●	●
	Outline			●	●	●	●	●	●	●	●
	References		●	●	●		●	●	●	●	●
	Marking		●	●	●	●	●	●	●	●	●
QLS	Sectioning			●		●	●	●		●	●
	Pagination			●			●	●			●
	Styling			●	◒	●	●	●		●	●
	Widgets			●	●	●	●	●			●
	Validation			●	●	●	●	●		●	●
Feature coverage (in percent)		24	44	88	74	82	88	97	59	65	94

Table 3: Implemented QL and QLS features per language workbench (● = "fully implemented", ◒ = "partially implemented").

- Different QL/QLS features may require varying amounts of effort, which may not be reflected in SLOC. Furthermore, the degree as to how much effort is needed for a particular feature may vary per language workbench. The coarse granularity of the QL feature model may obscure this even more. For instance, the feature model does not distinguish between the number of questionnaire data types that are supported.
- Even though, intuitively, more features would imply more effort, this relation is almost certainly not linear, since more features increase the risk of feature interaction. The SLOC/feature metric ignores this aspect.
- The SLOC count may be influenced by the developer's familiarity with the language workbench. For instance, some of the solutions have been developed by the language workbench implementors themselves (e.g., Más, SugarJ), whereas others are built by first-time (e.g., MPS) or second-time (e.g., Rascal) users of a language workbench. We did not record the time spent on a particular solution.
- Even if all risks above could be mitigated, our data set is to small to derive any statistically significant conclusions. Moreover, in the low end of the SLOC data set there are very few data points, and in the upper region of the data set there is high variability.

In summary, we are aware that the presented numbers are a gross simplification of reality. Nevertheless, juxtaposing the size, size per feature, and dependencies helps to

	SLOC / NME	SLOC/NME per feature	Compile-time dependencies	Runtime dependencies
Ensō	83 / –	21 / –	Ensō, NodeJS or Ruby 1.9	Ensō, NodeJS, browser with JavaScript, jQuery
Más	413 / 56	55 / 9	Más, browser with JavaScript	browser with JavaScript, jQuery
MetaEdit+	1 177 / 68	78 / 5	MetaEdit+	browser with JavaScript
MPS	1 324 / –	106 / –	MPS, JDK, Sacha Lisson's Richtext Plugins	JRE
Onion	1 876 / –	134 / –	Onion, .NET 4.5, StringTemplate	browser with JavaScript
Rascal	2 408 / –	161 / –	Rascal, Eclipse, JDK, IMP	PHP server, browser with JavaScript, jQuery and validator
Spoofax	1 420 / –	86 / –	Spoofax, Eclipse, JDK, IMP, WebDSL	WebDSL runtime, SQL database, browser with JavaScript
SugarJ	703 / –	70 / –	SugarJ, JDK, Eclipse, Spoofax	JRE
Whole	645 / 313	59 / 28	Whole Platform, Eclipse, JDK	JRE, SWT, Whole LDK
Xtext	1 040 / –	65 / –	Xtext, Eclipse, ANTLR, Xtend	JRE, JSF 2.1, JEE container

Table 4: Size metrics and dependency information on the QL/QLS solutions.

spot outliers and can enable interesting observations. Furthermore, this can guide future investigations by workbench users or implementors. In the next section, we present our findings based on the results above.

7 Observations

Completeness All solutions fulfilled the basic requirements of rendering and executing QL models. Furthermore, 9 out of 10 solutions provide IDE support for the QL language. Additionally, 7 of those solutions also provide IDE support for the optional QLS language. All of the solutions achieve these results with fewer than 2 500 SLOC; for the language workbenches based on non-textual notations, the raw SLOC count is below 1 200. For comparison, a simple QL implementation in Java, consisting of a (generated) parser, type checker and interpreter, roughly requires around 3 100 SLOC, *excluding* IDE support and QLS features[9]. This shows that state of the art language workbenches indeed provide advanced support for language engineering, and confirms earlier research providing evidence that the use of DSL tools leads to language implementations which are easier to maintain [31].

Diversity Reflecting upon Tables 1 and 4 we can observe a striking diversity among the tools, even though they perform more or less equally well in terms of the assignment. In our study, half of the workbenches are developed in an academic context (Ensō, Rascal, Spoofax, SugarJ, and the Whole Platform) and the other half in industry (Más, MetaEdit+, MPS, Onion, and Xtext). Feature coverage and SLOC per feature show no bias to either side. Similarly, the age of the language workbenches varies from 18 years (MetaEdit+) to 1 year (Onion). Yet, again there seems to be no bias towards a particular age category. It is to be expected that the maturity, stability, and scalability of industrial and academic tools differ; however, this has not been focus of our study. Indeed, scalability will likely be one of the focuses of the next LWC, from which we hopefully gain further insight into the field of language workbenches.

[9] This number is the median SLOC over 48 QL implementations of hand-written Java code together with a grammar definition using ANTLR, Rats! or JACC, constructed by master-level students of the Software Construction course at University of Amsterdam, 2013. See: `https://github.com/software-engineering-amsterdam/sea-of-ql`.

Another interesting distinction is whether a language workbench provides a single, generic metalanguage or a combination of smaller metalanguages. For instance, Rascal provides a unified language with domain-specific features (grammars, traversal, relational calculus, etc.) to facilitate the construction of languages. Similarly, apart from metamodels in Más and grammars and metamodels in Onion, these two language workbenches interface with general purpose languages for the heavy lifting (Xtend in Más, C# in Onion). Both MPS and Xtext provide escapes to Java should the need arise.

On the other hand, Spoofax provides a multiplicity of declarative languages dedicated to certain aspects of a language implementation (e.g., SDF3 for parsing and pretty printing, Stratego for transformation, NaBL for name binding, etc.). Along the same lines, MPS and SugarJ provide support for building such sub-languages on top of an open, extensible base language. In this way, SugarJ integrates SDF, Stratego and a language for type systems into the base language. MPS uses specialized languages for type system rules, transformation rules and data flow specification, among others.

Finally, considering editor model and notation style, there seems to be no predominant language-workbench style: textual, projectional and graphical notations are well represented and have been found equally able to realize the QL/QLS assignment. It is interesting to note however, that such boundaries are blurring. MPS already supports tabular, symbolic, and textual notations. Both MPS and Spoofax are currently working towards integrating graphical notations (see e.g., [55]). In the Onion language workbench, textual parsing is combined with projectional editing. Finally, Ensō apriori does not commit to one particular style and supports both textual and graphical editing. Thus there seems to be a convergence towards language workbenches where multiple, heterogeneous notations or editing modes may co-exist within one language, similar to the original vision of intentional programming [47].

Language reuse and composition An important goal of language-oriented programming [61] is the ability to combine different languages describing different aspects of software systems. The results on the QL/QLS assignment reveal first achievements in this direction. First of all, as indicated above, a number of language workbenches approach language-oriented programming at the meta level: language definitions in MPS, Spoofax, and SugarJ are combinations of different metalanguages. Second, some of the language workbenches achieve high feature coverage using relatively low SLOC numbers. Notably, the low SLOC/feature number of Ensō, MPS, Spoofax, SugarJ and Xtext can be explained by reusing existing languages or language fragments. The Ensō, MPS, SugarJ, and Xtext solutions reuse a language for expressions, thus getting aspects like syntax, type checking, compilation or evaluation for free. The Spoofax solution targets the WebDSL platform, thus reusing execution logic at runtime. In contrast, the Rascal solution includes full implementations of both syntax and semantics of expressions and the execution logic of questionnaires.

Another observation in line with language-oriented programming is the fact that all language workbenches considered in this paper are themselves compile-time dependencies for the QL/QLS IDE. This suggests that the goal of state-of-the-art language workbenches is not so much to facilitate the construction of independent compilers and IDEs, but to provide an extensible environment where those compilers and IDEs can live in. In Ensō, MetaEdit+, MPS, SugarJ, and the Whole Platform, new languages are really

extensions of or additions to the language workbench itself. MPS, Ensō and SugarJ even facilitate extension of the metalanguages. Furthermore, with the exception of Xtext, all language workbenches allow new languages or language extensions to be activated dynamically within the same instance of the IDE.

8 Concluding Remarks

To document the state of the art of language workbenches, we established a feature model that captures the design space of language workbenches. We positioned existing language workbenches in this design space by identifying the features they support. As our study reveals, all features of our feature model are realized by some language workbench, but no language workbench realizes all features. To investigate the 10 language workbenches of our study in more detail, we collected empirical data on feature coverage, size, and required dependencies of implementations of a language for questionnaires with styling (QL/QLS) in each language workbench. Based on the results, our observations can be summarized as follows:

- Language workbenches provide adequate abstractions for implementing a language like QL. The results show a marked advantage over manual implementation.
- The language workbench space is very diverse: different sets of supported features, age ranging from 1 to 18 years, single metalanguage or multiple metalanguages, industry or research, etc. Based on our results it is impossible to conclude that any particular category performs better than others.

Finally, we have observed trends towards:

- Integrating different notation styles (textual, graphical, tabular, symbolic) and editing modes (free-form and projectional).
- Reuse and composition of languages, leading to language-oriented programming both at the object level and meta level.
- Viewing language workbenches as extensible environments, instead of tools for creating other tools.

References

1. A. Alderson. Experience of bi-lateral technology transfer projects. In *Diffusion, Transfer and Implementation of Information Technology*, 1997.
2. M. Anlauff, P. W. Kutter, and A. Pierantonio. Tool support for language design and prototyping with Montages. In *CC*, pages 296–299. Springer, 1999.
3. B. R. T. Arnold, A. v. Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *Formal Methods Application in Software Engineering*, pages 6–13. IEEE, 1995.
4. D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
5. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. *SIGPLAN Not.*, 24(2):14–24, 1988.
6. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.
7. M. Chen and J. Nunamaker. Metaplex: An integrated environment for organization and information system development. In *ICIS*, pages 141–151. ACM, 1989.
8. S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.
9. V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: The MENTOR experience. Technical Report 26, INRIA, 1980.

10. S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: Implementing domain-specific languages for Java. In *GPCE*, pages 112–121, 2012.
11. S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universität Marburg, 2013.
12. S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *LDTA*, pages 7:1–7:8. ACM, 2012.
13. S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In *GPCE*, pages 167–176. ACM, 2011.
14. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA*, pages 391–406. ACM, 2011.
15. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-sensitive generalized parsing. In *SLE*, volume 7745 of *LNCS*, pages 244–263. Springer, 2012.
16. S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Haskell Symposium*, pages 149–160. ACM, 2012.
17. M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *SPLASH Companion*, pages 307–309. ACM, 2010.
18. B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *POPL*, pages 111–122. ACM, 2004.
19. M. Fowler. Language workbenches: The killer-app for domain specific languages? Available at `http://martinfowler.com/articles/languageWorkbench.html`, 2005.
20. J. Heering and P. Klint. Semantics of programming languages: a tool-oriented approach. *SIGPLAN Not.*, 35(3):39–48, 2000.
21. F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *ECMDA-FA*, pages 114–129. Springer, 2009.
22. Z. Hemel and E. Visser. Declaratively programming the mobile web with Mobl. In *OOPSLA*, pages 695–712. ACM, 2011.
23. M. Hills, P. Klint, and J. J. Vinju. Meta-language support for type-safe access to external resources. In *SLE*, volume 7745 of *LNCS*, pages 372–391. Springer, 2013.
24. Honeywell Technology Center. Dome guide, 1999.
25. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, CMU Software Engineering Institute, 1990.
26. J. Kärnä, J.-P. Tolvanen, and S. Kelly. Evaluating the use of domain-specific modeling in practice. In *DSM*, 2009.
27. L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463. ACM, 2010.
28. S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In *CAiSE*, volume 1080 of *LNCS*, pages 1–21. Springer, 1996.
29. S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.
30. P. Klint. A meta-environment for generating programming environments. *TOSEM*, 2(2):176–201, 1993.
31. P. Klint, T. van der Storm, and J. Vinju. On the impact of DSL tools on the maintainability of language implementations. In *LDTA*. ACM, 2010.
32. P. Klint, T. van der Storm, and J. Vinju. EASY meta-programming with Rascal. In *GTTSE III*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.
33. P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177. IEEE, 2009.
34. G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *SLE*, volume 7745 of *LNCS*, pages 311–331. Springer, 2012.
35. T. Kosar, P. E. M. López, P. A. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, 2008.

36. M. F. Kuiper and J. Saraiva. Lrc – a generator for incremental language-oriented tools. In *CC*, pages 298–301, 1998.
37. J. d. Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In *FASE*, pages 174–188. Springer, 2002.
38. A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Intelligent Signal Processing*, 2001.
39. D. H. Lorenz and B. Rosenan. Cedalion: A language for language oriented programming. In *OOPSLA*, pages 733–752. ACM, 2011.
40. F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In *ICFP*, 2013. to appear.
41. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
42. M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer. LISA: An interactive environment for programming language development. In *CC*, pages 1–4. Springer, 2002.
43. Meta Systems Ltd. Quickspec reference guide, 1989.
44. MetaCase. MetaEdit+ revolutionized the way Nokia develops mobile phone software. Online, 2007. `http://www.metacase.com/cases/nokia.html` (June 5th, 2013).
45. T. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.
46. T. Reps and T. Teitelbaum. The synthesizer generator. *SIGPLAN Not.*, 19(5):42–48, 1984.
47. C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOPSLA*, pages 451–464. ACM, 2006.
48. K. Smolander, K. Lyytinen, V.-P. Tahvanainen, and P. Marttiin. MetaEdit—a flexible graphical environment for methodology modelling. In *CAiSE*, pages 168–193. Springer, 1991.
49. E. Söderberg and G. Hedin. Building semantic editors using JastAdd: tool demonstration. In *LDTA*, page 11, 2011.
50. R. Solmi. *Whole platform*. PhD thesis, University of Bologna, 2005.
51. P. G. Sorenson, J.-P. Tremblay, and A. J. McAllister. The Metaview system for many specification environments. *IEEE Software*, 5(2):30–38, 1988.
52. D. Teichroew, P. Macasovic, E. Hershey III, and Y. Yamato. Application of the entity-relationship approach to information processing systems modeling, 1980.
53. J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *ICSE SEIP*, pages 671–680. ACM, 2011.
54. T. van der Storm. The Rascal Language Workbench. CWI Technical Report SEN-1111, CWI, 2011.
55. O. van Rest, G. Wachsmuth, J. Steel, J. G. Süss, and E. Visser. Robust real-time synchronization between textual and graphical editors. In *ICMT*, 2013.
56. E. Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE II*, volume 5235 of *LNCS*, pages 291–373. Springer, 2007.
57. M. Voelter and V. Pech. Language modularity with the MPS language workbench. In *ICSE*, pages 1449–1450. IEEE, 2012.
58. M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Journal of Automated Software Engineering*, 2013.
59. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an extensible C-based programming language and IDE for embedded systems. In *SPLASH Wavefront*, pages 121–140. ACM, 2012.
60. T. Vollebregt, L. C. L. Kats, and E. Visser. Declarative specification of template-based textual editors. In *LDTA*, 2012.
61. M. P. Ward. Language-oriented programming. *Software – Concepts and Tools*, 15:147–161, 1995.