M. Riebisch, M. Tropmann-Frick (Hrsg.): Modellierung 2022,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2022 93

Teaching the Use and Engineering of DSLs with JupyterLab: Experiences and Lessons Learned

Joel Charles[1], Nico Jansen[1], Judith Michael[1], Bernhard Rumpe[1]

Abstract: Domain-Specific Languages (DSLs) are tailored to a specific domain which requires them to provide domain-specific concepts and a sophisticated tooling for their engineering; aspects which we address with the language workbench MontiCore. As we use MontiCore for research and teaching, we are interested in reducing the entry barrier to use and engineer MontiCore DSLs. While there are approaches for ready-to-use learning environments such as web-based editors, only a few provide a tailored solution for specific DSLs. Within this paper, we present our experiences using JupyterLab in combination with the infrastructure of MontiCore for teaching the use and engineering of DSLs in an interactive manner. We have realized three practical courses and one conference tutorial applying this technical approach. The front-end provides immediate feedback and includes supporting explanations in an integrated manner. Initial feedback indicates that this approach can lower the entry barrier for DSL use and engineering for students and practitioners.

Keywords: Education; Domain-Specific Languages; Model-Driven Software Engineering; Software Language Engineering; JupyterLab; Jupyter Notebook

1 Introduction

Domain-Specific Languages (DSLs) become increasingly important in practice, e.g., the German tax forms [Ru21], in systems engineering [Gu21], architecture modeling for safety critical automotive software systems [SBS20], cyber attacks in the automotive domain [Wo21], TV program planning [Dr20], or simulation of marine ecosystems [JH17], just to mention some published examples in addition to the reports from our industry contacts. Thus, universities need to teach DSL engineering and usage. A domain-specific language is tailored to the needs of domain experts [KRR18], thus, often adopting domain terminology for its concrete syntax.

Reusing domain vocabulary within a modeling language mitigates entry barriers and enables its users to create conform models without much familiarization effort. However, learning a DSL is difficult for non-specialists since they first need to get familiar with the terminology of the domain [GM18]. In addition, being able to engineer a high-quality DSL requires a lot of practical hands-on experience [KRR18]. It requires the understanding of how to develop a suitable syntax and the knowledge of using a language workbench [Ba20]. In our context, the sophisticated language workbench MontiCore [HKR21] is used to engineer DSLs.

[1] Software Engineering, RWTH Aachen University, Germany, www.se-rwth.de
{charles,jansen,michael,rumpe}@se-rwth.de

doi:10.18420/modellierung2022-014

Until now, there exists no integrated learning environment to teach MontiCore-based DSLs. In the past, our teaching of the introduction to MontiCore DSLs separated the theoretical concepts from their practical application. Furthermore, the initial setup of the IDEs required manual steps by its users. This distracted them from their actual learning objectives.

In this paper, we present an interactive approach based on Jupyter Notebooks to improve the teaching of Software Language Engineering (SLE) based on our experiences. We have designed the architecture and realized the teaching infrastructure for MontiCore-based DSLs with Jupyter Notebooks. Moreover, we present our lessons learned from concrete labs using this infrastructure.

In Section 2, we explain the basics of SLE, as well as the language workbench MontiCore we use to teach it. Section 3 addresses the requirements we specified for the solution and how the introduction to MontiCore and the use of MontiCore-based languages has been taught until now. In Section 4, we consider related work. Section 5 examines possible approaches and Section 6 describes the selected solution. Section 7 explains the realization in several practical courses and one conference tutorial, whereas Section 8 discusses our lessons learned. The last Section concludes.

2 Preliminaries

Our work is based on the general notion of software language engineering and the particular realization of the MontiCore language workbench, including its generated infrastructure.

Software Language Engineering. The discipline of SLE investigates the efficient development, maintenance, and evolution of modeling languages. In general, modeling languages consist of a concrete and abstract syntax, a semantic domain, and a semantic mapping [Bu19, GRR09, He07]. The concrete syntax defines the possible sentences, while the abstract syntax comprises its structural essence. The semantic domain of a language describes the target application area, which typically depends on mathematical theory, such as Petri nets [Re12] or Focus [BS12] (i.e., formal modeling foundations for describing behavior and processes). A semantic mapping provides a meaning for the sentences, i.e., it maps sentences to the semantic domain, for instance, via formalized mathematical notation or graph transformations [HR04]. We distinguish between two types of modeling languages, General-Purpose Languages (GPLs) that are generally applicable and DSLs that correspond to an application domain [Cl15, Hö19].

MontiCore. MontiCore [HKR21] is a language workbench for designing textual modeling languages. It utilizes context-free grammars (CFGs) to define the abstract as well as the concrete textual syntax of a language in a single effort. The grammars feature an EBNF-like [Wi96] syntax and consist of multiple productions that declare the language. MontiCore processes the grammar and generates abstract syntax classes that describe the language's structure. Additionally, infrastructure for advanced language development

is provided [HKR21], including a parser, a framework for context conditions (CoCos), and a symbol table. The parser processes textual models and creates a corresponding abstract syntax tree (AST) comprising instances of the abstract syntax classes. An AST contains the information of the underlying model without syntactic sugar, which is used for further processing. For a consecutive step, MontiCore provides a CoCo framework, which are additional validation rules for the language. CoCos are constraints on the AST that cannot be defined well within the CFG, such as context-sensitive restrictions. A common example is checking whether a name starts with a capital letter. Furthermore, MontiCore provides a symbol management infrastructure, enabling to derive a symbol table for an AST. This symbol table lifts the tree structure of the AST to a graph structure, enabling cross-referencing, easy type checking, and quick navigation. For further customizability of the generated infrastructure, MontiCore provides the TOP mechanism [HKR21]. It is a realization of the generation gap pattern and allows for overriding generated artifacts. The mechanism automatically recognizes these handwritten artifacts and integrates these seamlessly into the generated architecture. Finally, MontiCore uses template-based code generators [Ad18] to process the AST and to transform the structured information into artifacts of a target language. MontiCore also supports language inheritance, embedding, and aggregation [Ha15], extending existing CFGs and thus reusing their concrete and abstract syntax. These features support more sophisticated language development by leveraging productions of existing languages for new DSLs, thus fostering reusability.

3 Challenges and Requirements for Teaching DSL Use and Engineering

The usage of a DSL and the engineering of a MontiCore-based DSL address different target audiences, resulting in different requirements for its learning materials [St15]. We show specific challenges of learning a DSL, the challenges of teaching the engineering of a MontiCore DSL, and introduce the former teaching approach and its shortcomings. The identified requirements are labeled within the text. While several publications report about requirements for (the engineering of) DSLs [CMP18], our analysis directly relates to teaching how to effectively create and use such modeling languages.

3.1 Using a MontiCore DSL

Since DSLs by their nature are tailored to a specific domain, a modeler must have a basic understanding of the domain to create valid models. To support the progression towards becoming a domain expert, this *basic understanding of the domain* should be taught as part of the intended solution (**Req. U1**). Thus, according to Bloom's Taxonomy [AK01], the learning goal of learning a new DSL is to *apply* gained knowledge concerning the corresponding tasks. Generally, we cannot assume that students and tutorial participants are already domain experts for every DSL they learn. Having internalized the essential key concepts of the domain, a modeler can formulate semantic sound statements within the

domain. In order to model these in a structured way in a DSL, he must *familiarize* himself with the *syntax* of the language (**Req. U2**). Since no special prior knowledge is required for the modeler, their knowledge level varies. Therefore, enabling continuous learning progress requires an educational environment to *provide immediate feedback* on model validity and potential improvements (**Req. U3**). A steep learning curve has to be managed to achieve this level of knowledge, therefore *no initial set-up effort* should distract from the focus on the learning process (**Req. U4**). In order to convey an understanding of the language infrastructure, it is necessary to be able to *integrate further language tooling*, such as a generator, into the solution in an executable manner (**Req. U5**).

3.2 Engineering a DSL with MontiCore

The requirements for developing a DSL are partly similar to those for modeling, but in the case of MontiCore, they are more sophisticated, as it contains various aspects (e.g., grammar, CoCos, etc.) that have to be implemented, often in different meta-languages. Thus, fundamental language quality criteria, as well as the methods for creating a DSL must be learned (**Req. E1**). To define the DSL, the characteristics of the used language workbench must be considered. In our use case we assume that the language engineer has no prior knowledge of MontiCore. Neither should a technical know-how be a prerequisite to get started with DSL engineering (**Req. E2**). Furthermore, it should also be possible to use sophisticated modeling features such as CoCo's and the TOP-mechanism. Although they are technically written in Java, they are an integral part of an advanced DSL development with MontiCore. The TOP mechanism is a programmatic adaptation of the generated code, accordingly a more in-depth understanding of the generation process is required in this case (**Req. E3**). Based on these requirements and an analysis of existing learning and modeling environments, we designed a solution tailored for our use case in MontiCore.

3.3 Previous Approach to Teach DSL Usage and Its Engineering

At RWTH Aachen, we teach DSL usage and engineering, e.g., in the corresponding SLE lecture, which includes the presentation of theory with slides, smaller exercises, and a large group project using MontiCore. The objective of the course is to learn to engineer a DSL. The skills learned are applied in practice as part of exercises. Moreover, the fundamentals of the language workbench are introduced. Weekly assignment sheets are given to students as part of the exercise. They include descriptions of how students need to set up their development environments. Furthermore, they are provided with projects as a starting point for their task, which they work on in an IDE. Consequently, students not only have to familiarize themselves with MontiCore, but also with an IDE that may be unfamiliar to them. Its initial configuration effort obstructs them from achieving the actual learning objective. For groups of people who do not attend the course, we provide them a website with an introduction to the use of the language workbench. For details, we provide the students a

handbook [HKR21] which includes a getting started section explaining configuration steps, and a detailed documentation of MontiCore's features. Consequently, no setup-free access to learn MontiCore was available for any target group.

Previously, we provided generated command line interface (CLI) tools for learning an existing DSL. While CLI-access is arguably an easy and efficient way to use and chain tools for experienced practitioners, it does not foster the initial learning of the language and, thus, is less suitable for inexperienced users [St15]. Default editor features such as syntax highlighting or autocompletion are not available.

In the case of teaching how to engineer a DSL, we generally provided a more sophisticated Gradle project containing the required source files and dependencies for developing a new modeling language. These projects were integrable into an IDE, such as IntelliJ[2] or Eclipse[3], which generally facilitates the engineering process. In MontiCore, generated artifacts (e.g., the parser), as well as handwritten artifacts (e.g., custom CoCos), are based on Java, for which extensive support is automatically provided. While the use of an IDE closest reflects the actual development of a language, it also comes with some issues, especially for novice users. As not all students have a computer science or programming background, an IDE might be quite overwhelming, resulting in practitioners spending more time getting used to the intricacies of the environment than with the language engineering task itself. Thus, using an IDE-based teaching method yielded issues such as dealing with different operating system (OS) distributions, different improper Java or Gradle versions, flawed IDE configurations, and general troubleshooting.

Our target group works with private hardware. Accordingly, a wide variety of OS platforms are used. Due to a lack of resources, we are not able to fully support all systems such that some distributions could not be used. Besides practitioners using incompatible Java or Gradle versions, MontiCore relies on the Java Development Kit (JDK). Even though the provided installation instructions explicitly guide towards the JDK, a common mistake is that users only installed the Java runtime environment during setup, resulting in redundant double-checking efforts for the teachers. Additionally, individual problems, such as reusing IDE configurations from an unrelated project or access rights, require the attention of the exercise instructors. For issues that did not occur during the pilot operation, a root cause must be found in the short term. This is particularly problematic for cases in which the origin of the error is outside of our area of responsibility. Generally, the accumulation of errors and their various causes generated an increased, often unmanageable, support effort resulting in a need for a solution independent of external factors such as hardware, environment setups, and different levels of practitioner knowledge.

[2] https://www.jetbrains.com/idea/
[3] https://www.eclipse.org/

4 Related Work

Other modeling tools are also faced with the challenge of learning them initially. For example, the MetaEdit+ Workbench [KLR96] and ADOxx [FK13] offer workshop events and webinars to help getting started. They are supplemented by videos, reference documentation, as well as slides, and instructions. Furthermore, there are forums where frequently asked questions can be answered and new questions can be posted.

With the Language Server Protocol (LSP) and its graphical extension (GLSP) [Ro18], respectively, with editors that implement these protocols, potential tools are available that can be used for modeling. The one-time implementation of the protocol makes it possible to support all (G)LSP editors, which is an advantage over the AtoMPM [Sy13] and WebGME [Ma14] approaches. Both are tools to create domain-specific editors.

All of the above approaches have in common that they do not take an integrated interactive learning approach. Instead, they only offer solutions to facilitate tool-based modeling. However, our use case considers the engineering and use of DSLs created with MontiCore. Both the DSL engineering and the resulting DSL are purely textual. The goal is for learners to be able to design languages with MontiCore and create textual models. This is not achieved when the learner uses a UI focused on diagram-based abstractions. Furthermore, it does not solve the problem that the editor still has to be mastered by the user. The extensive documentation of the above approaches shows that their use takes a similar time of familiarization as an IDE. For the acceptance of a model-driven approach (in industry), it is necessary to use suitable tools [TK16]. However, the features are often extensive and distract from the actual learning objective.

Another paper [BVG18] analyzes teaching the Object Constraint Language (OCL) [RG02] using different modeling tools (such as MagicDraw, Papyrus, etc.). This work aims at analyzing different modeling tools for a specific language. It confirms that direct feedback during modeling can positively affect the final result. Furthermore, their observations support that immediate feedback in an integrated learning environment improves the overall learning success.

Furthermore, [Te19] presents a browser-based modeling tool. Its main objective is to analyze practitioners and their learning behavior during modeling. The application tracks interesting events, such as interactions with an editor. The main goal of this approach is to gather data about how novice modelers obtain knowledge about the modeling language and modeling itself. While our platform focuses more on education than on gathering information about the learning experience, it is notable that the presented solution also uses a web-based application as a foundation for a proper learning environment. While future discoveries of this research observatory will provide relevant information for our work, the described setup already indicates that a directly functional application, without initial effort, is a suitable foundation for learners.

In general, there are several approaches documented that focus on teaching modeling

languages or their convenient provision via corresponding tools. The sources range from literature studies [RTS19] over DSL design guidelines [Ul13] to experience reports on the transition to distance learning [Bo21], with special concerns on educating practitioners. Our analyses indicate that providing extensive material in a self-learnable manner and an integrated learning environment with immediate feedback has positive effects on the learning results. Applications such as the bigER tool [GB21] suggest that integrated modeling with different views, here presented on textual and graphical representations of entity relationship diagrams, can additionally increase modeling efficiency and understanding of models simultaneously. In summary, it appears that many of our requirements for learning a modeling language or creating one, established in Section 3, is part of current research. Recurring features include immediate feedback, minimal initial set-up effort, a convenient look and feel for familiarizing with the language or modeling tool, and further integrated functionality, such as well-formedness checks or code generation.

5 Learning Environments for Teaching DSL Usage and its Engineering

When choosing a development environment, many considerations must be taken into account. First, a single cross platform IDE should be chosen to ensure a consistent user experience. Furthermore, the focus is on learning a DSL or its engineering. Accordingly, the initial setup process should be minimal. Only web-based solutions allow for a setup-free user experience, so we limit the candidates to these only [LH01, Ag00](see Req. U4).

One category of online environments are so-called playgrounds. The TypeScript Playground[4] and JSFiddle[5] are exemplary representatives of the category. Their main objective is rapid prototyping (see Req. U3). Applications are for instance to showcase a reproducible minimal example or to test a new feature of a programming language. Usually a collaboration is possible, e.g., via sharing a link. Since this category of editors is more aimed at providing a basis for discussions using small code snippets, it is not suitable for our use case [St15].

The next category is represented by online IDEs. Solutions like GitPod[6] and CodePen[7] are designed to streamline the development process by providing an always configured online IDE for developers (see Req. U4). They can be used to develop entire applications, while facilitating collaboration between developers.

With the Meta Programming System (MPS) [VP12] is a tool from JetBrains available to engineer DSLs. However, the tool is a editor only, which does not offer the integrated learning approach (see Req. U1, U2) we are targeting [Pr21].

These approaches do not include introductions to syntax and therefore require a pre-existing basic understanding of the respective language (see Req. E2). Additionally, they contain

[4] TypeScript Playground, Microsoft: `https://www.typescriptlang.org/play` (accessed: 2022-03-08)
[5] JSFiddle, JSFiddle: `https://jsfiddle.net` (accessed: 2022-03-08)
[6] GitPod GmbH, Gitpod: `https://www.gitpod.io` (accessed: 2022-03-08)
[7] CodePen, CodePen: `https://codepen.io` (accessed: 2022-03-08)

more functionality than needed for our use cases. Therefore, they are less suitable for newcomers. Furthermore, they fail to combine the code with the explanations for learning the language.

Another option is to develop a tailored IDE ourselves, using the IDE Platform Theia[8] for example. However, the development effort for this is high and involves long-term maintenance costs.

The open-source project Jupyter [PG15, GP21] focuses on web-based interactive computing for a wide variety of programming languages. It originates from the IPython project [PG07], which has pushed interactive computing for python. The core concept of the approach are so-called Jupyter Notebooks. The Jupyter Notebook is an UI which combines interactive computing with additional annotations in a tutorial-based approach (see Req. U1, U2, U3, E1). In the Jupyter context, a *tutorial* integrates information and input fields on one screen following the Read-Eval-Print-Loop (REPL) concept [Va20]. Figure 1 shows a sample notebook. It is composed of a set of typed cells. Cells are executable and are either of type

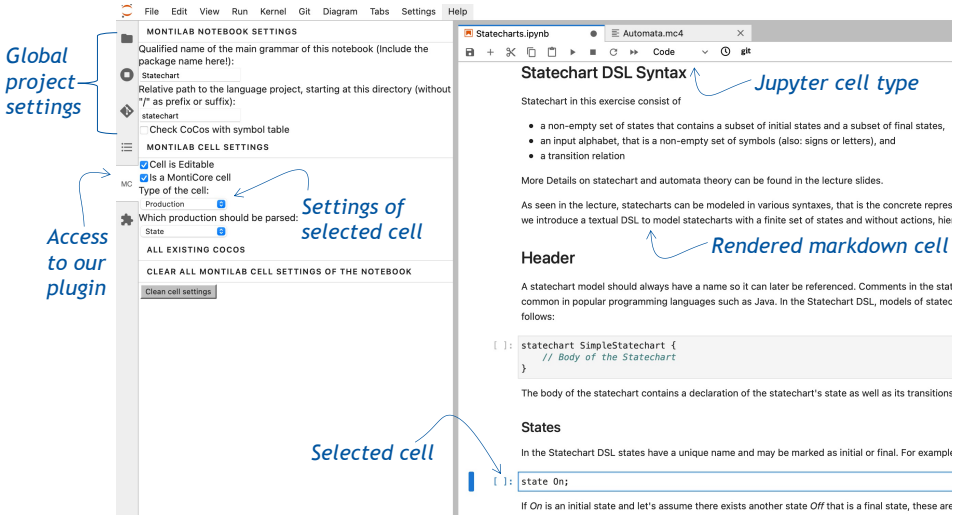

Abb. 1: A Jupyter Notebook edited in JupyterLab.

code, markdown or raw. The latter two types allow to insert additional information before, in between or after code cells. The execution of a markdown cell causes the markdown to be rendered in the cell. The raw cell only becomes relevant when the notebook is converted to another format, so this type will not be discussed in detail. Jupyter achieves the notebooks' language independence through the concept of kernels. The kernel contains all language-specific functionality and runs independent of the client used.

Within the project, JupyterLab represents the latest client. It is a web-based development

[8] Eclipse Foundation, Theia: `https://github.com/eclipse-theia/theia` (accessed: 2022-03-08)

environment (see Req. U4). The user interface is highly customizable through so-called plugins. Even the notebook itself in the right part in Figure 1 is such a plugin. The content of a code cell in the notebook is transferred to the kernel when it is requested to be executed. A kernel processes the request and transmits the result to the client. JupyterLab then renders the output under the executed cell (see Req. U5, E3). In this manner, domain specifics are separated from the language unspecific client. A notebook can be persisted in .ipynb format (json) and re-uploaded into JupyterLab. Thus, an executable documentation of the working process is available.

Compared to the previous solutions presented, JupyterLab is also suitable for less non-technical newcomers, as the functions are tailored to an interactive learning process [Bl19, Ma20]. In contrast, classic IDEs are tailored for the efficient development through experienced engineers. Compared to a custom solution with Theia, project Jupyter offers the advantage to build on an already established platform. At the same time, we benefit from enhancements at no cost to ourselves. For these reasons, we decided to use project jupyter for the realization. Table 1 summarizes the degree of requirements fulfillment of the available alternatives.

Approach	U1	U2	U3	U4	U5	E1	E2	E3
Playground	X	0	Y	Y	X	X	0	0
IDE	X	0	Y	Y	Y	X	X	0
Jupyter	Y	Y	Y	Y	Y	Y	Y	Y

Tab. 1: Fulfilled requirements by solution approach

6 Building the Jupyter Infrastructure

Jupyter Notebooks can be used both for engineering DSLs and for learning them. For the editing of Jupyter Notebooks we rely on JupyterLab. To enable multi-user support we use the JupyterHub which is part of project Jupyter. JupyterHub is a is a containerizable solution that provides user authentication and serves the configured JupyterLab. It suggests a deployment to an kubernetes cluster if more than 100 simultaneous users are targeted. However, otherwise the use of a single machine is suggested. We use the latter option as we initially assumed fewer than 100 users and could not justify the cost of a kubernetes cluster. However, we did not want to miss out on the advantages of a containerized deployment. Figure 2 shows the deployment we used. JupyterHub is executed inside a docker container, which is accessible via Https from the internet. When a user successfully authenticates, a container is created using JupyterHub's docker spawner. The spawner uses the same image for each user, which was created in advance on the host system. The image contains all the materials needed for the exercises. If the users manage to set their environment to a bad state while working on an exercise, the entire container can be restored. On the one hand, this is possible for administrators with access to the host system, but also for the users themselves, since communication of the user container with the JupyterHub container is established via a docker network. In principle, it is possible to store the files of the user

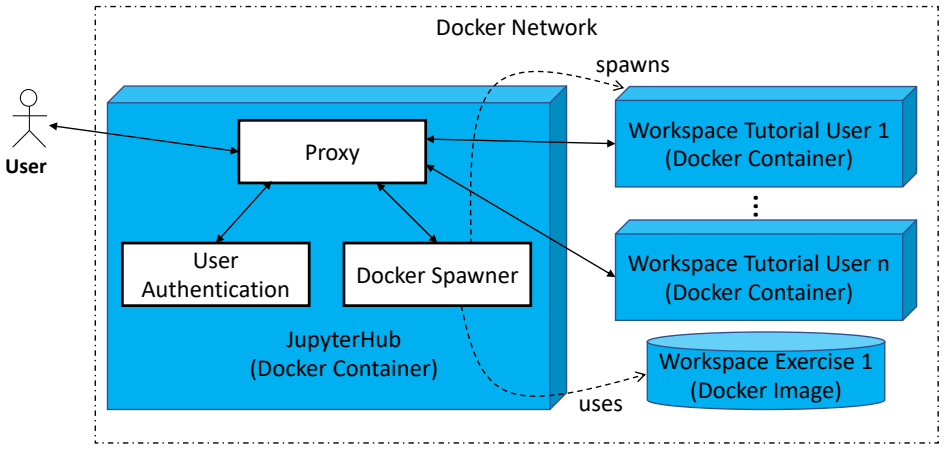

Abb. 2: The deployed JupyterHub architecture

containers via docker volumes. Then it is possible to remove the container in idle phases and save resources. As soon as the user returns, his data can be restored by mounting the volume. In our specific case, however, we have opted not to do so. On the one hand, we are able to keep the containers running for the period of the exercise with our user counts. On the other hand the files are no longer needed after the exercise is finished. However, if the learner want to persist the data, he can do so using the export function of JupyterLab. To avoid having to download individual files, a plugin is installed which downloads the files bundled in a zip archive.

Our objective is to process different artifacts within a single notebook (e.g. MontiCore grammars, handwritten Java classes). The fundamental idea is that JupyterLab orchestrates a language project in the background. This task is performed by a custom kernel developed by us. However, the code cell type is the only cell type that is passed to the kernel for execution. By default, JupyterLab does not distinguish between different types of code cells. Therefore a possibility had to be created either to send additional meta information about the cell content or to interpret cell's artifact type by the kernel. For some artifacts we need additional meta information in any case, which the kernel is unable to compute. For example, if a partial model is to be parsed in a cell, the kernel needs the meta-information which production of the grammar it should try to parse. This information should not be visibly included in the cell in advance, nor should the learner be forced to write it into the cell. For this reason we decided to develop a front-end plugin instead for a more robust solution. The plugin can be seen on the left in Figure 1 and ensures that the required metadata is sent to the kernel in addition to the cell content. In this way, only internally necessary information is hidden for the tutorial user.

7 Jupyter Notebook in SLE Teaching

By integrating explanations of concepts into an environment where they can be immediately applied, JupyterLab unlocks new opportunities to teach SLE. The following sections explain how JupyterLab was used to contribute to the teaching of SLE. Section 7.1 states details on building Jupyter notebooks for learning a DSL. Section 7.2 takes it a step further by explaining how engineering of DSLs can also be supported.

7.1 Using Jupyter Notebook for using a DSL

We have used Jupyter Notebook in the practical courses of the lectures Model-Based Software Engineering, and Model-Based Systems Engineering in the winter semesters 2020/21 and 2021/22 with about 60 students in each course. Complex domains were already modeled with MontiCore DSLs. It has been shown that expressive languages can result from this process, whose domain should be introduced iteratively according to our experience. For this reason, we have introduced the tutorial with markdown cells in which the domain is motivated. Building on this, terminology was introduced gradually. By transferring fragments of semantic statements of the domain into the syntax of the DSL via explanatory texts, keywords of the language and the syntax in which they are applied was taught. In code cells following the explanation, learners were able to immediately put the explanations into practice. By running the cell, they got instant feedback on whether their partial model conforms to the language. This procedure was repeated until the complete syntax was learned.

At this point, the tutorial user was able to create syntactically correct models, but those might be semantically invalid. Explanations in the notebook should point out this fact. By configuring a CoCo check in the front-end plugin by the tutorial creator in the subsequent cells, in addition to parsing the model, the semantic violations were experienced. The creators had the possibilities to deactivate individual CoCos. After the tutorial user created valid models, he was introduced to language tooling. Our plugin allows to configure a number of different operations on cell execution. The operations were sequenced one after the other. For example, if the parsing of a model was successful, CoCos were executed and upon success, the cell contents was used as input for a script or other language tooling. In this way, the generator generated artifacts based on the model. This can be multimedia content that is rendered below the cell or other kind of files that could be inspected via the file browser of JupyterLab. The notebook created during this process was saved by the tutorial user (e.g., as a pdf) and served as documentation for his learning achievements.

7.2 Using Jupyter Notebook for the engineering of a DSL

We have used Jupyter Notebook in the practical course of the lecture Software Language Engineering in the summer semesters 2021 with about 60 students and in a conference tutorial

at the Modellierung 2020 [Hö20]. Teaching the engineering of a DSL is fundamentally more challenging than using one (see differences in the cognitive processes for apply and create [AK01]). The language developer must consider not only the syntax of the resulting language, but also its maintainability [FR07, KRR18]. The language workbench MontiCore, whose handling must be trained, contains concepts to accomplish such a DSL. We used a similar approach as in Section 7.1 to getting started with MontiCore.

The learning process starts with a motivating introduction to SLE and the relevance of tailored DSLs. The objective of developing a DSL with the help of the notebook is formulated. A transition to the language workbench MontiCore follows. The MontiCore grammar is introduced as a central artifact of the engineering process. Through further explanations, the syntax of productions is introduced. To make the effect of a production experience-able, a code cell follows in which a grammar with a simple production is already given. Using our plugin it is possible to protect the cell from user modifications. When the cell is executed, MontiCore generates the infrastructure associated with the grammar. In the cell output area, a report appears showing the changes made to the underlying project by the generation step. The generated files are inspected via the file browser on demand. The subsequent markdown cell motivates the tutorial user to enter a partial model that matches the production. Upon execution, the learner is informed whether the input is valid and otherwise faced with a clear error message of the parser. In this way, the basic syntax of the grammar definition is introduced iteratively. At the same time, the tutorial user understands the impact of grammar changes on the generated infrastructure. The learner extends the given grammar with his own productions. The user investigates the correctness of the user-defined grammar by using predefined cells containing valid models. The underlying language project is in a valid state at all time which is ensured by our kernel.

After the grammar definition is completed, context sensitive conditions are introduced. The reason why these cannot be expressed in a context-free grammar is explained. In this advanced part of the tutorial, one can refer to Jupyter Notebooks which explains Java in its basics. Furthermore, inexperienced tutorial users can be supported by partially pre-filled code blocks in the code cell. The level of support was adjusted to the experience of the participants (see Req. E2). For example, our kernel is able to add package definitions or imports on its own. Furthermore, syntax highlighting and an integrated Java parser is used to communicate the source of errors. To avoid that a syntactically incorrect CoCo interrupts the generation process, the kernel adds it to the language project only if the Java parser accepts it. Handwritten extensions via the TOP mechanism are handled analogously. Finally, predefined models are used to check whether all checks have been implemented correctly. The language project created in this manner can be viewed by the tutorial user both in JupyterLab and exported to an IDE of his choice.

8 Lessons Learned and Discussion

The concept of Jupyter Notebooks allows to introduce the domain alongside the associated syntax of the DSL iteratively in markdown cells in an integrated manner. The insertion of code cells allow the practitioner to get immediate feedback when applying the acquired knowledge (see Req. U1, U2, U3). In the same manner language quality criteria can be teached without the need for prior tool experience (see Req. E1, E2). The custom developed kernel allows to respond upon a cell execution as needed and thus enables the integration of further language tooling (see Req. U5). For example, the kernel optionally outputs details of the generation process. JupyterLab, in turn, allows the generated artifacts to be inspected directly (see Req. E3). At the same time, the web-based approach eliminates the initial setup effort for the learner (see Req. U4).

The approach presented has already been used in three practical courses and in a conference tutorial. Lessons could be learned in the process of their realization. The lessons learned can be categorized into the perspective of the tutorial creator and the tutorial user.

Lesson learned 1: Initial setup effort. On the one hand the initial setup effort is increased for the tutorial creator. On the other hand, the setup process is completely eliminated for the tutorial user (see Req. U4). A custom kernel and a front-end extension have been implemented. An initial setup effort for the infrastructure is also required. However, once the containerized environment is established, it can be immediately reused in other contexts. This significantly reduces the effort required for follow-up courses which want to make use of the Jupyter infrastructure.

In addition, the tutorial content also needs to be created. A tutorial creator needs to get familiar with JupyterLab and the concept of interactive teaching of knowledge. However, a strategy for communicating the topics must be found in any case.

Lesson learned 2: Effort during execution. The unified development environment has resulted in significantly fewer inquiries from participants regarding the setup. In case of technical problems, the tutorial user have either been able to reset their environment themselves or have been able to ask an instructor to do so. The kernel ensures a valid state of the language project, therefore this case generally rarely occurs.

In one case, we accessed a user's container to reproduce his error pattern. It was caused by a bug in the kernel, which could be fixed by a redeployment. Our software architecture allowed us to apply the fix immediately for all users. In previous teaching units without JupyterLab, this would have required active actions from users. Furthermore, it would have been difficult to determine whether all participants had correctly applied the fix.

Lesson learned 3: Maintenance effort. The hosting of a server is accompanied by necessary maintenance work. Updates for JupyterLab, JupyterHub and other packages have to be installed. Since we want to provide the latest functionalities of MontiCore to the participants,

the kernel has to be maintained as well. Accordingly, the tutorial content may need to be extended in the future. However, this would also be the case in the traditional exercise mode.

Lesson learned 4: Required hardware resources. Developing models for a given DSL is not very resource intensive. However, as soon as sophisticated language infrastructure is used in the notebook, the CPU requirement increases significantly. Accordingly, the use of the MontiCore generator leads to an increase in the load of the system. The format of the exercise operation causes that at certain times a large number of users work on the system simultaneously. However, in other time frames, the server's capacity is idle. The load scenario advocates the use of a scalable cloud infrastructure.

Lesson learned 5: Overall feedback. Feedback on the use of Jupyter has been positive, both from tutorial creators and tutorial users. There are evaluations of the courses realized, which indicate a positive acceptance of the Jupyter Notebooks. Nevertheless, a full evaluation of the teaching approach has not yet been conducted by us. It has also been shown that at a certain experience level of the tutorial users a migration to a real IDE is desired. This is reasonable, for example, when complex language infrastructure is to be realized.

The availability of an interactive solution to teach SLE has prompted requests for more tutorial content. Since the expected user base is growing, we decided to move the solution to a kubernetes cluster.

9 Conclusion

Within this paper, we have shown how a technical infrastructure, namely JupyterLab, and its usage to teach the use and engineering of domain-specific languages. We have presented requirements for teaching DSLs and discussed them. Moreover, we have discussed lessons learned from its application in several practical courses and one conference tutorial.

To sum up, we think that JupyterLab is well useable for teaching the use and engineering of DSLs if one can cope with the increased need for server resources. The main advantages of this approach for *students* are that they have less set-up of the technology and can focus on the main teaching goals, they have information and input fields integrated on one screen, and they get interactive response of their solutions. The main advantages for *teachers* are that they need less technical supervision after the initial set-up and can focus on teaching DSL engineering and use. Moreover, they can reuse the Jupyter tutorial for following instances of the same practical course.

Literaturverzeichnis

[Ad18] Adam, Kai; Butting, Arvid; Kautz, Oliver; Pfeiffer, Jerome; Rumpe, Bernhard; Wortmann, Andreas: Retrofitting Type-safe Interfaces into Template-based Code Generators. In: 6th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD'18). SciTePress, S. 179 – 190, 2018.

[Ag00] Aggarwal, Anil K: Web-based Learning and Teaching Technologies: Opportunities and Challenges, Idea Group Publishing. Inf. Soc., 20(2):153–154, 2000.

[AK01] Anderson, Lorin W; Krathwohl, David R: A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives. Longman, 2001.

[Ba20] Barash, Mikhail: Example-driven software language engineering. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering. S. 246–252, 2020.

[Bl19] Blank, Douglas S; Bourgin, David; Brown, Alexander; Bussonnier, Matthias; Frederic, Jonathan; Granger, Brian; Griffiths, Thomas L; Hamrick, Jessica; Kelley, Kyle; Pacer, M et al.: nbgrader: A tool for creating and grading assignments in the Jupyter Notebook. The Journal of Open Source Education, 2(11), 2019.

[Bo21] Bork, Dominik; Fend, Andreas; Scheffknecht, Dominik; Kappel, Gerti; Wimmer, Manuel: From In-Person to Distance Learning: Teaching Model-Driven Software Engineering in Remote Settings. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, S. 702–711, 2021.

[BS12] Broy, Manfred; Stølen, Ketil: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer Science & Business Media, 2012.

[Bu19] Butting, Arvid; Eikermann, Robert; Kautz, Oliver; Rumpe, Bernhard; Wortmann, Andreas: Systematic Composition of Independent Language Features. Journal of Systems and Software, 152:50–69, June 2019.

[BVG18] Burgueño, Loli; Vallecillo, Antonio; Gogolla, Martin: Teaching UML and OCL models and their validation to software engineering students: an experience report. Computer Science Education, 28(1):23–41, 2018.

[Cl15] Clark, Tony; Brand, Mark van den; Combemale, Benoit; Rumpe, Bernhard: Conceptual Model of the Globalization for Domain-Specific Languages. In: Globalizing Domain-Specific Languages. LNCS 9400. Springer, S. 7–20, 2015.

[CMP18] Czech, Gerald; Moser, Michael; Pichler, Josef: Best Practices for Domain-Specific Modeling. A Systematic Mapping Study. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, S. 137–145, 2018.

[Dr20] Drave, Imke; Henrich, Timo; Hölldobler, Katrin; Kautz, Oliver; Michael, Judith; Rumpe, Bernhard: Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In: Modellierung 2020. GI, S. 173–188, 2020.

[FK13] Fill, Hans-Georg; Karagiannis, Dimitris: On the conceptualisation of modelling methods using the ADOxx meta modelling platform. Enterprise Modelling and Information Systems Architectures (EMISAJ), 8(1):4–25, 2013.

[FR07] France, Robert; Rumpe, Bernhard: Model-driven Development of Complex Software: A Research Roadmap. Future of Software Engineering (FOSE '07), S. 37–54, May 2007.

[GB21] Glaser, Philipp-Lorenz; Bork, Dominik: The bigER Tool-Hybrid Textual and Graphical Modeling of Entity Relationships in VS Code. In: 2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW). IEEE, S. 337–340, 2021.

[GM18] Gonnord, Laure; Mosser, Sébastien: Practicing domain-specific languages: from code to models. In: 21st ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Companion Proceedings. S. 106–113, 2018.

[GP21] Granger, Brian E.; Pérez, Fernando: Jupyter: Thinking and Storytelling With Code and Data. Comput. Sci. Eng., 23(2):7–14, 2021.

[GRR09] Grönniger, Hans; Ringert, Jan Oliver; Rumpe, Bernhard: System Model-Based Definition of Modeling Language Semantics. In: Formal techniques for distributed systems, S. 152–166. Springer, 2009.

[Gu21] Gupta, Rohit; Kranz, Sieglinde; Regnat, Nikolaus; Rumpe, Bernhard; Wortmann, Andreas: Towards a Systematic Engineering of Industrial Domain-Specific Languages. In: IEEE/ACM 8th Int. WS on Software Eng. Research and Industrial Practice (SE&IP). IEEE, 2021.

[Ha15] Haber, Arne; Look, Markus; Mir Seyed Nazari, Pedram; Navarro Perez, Antonio; Rumpe, Bernhard; Völkel, Steven; Wortmann, Andreas: Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In: Model-Driven Engineering and Software Development Conf. (MODELSWARD'15). SciTePress, 2015.

[He07] Herrmann, Christoph; Krahn, Holger; Rumpe, Bernhard; Schindler, Martin; Völkel, Steven: An Algebraic View on the Semantics of Model Composition. In: Conf. on Model Driven Arch. - Foundations and Applications (ECMDA-FA'07). LNCS 4530. Springer, 2007.

[HKR21] Hölldobler, Katrin; Kautz, Oliver; Rumpe, Bernhard: MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[Hö19] Hölldobler, Katrin; Michael, Judith; Ringert, Jan Oliver; Rumpe, Bernhard; Wortmann, Andreas: Innovations in Model-based Software and Systems Engineering. The Journal of Object Technology, 18(1):1–60, July 2019.

[Hö20] Hölldobler, Katrin; Jansen, Nico; Rumpe, Bernhard; Wortmann, Andreas: Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In: Modellierung 2020. GI, S. 189–190, 2020.

[HR04] Harel, David; Rumpe, Bernhard: Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Computer, 37(10):64–72, October 2004.

[JH17] Johanson, Arne N.; Hasselbring, Wilhelm: Effectiveness and Efficiency of a Domain-Specific Language for High-Performance Marine Ecosystem Simulation: A Controlled Experiment. Empirical Software Engineering, 22(4):2206–2236, 2017.

[KLR96] Kelly, Steven; Lyytinen, Kalle; Rossi, Matti: Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In: International Conference on Advanced Information Systems Engineering. Springer, S. 1–21, 1996.

[KRR18] Kautz, Oliver; Roth, Alexander; Rumpe, Bernhard: Achievements, Failures, and the Future of Model-Based Software Engineering. In: The Essence of Software Engineering, S. 221–236. Springer, 2018.

[LH01] Lin, Binshan; Hsieh, Chang-tseh: Web-based teaching and learner control: A research review. Computers & Education, 37(3-4):377–386, 2001.

[Ma14] Maróti, Miklós; Kecskés, Tamás; Kereskényi, Róbert; Broll, Brian; Völgyesi, Péter; Jurácz, László; Levendovszky, Tihamer; Lédeczi, Ákos: Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure. MPM@ MoDELS, 1237:41–60, 2014.

[Ma20] Manzoor, Hamza; Naik, Amit; Shaffer, Clifford A; North, Chris; Edwards, Stephen H: Auto-grading jupyter notebooks. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education. S. 1139–1144, 2020.

[PG07] Pérez, Fernando; Granger, Brian E: IPython: a system for interactive scientific computing. Computing in science & engineering, 9(3):21–29, 2007.

[PG15] Perez, Fernando; Granger, Brian E: Project Jupyter: Computational narratives as the engine of collaborative data science. Retrieved September, 11(207):108, 2015.

[Pr21] Prinz, Andreas: Teaching Language Engineering Using MPS. In: Domain-Specific Languages in Practice, S. 315–336. Springer, 2021.

[Re12] Reisig, Wolfgang: Petri Nets: An Introduction, Jgg. 4. Springer Science & Business Media, 2012.

[RG02] Richters, Mark; Gogolla, Martin: OCL: Syntax, semantics, and tools. In: Object Modeling with the OCL, S. 42–68. Springer, 2002.

[Ro18] Rodriguez-Echeverria, Roberto; Izquierdo, Javier Luis Cánovas; Wimmer, Manuel; Cabot, Jordi: Towards a language server protocol infrastructure for graphical modeling. In: 21th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems. 2018.

[RTS19] Rosenthal, Kristina; Ternes, Benjamin; Strecker, Stefan: Learning Conceptual Modeling: Structuring Overview, Research Themes and Paths for Future Research. In: 27th European Conference on Information Systems - Information Systems for a Sharing Society, ECIS 2019, Stockholm and Uppsala, Sweden, June 8-14. 2019.

[Ru21] Rumpe, Bernhard; Michael, Judith; Kautz, Oliver; Krebs, Roland; Gandenberger, Sabine; Standt, Janos; Weber, Uli: Digitalisierung der Gesetzgebung zur Steigerung der digitalen Souveränität des Staates, Jgg. 19 in Berichte des NEGZ. Nationales E-Government Kompetenzzentrum e. V., June 2021.

[SBS20] Schlichthaerle, Stefan; Becker, Klaus; Sperber, Sebastian: A Domain-Specific Language Based Architecture Modeling Approach for Safety Critical Automotive Software Systems. In: Software Engineering Workshops 2020. CEUR-WS.org, 2020.

[St15] Staubitz, Thomas; Klement, Hauke; Renz, Jan; Teusner, Ralf; Meinel, Christoph: Towards practical programming exercises and automated assessment in Massive Open Online Courses. In: 2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE). IEEE, S. 23–30, 2015.

[Sy13] Syriani, Eugene; Vangheluwe, Hans; Mannadiar, Raphael; Hansen, Conner; Van Mierlo, Simon; Ergin, Huseyin: AToMPM: A web-based modeling environment. In: Joint Proc. of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition. S. 21–25, 2013.

[Te19] Ternes, Benjamin; Strecker, Stefan; Rosenthal, Kristina; Barth, Hagen: A browser-based modeling tool for studying the learning of conceptual modeling based on a multi-modal data collection approach. In: Human Practice. Digital Ecologies. Our Future. 14. Internationale Tagung Wirtschaftsinformatik (WI 2019), February 24-27, 2019, Siegen, Germany. University of Siegen, Germany / AISeL, S. 1984–1988, 2019.

[TK16] Tolvanen, Juha-Pekka; Kelly, Steven: Model-driven development challenges and solutions: Experiences with domain-specific modelling in industry. In: 4th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD). IEEE, S. 711–719, 2016.

[Ul13] Ulrich, Frank: Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines. In: Domain engineering, S. 133–157. Springer, 2013.

[Va20] Van Binsbergen, L Thomas; Verano Merino, Mauricio; Jeanjean, Pierre; Van Der Storm, Tijs; Combemale, Benoit; Barais, Olivier: A principled approach to REPL interpreters. In: ACM SIGPLAN Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software. S. 84–100, 2020.

[VP12] Voelter, Markus; Pech, Vaclav: Language modularity with the MPS language workbench. In: 34th Int. Conf.e on Software Engineering (ICSE). IEEE, S. 1449–1450, 2012.

[Wi96] Wirth, Niklaus: Extended Backus-Naur Form (EBNF). Iso/Iec, 14977(2996):2–21, 1996.

[Wo21] Wolschke, Christian; Marksteiner, Stefan; Braun, Tobias; Wolf, Markus: An Agnostic Domain Specific Language for Implementing Attacks in an Automotive Use Case. In: 16th Int. Conf. on Availability, Reliability and Security (ARES 2021). ACM, 2021.