Language Extension and Composition with Language Workbenches

Markus Völter
Independent/Itemis, Germany
http://www.voelter.de
voelter@acm.org

Eelco Visser
Delft University of Technology, The Netherlands
http://eelcovisser.org
visser@acm.org

Abstract

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain. They provide linguistic abstractions and specialized syntax specifically designed for a domain, allowing developers to avoid boilerplate code and low-level implementation details.

Language workbenches are tools that integrate all aspects of the definition of domain-specific or general-purpose software languages and the creation of a programming environment from such a definition. To count as a language workbench, a tool needs to satisfy basic requirements for the integrated definition of syntax, semantics, and editor services, and preferably also support language extension and composition. Within these requirements there is ample room for variation in the design of a language workbench.

In this tutorial, we give an introduction to the state of the art in textual DSLs and language workbenches. We discuss the main requirements and variation points in the design of language workbenches, and describe two points in the design space using two state-of-the-art language workbenches. Spoofax is an example of a parser-based language workbench, while MPS represents language workbenches based on projectional editors.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Programming Environments; D.3.4 [*Programming Languages*]: Processors

General Terms Languages

Keywords Language Workbench, Domain-Specific Language, Meta-tooling, IDE, Projectional Editing, Parsing, Textual Language, Eclipse, Stratego, SDF, Spoofax, SGLR, MPS,

SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
ACM 978-1-4503-0240-1/10/10.

1. Domain Specific Languages

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain. They provide linguistic abstractions and specialized syntax specifically designed for a domain, allowing developers to avoid boilerplate code and low-level implementation details. The syntax can be graphical, textual, or even a mixture of the two. DSL code (often called a "model") is typically executed by an interpreter, or transformed into programming language code for subsequent execution. In addition to aligning notations closely with the domain, DSLs allow error messages using domain terminology and optimizations based on domain knowledge.

Since DSLs typically focus on a single aspect of software implementation, models in multiple DSLs or combinations of models and regular programs are needed to cover all aspects of a complete software system. Thus, DSL models need to interact with models expressed with other DSLs and with programs in general purpose programming languages. This requires the extension of domain-specific checking and optimization to combinations of languages.

To enhance acceptance of DSLs with their prospective users (programmers or domain experts), the languages must come with good IDE support, as we know it from tools like Eclipse, Visual Studio or IntelliJ. Modern IDEs increase developer productivity by incorporating many different kinds of editor services specific to the syntax and semantics of a language. They assist developers in understanding and navigating through the code, they direct developers to inconsistent or incomplete areas of code, and they even help with editing code by providing automatic indentation, bracket insertion, and content completion.

The development of new DSLs comprises many tasks, ranging from syntax definition to code generation to the construction of an integrated development environment (IDE). Language engineering tools are essential for productivity in each of these tasks. Specifically for project-specific DSLs, which are by their nature limited in scope, development must be efficient, so DSLs can be developed as part of real-world development projects.

2. Language Workbenches

Language workbenches [3] are tools that integrate all aspects of the definition of domain-specific or general-purpose software languages and the creation of a programming environment from such a definition. To count as a language workbench, a tool needs to satisfy the following basic requirements and preferably also support language extension and composition. Within these requirements there is ample room for variation in the design of a language workbench.

2.1 Basic Requirements

A language workbench should at least support the definition of individual languages. A language definition consists of the definition of the syntax and semantics of the language, as well as the editor services that form the IDE.

Syntax definition: defines the concrete notation used for models and their underlying structure, which is the basis for analysis and transformation.

Semantics definition: defines the analyses and transformations applied to the structures defined by the syntax definition, including error checking, transformations such as refactorings, and code generation to translate a model to an implementation in a target language.

Editor services definition: defines the editor services that bind the language to an integrated development environment (IDE), including syntax highlighting, outline view, bracket matching, automatic indentation, reference resolving, content completion, error marking, and refactoring. Editor services often depend on syntactic or semantic analyses of the structure of edited models.

Language workbenches typically provide high-level, declarative DSLs to make language definition efficient.

2.2 Extension and Composition

The next step beyond support for the basic language definition requirements is support for language extension and composition to cater for language evolution and software projects consisting of models in multiple languages.

Language extension: existing languages can be extended with new concepts, adapting them to more specific contexts.

Language composition: languages for different domains can composed, either by symbolic integration such that language concepts in one language can reference concepts defined in other languages, or by embedding, such that concepts from one language can be embedded in another one.

2.3 Variation Points

There are many decisions that must be made in the design of a language workbenches. The following is a list of variation points in the design space:

Concrete syntax: The main dividing line is between graphical and textual notations. While in the long run we want to be able to mix the two, currently most tools are either focused on one, or at least have a strong bias. In this tutorial, we focus on textual notations.

Parser-based vs. projectional: Textual notations can either be implemented based on parsers or based on a projectional editor.

Derivation of editor services: Some editor services can be automatically derived from the language specification (e.g. code completion), others may have to be customized by the developer (outline view icons, or custom syntax highlighting).

Storage: Storage can either be file-based, using existing version control tools for team collaboration, or repository-based, often supporting real-time collaboration by various users. Furthermore, storage can be based on the abstract or the concrete syntax.

Inconsistent definitions: The model editor may or may not allow models to be in an inconsistent or erroneous state. Supporting (temporary) inconsistencies improves agility of software development, not requiring the developer to tend to each detail immediately.

In the tutorial we discuss two state-of-the-art language workbenches. Spoofax is an example of a parser-based language workbench, while MPS represents language workbenches based on projectional editors.

3. Spoofax

The Spoofax language workbench [1, 6] is a platform for the development of textual (parser-based) domain-specific languages with state-of-the-art IDE support. Spoofax provides a comprehensive environment that integrates syntax definition, program transformation, code generation, and declarative specification of IDE components. The environment supports agile development of languages by allowing incremental, iterative development of languages and showing editors for the language under development alongside its definition. These editors can be used to view the abstract syntax of a program or to directly apply transformations on a selection of text. Spoofax is based on Eclipse, an extensible programming environment that offers many language-generic development facilities such as plugins for version control, build management, and issue tracking. Spoofax language definitions take the form of Eclipse plugin projects, and can be distributed to "end developers" using the Eclipse update site mechanism.

3.1 Syntax Definition

The grammar forms the heart of the definition of any textual language. It specifies the concrete syntax (keywords etc.) and the abstract syntax (data structure for analysis and transformations) of a language. In Spoofax, the syntax is also

used to derive customizable editor services, such as a default syntax highlighting service and an outline view service. Spoofax uses the modular syntax definition formalism SDF2 [4, 8] for the specification of grammars. SDF grammars are highly modular, combine lexical and context-free syntax into one formalism, and can define concrete and abstract syntax together in production rules.

3.2 Semantics Definition

Spoofax uses the Stratego program transformation language [2] to describe the semantics of a language. Stratego is based on rewrite rules for first-order terms, and strategies that control the application of these rules. During development, the abstract syntax view can be used as a reference for the first-order term representation of a language's abstract syntax. Rewrite rules may use string interpolation to conveniently generate text from textual templates. Alternatively, rules may rewrite to abstract syntax or may use syntax-checked concrete syntax expressions [9]. Code generation rules can be used to transform the DSL to a compilable form. They can be applied automatically as files are saved, or manually when triggered by the user. They can also be used to create views of the language. By default, views are automatically kept up-to-date and regenerated in the background as the source is changed. Stratego rewrite rules are also used to specify semantic editor services, such as error checking, reference resolving, and content completion.

3.3 Editor Services

Spoofax provides declarative editor descriptor languages for the definition of editor services. For many editor services, Spoofax generates default editor service descriptors from the syntax of the language, which can be combined with custom user-defined specifications in such a way that default descriptors can be re-generated when the syntax definition changes. Semantic editor services such as code generators and refactorings are declared by binding a user interface action to a semantics definition in Stratego.

3.4 Language Extension and Composition

Spoofax supports extension and composition of languages through the modularity of its underlying SDF and Stratego languages.

The syntax definition formalism SDF supports the full class of context-free grammars, which is the only class of grammars that is closed under composition. The definition of lexical syntax is integrated with the definition of context-free grammars, instead of using a separate language based on regular grammars for the definition of tokens. As a result, also lexical syntax definitions of SDF are closed under composition.

The semantics definitions in Stratego are also modular. The definitions of rules and strategies can be modularly extended to support new language constructs. More precise extensions can be achieved by extending hook definitions or instantiating parameters of transformation strategies. However, such extensions requires anticipation of extensibility in the design of the base language by including proper extension points.

Composition of languages is typically handled by means of a normalizing ('desugaring') transformation, which translates statements in an embedded language to an implementation in a common core language.

4. JetBrains MPS

JetBrains' Meta Programming System is a projectional language workbench [5] that has been developed over the last couple of years by JetBrains and is now available open source under Apache 2.0. MPS comes with an integration into popular version control systems. While the code is represented as XML files, the tool provides diff and merge facilities on the level of the concrete, projected syntax. Starting with version 1.5, MPS also comes with a facility to define debuggers for DSLs. The tool has been used extensively within JetBrains and is slowly getting traction outside of the company [7, 11]. An extensive, detailed tutorial for MPS can be found here [10]. An illustration of the capabilities of the tool is provided by the screencasts and papers listed on the mbeddr.com website [11].

4.1 Syntax Definition

MPS is a projectional editor. Consequently, language definition does not involve a grammar. Instead, language definition starts by defining the structure of the language through concepts. Secondly, projection rules, also known as editors, define the textual, tabular or graphical rendering of concepts.

The projectional approach has a couple of nice characteristics, in addition to fulfilling the requirements for language workbenches described above:

- Notations are more flexible than ASCII/ANSI/Unicode. Graphical, semi-graphical and textual notations can be mixed and combined. For example, a graphical tool for editing state machines can embed a textual expression language for editing the guard conditions on transitions.
- Since the model is stored independently from its concrete notation, it is possible to represent the same model in different ways simply by providing several projections. Different viewpoints of the overall program can be stored in one model, but editing can still be viewpoint specific. It is also possible to store out-of-band data, i.e. annotations on the core model/program, such as documentation, pointers to requirements (traceability) , or feature dependencies in the context of product lines.

4.2 Semantics Definition

In MPS, the structure of a program can be restricted using various kinds of constraints: scopes, determine the set of possible targets for references, type system rules calculate

types based on typing rules and an inference engine, and constraints check domain-specific properties of programs.

Transformations can be defined between arbitrary languages. Transformations are mappings from one language structure onto another one, i.e. transforming the underlying graph structure of a model. However, the concrete syntax of the target language can be used in transformations, making them look more like code generators.

Transformations can be cascaded and the MPS transformation engine incrementally reduces code until it cannot be reduced any further, at which point a text file is generated for subsequent compilation.

4.3 Editor Services

In parser-based environments where users basically enter text into a buffer, sophisticated editor services are optional — one can, in principle, use a simple text editor for editing. In a projectional environment this is different because editing requires the projection engine. Consequently, language definition requires the definition of IDE services. MPS does not even attempt to draw a line between the two: the definition of a language and it's editors automatically entails the creation of services for code completion, syntax highlighting, error markers, go-to-definition, and find references. While all of these can be customized, editor services can not be removed, since it would make editing models impossible.

4.4 Language Extension and Composition

In MPS, language definition is similar to object oriented programming in the sense that language concepts correspond to classes and models to objects. Thus, the principles for extension and composition from OO programming can be applied to languages. A language can inherit from another language, making the concepts from the base language available in the sub-language. The sub-language can then add new concepts, making the sub-language an extended version of the base language. Concepts in the sub-language can also extend concepts in the base language. This is the primary means of language extension: a base language might define a Procedure concept that contains a list of Statements. By defining sub-concepts of Statement, a sub-language can essentially plug into the base language, providing other kinds of statements usable in procedures.

The equivalent of delegation can be used to embed languages. A language can use another language and then define concepts that contain (as children) concepts from the used language. No special steps have to be taken to be able to integrate the languages syntactically, because no grammar and no parser is used.

There is another way of extending languages that closely resembles aspect oriented programming. A language can "contribute" additional properties to concepts defined in other languages, without invasively modifying this other language. This is very useful for all kinds of annotations such as documentation or traces to requirements.

Finally, the upcoming MPS 2.0 will allow sub-languages to define new notations for concepts inherited from a base language.

More details on language composition and extension with MPS can be found here [12].

References

[1] The Spoofax project. `http://www.spoofax.org/`.

[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. of Comp. Programming*, 72(1-2):52–70, June 2008.

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

[4] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF: Reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[5] JetBrains. Meta Programming System. `http://www.jetbrains.com/mps/`.

[6] L. C. L. Kats and E. Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In M. Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, 2010.

[7] Realaxy LTD. Realaxy action script editor. `http://www.realaxy.com/`.

[8] E. Visser. A family of syntax definition formalisms. In M. G. J. van den Brand et al., editors, *ASF+SDF 1995. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.

[9] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *LNCS*, pages 299–315. Springer-Verlag, October 2002.

[10] M. Völter. LWC 11 MPS Submission. `http://code.google.com/p/mps-lwc11/wiki/GettingStarted`.

[11] M. Völter and B. Merkle. mbeddr.com. `http://mbeddr.com`.

[12] M. Völter and K. Solomatov. Language composition with projectional language workbenches illustrated with mps. `http://www.voelter.de/data/pub/VoelterSolomatov_SLE2010_Language%20ModularizationAndCompositionLWBs.pdf`.