How to Bootstrap a Language Workbench

Andreas Prinz[1] and Alexander Shatalin[2]
[1]*University of Agder, Department of ICT, Agder, Norway*
[2]*JetBrains, Prague, Czech Republic*
andreas.prinz@uia.no, alexander.shatalin@jetbrains.com

Keywords: Language Workbench, Bootstrapping, Metamodelling

Abstract: Language workbenches are designed to enable the definition of languages using appropriate meta-languages. This makes it feasible to define the environments by themselves, as the meta-languages are also just languages. This approach of defining an environment using itself is called bootstrapping. Often, such bootstrapping is difficult to achieve and has to be built deeply into the environment.
The platform Meta-Programming System (MPS) has used bootstrapping for its own definition. In a similar way, the environment LanguageLab is using bootstrapping for its definition. This paper reports the implementation of LanguageLab in MPS thereby also porting the bootstrapping. From the experiences general requirements for bootstrapping language workbenches are derived.

1 Introduction

Meta-modelling (Gonzalez-Perez and Henderson-Sellers, 2008) is an approach to define languages using meta-languages. In turn, these languages can be used to define specifications (programs) of these new languages. Typically, one would use a meta-modelling environment, also called language workbench[1] (Fowler, 2005; Stoffel, 2010; Erdweg et al., 2013; Völter, 2014), to define languages and meta-languages. The meta-languages are also languages and can be defined using the same meta-languages. This process, called bootstrapping, is mainly an advantage for the workbench developers. It is typically not visible to the users. This paper will discuss two language workbenches, namely MPS (Campagne, 2014; Pech et al., 2013) and LanguageLab (Gjøsæter and Prinz, 2013).

The open-source industrial-strength *Meta-Programming System (MPS)* is provided by the company Jetbrains. MPS has several meta-languages covering a wide range of language-design elements in order to support comprehensive language design. All the meta-languages within MPS are defined using MPS itself making the platform bootstrapped. MPS features a language called Base Language resembling Java, which is then available for language extension and development inside MPS. For example, it is possible to define new constructs for Java (Base Language) within MPS. In the project mbeddr (Szabó et al., 2014), MPS provides an almost complete definition of C++, allowing C++ to be used and extended within MPS. This is exactly what mbeddr does: extending C++ in MPS with state machines and units for use in embedded device programming.

LanguageLab is an academic project at the university of Agder, Norway. It is not concerned with an industrial strength environment, but rather with the concepts that are needed to make meta-modelling user-friendly and feasible. The main goal of LanguageLab is to show the essential concepts of meta-modelling in a clear and understandable tool, such that it can be used in university teaching (Gjøsæter and Prinz, 2011). In the same way as MPS, also LanguageLab is bootstrapped such that the few meta-languages of LanguageLab are defined within LanguageLab.

MPS and LanguageLab have different focus. The general idea of MPS is to provide the user with as much as possible help in defining languages and using known notation when possible. This leads to the fact that MPS is based heavily on Java-like languages for the definition of most aspects in the platform. LanguageLab is focused on a clean environment, and in this context, it tries to avoid exposing the underlying language of the platform to the user. Although both platforms are implemented in Java, this is very visible in MPS, but not at all visible in LanguageLab.

Based on the strengths of MPS, a project was run

[1]There are also grammar-based language workbenches in addition to the meta-modelling environments.

to implement LanguageLab in MPS. This is of interest for LanguageLab, because it puts LanguageLab onto an industrial-strength platform, thereby improving the quality of LanguageLab. This is also of interest for MPS, as it challenges MPS to handle a new set of meta-languages.

In this paper, we describe this implementation, its results and general conclusions from it. We start with an introduction of essential terms and concepts of meta-modelling in Section 2. In Section 3 we discuss how LanguageLab can be described using MPS. Then we describe the process how to bootstrap LanguageLab in MPS in Section 4. In Section 5 we extract the general requirements for successful bootstrap. We discuss the project and possible alternatives in Section 6. Finally, we conclude in Section 7.

2 Meta-modelling in MPS and LanguageLab

This section introduces the concepts and terms used in meta-modelling in general, and in LanguageLab and MPS in particular. We normally use the terms of MPS in this paper, and indicate the different terms used in LanguageLab. We also indicate the terms used in Xtext for convenience. Meta-modelling is an application of model-driven development (Atkinson and Kühne, 2003) to language engineering and compiler construction. Instead of implementing language handling tools, e.g compilers, a model of the language is created. From this model, tools are automatically generated (Nytun et al., 2006). This way, languages can be designed quickly for the programming tasks at hand (Ward, 1994).

A *language model* has several different aspects, which together give a complete description of all important properties of the language. There are the aspect groups of structure, syntax, and semantics, as well as a group of tool-related aspects. In this section, we only look into the aspects that have been used in the project.

2.1 Structure (Abstract Syntax)

The structure (abstract syntax, meta-model) aspect defines the *concepts* that are used in the language and their relationships with each other. LanguageLab uses the term 'type' instead of 'concept'. Xtext uses EMF for the definition of the meta-model[2], and a 'concept' is represented by a 'class'. Concepts can own *properties* (of basic types) and *children* (enclosed concepts). Moreover, concepts can have *references* to other concepts. Properties are called 'attributes' in LanguageLab and EMF, while children and references are called 'aggregates' and 'references', respectively, in LanguageLab. In EMF, references and children are represented by associations, where children are contained associations. A concept can have a *parent* concept in terms of inheritance.

A language can be used by creating a specification (a program) in it. In this case, language concepts are instantiated into a node structure, where each node is an instance of a concept. Nodes have values (instances of properties), sub-nodes (instances of children), and links to other nodes (instances of references).

2.2 (Concrete) Syntax

The concrete textual syntax, called *editor* in MPS, defines how the specifications should look for the user. In MPS and LanguageLab the syntax is defined based on the structure, while for Xtext the structure is derived from the syntax (grammar)[3]. MPS uses a projectional editor (Völter et al., 2014), which means that the editor is presenting the internal node structure based on the syntax description. LanguageLab is based on a grammar-approach with LR analysis. The differences between the two approaches are quite serious in terms of user experience and implementation. However, for the sake of bootstrapping, the difference is less important.

2.3 (Dynamic) Semantics

The meaning of the specifications of our new language is defined in the semantics. There are two main ways to do this, namely transformations (compiler) and executions (interpreter). For reasons of simplicity, only *generators* are considered here, which are called 'transformations' in LanguageLab. Semantics is out of scope of Xtext, which is mostly concerned with the front-end, but can connect to semantics tools within Eclipse. MPS features a template-based generation with very sophisticated expressions, while LanguageLab has a very simplistic pattern-based transformation language.

[2]Please note that Xtext often auto-generates the meta-model from the grammar.

[3]There is an option in MPS to derive the structure from a grammar. Similarly, Xtext allows to base a grammar on an existing meta-model (structure).

3 LanguageLab in MPS

The first step for moving LanguageLab (LL) to MPS is to define LanguageLab in MPS as shown in Figure 1. The LL generator, structure, and editor meta-languages are defined in MPS. Finally, an LL Petrinet language is defined as a sample test language on top of those meta-languages.

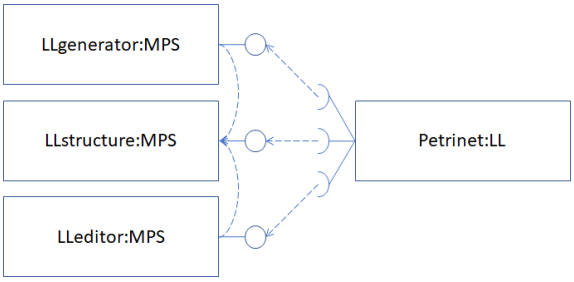

Figure 1: Defining Petrinet in LanguageLab, which is defined in MPS.

3.1 Structure Language

The initial LanguageLab structure language is defined using standard MPS. In the first place, the language is not yet a meta-language, but just a plain language providing the following concepts: Type, Property (with specializations Attribute, Aggregate, and Reference), AttributeType, Enumeration, and EnumElement.

When the types of the LanguageLab structure language are used, they are translated into concept declarations of the MPS structure meta-language. This turns the LanguageLab structure language into a meta-language.

3.2 Editor Language

Textual syntax in LanguageLab is based on a grammar language that is finally translated into an LALR parser. The editor is using the parser to read input and a pretty printing tool to show the current text. MPS works differently. Here, the editor is projectional, and no parser is needed.

Fortunately, the descriptions of an MPS editor and a grammar are similar enough to be able to translate a LanguageLab description into an MPS description. This is mostly due to both of them being quite declarative. Again, the LanguageLab textual syntax language is so far only a language. It becomes a meta-language because the editor descriptions are translated into MPS editor language constructs.

3.3 Generator

The transformation language of LanguageLab is very simplistic compared with the generator language used by MPS to describe model-to-model transformations. Therefore, the generator language for LanguageLab in MPS is an adaptation of the MPS generator language, adapted to the LanguageLab structure primitives. It is simplified and only provides the generator primitives needed for the bootstrap, i.e. the LanguageLab and Petrinet generators.

3.4 Testing with Petrinets

A simple Petrinet language was first built within plain MPS, and then again with the three LanguageLab languages explained in this section. With both versions it was possible to write Petrinet specifications, generate Java code from them and finally run them.

In the case of LanguageLab, the Petrinet language description was first generated into MPS meta-languages, and from there into Java.

The user experience is almost the same for the MPS version and the LanguageLab version, which indicates that the two versions are very similar. In addition, the code generator from Petrinet to Java generates the same code in both cases.

4 Bootstrapping

The definition of LanguageLab in MPS allows other languages to be defined in LanguageLab, as shown for the Petrinet language. But can we do the same for the LanguageLab languages? The process would be the same as for Petrinet: first, the description in LanguageLab is translated to MPS, and then down to Java, which is finally executed (Figure 2). The initial languages defined in Section 3 relate to the last step. Defining LanguageLab in LanguageLab (the first step) introduces a definitional cycle which might spoil the proper function of LanguageLab.

Figure 2: Generation steps.

In describing the bootstrap, we distinguish between the bootstrap situation we want to achieve, and the creation of this situation in the tool. We start by discussing the bootstrap situation, before we jump into the process to create this situation.

4.1 Bootstrap Situation

The final state after bootstrapping is to have a complete description of LanguageLab using LanguageLab, as shown in Figure 3. This figure depicts the

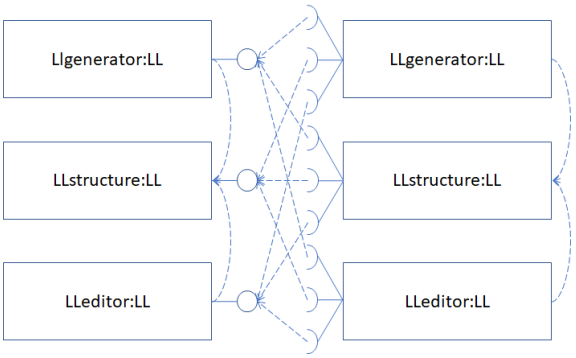

Figure 3: Ideal bootstrap situation. All languages are defined using LanguageLab (LL).

three LanguageLab languages in their meta-language role on the left, and in their language role on the right. Still, the languages are the same as indicated by the same name.

There are two kinds of dependencies in Figure 3: use dependencies and references. A use dependency is a connection between a meta-language and a language, i.e. an arrow from the right to the left. Essentially it means that a description is dependent on the language it is written in. A reference is given between languages on the same meta-level. In our case, both the editor language and the generator language depend on the structure language, as the editor and the generator descriptions always refer to their concept description.

We see in Figure 3, that the structure language is central to the bootstrap because of several reasons.

- It has five incoming dependencies.
- It is the only language with incoming references.
- It is needed already for the bootstrap situation.
- In addition, it is the only language with an internal definition loop (Type being defined by a Type).

Due to this central role of the structure language, we focus on it now.

Based on the initial languages described in Section 3, we can define the LanguageLab structure language using the initial LanguageLab structure, editor, and generator languages, see Figure 4.

The task for bootstrapping the structure aspect is to replace the LLstructure:MPS language to the left in Figure 4 with the LLstructure:LL language.

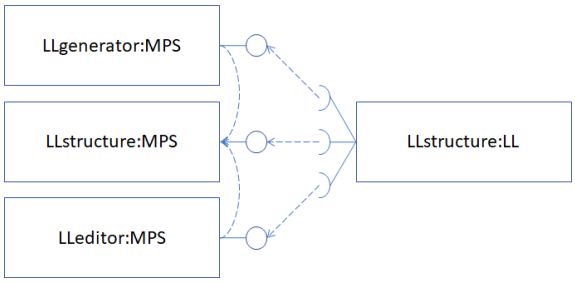

Figure 4: Preparing bootstrap: Defining LanguageLab on itself, but using MPS.

In order to make this new structure language usable, it has to be connected to adapted editor and generator languages, as shown in Figure 5. The adaptation is quite simple aligning the editor and generator languages with the references structure language, which is structurally the same in all cases.

The original LanguageLab meta-languages to the left in Figure 5 are defined using MPS. Using these languages, a bootstrapped version of the structure language is defined and connected to copies of the generator and editor languages with the change in their reference dependencies. This step can be repeated as shown in Figure 5. For the bootstrap process, we need to replace the initial structure language with the bootstrapped structure language in order to complete the cycle. When this is done, the other meta-languages in the middle column of Figure 5 need to be adapted by way of their references.

4.2 Bootstrap Process

We could extend the process depicted in Figure 5 several times, but it only creates a sequence of languages and not a loop (a language defining itself) as needed for the bootstrap. For the creation of the bootstrap situation, we first look at how a bootstrap process is done in LanguageLab and MPS individually in order to construct a process for LanguageLab in MPS.

LanguageLab has two features to allow a bootstrapping process. The first feature is that LanguageLab is interpreted, that is the language description (file) is used as it is. This can be exploited by simply changing the dependency in the language description file (i.e. outside the LanguageLab platform) in order to establish the bootstrapping situation. A second helpful feature of LanguageLab is that each language use is connected to its language definition via an interface, where the actual language can be exchanged when the instance is loaded. This means that there is no direct connection between language use and language definition. With these interfaces, bootstrapping is even possible within the platform itself.

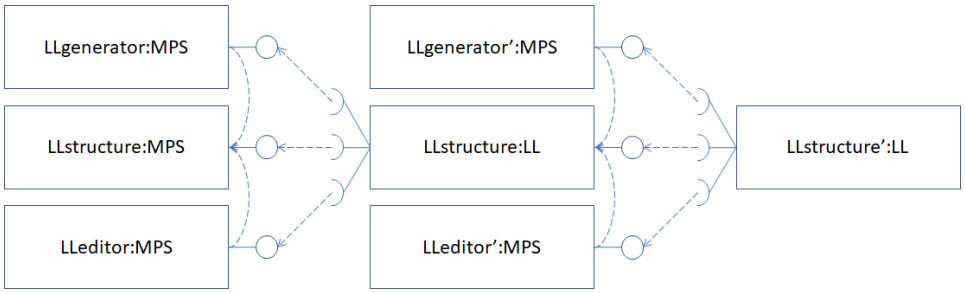

Figure 5: Situation before bootstrap.

MPS is a generated platform, where the language description is translated into Java code, see also Figure 2. MPS features a very strong connection between nodes and their concepts, such that it is essentially impossible to change the concept of a node. This implies that the bootstrap process can only happen by influencing the generated code, which was done for the MPS bootstrap.

For LanguageLab in MPS, we therefore have to generate the same Java code for the concept declarations of the bootstrapped LanguageLab structure language as we have already generated for the initial LanguageLab structure language. In MPS, two concept declarations are generated to the same code when they have the same name, the same concept ID and are in the same language[4]. We achieve the last point by collecting all the initial and bootstrapped concepts in one language. In this combined LanguageLab structure language the two versions of the language have different names to begin with.

For the generation of the concept IDs in MPS, the node ID of the concept definition is used as default. This can be changed to the node ID of the initial LanguageLab definition.

Based on these general considerations, the bootstrap process for the structure language has three stages: prepare, bootstrap, and beautify.

4.2.1 Preparation

In the first stage, the source is changed, but the generated code is left untouched. This allows running the old code. There are three steps as follows.

Source code control Put the generated code into source code control (SCC). Normally, only the language description is put into SCC. However, for the bootstrap it is important to keep the generated code, as this is used to load the languages. Without the generated code, it is impossible to load the bootstrapped language, because the code is used in loading and it cannot be generated without the code itself.

ID handling Insert the initial concept IDs for the generation of the bootstrapped concepts including IDs for the attributes, references and children. This step relates to the identifying property for concepts consisting of name, ID, and language.

Remove initial Change the names of the bootstrapped concepts to be the same as the initial concepts. Remove the initial concepts and related editor and generator. This is safe to do as the generated code and the loaded code are still available. Both of them will be replaced by the newly generated code from the bootstrapped concepts in the next step.

4.2.2 Bootstrapping

In the second stage, the generated code is changed. This will invalidate some of the definitions, which have to be fixed. It involves the following four steps.

Regenerate In this step, the old generated code is removed and new code is generated. The newly generated code for the bootstrapped concepts is more or less a copy of the old generated code for the initial concepts.

Fix meta-language references This step is related to the type of the references between meta-languages. The editor and the generator meta-languages reference structure concepts that used to be defined by the initial concepts. After bootstrap, they are defined by the bootstrapped concepts, and therefore the two languages have to be adapted. Due to the structural equivalence of the initial and the bootstrapped concepts, this is a simple mechanical replacement and also the generated code is almost the same as before.

Fix language descriptions After we have changed the editor and generator languages, we also need to adapt the LanguageLab editor and generator definitions, which is again a simple mechanical replacement.

[4]In MPS, languages are defined in so-called *modules*.

Regenerate again Finally, LanguageLab is generated by LanguageLab. This completes the bootstrapping process and establishes the correct bootstrap situation for the structure meta-language.

4.2.3 Beautifying

Although we have achieved the bootstrap situation, we still have a special ID handling in place. This is removed in the third stage as follows.

Nice IDs Change the concept IDs of the bootstrapped concepts to the IDs of the initial concepts (that are now deleted).

Regenerate and finish Finally, the bootstrap process is finished and we can rerun it as often as needed, even with changes in the definitions. We are back to the standard handling in MPS.

The process worked out exactly as planned. After aligning the Petrinet language definitions to the new LanguageLab, also the tests with the Petrinet language still worked out. All the code can be found in the LanguageLab git repository (LanguageLab, 2018) and the project is built on the JetBrains build server (JetBrains, 2018).

4.3 Bootstrapping the Editor

After the central part of bootstrapping the structure language is done, the remaining languages are easy to add. To test this, a bootstrapped editor language is created using the following steps.

Defining the editor structure The LanguageLab structure is used to define the structure of the LanguageLab editor language.

Defining the editor generator The generator for the editor language is defined using the LanguageLab generator language.

Generating Code for the editor language is generated such that the language is usable.

Defining new editors for structure and editor New definitions of the editors for the structure language and the editor language are defined.

Regenerate The bootstrap is more complete with this regeneration as now we have a bootstrap of both structure and editor.

A similar process works also for the generator language, completing the bootstrap for all three LanguageLab meta-languages. The success of the project shows that MPS is able to facilitate bootstrapping of another language workbench.

5 Needed Technology

Based on the project, we extract general conclusions about bootstrapping for meta-model-based language workbenches. In particular, we look into the elements needed for a bootstrap to work in the two parts: bootstrap situation and bootstrap process. Moreover, we consider a general process of porting a bootstrap to another workbench.

5.1 Bootstrap Situation

When we have created a bootstrap situation, we can continue using it as is or change elements of the bootstrapped language.

Keeping the cycle means that the language workbench is able to express the cyclic dependency of the bootstrap situation.

Changes A change in the bootstrap situation can be a problem because the definition could be invalidated by the changes. This is about using the old definition in order to create the new one having a distinct point where the change is taken.

Keeping a bootstrap situation is easy for language workbenches, as the connection between language definition and language use is the core of a language workbench. One way to make it work is to have two different representations of the language: as a language description for the definition role and as code for the use role, as in MPS and EMF/Xtext. Alternatively, the language can be loaded twice for these two roles, as in LanguageLab.

For handling changes, in case of MPS and EMF/Xtext, the point of change of the language is when the new code is generated. In LanguageLab, the point of change is when reloading the definition. In both cases, the language definition is used in a read-only mode thus allowing changes to the new definition using the old definition.

5.2 Creating Bootstrap

Creating a bootstrap situation requires the ability to construct a type-instance loop.

Circularity There needs to be a way to change the association between type and instance for the bootstrapped languages.

In normal operation, it is impossible to create circular definitions, as a language workbench would require a concept to be existing in order to create an instance of it. This way, 'concept' cannot easily be an instance of 'concept'. The creation of the circular definitions is the core of the bootstrapping process.

In MPS (and EMF), there are three versions of any language: first, the language description as instances of the meta-languages. Second, this description is generated to Java. Third, the compiled Java code can be loaded (and thereby activated) in MPS. The two first versions are available as files and can be stored in a SCC, while the last one is transient. This means in order to create the cycle it is most easy to work on the generated code as we have done here.

The situation in LanguageLab is slightly different, given by its interpretative nature. The language description is loaded directly, and without an intermediate compilation step. Still, the defining language is loaded read-only and before its instances. Changes to this language are not done until the next load of the language. This way, in order to create a cycle the definition file can be changed.

In order to make a language workbench ready for bootstrap, it is important to provide an operation to build an "empty" circular definition, which can then be extended at will. Such a definition eases the bootstrap process considerably.

5.3 Porting Bootstrap

Collecting the experiences from our project, we recommend the following steps for the porting.

1. Before jumping into the circularity, the first step of the porting is the creation of the meta-languages without the circularity using the meta-languages of the available language workbench.
2. Define a second set of meta-languages which are similar, but now defined on the first set. Still, the meta-languages are not circular.
3. As described before, the most critical part of the bootstrap port is the handling of the define-use circularity. Sort out the circularity by either adapting the code generation or by creating a loop in the workbench if that is possible. This will solve the internal circularity of the structure meta-language.
4. Circularities that involve define, use, and reference links can now be established stepwise. The editor and the generator can first be defined non-circular in order to get the structure loop working. Then they can be re-defined in a circular way and replace their previous definitions.

6 Discussion

In this chapter, we discuss two issues we encountered during bootstrapping.

6.1 Complete Bootstrap

The bootstrap in LanguageLab is complete in the sense that only the language descriptions are used, and the rest is given in the MPS platform. The situation is different for the MPS platform itself, which has all its meta-languages defined in MPS and generated down to Java, but in addition some extensions to the underlying IntelliJ IDEA platform (Fields and Saunders, 2006).

In the earlier LanguageLab, the bootstrap was also given by a set of languages together with a platform. In general, any bootstrap situation has to rely on a platform, and the question is merely how much of the platform has to be adapted to make the bootstrap work. Because of the power of the MPS platform, the LanguageLab bootstrap on MPS did not need any extra code.

6.2 Forward Loop

Currently, the bootstrap is created by wrapping the loop backwards, i.e. the IDs of the bootstrapped concepts are changed to fit the IDs of the initial concepts. Alternatively, we could work the other way around. We could change the IDs of initial concepts such that they fit with the bootstrapped concepts.

In general, these two possibilities look very similar and they have the same effect. In practice, a forward loop changes the initial concepts. This invalidates all their instances. This means all these instances have to be recreated. In addition, there is a risk that the changed initial concept IDs crash with the existing bootstrapped IDs. In order to reduce the risk and to keep some sort of stability, the decision was taken to wrap the loop backwards.

7 Conclusion

In this paper, we have shown how LanguageLab can be bootstrapped in MPS. The experiences have been generalized to requirements that have to be in place for a bootstrap to work.

We have bootstrapped the LanguageLab structure language including its internal definition-use cycle. On the base of the structure language, also the LanguageLab editor language was bootstrapped. The languageLab transformation language has not been bootstrapped, but is still partly based on MPS meta-languages. As it does not introduce new references to the bootstrap, we expect that it can be added with the same ease as the editor language.

Apart form the general possibility of porting the LanguageLab bootstrap to MPS, several technical tricks had to be used to make LanguageLab fit into MPS. This is mostly due to the different philosophy behind the two language workbenches, in particular their differing views on language aspects.

For making a bootstrap work in general, a language workbench needs to be able to use the same language as defining and defined specification, which is a special case of the standard language workbench functionality. Moreover, it has to keep the used language stable and to allow changing the type of an instance for introducing a loop. MPS, Xtext and LanguageLab fulfill all these requirements.

Acknowledgements

We thank the reviewers for their helpful comments.

REFERENCES

Atkinson, C. and Kühne, T. (2003). Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41.

Campagne, F. (2014). *The MPS Language Workbench: Volume I*. Fabien Campagne.

Erdweg, S., Storm, T. v. d., Völter, M., Boersma, M., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G. D. P., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V. A., Visser, E., Vlist, K. v. d., Wachsmuth, G. H., and Woning, J. v. d. (2013). The state of the art in language workbenches. In Erwig, M., Paige, R. F., and Wyk, E. V., editors, *Software Language Engineering*, number 8225 in Lecture Notes in Computer Science, pages 197–217. Springer International Publishing.

Fields, D. and Saunders, S. (2006). *IntelliJ Idea In Action*. Dreamtech Press.

Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? http://www.martinfowler.com/articles/languageWorkbench.html.

Gjøsæter, T. and Prinz, A. (2011). *Teaching Computer Language Handling - From Compiler Theory to Meta-modelling*, pages 446–460. Springer Berlin Heidelberg, Berlin, Heidelberg.

Gjøsæter, T. and Prinz, A. (2013). Languagelab 1.1 user manual. Technical report, University of Agder.

Gonzalez-Perez, C. and Henderson-Sellers, B. (2008). *Metamodelling for Software Engineering*. Wiley Publishing.

JetBrains (2018). Languagelab build server. https://teamcity.jetbrains.com/viewType.html?buildTypeId=MPS_LanguageLab_LlAll.

LanguageLab (2018). Languagelab git repository. https://tools.uia.no/stash/scm/projects_2015/mps-languagelab.git.

Nytun, J. P., Prinz, A., and Tveit, M. S. (2006). Automatic generation of modelling tools. In Rensink, A. and Warmer, J., editors, *Model Driven Architecture Foundations and Applications*, number 4066 in Lecture Notes in Computer Science, pages 268–283. Springer Berlin Heidelberg.

Pech, V., Shatalin, A., and Völter, M. (2013). JetBrains MPS as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 165–168. ACM.

Stoffel, R. (2010). Comparing language workbenches. In *MSE-seminar: Program Analysis and Transformation*, pages 18–24.

Szabó, T., Voelter, M., Kolb, B., Ratiu, D., and Schaetz, B. (2014). Mbeddr: Extensible languages for embedded software development. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 13–16, New York, NY, USA. ACM.

Völter, M. (2014). *Generic tools, specific languages*. PhD thesis, TU Delft, Delft University of Technology.

Völter, M., Siegmund, J., Berger, T., and Kolb, B. (2014). Towards user-friendly projectional editors. In Combemale, B., Pearce, D. J., Barais, O., and Vinju, J. J., editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 41–61. Springer International Publishing.

Ward, M. P. (1994). Language oriented programming. *Software-Concepts and Tools*, 15(4):147–161.