Evaluating and Comparing Language Workbenches

Existing Results and Benchmarks for the Future

Sebastian Erdweg[d], Tijs van der Storm[a], Markus Völter[e], Laurence Tratt[b], Remi Bosman[f], William R. Cook[c], Albert Gerritsen[f], Angelo Hulshout[g], Steven Kelly[h], Alex Loh[c], Gabriël Konat[i], Pedro J. Molina[j], Martin Palatnik[f], Risto Pohjonen[h], Eugen Schindler[f], Klemens Schindler[f], Riccardo Solmi[m], Vlad Vergu[i], Eelco Visser[i], Kevin van der Vlist[k], Guido Wachsmuth[i], Jimi van der Woning[l]

[a]CWI, The Netherlands
[b]King's College London, UK
[c]University of Texas at Austin, US
[d]TU Darmstadt, Germany
[e]voelter.de, Stuttgart, Germany
[f]Sioux, Eindhoven, The Netherlands
[g]Delphino Consultancy
[h]MetaCase, Jyväskylä, Finland
[i]TU Delft, The Netherlands
[j]Icinetic, Sevilla, Spain
[k]Sogyo, De Bilt, The Netherlands
[l]Young Colfield, Amsterdam, The Netherlands
[m]Whole Factory, Bologna, Italy

Abstract

Language workbenches are environments for simplifying the creation and use of computer languages. The annual Language Workbench Challenge (LWC) was launched in 2011 to allow the many academic and industrial researchers in this area an opportunity to quantitatively and qualitatively compare their approaches. We first describe all four LWCs to date, before focussing on the approaches used, and results generated, during the third LWC. We give various empirical data for ten approaches from the third LWC. We present a generic feature model within which the approaches can be understood and contrasted. Finally, based on our experiences of the existing LWCs, we propose a number of benchmark problems for future LWCs.

Keywords: language workbenches, domain-specific languages, questionnaire language, survey, benchmarks

1. Introduction

Language workbenches, a term popularized by Martin Fowler in 2005 [1], are tools that lower the development costs of implementing new languages and their associated tools (IDEs, debuggers etc.). As well as easing the development of traditional stand-alone languages, language workbenches also make multi-paradigm and language-oriented programming environments (see e.g. [2, 3]) practical.

Email address: `erdweg@cs.tu-darmstadt.de` (Sebastian Erdweg)

For almost as long as programmers have built languages, they have built tools to ease the process, such as parser generators. Perhaps the earliest tool which we would now think of as a language workbench was SEM [4], which was later followed by tools such as MetaPlex [5], Metaview [6], QuickSpec [7], and MetaEdit [8], Centaur [9], the Synthesizer generator [10], the ASF+SDF Meta-Environment [11], Gem-Mex/Montages [12], LRC [13], and Lisa [14]. Most of these systems operated on textual languages and were intended to work with formal specifications of General Purpose Languages (GPLs) [15]. Nevertheless, many of them were used to build practical Domain-Specific Languages (DSLs) [16].

Informally, modern language workbenches are often referred to as being textual, graphical, or projectional. Extant textual workbenches like JastAdd [17], Rascal [18, 19], Spoofax [20], and Xtext [21] can be seen as successors of the original language workbenches, often making use of advances in IDE or editor technology. Many extant graphical workbenches such as MetaEdit+ [22], DOME [23], and GME [24] were originally developed for box and line style diagrams. Projectional workbenches are a recent addition, with JetBrains MPS [25] and the Intentional Domain Workbench [26] reviving and refining the old idea of syntax directed editors [27], opening up the possibility of mixing textual and non-textual notations.

Since language workbenches have come from industry, it is perhaps unsurprising that many real-world projects have used them. As an indicative sample (in approximate chronological order): the Eurofighter Typhoon used IPSYS's HOOD toolset [28]; Nokia's feature phones [29] and Polar's heart rate monitors[30] used MetaEdit+; WebDSL [31] and Mobl [32] were developed using Spoofax.

In short, not only are the uses for language workbenches growing, but so are the number and variety of the workbenches themselves. One disadvantage of this growing number of systems is that the terminology used and features supported by different workbenches are so disparate that both users and developers have struggled to understand common principles and design decisions. Our belief is that a systematic overview of the area is vital to heal this rift.

The Language Workbench Challenge (LWC) was thus started to promote understanding of, and knowledge exchange between, language workbenches. Each year a language engineering challenge is posed and submissions (mostly, but not exclusively, by the developers of the tools in question) implement the challenge; documentation is required, so others can understand the implementation. All contributors then meet to discuss the submitted solutions. Tackling a common challenge allows a better understanding of the similarities and differences between different workbenches, the design decisions underlying them, their capabilities, and their strengths and weaknesses.

Contributions and Structure In this paper, we describe the challenges posed by the 4 LWC editions run so far (Section 2), before explaining why we focus on the results generated by its third incarnation, LWC'13. We then make the following contributions:

- We establish a feature model that captures the design space of language workbenches as observed in the previous LWCs (Section 3).
- We present and discuss the 10 language workbenches participating in LWC'13 by classifying them according to our feature model (Section 4).
- We present empirical data on 10 implementations of the LWC'13 assignment: a questionnaire DSL (Section 5).
- Based on the experiences from the previous LWCs, we propose benchmark problems to be used in future LWCs (Sections 6.1 and 6.5) We also two examples of evaluating the benchmarks in Section 7.

This paper is an extended version of [33]. The discussion of the various editions of the LWC (at the beginning of Section 2) as well as the benchmarks (in Sections 6.1 and 6.5) are new in this version.

2. Background

The idea for the LWC came from discussions at the 2010 edition of the Code Generation conference. Since then, four LWCs have been held, each posing a different challenge. We first describe each year's challenges, before explaining why we focus in this paper on data collected from the third LWC. We start out with a note on terminology.

2.1. Terminology

In this paper we use terminology from different areas, including DSL engineering ("program", "abstract syntax"), model-driven engineering ("metamodel", "model-to-text", "model-to-model"), and language-oriented programming ("language extension"). The reason is that the various tools as well as the authors come from this variety of backgrounds. We decided to not try to unify the different terminologies into a single one because doing this well would amount to its own paper. We believe that each of the terms is clear in whatever context it is used in the paper.

2.2. The LWC challenges

LWC'11 The first challenge 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 GPL. 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 required the basic solution to be extended, but not modified.

LWC'12 The second challenge required two languages to be implemented. The first language captured piping and instrumentation models 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 behaviour of piping and instrumentation models. Developers were expected to combine the two languages to enable the simulation of piping and instrumentation systems.

LWC'13 The third challenge consisted of a DSL for questionnaires, which had to be be rendered as an interactive GUI that reacted to user input, and presented additional questions. The questionnaire definition was expected to be validated, detecting errors such as unresolved names and type errors. In addition to basic editor support, participants were expected to modularly develop a styling DSL that could be used to configure the rendering of a questionnaire. We describe the details of this challenge in Section 5.

LWC'14 The fourth challenge was based on the third, but emphasised nonfunctional aspects, particularly teamwork (several developers editing a model concurrently) and scalability (in terms of model sizes).

2.3. Focusing on LWC'13

Designing language workbench challenges is tricky. The challenge must be small enough to encourage wide participation, modular so that participants are not forced to address problems that their tools were never designed for, and sufficiently specified to make comparison possible. Our subjective overview of the various challenges is as follows:

LWC'11 had a simplistic challenge and few participants. The results are not detailed or representative enough to warrant deep scrutiny.

LWC'12 had a large, but loosely specified, challenge. The resulting implementations were too diverse to allow sensible comparison, and thus no meaningful results were generated.

LWC'13 had a clearly defined task that spanned many aspects of DSL development while permitting compact implementations. This facilitated detailed comparison and meaningful results for the 10 participating workbenches.

LWC'14 had a focus on nonfunctional properties. It was hard to meaningfully quantify and compare the results.

As this suggests, in our opinion LWC'13 generated the most useful results. We therefore use that as a basis for the rest of this paper.

2.4. Survey methodology

This paper's first contribution is to document the state of the art of language workbenches in a structured and informative way. We do that by using the results of the LWC'13 challenge. This section sets out the methodology used. In essence, we carried out a post-LWC survey, leading to a qualitative domain analysis of language workbenches, and a quantitative analysis of empirical data about the LWC solutions.

Qualitative domain analysis. The first part of our methodology provides a framework for discussion about LWBs. We asked all LWC'13 participants to provide a detailed list of features supported by their language workbench. The first three authors used this data to produce an initial feature model [34], which was then presented to all participants for feedback. The refined feature model presented in Section 3 provides the first mechanism to categorize and discuss language workbenches according to which features they support.

Empirical data. We constructed a feature model of the questionnaire DSL's features and asked the participants to indicate which features they had tackled in their solution. We present a description of the assignment and the feature model in Section 5. Using each solution's source code, we were able to answer the following questions:

- *What is the size of the solution?* This question aimed to gain a simple understanding of the complexity of the solution each tool required. The suggested metric for the answer was Source Lines of Code SLOC, a count which excludes comments and empty lines. Non-textual solutions were unable to use SLOC, and we consider this issue in more detail in Section 5.2.

- *What are the compile-time dependencies?* This question aimed to understand the difficultly a user might face in installing a language workbench and generating a questionnaire DSL (e.g. the various libraries, frameworks, and platforms that are needed to run the compiler and IDE).

- *What are the run-time dependencies?* This question aimed to understand the difficulty an end-user might face in running a generated questionnaire GUI.

The data for these questions is presented in Section 5.2. We then discuss the various workbenches in light of the data in Section 5.3.

2.5. Survey Generality

Although 10 language workbenches entered LWC'13, not every extant language workbench was present. As Section 4 clearly shows, the language workbenches in LWC'13 support a diverse range of features, which gives us confidence that our feature model is reasonably complete and representative of a wider class of language workbenches. Indeed, to the best of our knowledge, the feature model adequately covers the language workbenches that were not present in LWC'13.

3. A Feature Model for Language Workbenches

We currently lack both common terminology when discussing language workbenches and an understanding of what features different workbenches support. To this end, we took data generated by LWC'13 and derived a feature model from it, capturing the major features of language workbenches. Our initial feature model was passed for feedback to the LWC participants, altered as appropriate and the final version shown here.

The feature model is shown in Figure 1. We use standard feature-diagram notation and interpretation [35]. For those unfamiliar with this notation, a brief summary is as follows. To describe which features a language workbench has, one starts at the top node (*Language workbench*) and selects it (i.e. says the tool is indeed a language workbench). One then walks the tree from top to bottom, selecting nodes as appropriate. Edges which end in a filled circle are mandatory, and if the parent node is selected, so too must the child node. Edges which end in a hollow circle are optional, and the child node can be selected if appropriate, but this is not required. Edges connected with a filled wedge are 'or' nodes, and if the parent node is selected, at least one of the child nodes must be selected. At the end of this activity, the selected nodes tell you which features the language workbench implements. Note that while the feature model captures the design space of language workbenches, it does not tell one *how* a specific language workbench might support a given feature.

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.

Notation. 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.

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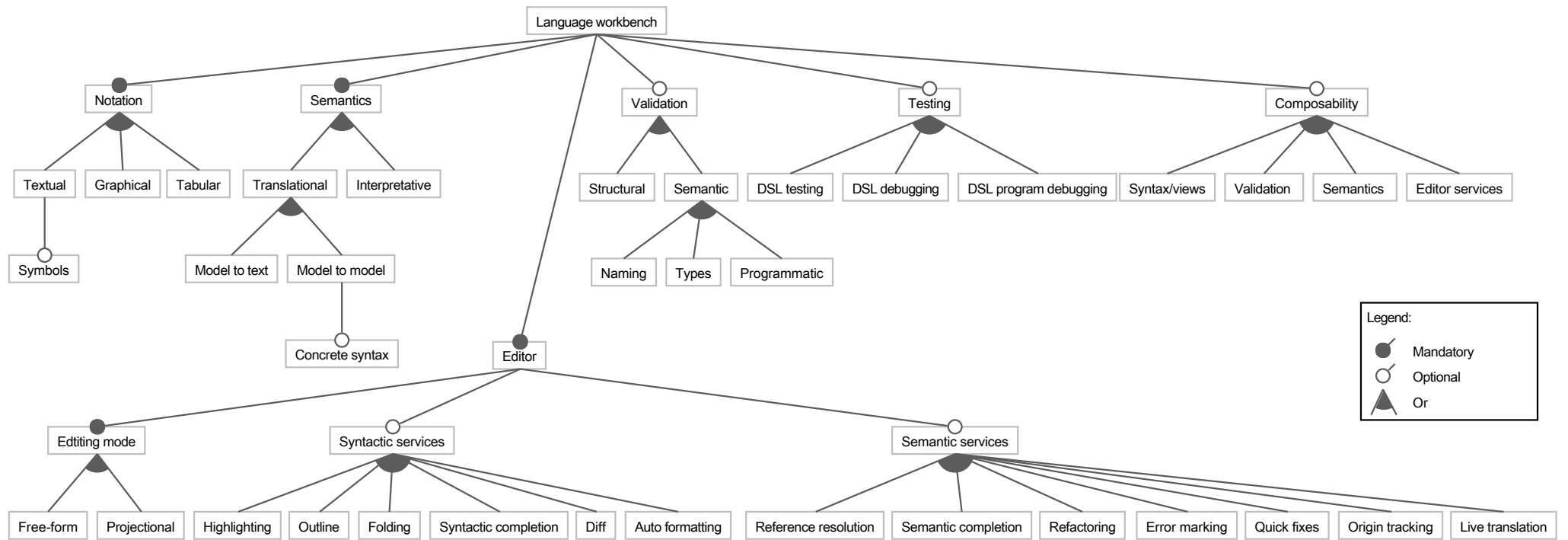
Figure 1: Feature model for language workbenches. With few exceptions, all features in the feature model apply to the languages that can be defined with a language workbench, and not to the definition mechanism of the language workbench itself.

trees or graphs. For model-to-model translations, we distinguish those which are able to use a language's concrete syntax in specifying translations from those that cannot.

Editor support. Editing is a central pillar of language workbenches [1] 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 as text), and projectional editing, where the user edits a projection of the persisted model in a fixed layout (text-like or otherwise). 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 optionally 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.
- *Auto formatting*, restructuring, or aligning of a model's presentation.

Semantic editor services include:

- *Reference resolution* to link different concepts, such as declarations and usages of variables, together (e.g. for type checking).
- 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.
- *Error markers* allow errors in the model to be clearly highlighted to the user, pinpointing a specific model element(s) and displaying an appropriate error message.
- *Quick fixes* may propose ways of fixing such an error. When the user selects a proposed fix, 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 behaviour 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.

Validation. In addition to the above, most language workbenches can display information about the result of language-specific validation. We distinguish validations that are merely structural, such as containment or multiplicity requirements between different concepts, and validations that are more semantic in nature, 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 mechanism and instead allow validations to be implemented in a normal GPL.

Testing. Testing a language definition may be supported by unit-testing different language aspects: the syntax (parser or projections), semantics (translation or interpretation), editor (completion, reference resolution, refactoring, etc.), or validation (structure or types). Some language workbenches also 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.

Composability. Finally, composability of language definitions is a key requirement for supporting language-oriented programming [2, 3] where software developers use multiple languages to address different aspects of a software system. Language workbenches may support *incremental extension* (modularly adding additional language constructs to a base language) and *language unification* (independently developer languages can be unified into a single language) [36]. Ideally, all aspects of a language – syntax, validation, semantics, and editor services – should be composable.

4. The Language Workbenches of LWC 2013

In this section, we briefly introduce the LWC'13 language workbenches, before describing their fit to our feature model.

4.1. 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 pitches itself at "non-developers", using projectional editing to provide an appropriate style of editing. Language semantics are 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 [22]. MetaEdit+ aims to remove accidental complexity, allowing users to concentrate on creating productive languages and good models. MetaEdit+ is commercially successful, used by customers in both industry and academia.

MPS (since 2003, http://www.jetbrains.com/mps/) is an open-source language workbench. Its most distinctive feature is a projectional (roughly speaking, a modernised syntax directed) editor that supports integrated textual, symbolic, tabular and graphical notations [37], as well as wide-ranging support for composition and extension of languages and editors. In contrast to legacy projectional editors, its editor usability is acceptable to users [38].

Onion (since 2012) is a language workbench and base infrastructure implemented in .NET for assisting in the creation of DSLs. Onion evolved from Essential (2008), a

textual language workbench with a focus on model interpretation and code generation. The main goal 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 [18, 19, 39, 40]. Rascal combines and unifies features found in several 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. The tool is accompanied with interactive online documentation and is regularly released as a self-contained Eclipse plugin. Rascal is currently used as a research vehicle for analyzing existing software and the implementation of DSLs.

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 [20]. In Spoofax, languages are specified in declarative meta-DSLs for syntax (SDF3 [41]), name binding (NaBL [42]), editor services, and transformations (Stratego [43]). 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.

SugarJ (since 2010, http://www.sugarj.org/) is a Java-based extensible programming language supports extension of the base language with custom language features [44, 45]. Extensions are defined with declarative meta-DSLs (SDF, Stratego, and a type-system DSL [46]) 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 [47] that can be customized via library import on a file-by-file basis. An extension can use arbitrary context-free and layout-sensitive syntax [48] that does not have to align with the syntax or semantics of the base language (Java).

Whole Platform (since 2005, http://whole.sourceforge.net/) is a mature projectional language workbench [49]. 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.

Xtext (since 2006, http://www.eclipse.org/Xtext/) is a mature open-source framework for developing 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 [50]. Its architecture allows developers to start by reusing well-established and commonly understood default semantics for many language aspects.

These descriptions reveal a striking diversity. 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). Similarly, the age of the language workbenches varies from 18 years (MetaEdit+) to 1 year (Onion). 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.

4.2. Language Workbench Features

Table 1 shows the LWC'13 language workbenches relative to our feature model. In the remainder of this subsection, we reflect on various interesting findings that Table 1

9

Category	Feature	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)

brings out.

Notation and editing mode. Most language workbenches provide support for textual

notations[1], with the exception of MetaEdit+. 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. In addition to textual projections, MPS also supports graphical notations, and notations can be arbitrarily mixed [37]. Ensō combines textual and graphical notations by providing support for custom projections into diagram editors. The remaining 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.

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 colouring, 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 projection rules, so this feature is implicit. Textual free-form language workbenches can use the traditional `diff` program (e.g. for version control). 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 modellers can inspect recent changes; for version-control a shared repository is used.

Semantics. With the exception of Ensō, all language workbenches support 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. Just over half of the workbenches provide a programmatic interface for implementing validation routines. Some language workbenches lack dedicated support for type checking and/or constraints. These concerns are either dealt with by manually programming a type-checker or by assuming that models are correct by construction (e.g. are instances of a specific meta-model). MPS, SugarJ [46], and Xtext provide declarative languages for the definition of type systems. Spoofax has a declarative language for describing name binding rules [42].

Testing. MPS, Spoofax, and Xtext have dedicated sub-languages 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 support some form of debugging of the language specification. 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 meta-language 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,

---

[1]All tools allow users to type raw text. However, by "support" we mean IDE support that includes syntax color, code completion or error markup.

```
form taxOfficeExample {
    "Did you sell a house in 2010?"
        boolean hasSoldHouse
    "Did you buy a house in 2010?"
        boolean hasBoughtHouse
    "Did you enter a loan?"
        boolean hasMaintLoan
    if (hasSoldHouse) {
        "What was the selling price?"
            money sellingPrice
        "Private debts for the sold house:"
            money privateDebt
        "Value residue:"
            money valueResidue =
                (sellingPrice - privateDebt)
    }
}
```

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

reusable building blocks. The composition of textual languages involves different trade-offs (see [51] for an overview) and not all workbenches follow the same approach. Ensō, Rascal, Spoofax, and SugarJ choose to support arbitrary syntax composition which requires the use of possibly ambiguous generalized parsing technology. In contrast, Xtext, which uses ANTLR's LL(*) algorithm [52] does not introduce ambiguity, but cannot compose arbitrary grammars. Onion uses PEGs [53] which allow arbitrary ambiguity-free composition, but without any guarantees about whether the composed language can accept inputs that the components could. MPS and MetaEdit+, which do not use parsing at all, allow arbitrary notations to be combined, but lose the 'feel' of traditional text editing.

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; whereas the composition of transformations relies on the pair-wise specification of relative priorities between transformation rules.

5. LWC'13: Setup and Results

In this section, we first introduce the LWC'13 challenge before detailing which solutions implemented which part of the challenge.

5.1. The challenge

LWC'13 asked participants to create a Questionnaire Language (QL)[2]. The questionnaire language was selected based on the expectation that it could be completed with limited effort, that it provides enough discriminatory power to identify interesting features, and that it would not be biased towards any one style of language workbench. Anecdotally, we have not received any feedback indicating that the assignment was unfeasible or unsuitable.

---

[2]The original assignment can be found at
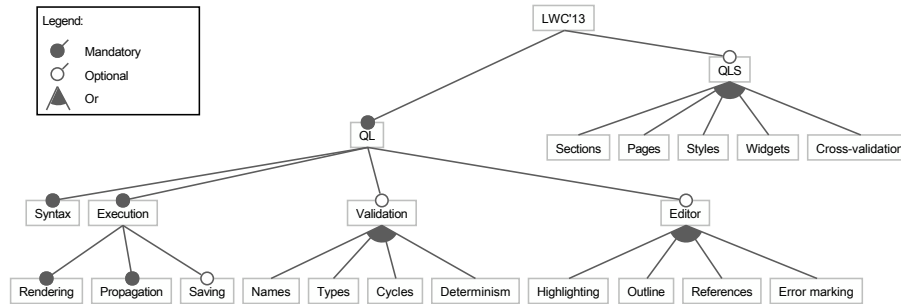http://www.languageworkbenches.net/wp-content/uploads/2013/11/Ql.pdf

Figure 3: Feature model of the QL assignment.

A summary of the challenge is as follows. A questionnaire consists of a sequence of questions and derived values presented to a user by rendering it as a GUI (an example is shown in in Figure 2). Later questions may only be visible depending on answers given to earlier questions. Users must be able to specify questionnaires in a tool and generate GUIs from them. Participants were then given a list of features which they could choose to implement, with the explicit expectation that a QL language and IDE implementation would be generated. The complete list of features is as follows:

- 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.
- Cycles: detect cyclic dependencies through conditions and expressions.
- Determinism: check that no two versions of equally-named questions are visible simultaneously (because of conditional questions, this requires requires SMT solving or equivalent).
- 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.

Participants were also asked to develop an optional second language called QLS for declaring the style and layout of QL questionnaires. QLS styles were asked to be applicable to a QL specification without the latter having knowledge of the former. In more detail, QLS was expected to have 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.

A feature model for the challenge is shown in Figure 3. Together, QL and QLS define 17 features, of which 3 are mandatory and the rest optional.

		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 (%)		24	44	88	74	82	88	97	59	65	94

Table 2: Implemented QL and QLS features per language workbench (● = fully implemented, ◐ = partially implemented/limited support).

5.2. Results

Table 2 shows which features each solution supports. The completeness of a solution is a simple ratio of the supported features (where ● = 1, ◐ = 0.5), over the total number of features (17). The feature-based comparison helps in the interpretation of the quantitative – but sometimes not easily comparable – metrics shown in Table 3. Table 4 provides download links for the solutions so that readers can verify our interpretation

Table 3 summarizes the quantitative facts about solutions. Where possible we use SLOC, using cloc[3]. Where non-textual notations are used for some, or all, of a solution we report the Number of Model Elements (NME) used. We define this as any kind of structural entity that is used to define aspects of a language (e.g. in MetaEdit+ this includes graphs, objects, relationships, roles, and properties).

Although MPS is purely projectional, the languages used for language definition are rendered textually. We therefore use an approximate SLOC count (based roughly on [54]): for each language construct, we define an SLOC factor based on the construct's textual representation. Each model instance is then multiplied by the corresponding SLOC factor to obtain a total SLOC. In addition we report the SLOC/NME per feature

---

[3]http://cloc.sourceforge.net

	SLOC / NME	SLOC/NME per feature	Compile-time dependencies	Run-time 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, Sascha 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	1 170 / −	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 3: Size metrics and dependency information on the QL/QLS solutions.

LWB	Links to the corresponding QL solutions
Ensō	https://github.com/enso-lang/enso/tree/master/demos/Questionaire
Más	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/sugar-lang/case-studies/tree/master/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 4: Download locations for each of the LWC'13 solutions.

ratio (though since not all features are of equal size, we caution against over-interpreting this number). Finally, we show the compile-time and runtime dependencies of each solution, as an indication of deployment complexity.

While the measures in Table 3 are useful in giving a sense of the different solutions and the workbenches that led to them, we caution against over-interpretation. In particular, the numbers we present are unsuited to ranking workbenches, for several reasons. First, even were one to assume that SLOC is a perfect measure for textual systems, there is no meaningful way of comparing it to NME. Second, the architecture and design is often substantially different across QL/QLS solutions. Client-server web architectures may impose substantially different costs on solutions than a desktop GUI design, for example. Third, the SLOC/feature metric ignores issues such as increased interaction between features as their number increases; put another way, the more features one has, the more code tends to be dedicated to managing their interactions. Fourth, those implementing the solutions varied in experience. Some of the solutions were developed by the language workbench developers themselves (e.g. Más and SugarJ), whereas others are built by first-time (e.g. MPS) or second-time

(e.g. Rascal) users of a language workbench.[4] Finally, our data set is too small to derive any statistically significant conclusions. Moreover, in the low end of the SLOC data set there are few data points, and in the upper region of the data set there is high variability.

5.3. Observations

Bearing in mind the various warnings we have made about the numbers presented in the previous section, they are still sufficiently useful to allow us to make a number of interesting observations.

Completeness. All solutions fulfilled the basic requirements of rendering and executing QL models. Furthermore, 9 out of 10 solutions provided IDE support for the QL language. Additionally, 7 solutions also provided IDE support for the optional QLS language. All text-oriented language workbenches achieve these results with fewer than 2,500 SLOC; those language workbenches based on non-textual notations achieve this with fewer than 1200 NMEs.

An interesting question then arises about the size of the solutions relative to traditional approaches. For comparison, a reasonable Java solution from our students came to about 2,200 SLOC, *excluding* IDE support and QLS features.[5] This can be seen as confirmation that, by lowering language creation costs, language workbenches can make software developers' lives easier (see [55]).

Single or multiple meta-languages. Another interesting distinction is whether a language workbench provides a single 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). Except for the grammar, Xtext relies on Java or Xtend for the language implementation.

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, there seems to be a convergence towards language workbenches where multiple, heterogeneous notations or editing modes co-exist within one language: MPS already supports graphical, tabular, symbolic, and textual notations; Spoofax is currently working towards integrating graphical notations (see e.g., [56]); and in the Onion language workbench, textual parsing is combined with projectional editing.

Language reuse and composition. A long-standing goal of language-oriented programming is the ability to compose different languages. 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

---

[4]We did not record the time spent to create any solution, partly for this reason.

[5]The median solution was 3,100 SLOC, which is the median 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.

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 an existing expression language (which, in cases such as Xtext, can be fairly large), thus obtaining 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.

Extensible Workbenches. 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.

6. Benchmark Problems for the Future

Although the LWC has provided the basis for much of this paper, we believe that future LWCs can do an even better job. This section proposes a set of specific, detailed benchmarks for some aspects of language workbenches. We first motivate the need for better language workbench comparisons in Section 6.1. Then we discuss desirable characteristics of benchmark problems in Section 6.2. The criteria for evaluating the proposed benchmarks are given in Section 6.3; we show two example evaluations in Sections 7.1 and 7.2. The actual benchmark problems are contained in Section 6.5, and the example languages they rely on are introduced in Section 6.4.

6.1. Detailed Workbench Comparison Requires Benchmarks

The feature model in Figure 1 is, inevitably, relatively coarse-grained. For example, the editing mode provides *free-form* and *projectional* definitions, but does not go into detail on either, nor does it provide an explicit way to define hybrid approaches. In large part this simply shown that the LWC challenges were unable to drill into enough detail to bring out more fine-grained differences.

This is not to suggest that the LWC is not the right general idea, however. It is easy to claim that a particular approach or technology can do this or that, without ever precisely defining what 'this' or 'that' are. To put such claims on a more sound footing – and ultimately to enable repeatable, reproducible experiments – they need to be well-defined and standardized. This activity has a long tradition in computing research in the form of benchmarks. For instance, benchmarks exist for evaluating the performance of theorem provers [57] and XML data management [58], the RDF query language SPARQL [59], or the integration between Java and native code through JNI [60]. Not all benchmark problems address performance concerns, however. The

so-called 'expression problem'[6] is a well-known benchmark addressing modularity and extension in programming languages. Competitions like POPLmark[7], the Language Workbench Challenge [33], Transformation Tool Contest [61], or the MSR Data Mining Challenge[8] propose challenge problems for specific communities to take on, and address a variety of aspects, unrelated to performance. The benchmark problems presented in this paper fit in that latter category.

6.2. Desirable Benchmark Characteristics

We believe that good benchmark problems have certain characteristics:

Self-contained benchmark problems highlight a specific tool capability as independently from other capabilities as possible. Instead of requiring, for instance, the implementation of particular language as a whole, one can try to isolate specific features of editor technology.

Implementable benchmark problems are described sufficiently clearly to make solutions practical. Each problem is illustrated using concrete, exemplary language features.

Feasible benchmark problems are those whose solutions are small enough to permit implementation in reasonable time.

Indicative benchmark problems are those that can reasonably lead to solutions which can be clearly evaluated against the benchmark. Put another way, indicative benchmark problems give the ability to clearly analyse whether a solution implements the benchmark problem or not.

State of the art benchmark problems are those that have particular relevance for differentiating language workbenches. Put another way, such problems should avoid commonplace features of editor technology such as syntax highlighting, completion, and error marking.

Some benchmark problems, particularly those that address performance, are quantitative in nature. However, most – including those introduced in this paper – are partly or wholly qualitative: their results are captured in a structured description. However, by using a structured description, comparisons are still possible and relatively easy.

The authors of this paper all have a stake in language workbenches. The benchmark problems were written by authors who represent five separate workbenches; most benchmark problems were thus, explicitly or implicitly, written with a particular language workbench's strengths or weaknesses in mind mostly Rascal or MPS, but also Spoofax, SugarJ, or Eco [62]. Although there is inevitably some bias towards these tools, the fact that they represent five very different points in the design space reduces this bias to what we consider an acceptable level.

6.3. Evaluation Criteria

Evaluating benchmark solutions requires a clear definition of the criteria to be used. In this subsection we thus define the minimal set of criteria needed to make the benchmark problems we propose comparable. Section 7 contains two example evaluations based on these criteria.

---

[6]http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt
[7]http://www.seas.upenn.edu/ plclub/poplmark/
[8]http://2014.msrconf.org/challenge.php

```
statemachine TrafficLights {        state greenForCars {            state greenForPeds {
  in event pedestrianButton           entry {                         entry {
  in event timeout                      send carLightsGreen             send carLightsRed
  out event carLightsRed                send pedLightsRed               send pedLightsGreen
  out event carLightsGreen            }                               }
  out event pedLightsRed              on pedestrianButton             on timeout
  out event pedLightsGreen                -> greenForPeds                 -> greenForCars
}                                   }                               }
```

Figure 4: An example state machine.

Assumptions Are there any assumptions or prerequisites relevant to the implementation of the solution?

Implementation What are the important building blocks for defining the solution? What does it take to implement the solution to the problem?

Variants Are there any interesting and insightful variants of the implementation? What small change(s) to the challenge would make a big difference in the implementation strategy or effort?

Usability What is the resulting user experience? Is it convenient to use? Is it similar to other kinds of notations? Does it feel 'foreign' to experienced users of the particular editor?

Impact Which artefacts have to be changed to make the solution work? Are changes required to (conceptually) unrelated artefacts? How modular is the solution?

Composability To what degree does the solution support composition with solutions to other benchmark problems, or with other instances of the same problem?

Limitations What are the limitations of this implementation?

Uses and Examples Are there examples of this problem in real-world systems? Where can the reader learn more?

Effort (optional). How much effort must be spent to build the solution, assuming an experienced user of the technology? We are realistic that precise effort levels are almost impossible to record and we therefore consider this criteria optional.

Other Comments Anything that does not fit within the other categories.

6.4. Example Languages used in Benchmark Problems

We use three 'standard' languages as the basis for the implementations of the benchmark problems in the *assignment*; we describe them below:

A traditional programming language. This represents a 'standard' mainstream programming language that uses expressions, variable declarations, statements, and functions (e.g. C, Java, Python). A simplified subset of one of these languages is sufficient to address the benchmarks. For simplicity, we refer to this language as *MiniJava* (a homage to [63]).

A state-machine DSL. A simple state-machine language has the advantage of being manageable whilst having enough concepts to be interesting from the perspective of language workbenches. For example, Figure 4 shows an example state machine; the details of the syntax are unimportant in our context. The language should include guard conditions (boolean expressions) and state entry actions (statements), for example, similar to Fowler's state-machine DSL [64]. We refer to this language as the *state-machine language*.

An end-user query language. We define this as a simple SQL-like language along the lines of FROM <table> SELECT <fields> WHERE <condition>. In our context, such

simple queries are enough; `<table>`, `<fields>` and `<condition>` can just be identifiers for the purpose of the benchmarks. We refer to this as the *query language*.

In addition, we reference several additional languages as a means to illustrate real-world relevance of the benchmark problems in the *example* and *relevance* sections of each problem. Since they only serve illustrative purposes, we do not introduce them here.

6.5. Benchmark Problems for Language Workbenches

In this subsection we introduce benchmark problems for language workbenches. The problems have been derived from the investigation of real-world DSLs, as illustrated by the examples in each benchmark. We group the problems into three broad categories: notation, evolution and reuse, and editing. There is inevitably some overlap between the categories and we caution readers not to interpret them too literally.

For each benchmark problem, we give a name and a description. Since precisely specifying benchmarks down to the finest detail would be futile (in terms of our effort) and limiting (ruling out the possibility of creative solutions), the challenges should be read as indicative of the class of problems specified by their name: we consider solutions to fully satisfy the challenge if they are similar in spirit to the one described in the benchmark problem. In addition to the description of the challenge, we discuss the *relevance* to real-world systems, show a concrete *example* and an explicit *assignment* to be clear about the expected form of solutions. Where it makes sense to do so, we have also added illustrations of what a solution may look like; again these should be interpreted as aides to understanding rather than as prescriptive definitions.

6.5.1. Notation

The following benchmark problems address the *Notation* feature of Figure 1, as well as its subfeatures.

Mathematical Symbols. MiniJava should be extended such that users can use conventional mathematical notation such as sum symbols, fractions, and square root symbols. Normal expressions from MiniJava must be allowed inside the mathematical notations (e.g. in the sum index, the numerator or denominator of fractions, and under the square root).

Relevance: Mathematical calculations are an important ingredient to many systems; the ability to use mathematical symbols enhances readability.

Example: The following code shows C extended with mathematical notations, implemented in JetBrains MPS [65].

```
int32 averageIntArray(int32[] arr, int32 size) {
                size
                 ∑   arr[i]
               i = 0
    return   ─────────────── ;
                 size
}
```

Assignment: Add fraction bars and summation symbols to MiniJava.

Tabular Notation. Represent state machines in form a transition table[9] instead of the textual representation shown in Figure 4. States could correspond to rows; events to

---

[9]`http://en.wikipedia.org/wiki/State_transition_table`

columns; and transitions to the appropriate row/column cell. The table should be fully editable.

Relevance: Many aspects of programs, including decision tables, state machines, lookup tables and data collections are naturally represented as tables.

Example: A (non-state machine) example of a tabular notation from a health system is shown below.

Assignment: Represent state machines as tables.

Metadata Annotations. Users should be able to attach annotations to arbitrary program elements without affecting the program's core meaning. A full solution should allow the annotation itself to have structure enforced by the workbench.

Relevance: Metadata such as the ones mentioned above are are useful for documenting software (revisions, author) and for declaratively specifying the integration with other systems (permissions, transactions).

Example: It should be possible to require a `@revision` annotation to adhere to the versioning scheme *major.minor*, or the `@author` annotation should support picking an author from a predefined list of names.

Assignment: Define an `@author` annotation that can be attached to any program element in MiniJava.

Optional Hiding. Aspects or parts of programs should be hidden optionally. Hidden aspects must be stored and users must be given the option to redisplay them later; the non-hidden parts must remain fully editable. A full solution should allow users to customize which parts of a program they can hide.

Relevance: This is an example of the widely-used concept of views, where different subsets of a program or model are made available to different stakeholders.

Example: Consider the textual notation for state machines shown in Figure 4. Users should be able to hide parts of the state machine so that they can concentrate on the big picture (e.g. hiding the `entry` clauses).

Assignment: Implement a way for users to optionally and temporarily hide the `entry` clauses of states.

Multiple Notations. This one addresses editing the same structure using different notations, based on a user's preference. A full solution will allow the free switching between multiple notations. As an example, mbeddr's state machines can be edited as text, diagram or table.

Relevance: Switching between different notations helps integrating different stakeholders or supports different use cases by focussing on different aspects of a data structure.

Example: A state machine can be represented as text (cf. Figure 4) or as a table. Both cases support editing of the same underlying state machine structure.

Assignment: Support two notations for state machines. For instance, the textual notation of Figure 4 and a tabular notation.

Computed Properties. Consider a method in MiniJava that has no side effects. The user should be able to have the editor automatically mark the method with a `pure` annotation which is visible as a keyword before the method name when this property is true. When the property ceases to be true, the annotation should be removed automatically. The annotation should be read-only. This challenge can be seen as embodying a general concept of enriching the visual presentation of an element to convey information computed from the element itself; it is similar to syntax highlighting or error highlighting, but is more general than simple styling.

Relevance: Showing computed, read-only values in the editor helps users understand the program; in contrast to hovers, they are always visible and do not require a mouse-over. This allows users to glance at the whole set of computed values.

Example: An insurance DSL uses this approach to show the inferred types of complex expressions directly in the program code.

Assignment: Show the `pure` annotation on MiniJava methods based on some kind of analysis (e.g., whether it has side effects or not).

Computed Structures. This challenge can be considered as extending *Computed Properties*. In the context of MiniJava, consider a program with two classes `A` and `B`. Class `A` defines a method `m1` and class `B` defines method `m2`. Moreover, class `B` inherits from `A`. Even though class `B` inherits method `m1`, the source code of `m1` is only visible within class `A`. The goal of this challenge is to enable viewing inherited methods within the inheriting class, that is, view all of `A`'s methods in `B` while editing `B`. Editing the `m1` method viewed in `B` should edit the underlying `m1` method in `A`. Note that, even though the method `m1` is shown inline in class `B`, it still is part of class `A` and still has access to `A`'s private fields, which is not the case for `m2`. We leave it open to solutions how to communicate such private fields to the user.

Relevance: The more sophisticated decomposition mechanisms a language provides (such as specialization or aspects), the more important it becomes to show the resulting, composed structure of a program.

Example: The same insurance DSL mentioned above uses this approach show flatten the inheritance hierarchies of insurance contracts for some stakeholders.

Assignment: Allow inherited methods to be viewed and edited inline in the inheriting class in MiniJava.

Skeleton Editing. Predefined templates (or 'skeletons') can be useful to guide data entry for inexperienced users (i.e., those who are unfamiliar with a new language). For example, the query language may allow queries of the form `FROM` *table* `SELECT` *field_1, ..., field_n* `WHERE` *expr*, where `FROM`, `SELECT` and `WHERE` represent the skeleton. Full skeleton editing will automatically expand the skeleton with holes and automatically guide the user from one hole to another. The challenge is to provide skeletons whose core contents (e.g., keywords such as `FROM`) not only act as scaffolding,

```
Rule Set Type DemoRuleSetType          Rule Set Type DemoRuleSetType


Business objects                       Business objects
person : Person                        <no business objects>
policy Policy :

Variables:          Parent            Variables:          Parent
PRMI    : int       <no parent>       <no variables>      <no parent>
FR      : int
NN      : int
TT      : int       Libraries                             Libraries
J       : int       Standard                              <no libraries>
A3      : int       Extra
G3      : int
ANUI    : int
X       : int
```

Figure 5: The left half of this figure shows a *rule set type*, essentially a data type definition used in an insurance system. The right side shows a new, "empty" rule set type. Notice how the sections where users can enter variables, libraries or business objects are already predefined as a skeleton. Users cannot delete them; they can only "fill in the blanks" marked by the `<...>` placeholders.

but also cannot be deleted.

Relevance: Skeletons guide users during editing. In particular, our experience tells us that business users (i.e., non-programmers) appreciate such skeletons.

Example: Figure 5 shows an example from a business application.

Assignment: Allow the query language to be edited using skeletons.

Embedding Code in Prose. Support arbitrary embedded program nodes within free-flowing text so that the embedded nodes are treated as ordinary program code. Refactorings such as renames should be reflected in such nodes, and editor services such as syntax highlighting, code completion, reference hyperlinking, and error marking should be supported.

Relevance: Languages often feature elements containing free-form prose, such as comments or documentation text. Inside the text, however, program elements must be embedded or referenced.

Example: Javadoc-style multi-line comments mostly consist of natural language text. Inside the text users can refer to method parameters of the documented method. Another example is shown in the figure below: it shows references to program elements inside comments of a DSL for questionnaires, implemented in Rascal [18].

```
 1
 2  form addition {
 3
 4      "Enter number 1:" integer x
 5      "Enter number 2:" integer y
 6
 7      /*
 8       * The result is computed based on <x> and <y>
 9       * using summation, which is not <x && y>
10       */
11
12      "The sum is: " integer sum = x + y
13
14  }
```

Assignment: Allow methods to be referenced in MiniJava comments.

Embedded Black Boxes. Though most of our challenges have explicitly considered only textual editing, embedding non-textual items can be useful. At the simplest level, one may embed images into text as shown below in example implemented in Racket [66].


```
(define under-construction        )

(define (home-page)
  (html (head (title "Under construction"))
        (body (img under-construction)))))
```

At the other end of the spectrum one finds the encapsulated embedding of entire editors (e.g. spreadsheets or graphical editors)

Relevance: Embeddeding "foreign objects" into applications has always played an important role in end-user applications (such as Microsoft OLE[10] or Apple's OpenDOC[11]. Essentially, this benchmark addresses the same goal for languages and IDEs.

Example: mbeddr embeds full graphical editors for state machines in otherwise textual C programs in order to cater for different stateholders.

Assignment: Allow the use of bitmap images as expression literals in MiniJava.

6.5.2. Evolution and Reuse

The following benchmark problems address modular extension of languages (the *Composability* feature and its subfeatures in of Figure 1) as well as their evolution over time (i.e. moving from one version of a DSL to another version of the same language while keeping the programs written with the language valid).

Language Extension. One form of language composition involves adding additional constructs to existing syntactic categories. For instance, one could add support for an `unless` statement to MiniJava. The structure is similar to an `if` statement and the semantics correspond to `if (!<cond>)`. In contrast to the embedding (discussed below), the `unless` extension is defined *specifically* for use with MiniJava, that is, the extension may depend on the extended language. How can such extensions be supported in a modular fashion [67], that is, without invasively changing MiniJava?

Relevance: Extension supports incrementally growing a language into a particular domain [47] by adding language constructs relevant to that domain.

Example: mbeddr uses various language extensions of C to make it more useful for embedded software development. Example extensions include state machines, interfaces and components or physical units.

Assignment: Add an `unless`-statement to MiniJava in a modular way.

Language Embedding. Consider an existing implementation of MiniJava and an existing implementation of the query language. There are no dependencies between these languages and each can be used without the other. How can the query language be embedded [67] into MiniJava, so that it can be used as an expression such as `String name = SELECT name FROM customers WHERE ...`? An important challenge here is to achieve the embedding with minimal "glue" code to tie the languages together, and without invasively changing either of the languages.
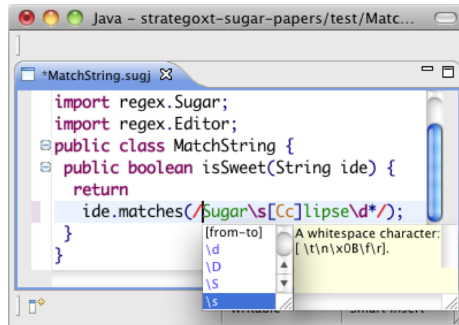
---

[10] http://en.wikipedia.org/wiki/Object_Linking_and_Embedding
[11] http://en.wikipedia.org/wiki/OpenDoc

Relevance: Many languages support embedded query languages. Examples include SQL/J or Microsoft Linq. Such embeddings should be modular and come with the expected IDE support.

Example: The figure shows an embedding of regular-expression syntax into Java using SugarJ [47].

Assignment: Embed the query language into MiniJava.

Extension Composition. Let us assume multiple MiniJava extensions have been built independently, such as the `unless` extension mentioned above and a new extension that provides complex number for MiniJava. What has to be done to support extension composition [36], that is, to use these independent extensions in the same MiniJava program?

Relevance: Only if extension composition is supported does it become possible to combine independently developed extensions in a single program.

Example: The C extensions mentioned for mbeddr can be combined in the same program without any glue code.

Assignment: Define another modular MiniJava extension similar to the `unless` statement. Show how to use this one and `unless` in the same program.

Beyond-Grammar Restrictions. Language grammars partially restrict the programs that can be expressed. However, in some cases, additional constraints may be necessary to statically reject semantically undesirable programs. Consider a state machine language extension to MiniJava. The entry actions in states should be able to contain Mini-Java statements. However, there are specific statements such as `triggerself <event>` with which the state machine can inject events into itself. Syntactically, `triggerself` is a statement like any other MiniJava statement to make sure it can be used in actions. However, it does not make sense to use it outside a state's entry block and the editor should therefore highlight such uses as erroneous. How can we restrict the context in which a language concept can be used?

Relevance: The tighter it can be specified which contructs are allowed in which context, the better users an be guided (for example, through code completion).

Example: In the insurance DSL mentioned earlier this feature is used to restrict certain kinds of expressions to specific contexts.

Assignment: Define a `triggerself` statement and restrict its use to actions inside state machines.

Syntax Migration. Consider the textual state machine language. A user wishes to make a purely syntactic change to the language (e.g. changing the `on` keyword to `event`),

while keeping the underlying structure the same. How can programs be migrated from the old to the new syntax?

Relevance: Syntax-only migration is commonplace, especially in DSLs, where changing requirements as expressed by end users often lead to a need for different syntax.

Example: The syntax of the data type definition in Figure 5 had to be changed several times over the course of the project (keywords, arrangement) while keeping the underlying structure unchanged.

Assignment: Change the state machine language definition to use `event` instead of `on` for transitions. Discuss the migration of existing programs.

Structure Migration. This challenge is, in a sense, the inverse of *Syntax Migration*. Instead of adjusting a language's concrete syntax, consider a change to the structure of the underlying program representation while keeping the syntax the same. For example, in the state machine language, change the representation of send commands from `send(Event)` to `send(Statement(Call(Event)))`. Again, the question is how can existing programs be migrated to the new program representation?

Relevance: Structure migration is common as a means to refactor the structure of the AST, for example, to support certain kinds of extensibility.

Example: In mbeddr's state machine language, the structure of transitions had to be changed to support triggers other than events while keeping the syntax for event-based triggers unchanged.

Assignment: Change the program representation of send commands in the state machine language to `send(Statement(Call(Event)))`. Discuss how existing programs can be migrated.

6.5.3. Editing

The following benchmark problems address the *Editor* feature, and its subfeatures, from Figure 1.

Editing Incomplete Programs. Consider editing a program in MiniJava. A developer gets half-way through writing a variable assignment (e.g., `int x =`) and then realizes that they need to add another variable earlier in the program. Are they forced to make the first variable assignment x syntactically valid (e.g., by adding an initial value and a semicolon after the equals sign) before they can move to an earlier part of the program and add another variable assignment?

Relevance: It is tedious for users to be forced to first 'finish' typing one part of the program before being allowed to edit other parts.

Example: Simplistic projectional editors may not be able to deal with half-finished program constructs.

Assignment: Discuss how syntactically incomplete programs are dealt with.

Referencing Missing Items. Some editor technologies aim to ensure referential consistency by construction, that is, references refer to valid program elements at all times. For example, in MiniJava, can the user write a function call before declaring the function to be called? In this inconsistent state, are further edits possible? Is the reference automatically resolved when the function definition is eventually added?

Relevance: Forcing users to create every definition before it can be referenced can make life difficult for the user, because it prevents top-down programming.

Example: In mbeddr users can use a quick fix to create reference targets (function arguments, local variables or global variables) on demand from the location of a reference.

Assignment: Discuss how references can be written when the reference target is not available at the time of writing the reference.

Structure-agnostic Copy/Cut/Paste. Can the user select parts of a program which cut across the abstract syntax tree? Consider an expression such as 1+2*3. Can the user select and copy or cut 1+2* (without the 3) and paste it at other places? Does this retain the tree structure when possible?

Relevance: Users expect to be able to copy/paste arbitrary parts of programs, based on the concrete syntax. Restrictions based on the underlying structure are hard to accept.

Example: Some projectional editors (such as MPS) may only support selection along the boundaries of the underlying tree structure.

Assignment: Discuss how to select and duplicate program fragments that do not respect the underlying syntax tree.

Restructuring. Consider again the expression 1+2*3. The multiplication * binds stronger than addition +; the program is equivalent to (1 + (2 * 3)). How can the user edit the program to make it equivalent to (1+2)*3? This requires adding parentheses around nodes that were not siblings of the same parent.

Relevance: If restructuring were not supported, users would be required to tediously retype the complete expression.

Example: Again, some projectional editors do not support cross-tree editing (inserting parentheses is a good example of cross-tree editing). MPS supports this case.

Assignment: Discuss editing operations that rearrange the tree structure of the underlying syntax tree.

Language Demarcation. Consider the embedding of the query language into MiniJava as described above. Does the embedded query language require explicit 'begin' and 'end' markers (e.g., brackets such as ⟦ ⟧) to delimit the embedded language? Or is the user required to explicitly tell the editor that they wish to switch from editing MiniJava to editing the query language? Or is this fully transparent?

Relevance: Explicit demarcatoin can clutter the syntax of the program, especially when used on a fine-grained (expression) level.

Example: Converge [68] requires explicit markup of program fragments written in an embedded DSL, enforcing limiting a coarse-grained granularity of embedding. In contrast, MPS does not require explicit demarcation.

Assignment: Explain how language demarcation and disambiguation are handled.

Delayed Decisions. Again consider the embedding of the query language into MiniJava as described above. A query expression has the syntax `select <expr> from ....` Now also consider a new kind of MiniJava expression that has the syntax `select <expr> of ....` Both expressions start with `select <expr> ...` and the editor can only decide which concept a user referred to by inspecting the second keyword (`from` or `of`). How does the editor support languages that require this kind of 'delayed' decision?

Relevance: If this were not supported by an editor, users would be forced to make the decision up-front, for example, through code completion.

Example: In projectional editors such as MPS, users typically have to decide which construct they want to instantiate as the construct is entered.

Assignment: Describe how to implement the two different `select` expressions so that they can be used in the same language.

End-User-Defined Formatting. In any editor technology, the structure of a program is presented to the user in some visual form. This challenge addresses whether it is possible for users to change, reformat, or relayout the presentation, without affecting the underlying structure. For instance, this would allow different indentation styles or placement of open and closing grouping constructs. A full solution to this challenge persists and updates the different presentations.

Relevance: Different users may have different preferences; *one* formatting style on all users is hard to enforce.

Example: A parser-based language implementation typically ignores whitespace, and users can format programs as they please. In contast, a projectional editor typically forces one particular formatting and does not give any formatting freedom to users.

Assignment: Explain the mechanisms available for users to reformat existing code (without changing the program's AST or the language definition).

Specification of Default Formatting. Even though editor technology may support varying degrees of end-user formatting (cf. the previous challenge), it is often desirable to have a default formatting tool to automatically style and layout a program. How can such formatters be specified for a language?

Relevance: Especially for DSLs, users may not initially see the need for good formatting. An efficient, automated way to "clean up" the code after it has been written is essential.

Example: Languages defined in Xtext may define a formatter that can be invoked on existing programs. In contrast, since projectional editors include the formatting in the projection rules, no additional formatter is required.

Assignment: Describe how formatting specifications are integrated with language definitions.

Formatting Preservation. Refactoring and quick-fixes are examples of automated transformations that change a program's structure. The question addressed by this challenge is: What happens to the visual presentation of the program when such transformation is applied? Is the program reformatted according to the default formatting? Or is the user's formatting preserved as much as possible? Can the behaviour be specified for a specific language?

Relevance: Automated refactorings are much less useful if they destroy the formatting of the refactored program.

Example: In projectional editors this is a non-issue because they *always* render a program based on the AST. In parser-based systems, however, explicit care must be taken to enable this feature.

Assignment: Discuss how manually created formatting information is stored in the program and preserved by transformations.

7. Example Solutions to Challenges

The previous section introduced concrete challenges as the basis for comparing language workbenches in detail. In this section we provide two possible solutions to two of them to illustrate the use of the evaluation criteria introduced above. First, Section 7.1 shows how the *Metadata annotations* challenge is addressed in JetBrains MPS [65]. Second, Section 7.2 details a a Rascal [18] solution to the challenge of *Persistent User-defined Formatting*.

7.1. An Example Solution to Metadata Annotations

This section provides a sample solution to the *Metadata Annotations* problem, including picking the authors from a predefined list of names. The metadata annotations problem is one that has a very specific assignment. This is in contrast to the problem discussed in the next subsection.

The example is built with JetBrains MPS[12], a projectional editor. We would expect each description of a solution to provide a brief overview of the tool, but in this section we have insufficient space, and thus refer the reader to [67].

Assumptions We assume that an MPS model of users is available. Each user is identified by a unique name, and may have any number of additional attributes. We call the concept representing each user `AuthorDef`.

```
users {
  markus: Markus Voelter,      voelter@acm.org    // each line is an AuthorDef
  tijs:   Tijs van der Storm,  storm@cwi.nl
  laurie: Laurence Tratt,      laurie@tratt.net
}
```

Implementation The `AuthorAnnotation` language concept is used to represent the `@author` tag. To make it attachable to program nodes without the program node's definition knowing about it in advance, it must extend the predefined MPS concept `NodeAttribute`:

```
@annotated: <any> as role: author
concept AuthorAnnotation extends NodeAttribute
  references:
    authorRef: AuthorDef[1]
```

Annotations are stored under the node they are annotated to, but without the original definition of the annotated node declaring the child. The `role` property specifies the name used to store the annotation under the annotated node. The `annotated` property specifies which language concepts can hold `AuthorAnnotations` as children. In the case of the `@author` tag, any language concept can be annotated, and the annotation is stored in the `author` property. The `AuthorAnnotation` also references exactly one `AuthorDef` in its `authorRef` reference. Next we define the editor:

```
editor for concept AuthorAnnotation:
  [> [annotated_node] | @author | (%authorRef% -> name) <]
```

MPS' projection engine uses *editor cells* as the building block of editors and editor definitions are defined collections of cells, arranged in various ways. The editor definition above represents an `AuthorAnnotation` as a horizontal list of cells (`[> .. <]`) with the following ingredients: first, the node to which the annotation is attached to is shown

---

[12]http://jetbrains.com/mps

([annotated_node]). Then a constant @author is projected. Finally, the reference to an AuthorDef is rendered by showing the authorRef's name property.

The original definition of MiniJava does now know about the @author annotation, so it cannot just be typed into a program. Instead, an intention must be defined that attaches a @author annotation to a program node. Once defined, a user can select the intention from a special popup menu that is activated by pressing Alt-Enter. The implementation of the intention uses the special @ notation to store a new instance of AuthorAnnotation under the annotated node in the authorRef slot.

```
intention addAuthor for BaseConcept {
  description(node, editorContext) -> string {
    "Attach @author";
  }
  execute(node, editorContext) -> void {
    node.@author = new node<AuthorAnnotation>();
  }
}
```

Variants The solution above supports adding the annotation to any program node. This may be too flexible, and the allowed targets could be restricted in two ways. First, if there is a an existing language concept to which it should be restricted (say, ClassDefinition), then the @annotated property of the AuthorAnnotation could be changed to this concept (replacing <any> with ClassDefinition), and the intention should be defined for ClassDefinition instead of BaseConcept. The second alternative applies to the case where there is no existing concept, but a restriction is still necessary. For example, the @author tag may be allowed for classes and methods in MiniJava. Since there may not be a common superconcept of ClassDefinition and MethodDefinition, the first alternative cannot be used. However, one can add an isApplicable function to the intention, limiting the program locations at which the intention can be invoked:

```
intention addAuthor for BaseConcept {
  // ... like before ...
  isApplicable(node, editorContext) -> boolean {
    node.isInstanceOf( ClassDefinition )
      || node.isInstanceOf(MethodDefinition);
  }
}
```

Effort ca. 15 minutes.

Usability As mentioned, the @author annotation cannot just be typed in. Instead, they have to be attached with an intention: users have to press Alt-Enter on the to-be-annotated element and select Attach @author. The annotation is then attached to the element, and the author can then be selected via code-completion. While the requirement of using an intention can be seen as a limitation, intentions are quite natural to MPS users, since the projectional editor in MPS heavily relies on them.

Impact The only artefact that needs to be changed is the language that defines the @author annotation, which can be different from MiniJava. The language that contains the concepts to which @author annotations should be attached to (i.e. MiniJava in the example) need not be changed. To use the @author annotation, users add the language that defines the annotation to a program written in MiniJava.

Composability Syntactic composability in MPS is unlimited, and since the semantics of @author annotations are not relevant to MiniJava's semantics, no semantic interactions are possible. In particular, the structure of the AST (as defined by the original

language) is not affected; existing transformation or analysis tools will continue to work unchanged.

Limitations The approach can only be used inside the MPS projectional editor.

Uses, Examples A similar approach is used to attach requirements traces to mbeddr program elements; mbeddr is an extensible set of integrated languages for embedded software engineering [69] built on top of MPS.

7.2. An Example Solution to Persistent User-defined Formatting

This subsection illustrates a second example; it shows a solution to the *Persistent User-defined Formatting* problem. This benchmark problem has a more open-ended assignment, and we decided to show how user-defined formatting is preserved during a rename refactoring, implemented for the state machine language. This example uses Rascal, a language workbench and functional programming language for meta programming [18].

Assumptions We assume the state machine language is defined using Rascal's built-in context-free grammars Parsing an input file using the grammar results in a *parse tree* (i.e. a concrete syntax tree). Unlike abstract syntax trees (ASTs), parse trees contain all information regarding layout: white space, comments, keywords, punctuation etc.. Each subtree of the parse tree is annotated with *source location* values, to relate subtrees to the textual fragment they correspond to. Source locations are a built-in data type in Rascal, named `loc`. An example source location value is the following:

```
|project://statemachines/input/traffic.stm|(8,5)
```

A source location consists of the URI of the source resource, and an offset (e.g., 8) and length (e.g., 5) to identify the textual fragment within the referenced resource. Source locations function conveniently as unique identifiers of parse trees, and as such can be used when implementing name analysis.

Name analysis binds *use* occurrences of names to their *definition* occurrences. Such information is encoded using Rascal's built-in tuple and relation types. The following declaration defines a type alias `RefGraph`, which captures the required information:

```
alias RefGraph = tuple[set[loc] defs, set[loc] uses, map[loc, loc] refs];
```

In the remainder of this example, we assume there is a function (e.g., `resolve`) which computes a `RefGraph` from a state machine parse tree. The locations in a `RefGraph` will be derived from nodes in the parse tree representing names (i.e. identifier nodes).

Implementation The rename refactoring is implemented as a single function, `rename`, which takes a state machine parse tree `m`, the location of the identifier where the user invoked the refactoring (`x`), the new name (`new`) and the reference graph (`g`). The function `rename` renames all name occurrences that refer to, or are referenced by, the name located at `x` to the new name `new`. Note that the original name is not needed, since edges in the reference graph exist only between name occurrences with the same textual value. The `rename` function is defined as follows:

```
StateMachine rename(StateMachine m, loc x, str new, RefGraph g) {
  loc def = x in g.uses ? g.refs[x] : x;
  set[loc] toRename = {def} + { use | use ← g.uses, g.refs[use] == def };
  return visit (m) {
    Id x ⇒ [Id]new when x@\loc in toRename
  }
}
```

The first line retrieves the definition site of x if it is not a definition itself. Then, the set of locations that need renaming is computed in `toRename` using a comprehension over the reference graph. The comprehension finds all use occurrences referring to the definition `def`. The actual rename is performed using Rascal's **visit** statement, which is used to traverse trees in a structure-shy fashion. Whenever **visit** visits an `Id x` which has its location in `toRename`, it is rewritten to the new name. The notation `[S]x` parses the string value x to a parse tree of type `S`. Because the transformation is performed on parse trees, and transforming a parse tree back to text is automatic, no user defined layout is lost.

Variants The `rename` function described above is specific for the state machine language, since it refers to concrete state machine types (`StateMachine` and `Id`). It could be made fully generic using additional type abstraction. The resulting rename refactoring can then be used for any language, provided the required reference graph is available. This is a reasonable requirement because name analysis is needed anyway for type checking, identifier hyperlinking, and other semantic tasks. The details of such a generic rename, however, are outside the scope of this example

Instead of transforming parse trees, the rename refactoring could als be implemented generically on the source text directly. In this style, the source locations in the reference graph are interpreted as pointers into the source text. Instead of transforming the parse tree, the locations in `toRename` are substituted directly in the original source text.

Effort Given an existing name analysis, ca. 15 minutes.

Usability Currently, integration of refactorings implemented in Rascal with the refactoring framework of Eclipse is still pending. Users have to select a name and invoke a menu action to enter a new name. The textual update of source files participates in Eclipse's undo stack. This means refactorings can be undone, as expected.

Impact There's no impact on the rest of the language implementation. The only required glue code is registering the transformation with Rascal's IDE framework.

Composability The `rename` refactoring for state machines is very abstract: it only refers to the concrete `StateMachine` and `Id` types. The other dependencies are fully generic. As a result, the implementation is resilient to a large class of changes or extensions to the language. The only change that will affect this implementation is when additional types of identifiers are introduced, which seems unlikely in practice.

Limitations The approach sketched here requires the program to be represented using concrete parse trees, not ASTs, which is more common, and sometimes more convenient. Rascal supports ASTs through an explicit `implode` step which converts parse trees to typed ASTs. In general, however, transforming ASTs is not sufficient for refactoring, since it is important to maintain end-user layout and comments as much as possible.

The sketched solution only work on single file refactorings. For multiple files, the string substitution variant described above is more flexible: it just needs to load the files corresponding to the locations that should be renamed.

The described rename refactoring is not name-safe. In other words, it could introduce inadvertent name captures in the transformed code. A language-parameteric solution to address this is described in [70], which can be used to fix name captures after the transformation.

Uses, Examples A demo project illustrating DSL development in Rascal using a simple state machine language includes a slight variation of the rename refactoring[13].

---

[13]https://github.com/cwi-swat/missgrant

8. Concluding Remarks

Language workbenches have seen substantial innovation over the last few years, both in industry and academia. To document the state of the art of language workbenches, we established a feature model that captures the design space of language workbenches.

We evaluated data from LWC'13 and positioned 10 existing language workbenches in this design space by identifying the features they support. As our study reveals, at least one language workbench implements any given feature in the feature model, but no language workbench realizes all features. While some features are supported by almost all workbenches (textual notation, model-to-text transformation, structural validation, syntax composition, syntax highlighting, folding, auto formatting, semantic completion, and error marking), others are less widely supported (type checking, DSL program debugging, quick fixes, and live translation).

Also based on LWC 2013, 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, we observe that language workbenches provide adequate abstractions for implementing a language like QL. The results show a marked advantage over manual implementation. We also observe that the language workbench space is very diverse: different sets of supported features, maturity ranging from 1 to 18 years of development, single metalanguage or multiple metalanguages, industry or research. However, our results are not intended, and do not, allow us to conclude that any approach, or category, performs better than others.

To enable an even more detailed comparison and discussion of language workbenches and their features, as well as to inspire future LWC instances, we proposed a set of benchmark problems to compare support for flexible notations, evolution and reuse, as well as editing; by taking input from authors representing different editing approaches, we have attempted to make the benchmark problems as unbiased as possible. They are in line with the following trends we observed in the field of language workbenches:

- Integration of different notational 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 metalevel.
- Language workbenches are extensible environments, and not just tools for creating other tools.

Acknowledgements

[1] M. Fowler, Language workbenches: The killer-app for domain specific languages?, Available at `http://martinfowler.com/articles/languageWorkbench.html` (2005).

[2] S. Dmitriev, Language oriented programming: The next programming paradigm, JetBrains onBoard 1 (2).

[3] M. P. Ward, Language-oriented programming, Software – Concepts and Tools 15 (1995) 147–161.

[4] D. Teichroew, P. Macasovic, E. Hershey III, Y. Yamato, Application of the entity-relationship approach to information processing systems modeling (1980).

[5] M. Chen, J. Nunamaker, Metaplex: An integrated environment for organization and information system development, in: ICIS, 1989, pp. 141–151.

[6] P. G. Sorenson, J.-P. Tremblay, A. J. McAllister, The Metaview system for many specification environments, IEEE Software 5 (2) (1988) 30–38.

[7] M. S. Ltd., Quickspec reference guide (1989).

[8] K. Smolander, K. Lyytinen, V.-P. Tahvanainen, P. Marttiin, MetaEdit—a flexible graphical environment for methodology modelling, in: CAiSE, 1991, pp. 168–193.

[9] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, Centaur: the system, SIGPLAN Not. 24 (2) (1988) 14–24.

[10] T. Reps, T. Teitelbaum, The synthesizer generator, SIGPLAN Not. 19 (5) (1984) 42–48.

[11] P. Klint, A meta-environment for generating programming environments, Transactions on Software Engineering Methodology (TOSEM) 2 (2) (1993) 176–201.

[12] M. Anlauff, P. W. Kutter, A. Pierantonio, Tool support for language design and prototyping with Montages, in: CC, 1999, pp. 296–299.

[13] M. F. Kuiper, J. Saraiva, Lrc – a generator for incremental language-oriented tools, in: CC, 1998, pp. 298–301.

[14] M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, LISA: An interactive environment for programming language development, in: CC, 2002, pp. 1–4.

[15] J. Heering, P. Klint, Semantics of programming languages: a tool-oriented approach, SIGPLAN Not. 35 (3) (2000) 39–48.

[16] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344.

[17] E. Söderberg, G. Hedin, Building semantic editors using JastAdd: tool demonstration, in: Workshop on Language Descriptions, Tools and Applications (LDTA), 2011, p. 11.

[18] P. Klint, T. van der Storm, J. J. Vinju, RASCAL: A domain specific language for source code analysis and manipulation, in: SCAM, 2009, pp. 168–177.

[19] P. Klint, T. van der Storm, J. Vinju, EASY meta-programming with Rascal, in: GTTSE III, Vol. 6491, 2011, pp. 222–289.

[20] L. C. L. Kats, E. Visser, The Spoofax language workbench: Rules for declarative specification of languages and IDEs, in: OOPSLA, 2010, pp. 444–463.

[21] M. Eysholdt, H. Behrens, Xtext: Implement your language faster than the quick and dirty way, in: SPLASH Companion, 2010, pp. 307–309.

[22] S. Kelly, K. Lyytinen, M. Rossi, MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment, in: CAiSE, Vol. 1080, 1996, pp. 1–21.

[23] Honeywell Technology Center, Dome guide (1999).

[24] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi, The generic modeling environment, in: Intelligent Signal Processing, 2001.

[25] M. Voelter, V. Pech, Language modularity with the MPS language workbench, in: ICSE, 2012, pp. 1449–1450.

[26] C. Simonyi, M. Christerson, S. Clifford, Intentional software, in: OOPSLA, 2006, pp. 451–464.

[27] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, Programming environments based on structured editors: The MENTOR experience, Tech. Rep. 26, INRIA (1980).

[28] A. Alderson, Experience of bi-lateral technology transfer projects, in: Diffusion, Transfer and Implementation of Information Technology, 1997.

[29] MetaCase, MetaEdit+ revolutionized the way Nokia develops mobile phone software, Online, `http://www.metacase.com/cases/nokia.html` (June 5th, 2013) (2007).

[30] J. Kärnä, J.-P. Tolvanen, S. Kelly, Evaluating the use of domain-specific modeling in practice, in: Workshop on Domain-Specific Modeling (DSM), 2009.

[31] E. Visser, WebDSL: A case study in domain-specific language engineering, in: GTTSE II, Vol. 5235, 2007, pp. 291–373.

[32] Z. Hemel, E. Visser, Declaratively programming the mobile web with Mobl, in: OOPSLA, 2011, pp. 695–712.

[33] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al., The state of the art in language workbenches, in: Software Language Engineering, Springer, 2013, pp. 197–217.

[34] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Tech. rep., CMU Software Engineering Institute (1990).

[35] D. S. Batory, Feature models, grammars, and propositional formulas, in: SPLC, Vol. 3714, 2005, pp. 7–20.

[36] S. Erdweg, P. G. Giarrusso, T. Rendel, Language composition untangled, in: Workshop on Language Descriptions, Tools, and Applications (LDTA), 2012, pp. 7:1–7:8.

[37] M. Voelter, S. Lisson, Supporting diverse notations in mps' projectional editor, GEMOC Workshop 2014 (2014) 7.

[38] M. Voelter, J. Siegmund, T. Berger, B. Kolb, Towards user-friendly projectional editors, in: SLE, 2014, pp. 41–61.

[39] T. van der Storm, The Rascal Language Workbench, Tech. Rep. SEN-1111, CWI (2011).

[40] M. Hills, P. Klint, J. J. Vinju, Meta-language support for type-safe access to external resources, in: SLE, Vol. 7745, 2013, pp. 372–391.

[41] T. Vollebregt, L. C. L. Kats, E. Visser, Declarative specification of template-based textual editors, in: Workshop on Language Descriptions, Tools and Applications (LDTA), 2012.

[42] G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, E. Visser, Declarative name binding and scope rules, in: SLE, Vol. 7745, 2012, pp. 311–331.

[43] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, Sci. Comput. Program. 72 (1-2) (2008) 52–70.

[44] S. Erdweg, Extensible languages for flexible and principled domain abstraction, Ph.D. thesis, Philipps-Universität Marburg (2013).

[45] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, SugarJ: Library-based syntactic language extensibility, in: OOPSLA, 2011, pp. 391–406.

[46] F. Lorenzen, S. Erdweg, Modular and automated type-soundness verification for language extensions, in: ICFP, 2013, pp. 331–342.

[47] S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, E. Visser, Growing a language environment with editor libraries, in: GPCE, 2011, pp. 167–176.

[48] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, Layout-sensitive generalized parsing, in: SLE, Vol. 7745, 2012, pp. 244–263.

[49] R. Solmi, Whole platform, Ph.D. thesis, University of Bologna (2005).

[50] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hassel-bring, M. Hanus, Xbase: Implementing domain-specific languages for Java, in: GPCE, 2012, pp. 112–121.

[51] L. Diekmann, L. Tratt, Eco: A language composition editor, in: SLE, 2014, pp. 82–101.

[52] T. Parr, R. W. Quong, ANTLR: A predicated-LL(k) parser generator, Software Practice and Experience 25 (7) (1995) 789–810.

[53] B. Ford, Parsing expression grammars: A recognition-based syntactic foundation, in: POPL, 2004, pp. 111–122.

[54] M. Voelter, D. Ratiu, B. Schaetz, B. Kolb, mbeddr: an extensible C-based programming language and IDE for embedded systems, in: SPLASH Wavefront, 2012, pp. 121–140.

[55] P. Klint, T. van der Storm, J. Vinju, On the impact of DSL tools on the maintainability of language implementations, in: Workshop on Language Descriptions, Tools and Applications (LDTA), 2010.

[56] O. van Rest, G. Wachsmuth, J. Steel, J. G. Süss, E. Visser, Robust real-time synchronization between textual and graphical editors, in: ICMT, 2013, pp. 92–107.

[57] G. Sutcliffe, C. Suttner, The TPTP problem library, Journal of Automated Reasoning 21 (2) (1998) 177–203.

[58] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, R. Busse, XMark: A benchmark for XML data management, in: Very Large Data Bases, 2002, pp. 974–985.

[59] M. Schmidt, T. Hornung, G. Lausen, C. Pinkel, Sp2bench: A SPARQL performance benchmark, in: ICDE, 2009.

[60] D. Kurzyniec, V. Sunderam, Efficient cooperation between Java and native codes–JNI performance benchmark, in: The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications, 2001.

[61] E. Jakumeit, S. Buchwald, D. Wagelaar, L. Dan, Á. Hegedüs, M. Herrmannsdörfer, T. Horn, E. Kalnina, C. Krause, K. Lano, M. Lepper, A. Rensink, L. M. Rose, S. Wätzoldt, S. Mazanek, A survey and comparison of transformation tools based on the transformation tool contest, Sci. Comput. Program. 85 (2014) 41–99.

[62] L. Diekmann, L. Tratt, Eco: A language composition editor, in: Software Language Engineering, Springer, 2014, pp. 82–101.

[63] E. Roberts, An overview of MiniJava, SIGCSE Bull. 33 (1) (2001) 1–5.

[64] M. Fowler, Domain Specific Languages, 1st Edition, Addison-Wesley Professional, 2010.

[65] V. Pech, A. Shatalin, M. Voelter, Jetbrains MPS as a tool for extending Java, in: Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, 2013, pp. 165–168.

[66] M. Flatt, Creating languages in Racket, Communication of the ACM 55 (1) (2012) 48–56.

[67] M. Voelter, Language and IDE development, modularization and composition with MPS, in: GTTSE 2011, 2011.

[68] L. Tratt, The converge programming language.

[69] M. Voelter, D. Ratiu, B. Kolb, B. Schaetz, mbeddr: Instantiating a language workbench in the embedded software domain, Automated Software Engineering 20 (3).

[70] S. Erdweg, T. Van Der Storm, Y. Dai, Capture-avoiding and hygienic program transformations, in: ECOOP 2014–Object-Oriented Programming, 2014, pp. 489–514.