Experiences with Teaching MPS in Industry
Towards Bringing Domain Specific Languages Closer to Practitioners

Daniel Ratiu*, Vaclav Pech† and Kolja Dummann‡
*Siemens, Munich, Germany, Email: daniel.ratiu@siemens.com
†JetBrains, Prague, Czech Republic, Email: vaclav@jetbrains.com
‡itemis, Stuttgart, Germany, Email: dummann@itemis.de

***Abstract*—Domain specific languages (DSLs) bring substantial increase in productivity and quality and thus look very appealing to software engineering practitioners. Because language workbenches can drastically reduce the cost of building and maintaining DSLs and associated tooling, they catch the attention of technical leads and project managers in the industry. Effective use of language engineering technologies for software development requires specific knowledge about building DSLs in general and about language workbenches in particular. Practicing software engineers need to enrich their skills with a new software development approach and supporting tools. In this paper we present our experiences with training and coaching software practitioners in the field to develop domain specific languages and tooling by using Jetbrains' Meta-Programming System. We distill the experience gained in the last three years during 16 trainings which are offered by three organizations. The trainings were absolved by over 50 developers who belong to different business domains and have a wide variety of technical backgrounds, previous experiences and concrete needs. We present a set of challenges faced with teaching language engineering technologies in the industry. To address these challenges we developed a curricula containing increasingly complex topics and an approach which combines classical trainings with continuous coaching either remote or on site. Based on our experience we distill a set of lessons learnt about the dissemination of language engineering technologies to practitioners and about the concrete needs which need to be fulfilled to enable their broader adoption in practice.**

I. Introduction

Domain specific languages drastically improve the efficiency and quality of the developed systems by providing adequate abstractions, which mirror concepts from the business domain. Business domain experts can use the DSLs to directly build domain models with the help of domain concepts, reducing accidental complexity. Furthermore, domain specific languages are enablers of domain specific tooling [1]. Once domain models are expressed in semantically rich languages, then advanced analyses (e.g. consistency checks), higher automation (e.g. generation of other artifacts) and support for authoring models (e.g. meaningful auto-completion and smart editor actions) become possible.

Motivation for building DSLs: There is a wide variety of DSLs, which address the needs of different business domains. Software engineers want to build their own DSLs to simplify their development work (e.g. generate code from statemachines). Business domain experts (without programming knowledge) are interested in DSLs, which describe their domain specific models like medical devices, energy networks or taxation laws. In many cases, domain experts have already built tooling (e.g. Excel + Visual Basic for Applications) to help them with their daily work. Due to the fact that domain agnostic tools capture domain models only implicitly, they cannot adequately address the complexity of the domain, are fragile and hard to maintain. Domain experts soon recognize the potential of DSLs and domain specific tooling.

Language engineering and language workbenches: Language engineering brings languages development from the computer science field into software engineering – developing languages can be part of classical software development projects and processes. Language workbenches are tools, which enable a cost effective development of DSLs and associate tooling. In the past several years there has been organized the "Language Workbench Challenge", in which different features of language workbenches are systematically benchmarked [2]. The organization of tools-oriented benchmarking of approaches, with participants from industry, shows a high maturity of the technology and readiness for adoption.

Adoption in practice: Attracted by promises of language workbenches, software engineering practitioners and also domain experts with programming affinities become more and more willing to use the new technologies. Widespread adoption of DSLs requires that a big number of software engineering practitioners possess language engineering skills. The skills necessary to efficiently use language workbenches to design and develop DSLs are however only recently taught in universities. Practicing software engineers in the field have not received formal education about software language engineering topics, about designing and implementing DSLs and about using meta-tools like language workbenches.

Language development requires a good understanding of concepts related to programming languages such as abstract syntax, concrete syntax, type systems, or generators. Furthermore, classical topics like processes, quality assurance, maintenance and evolution must be applied also when languages are developed. One way to close this gap for engineers in the field is to attend specialized trainings.

Stakeholders for Trainings: Usually, trainees belong to one of the following there categories. *Software engineering generalists*, who will develop domain specific tooling around domain specific languages for different customers in different domains. *Software engineering from particular business domains* will develop DSLs as part of their ongoing software development projects in their business domain. *Domain*

experts with inclination towards programming but without formal education in computer science will build or customize themselves DSLs, which address the needs in their domains.

Our experience: This paper is an experience report about our training and coaching activities for disseminating language engineering technology in general, and MPS tooling in particular, to practitioners. The authors of this paper each have several years of experience with training language engineering with MPS and belong to three different kinds of organizations. *JetBrains* is a company which provides the MPS technology – the main know-how is about developing the language engineering technology (meta-meta level). *itemis* is a company specialized in providing solutions, which use model based development – the main focus is on building languages and tooling for customers (meta-level). *Siemens* is a large company active in many business domains and a typical targeted end-user of language engineering technologies (model-level). Within Siemens, the trainings have been performed by Siemens Corporate Technology, a central research and technology transfer unit. Even if there are slight differences between our experiences (inherent to the fact that the trainings evolved independently and are given in different settings), we are presenting in this paper a unified view.

Contributions: We make four contributions to the state of the art. Firstly, we present our experience with training and coaching language engineering technology with MPS to professionals, who work in industry. To the best of our knowledge there are no previously published experience reports about teaching DSLs to practitioners. Secondly, we describe the challenges that are faced by software practitioners when using for the first time language engineering technologies for developing DSLs and how we address them in our trainings. Thirdly, we present our lessons learnt which can be used as input in similar technology dissemination programs and we identify a set of open topics specific to building DSLs which need further research. Last but not least, we present a curriculum, specific to MPS, which aims at bringing generalist software engineering professionals up to speed with language development. We think that our experience is helpful both for educators teching DSLs in academia or in industry, as well as for researchers to gain insights about dissemination of technologies in industry.

Structure of this paper: In Section II we present the most important particularities of teaching language engineering technologies in industry as opposed to classical academic programs. In Section III we give a brief overview on how does the Jetbrains' MPS language workbench support different aspects of language development. In Section IV we present a set of topics that we address to teach generalist software engineers about developing DSLs and a concrete curricula that we use for training MPS. In Section V we present a set of lessons learnt and challenges from teaching MPS in the last years. In Section VI we discuss several variation points and address possible threats to validity about our findings. Section VII contains the related work and the following section concludes the paper.

II. Teaching Language Engineering in Industry

We have investigated the existing literature on teaching MDE and language engineering (Section VII). To our surprise we could not find any papers describing experiences or approaches for teaching language engineering topics to practitioners working in industry but only about teaching in the university context. Universities aim to educate for a lifetime or further learning, whereas our trainings are about a particular technology, concrete and immediate needs. Thereby, there are substantial differences between teaching in industry versus in academia both between trainees, trainers and structures of trainings as detailed in the following subsections.

A. Trainees

Trainees are professionals working in the field for many years. In the following paragraphs we illustrate differences between our trainees and students in universities.

Technical Background: The background of trainees varies a lot both between different teams, which attend our trainings and between members of the same team. Not being part of a formal degree, our trainings cannot assume that the participants have attended courses on compiler construction or fundamental concepts of programming languages, thereby the material is mostly self contained. We assume though that the trainees have some background in object-oriented design and development.

Experience: There is a considerable difference in practical experience between typical university students and our target audience. Our trainees have typically several years of experience in the field, have hands-on knowledge about different technologies, are very aware about "essential complexity", which is intrinsic to any domain and about the "accidental complexity" of very large or long-lived software projects or due to bad organization and management. Some of the people have (inevitably) had during their engineering career also experience with software projects which went wrong and thereby tend to be very sensitive to aspects like software maintenance and evolution, quality management, organizational constraints and support for distributed teams or their development processes. These aspects need therefore to be addressed in our trainings in order to make the trainees confident that the taught approaches can be applied also in their working context.

Integration into daily work: Experienced practitioners have developed their "own" working (programming) style based on common best practices or processes from their company. Many of them use agile processes and tests-driven development. For these people, it is important to understand how a new technology fits in their daily work.

Timeline of expectations: Practitioners are interested in how new technologies can fullfil their concrete project needs on a short time. Furthermore, the time and effort, which practitioners are able and willing to allocate is rather small (a few days of trainings) when compared to a classical university lecture (spanning over 12 weeks, with structured lectures,

seminars and homeworks). In order to fulfill these expectations, our trainings cover practical topics, which have been proven to work and have a high technology readiness level. Mature tool support is indispensable to use new technologies in production; our trainings are done hands-on with MPS. Classical trainings are complemented with coaching activities to support engineers (see below).

Motivation: Practitioners trainings are not part of any official degree, participants do not get marks and do not have to pass exams. Their motivation is therefore rather to learn new technologies either to master new challenges in their projects or as part of a continuous education program for example of their companies. Some of our trainees are technical leads in their organization and thereby early adopters and scouts for new technologies. In order to keep the trainees motivated, we must adapt the trainings dynamically to fullfil their expectations.

Alignment in motivation: Trainees coming from a single organization have typically their motivations aligned – they work in the same domain and have similar goals for what knowledge needs to be acquired and what exercises will be relevant to them. This is an advantage over groups formed from individual participants (open trainings). One that allows for more engagement and faster progress.

B. Trainers

Trainers are different as well when compared to educators in the academic context as described below.

Trainers are practitioners: Trainers are people employed in the industry, who offer trainings as parts of a larger consulting portfolio. They are experienced users of language engineering technologies. Usually trainers do not have much logistics support from the organization that provides the trainings. The preparation time for a three days training is usually short (one-two days).

Roles blending: After the initial training, the trainers are often (part-time) part of the ramping-up development team for several months acting as coaches, consultants or experienced developers. Even if the setting changes (trainer becomes part of the team), the goal remains the same: transfer of know-how and dissemination of language engineering technologies to software engineers in the field.

C. Structure of the Training

Structure of trainings are different when compared to lectures at universities, as described in the following paragraphs.

Training format: Trainings are compressed in a few intensive (full-day) modules. Trainees typically do not get homeworks, and the time amount dedicated to a training (usually 3 days, 8 hours per day) is much lower when compared to normal lectures in universities (usually 12 weeks, 15 hours per week including lecture, seminars and homeworks). For each topic we provide a short theoretical overview followed by explanation of how the MPS tool supports that concept. The focus, however, lies on practical hands-on exercises, which are done under guidance of the trainer. Tasks that the trainees do on their own individually are used sparingly, mostly to gain instant feedback on how well the trainees internalized the knowledge. For introductory trainings we additionally recommend the trainees to study the on-line MPS tutorials[1] prior to joining the course. This typically levels up the up-front knowledge of the trainees and shortens the time spent on entry-level exercises during the actual course.

From frontal trainings to coaching: Besides the frontal trainings per se, practitioners need continuous support in form of coaching on special problems they face. Coaching takes different forms like pair-programming sessions or hackatrons. Most times coaching topics address concrete problems that practitioners face in the projects they work on – like for example designing modular languages, implementing complex generators, debugging the generators, evolving languages or implementing complex analyses on models.

High interaction: Our customers often desire to have the exercises tuned around their domain of expertise to increase relevance and motivation for the trainees. Many times the trainings happen on site at customers' offices. The nature of hand-on exercises demands intense interaction, the trainer has to constantly walk around the room and fix problems in trainees' code. These little issues that trainees run into frequently initiate inspiring and educative discussions relevant to some or all of the participants.

Limited resources: Due to the fact that none of our organizations is specialized exclusively on trainings, there is usually only limited resources available for preparing trainings. Our trainings and coaching are rather knowledge sharing among practitioners than formal seminars.

III. JetBrains MPS

Since our training work is very much centered around the MPS tool, we briefly describe in the this section how MPS embodies different language engineering concepts.

Jetbrains' Meta-Programming System[2] is a language workbench which offers comprehensive support for all concerns of the development of DSLs and associated tooling. In MPS, a language implementation consists of several *language aspects*. MPS ships with a dedicated DSL for implementing each language aspect. For reasons of brevity, we will not describe any of the aspect-specific languages in detail; we refer the reader to [3] and [4].

A. The Structure Definition Aspect

The structure aspect uses a declarative DSL to describe the AST of a language. Each AST element is called a *concept*, and concepts can have children (that constitute the tree), references (cross-references through the tree), and properties (primitive-typed values). In addition, a concept can extend another concept and implement several concept interfaces. Like any framework, MPS provides several interfaces and base concepts which have special meaning and need to be known by language developers. In order to build domain specific tooling,

[1] https://confluence.jetbrains.com/display/MPSD20171/Fast+Track+to+MPS

[2] https://www.jetbrains.com/mps/

language developers also need to read or modify the models. To this end, MPS supports a rich API wrapped in the "smodel" language.

The structure of MPS languages can be compared to a regular object-oriented design, and is closely aligned with EMF or EMOF. The terminology used by MPS is however a little different and thereby confusing for beginners – e.g. "meta-classes" from EMF are named "concepts" in MPS.

B. The Editor Definition Aspect.

MPS features at its core a projectional editor to display the models. Projectional editors do not use parsers; instead, they render, or project, a program's AST in a notation defined by the language developer. MPS' editor aspect defines this notation – it is made up of cells arranged in a hierarchy. Language engineers can extend the editors and define their specific notations appropriate for the DSLs they build – e.g. tabular, diagrams, mathematics or trees. Each of these notations is supported by a special DSL which lower the complexity of writing editors.

MPS supports the definition of several editors for a single concept that can be switched at runtime. In addition, editor definitions can contain logic to project the same concepts in different ways (i.e. possibly using different notations), depending on context or user preference.

Being a projectional editor, MPS does not feel like text editors when users edit their models. In order to increase the fluency of models creation and modification, MPS allows advanced customization of editors actions – e.g. what happens when the user presses "backspace" in a certain editor cell, or how are linear sequences of lexical items transformed into models.

C. The Type System Definition Aspect.

Type systems are specified using declarative typing equations; MPS processes them using a solver. Various different kinds of equations are supported, the most important one is the type inference rule. It acts as type inference, but also acts as a type checker if several equations require different types for the same AST nodes.

MPS also supports checking of context sensitive constraints with the help of *checking rules*; these are essentially if-statements that check some property of the AST and report errors if invalid code is detected. The analyses performed by checking rules can be arbitrarily complex. Once an error is identified, language engineers have also the possibility to implement support for the language users to fix these errors via IDE automation – known as *quick-fixes*.

D. The Constraints Definition Aspect.

Typing and checking rules inspect an AST and report errors if invalid structures or types are found. In contrast, scopes and constraints prevent the user of a language from constructing invalid models up front. Remember that in a projectional editor, users can only enter code that is proposed through the code completion menu. The structure of a language determines the concept that can be used in a child or reference link. Scopes and constraints further constrain the valid nodes, beyond the concept. This is the main means of MPS to support building of only meaningful models in a pure constructive manner.

E. The Generators Definition Aspects.

MPS supports two kinds of transformations. Like many other tools it support text generation from an AST. The DSL to achieve this essentially lets developers add text to a buffer and helps with indentation. However, most of MPS' transformation work is done by mapping one AST to another one (model-to-model); only at the very end are text generators used to output text files for downstream compilation.

Each language used to create a particular model can contribute its specific generators to the generation stack. The (partial) ordering of generation steps in the generation stack is statically defined before the generation starts based on pair-wise priorities between different generators. The set of generation steps along with partial ordering among them form the generation plan. Generators are built with a set of expressive DSLs. For building more complex generators, MPS also exposes a rich API which can be used to implement the transformations.

F. IDE Extensions

MPS also allows the definition of IDE extensions such as new menus or views; laguage engineers use these extensions extensively for building domain specific tooling or to integrate external tools and present their output in a window inside MPS. The IDE extensions are implemented via regular Java/Swing programs and a couple of MPS-specific extension points.

G. Other Aspects

MPS ships with a few more standard language definition aspects for implementing automation via intentions, data flow analyses, refactorings of models and migration of models when languages evolve. The standard aspects are covered in our trainings, we refrain here from further details due to space limitations. The set of language definition aspects can be further extended in a modular manner by language engineers in order to fulfill recurrent language development needs – this requires however deeper MPS knowledge and is not covered in our trainings.

IV. Approach for Teaching DSLs

In this section we present the topics of our trainings: in the first subsection we tackle a set of coarse granular topics about language engineering and in the second subsection we describe a curriculum for teaching MPS.

A. Coarse-Granular Topics

In the following paragraphs we discuss the particularities of DSLs development using language workbenches when compared to classical software development. These particularities make it challenging for software engineers without previous

experience with DSLs development to use language engineering technologies. We cover these topics accross different parts of our trainings.

a) Extracting the Right Domain Abstractions: Identifying the right domain abstractions and expressing them as language constructs is a prerequisite step for each DSLs development. In contrast to software engineering, where abstractions are often used for the purpose of reuse (technical concerns), in DSLs the abstractions are much more focused on the domain concepts themselves (domain concerns). Furthermore, some of our trainees (especially domain experts without deep programming background) tend to think in terms of interchange formats. This happens especially in domains where such data interchange formats are standardized (usually in XML) in order to facilitate the exchange of data between different tools from a domain. These exchange formats contain a super-set of domain concepts used by different existing engineering tools. Practitioners tend to take these interchange formats as basis for their DSLs. It is thereby important to discuss with trainees that depending on their use-cases (i.e. what they aim to do with the DSLs) the interchange formats simply lifted as DSLs might not be a good approach.

b) Choosing the Adequate Notation: Choosing the set of notations (concrete syntax) and making editors look appropriate to the domain experts is highly important for acceptance of the built DSLs by domain experts. MPS allows language developers to define and combine multiple notations (e.g. textual, tabular, graphical or diagramatic) for the same language. Making use of this MPS feature at its full potential is very important to increase the readability of domain specific models or lower the learning effort required from new users. Furthermore, depending on a concrete use-case of the language user, one notation or the other might be preferable (e.g. graphics for overview, textual for rapid development of code-like models). Thereby we are constantly making aware the trainees about different options they have in order to choose an appropriate notation.

c) Constructive Support for Building Well-formed Models: Domain specific tools guide the users when they author domain specific models. Thereby, advanced constraints and scoping mechanisms which prevent creation of wrong models are highly important. Furthermore, these mechanisms can be used to implement low-level workflows for creating domain specific models and thereby lower the learning curve of a new DSL for domain experts. We are constantly making trainees aware about these advantages of DSLs when compared to general purpose languages or domain agnostic tools. An important part of the training is about MPS mechanisms which guide language users to build meaningful models, to prevent making mistakes up-front and to build domain specific workflows.

d) On-the-fly Analyses: Semantically rich models offer a plethora of possibilities to implement deep consistency checks. These checks usually implement invariants in the domain or reflect different design decisions of the language – e.g. when generators make some assumptions about the form of the input models, these assumptions must be made explicit through context-sensitive checks in the editor. We teach the trainees about different use-cases which can benefit from the implementation of on-the-fly consistency checks. Many times, there is a meaningful default fix for violations of consistency checks and this represents a great use-case for automation.

e) High Automation during Model-authoring: DSLs users can be substantially helped in creating their domain specific models by editor automation. Since increased automation is one of the main drivers for using DSLs, we teach the trainees about different possibilities (e.g. auto-completion, quick context sensitive editor actions, shortcuts, meaningful model templates at a certain point in the AST) available to increase the automation of models construction.

f) Model Transformations: Many models created with a DSL are used to produce some other artefacts (e.g. code, configuration files). This usually involves some form of model transformation. The knowledge of model transformation principles is, however, not very common among the practitioners. Even students with a strong software engineering background usually did not have much exposure to model transformation and thereby we cover the M2M basics in our trainings.

g) Language Lifecycle and Evolution: Languages need to evolve in order to reflect the changes in the domain or to accomodate new requirements. Existing models need to be migrated such that the invested effort in creating them is preserved. Planing and designing a language evolution strategy is a skill specific to DSLs and we cover this in detail in the advanced trainings.

h) Language Quality: Building production strength tools require systematic quality assurance. In case of languages, the quality characteristics concern aspects like domain appropriateness, modular language design and functional correctness of language implementations. In different parts of the trainings we make engineers aware about quality issues, which occur more often and how they can be detected. For example, after each language definition aspect we train developers also on writing unit-tests for the functional correctness of the implementation of that functionality; we also make them aware about the management of dependencies and modularization issues.

i) Variation Points about Language Design Decisions: Many times there are several possibilities to implement a functionality. We make trainees who already have some experience in developing languages with MPS aware about these possibilities and the trade-offs they imply. For example, language developers need to be aware of the two means of supporting users in building correct models: constraints, which up-front prevent entering wrong models, and on-the-fly checks, which give users feedback whenever context sensitive constraints are violated. The first solution might feel strange to the language users when they try to enter a model fragment which is incorrect – MPS will simply not allow to enter the model at all without any explanation and this might feel frustrating to users – many situations perceived as bugs – "it does not work". The on-the-fly checks and errors reporting in IDE allows users to enter the wrong models, but then marks

the wrong parts as erroneous with a sensible error message – the user has more support to understand what is wrong and correct the model.

j) Languages Design Idioms: Like any software, there are specific idioms and best practices when developing languages. Some of the idioms are generic, other idioms are specific to the base technologies behind MPS – e.g. projectional editors. We make trainees aware continuously about language design idioms.

B. A Curriculum for Teaching MPS

In the following we describe the curriculum that we cover in our MPS trainings: an introductory training for beginners, and an advanced training for language engineers, who already have experience with MPS. We designed the first training for three intensive days. Once the people get started with MPS, many of the advanced topics are disseminated as part of coaching sessions on concrete problems – most times thus the second part is not a single training session per se, but scattered across a series of coaching sessions.

a) Introductory Module: Upon completion of the introductory module, the trainees are expected to possess the basic skills necessary for building DSLs. More precisely, they are expected to be able to:

- define abstract syntax of new languages, extend existing languages with new constructs
- define textual editors for language constructs; understand how other notations can be used and how switching between notations work
- implement basic constraints, which prevent up-front the instantiation of nodes; implement checking rules which are displayed in the editors as errors; create basic type-system rules
- implement basic automation using editors actions, intentions and quick-fixes
- implement basic generators; understand how low level API of MPS can be used to implement transformations
- use MPS unit-testing framework to implement tests for the covered language definition aspects

In Table I we present an overview over the topics taught in the introductory module. Before trainees start with our courses, we encourage them to have a closer look at several examples of DSLs built with MPS – e.g., the screencasts from mbeddr [3] provide a good overview of what can be technically achieved with MPS.

Day 1: In the first hour of the training we present general basic principles of language engineering like the responsibility of roles (language developer vs. language user), language oriented development and discuss a few examples of successful DSLs implemented with MPS. The next hour is dedicated to the topics related to MPS tooling like e.g. the user interface, basic MPS configuration files, creating projects, solutions, models and writing first code using existing DSLs. Trainees also need some time to get used to the projectional editor and then learn a few essential keyboard shortcuts that assist them in the editor, such as for code completion, navigate to definition, code selection and the intentions pop-up menu. It is advisable to spend the time learning these in a relatively easy-to-grasp language, rather than struggle later in complex languages when defining an editor or a generator. The Node Explorer, Reflective Editor and the Hierarchy View are three tools that get also covered early on during the first day. These provide indispensable means to navigate through the model and are likely to help the trainees during the course. After the introduction of the tooling, we give a quick overview over the core MPS language definition aspects. Trainees create themselves a very small language (syntax, editors, simple constraints and simple generator). This step serves as an overview over the entire language creation process. The rest of the training is dedicated to looking in detail to the core language definition aspects of MPS. We teach the syntax and editor aspects together such that the trainees can immediately experiment with their languages. At the end of the first day, the trainees are able to define by themselves the syntax (abstract and concrete) of simple DSLs.

Day 2: We start the second day with more advanced topics about making a good editing experience for language users. This involves for example the definition of actions on specific editor cells, or transformation of the syntax tree when content is entered or deleted. The second class of topics is about mechanisms of MPS for defining context sensitive constraints over models (e.g. checking rules, type-system). This inevitably leads to programatically manipulating the abstract syntax, so the essentials of the smodel language are also covered and practised. The MPS Console tool provides a convenient way to teach the principles of navigating and changing models.

Day 3: We reserve an entire day to building generators, one of the most complex topics of MPS. We teach the trainees about the generation stack of MPS and about building basic generators. Besides the models-to-model generation capabilities, we teach also the model-to-text generation. There are several cases, in which the MPS's generator DSL is not powerful enough to cover all use-cases (e.g. when the generation is convoluted) – in these cases, MPS provides a low-level API and DSL to navigate and manipulate the syntax tree. After the training from day 3 our trainees should be able to build generators, which satisfy most of their functional needs.

Scattered Topics: There are a few topics which are scattered over the three days and covered when needed. For example, we make trainees continuously aware about different testing facilities of MPS for each language aspect, or about variation points in design and implementation.

b) Advanced Module: While the introductory module is focused on basic language development skills, the advanced training targets topics like design of modular DSLs, management of the DSLs life-cycle and extending the user interface. Upon completion of these topics, the trainees are expected to understand the concepts and possess the skills necessary for building complex and modular DSLs:

- define modular DSLs, understand modularization mech-

[3] http://mbeddr.com/screencasts.html

Topic	Duration	Learning Objective
Preparation (prior to the course)	2	People install MPS, watch motivational screencasts with DSLs at work
Introduction	1	Basic principles of language engineering, explanation of projectional editing,
MPS Tool	1	Understanding the directory structure of MPS tooling and main configuration files Understanding the MPS user interface, effective navigation around code, MPS projects structure
Overview of language definition	1	A tour through the DSLs that are used to describe the individual aspects of DSLs
Structure aspect	1	Abstract syntax definition, special MPS interfaces, modularization of the syntax, smart references
Editor aspect	3	Textual editors, different notations, editors styles, tipps for making editors look good
Editor actions	2	Editor actions to make projectional editing fluent; node factories
Constraints aspect	1	Constraints, reference and inherited (hierarchical) scoping, helpful scope implementations
Context-sensitive checks	2	Implement checks about well-formedness of models, quick-fixes SModel API to access and navigate through models, quotations and anti-quotations
Type-system aspect	1	Type-system DSL, principles of type-system evaluation
Dataflow aspect	1	The dataflow DSL and the dataflow diagram
Intentions aspect	1	The intentions DSL and its use
The MPS console tool	1	Using the Console to query and update models programmatically
Models generators	6	High-level model-to-model transformation, debugging model generators, generation priorities and plans Low-level API for transforming models, tracing of nodes through generation
Text generators	1	Text generator aspect
Basic DSLs testing	2	Basic testing of different aspects of the DSLs implementation

TABLE I
LECTURE PLAN FOR TEACHING DSLS WITH MPS - INTRODUCTORY LEVEL

Topic	Learning Objective
Introduction	Language modularization, language extensions
Structure aspect	Specialised links, node attributes, commenting out code Enhancing DSLs with end-user-editable functions
Constraints aspect	Tuning structure with advanced constraints and scoping
Editor aspect	Multiple editors for a concept, modularization of editors (e.g. editors components, styles), advanced DSLs to define editors (e.g. grammar-cells DSL)
UI plugins	Extending the MPS UI - menus, tool windows, editor tabs, preferences
Migrations	Evolving languages, migrating language usages
Refactorings	Creating automatic refactorings
The build language	Details of the build language
Building language IDEs	Building MPS and IDEA plugins, building standalone IDEs
Dependency analysis	Using the module and model dependency analysers
Advanced models generators	Explicit generator plans, cross-model generation, switch templates
Advanced testing of DSLs	Testing different generator aspects - editor, constraints, scope, type-system, dataflow Debugging models and language definitions
Handling attributes in generators	Handling node attributes in model transformations and text generators
Common language patterns	Examples of recurring needs and their common solutions

TABLE II
MPS TRAINING TOPICS - ADVANCED LEVEL

anisms like language extensions and embedding
- define complex editors, understand modularization features for editors and multiple projections
- implement modular context-sensitive constraints
- create advanced IDEs by adding new views and menus
- create complex and modular generators
- implement support for refactorings of models
- manage the life-cycle of DSLs (e.g. perform migrations of languages and models)
- deploy languages in standalone tools
- know common language design patterns and MPS idioms

In Table II we present an overview of the topics taught in the advanced module. As mentioned earlier, these topics are taught often during coaching sessions as needed by the practitioners. The first overarching theme of the advanced trainings are different modularization mechanisms of language definition aspects (e.g. structure, editor, constraints, generators) in order to avoid code duplication and enable reuse and extensions. The second theme is about building domain specific tooling using MPS. This requires extensions of the UI (e.g. with new menus and views) as well as the deployment of languages in standalone tools or integration into InteliJ IDEA as plugins. The third theme is about managing the lifecycle of languages for example enabling language evolution and migration of existing models. Last but not least we teach the management of different quality aspects of DSLs implementation (e.g. dependencies between languages, advanced testing). Along with this we provide recipes for good language design and implementation in form of MPS idioms.

V. LESSONS LEARNT AND OPEN POINTS

In this section we present a set of lessons learnt (Section V-A) and open points (Section V-B) distilled from our experience with disseminating language engineering technologies to practitioners in the field.

A. Lessons Learnt

a) From Training to Continuous Coaching: After training we support the trainees for some time remotely by continuously coaching them when they are faced with difficult tasks and unlock them when they get stuck. One successful form of continuous coaching proved to be *hackathons* in which an

experienced MPS developer develops for several days together with a ramping-up team. This on-site sharing of best practices with language design and development and with MPS tooling in particular proved to be highly valuable.

b) MPS has a Steep Learning Curve: There are a lot of concepts that trainees new to development of DSLs have to learn and start actively using during the trainings. The terminology mismatch between MPS and other tools further increases the complexity. Trainees coming from the programming background frequently struggle with projectional editing, especially in the Editor and Generator definition languages. Their habits of free-form text editing must first be un-learned, ideally early-on in the course. Trainees with modelling background, who are not professional programmers, struggle with the developer-centric world-view of MPS - the terminology used, the layout of the UI, the notations used in language design, error messages, the emphasis on keyboard shortcuts. Gradual exploration through the menus, explanation of the UI and focus on the visual aids like Context Assistant and Context Actions Tool help bridge this gap.

c) Language Implementation vs. Language Design: Teaching the language implementation proves to be much easier than training good language design. Language implementation is more mechanical, language design requires experience and taste and these are much more difficult to convey [4]. We think that catalogues of patterns about language design topics would help a lot to increase the maturity of the language development for beginners.

d) Pragmatic Concerns: Our trainees are constantly concerned about industrial pragmatic topics like integrating the language development in an agile setting, using test-driven development, integrating in the nightly build, support for distributed development of languages, support for distributed work for domain experts building models, concerns about performance issues in case of big models, possibilities to package and deploy the languages to end-users or maintaining and evolving languages. Failing to support these concerns is regarded most of the times as show-stopper for the adoption of a technology in practice. This is why, besides "classical" language engineering topics (e.g. syntax, generators, etc.), we address these pragmatic concerns continuously as first class topics in our trainings.

e) Higher-level DSLs for Language Definition: Higher-level language definition DSLs like those part of the base libraries of mbeddr[4] (e.g. grammar cells [5], diagramatic editors) drastically ease the definition of languages in most commonly encountered cases and increase the productivity. For instance the grammar cells hide many of the low level aspects of otherwise sometimes tedious editor definition and therefore allow us to get to usable results faster. Some of these higher-level languages hide the basic concepts of MPS. We prefer to teach the MPS native concepts and then make trainees aware that there exist further DSLs which can substantially simplify many language implementation tasks.

[4]http://mbeddr.com/platform.html

f) Digest Basics before Starting with Advanced Topics: It is very useful to give students time to digest and practice basic DSLs development aspects before teaching advanced topics. We do this by offering a three days basic course which covers the major MPS topics such that the trainees are able to build languages themselves after the training ends. The advanced topics are addressed either on demand (e.g. when the trainer has a continuous relation with the team) or after several months in which the students have had the chance to broadly practice the basics and identify by themselves the need for advanced topics like good language modularization, or performing language evolution and models migration in a disciplined manner.

g) Trainers Continuously Learn from Trainees: Due to the fact that many trainees have broad experience with real software projects in the field, their feedback on the adequacy of certain proposed solution to their problems is invaluable. Parts of the trainings are often veritable exchange of experience. Their suggestions to improve the methods and tools in order to better address challenges are an important source for requirements elicitation for the new versions of the MPS tool.

B. Open Points

In the following paragraphs we present several themes which are very important for practicing engineers and which we would like to teach deeper, but which are still open.

a) Requirements Engineering for DSLs: Practitioners are often facing challenges about requirements elicitation and specification for DSLs. Finding adequate domain abstractions to serve as DSLs constructs involves approaches from domain engineering. How these aspects blend in the case of DSLs development processes remains to be investigated.

Once DSLs requirements have been identified, a major issue regards the capability to specify DSLs semantics in a systematic and precise (e.g. such that DSLs can be further used for development of critical systems) yet practical manner (i.e. the specification must be understandable by practitioners). We feel a strong need for practical yet rigorous approaches to specify semantics of domain specific languages.

b) Documentation: Practitioners frequently bring up the question of documenting DSLs. Just like libraries document their API to ease their use, languages should have their concepts documented on a fine level of granularity. More so that it may not be always obvious how to enter some parts of the syntax in a projectional editor, so the workflows need to be documented as well. Most languages implemented in MPS to date rely on textual description of the syntax, examples of usage, common sense and the rule of least surprise, but a descriptive and convenient way to document languages and to make their options easily discoverable to the users has yet to be invented.

c) Patterns Language for Language Design: During our trainings we are constantly teaching best practices in form of patterns and idioms about domain specific languages development topics. We feel however that there is more to be investigated in this direction and a patterns language covering

several essential language design aspects (e.g. AST, context-sensitive constraints, generators) would be highly helpful.

d) Quality Assurance of DSLs: Complex DSLs which are used in a productive setting, require a high quality. Thereby, we would like to extend our advanced trainings with topics about characterizing and measuring quality attributes (e.g. usability, maintainability, reliability) of domain specific languages. Furthermore, complex model-to-model transformations need to be robust enough in order to ensure the correctness of the generated artifacts. We feel that both general quality assurance topics about DSLs in general and highly assured generators in particular still need more research.

e) Critical Systems Development: An increasingly pressing issue regards using DSLs for the development of critical systems. On the one hand, better abstractions, the possibility of eliminating up-front errors in a constructive manner or the deep analyses which are enabled by domain abstractions are very appealing. On the other hand, the resulting domain specific tools need to be certified themselves to be fit for purpose for the development of safety critical systems. For example, if production code is generated from high level DSLs, then the expectations about the quality and maturity of implementation are similar to those of general purpose languages and their compilers. Using language engineering technologies for development of tools used in safety-critical systems needs more research.

f) Language-Oriented Software Development: Besides techniques which address single problems, we are increasingly feeling the need for a comprehensive method for creating software around domain specific languages. We envision this as a new holistic software development paradigm spanning and gluing all development phases from requirements to evolution and maintenance [6].

VI. Discussion

In previous sections we have presented our experiences and lessons learnt with dissemination of language engineering technologies to practitioners. In the following we discuss a set of variation points and the measure in which our lessons learnt can be generalized.

a) Becoming a Proficient Language Engineer: Our trainings are focused on explaining language engineering concepts as they are reflected in MPS. Following our introductory trainings, the participants possess the skills necessary for developing simple but useful domain specific languages. There is however a long way to go after our trainings until becoming an experienced and productive DSLs developer.

b) Background of Trainers: This paper distills the experiences of three trainers belonging to three different kinds of companies: JetBrains is a typical provider of language engineering technologies, Siemens is a typical user of language engineering technologies and itemis is a company focused on consultancy around model-driven development. These organisations are representative for the spectrum of trainers and give us confidence that our training approach and lessons learnt have a high generalisation power.

All of the authors of this paper have several years of experience with giving DSLs building trainings in an industrial context. The lessons learnt in this paper is a summary of the 16 trainings given in the last three years which were absolved by 59 trainees acting as software professionals in different companies and business domains. Two of the authors have had previous experience with teaching seminars and exercises in university context.

c) Background of Trainees: We have had trainees with most different backgrounds: from professional programmers with high experience with object-oriented technologies to modeling professionals experienced with UML/SysML technologies and to domain experts and beginners with less software engineering experience. We think thereby that the spectrum of trainees is representative for a wide range of cases.

d) Particularities of MPS: All our trainings are around the MPS technology. MPS is a highly mature language workbench, but has some strong particularities. One of the most important particulaties of MPS is that it features a projectional editor. Thereby, parsing techniques, which are large topics classical in language engineering and which require considerable effort in teaching, are not relevant for MPS. Instead, we put a lot of effort in teaching the building of editors such that the editing experience of users is improved. Furthermore, the projectional editor is an enabler for multiple notations and for implementing modular and extensible languages. Large parts of our trainings are focused on these technical topics which are central to the MPS way of building DSLs.

e) Particularities of Trainings: During the writing of this paper we realized that there are slightly different approaches for giving MPS trainings by Jetbrains, itemis and Siemens – examples of differences are in the ordering of topics, or emphasis on different aspects. Some of the differences are due to the fact that the trainings evolved independently in time, other distinctions are due to different backgrounds and needs of participants. This paper presents an unified and consolidated view over our trainings.

VII. Related Work

To our surprise he have not found any literature, which describes experiences from teaching language engineering topics to practitioners in industry. The lack of experience reports might be due to the fact that industry trainings are done by people employed in the industry and their motivation to publish papers and share experience (possibly to competitors) may thus be low. We believe however that it is important to share practical experiences from disseminating technology to practitioners since this can act as a catalyst for transferring proven research results from academia into industry.

In the following paragraphs we present literature and its relations to our work.

Pedagogical Patterns: There exists a large body of literature about pedagogical patterns for teaching object-oriented technologies [7], [9]. We use this patterns as a source of inspiration for our trainings, but our work is different since

we focus on language engineering technologies and on training and coaching practitioners in the field.

DSLs Design and Implementation Patterns: [10], [11], [12] present a set of DSLs design guidelines. These guidelines are high-level and generic and it is often unclear how to refine them when a specific business domain or language engineering technology (e.g. projectional language workbenches) are considered. [13] represent valuable best practices which can be used by practitioners who implement domain specific languages. We use some of these patterns in our trainings and instantiate them in the context of MPS. Furthermore, all works mentioned above address the design and implementation of DSL and not their quality assurance, management of lifecycle or the use of meta-tools like language workbenches and their instantiation as domain specific tools.

Teaching MDE and Language Engineering in Academia: There is considerable published work about teaching model driven development in universities both to bachelor and to master students [14], [15], [16], [17]. Compared to these works, ours addresses the meta-level (teaching DSLs) and is targeted mostly towards experienced engineers and practitioners from the field.

In many universities language engineering topics are still scattered in classical lectures like compiler construction or fundamental concepts of programming languages. [18] reflects on several language engineering lectures held at universities in Bergen and Koblenz[5]. Our paper presents experiences from teaching language engineering techniques in the industry to software professionals by using MPS. As discussed in Section II, there are substantial differences between teaching in academia versus training practitioners in the field and we are convinced that industry trainings deserve attention as well.

Experiences with DSLs building and use: Several reports have been published about practical experiences with development of different DSLs, languages eco-systems and domain specific tooling [19], [20]. These papers describe what language engineering experts can achieve whereas the focus of this paper is about disseminating technology to practitioners. We are continuously learning from these experiences in order to enrich and refine our trainings.

VIII. Conclusions

We strongly believe that in the age of fast software technological advancements continuous education plays an increasingly important role. The measure in which new technologies are adopted in industry directly depends on the capacities of practitioners to use them effectively. Dissemination of (relatively new) research results to a broad audience of practitioners requires training and coaching approaches.

In this paper we presented our experiences with disseminating language engineering technologies to software engineering professionals in industry. Our technical means for our work is represented by the Jetbrains' MPS language workbench. We pinpointed a set of differences and challenges with training practitioners as opposed to classical academic courses. We presented our approach to perform the trainings both as block seminars as well as continuous coaching of practitioners when they encounter problems. We have distilled a set of lessons learnt and open points which are above our possibilities to solve and which require the effort of the modeling community.

[5]http://slecourse.github.io/slecourse/

Acknowledgements: We would like to thank Markus Voelter for the precious discussions and feedback about this paper.

References

[1] M. Voelter, "Generic tools, specific languages," Ph.D. dissertation, 2014.
[2] "LWC@SLE 2016 - language workbench challenge," 2016. [Online]. Available: http://2016.splashcon.org/track/lwc2016
[3] F. Campagne, *The MPS Language Workbench, Volume I and II*, 2016.
[4] M. Voelter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*, 2013.
[5] M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, and T. Berger, "Efficient development of consistent projectional editors using grammar cells," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2016, pp. 28–40.
[6] J.-L. Sierra, "Language-Driven Software Development (Invited talk)," in *3rd Symposium on Languages, Applications and Technologies*, ser. OpenAccess Series in Informatics (OASIcs), Dagstuhl, Germany, 2014.
[7] J. Eckstein, "Pedagogical patterns: capturing best practices in teaching object technology," *Software Focus*, vol. 2, no. 1, pp. 9–12, 2001. [Online]. Available: http://dx.doi.org/10.1002/swf.19
[8] R. Rodin, "Meta patterns for developing a minimalist training manual," in *Proceedings of the 19th Conference on Pattern Languages of Programs*, ser. PLoP '12. USA: The Hillside Group, 2012, pp. 25:1–25:15.
[9] A. Fricke and M. Völter, "SEMINARS: A pedagogical pattern language about teaching seminars effectively," in *Proceedings of the 5th European Conference on Pattern Languages of Programms (EuroPLoP)*, 2000.
[10] D. S. Kolovos, R. F. Paige, T. Kelly, and F. A. Polack, "Requirements for domain-specific languages," in *Workshop on Domain-Specific Program Development (DSPD)*, 2006.
[11] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schneider, and S. Vlkel, "Design guidelines for domain specific languages," in *Proceedings of the 9th Workshop on Domain-Specific Modeling (DSM)*, 2009.
[12] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, and J. de Lara, "Pattern-based development of domain-specific modelling languages," in *Model Driven Engineering Languages and Systems (MODELS)*, 2015.
[13] T. Parr, *Language implementation patterns: create your own domain-specific and general programming languages*. Pragmatic Bookshelf, 2009.
[14] B. Tekinerdogan, "Experiences in teaching a graduate course on model-driven software development," *Computer Science Education*, vol. 21, no. 4, pp. 363–387, 2011.
[15] J. Cabot and M. Tisi, "The MDE diploma: first international postgraduate specialization in model-driven engineering," *Computer Science Education*, 2011.
[16] R. F. Paige, F. A. C. Polack, D. S. Kolovos, L. M. Rose, N. D. Matragkas, and J. R. Williams, "Bad modelling teaching practices," in *Proceedings of the MODELS Educators Symposium*, 2014.
[17] S. Mosser, P. Collet, and M. Blay-Fornarino, "Exploiting the internet of things to teach domain-specific languages and modeling: The arduinoml project," in *Proceedings of the MODELS Educators Symposium*, 2014.
[18] A. H. Bagge, R. Lämmel, and V. Zaytsev, "Reflections on courses for software language engineering," in *Proceedings of the MODELS Educators Symposium*, 2014, pp. 54–63.
[19] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," *Software: Practice and Experience*, 2009.
[20] M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen, "Lessons learned from developing mbeddr: a case study in language engineering with mps," *Software & Systems Modeling*, pp. 1–46, 2017.