After having explained the context of our research , we summarize the contribution of this paper .

Finally, we provide an overview of the rest of this paper .

1.1 Context In model-driven software engineering , software systems are developed ¨ by creating high-level models which are analyzed, simulated, executed, or transformed into code.

In this context, models are structured artifacts which are instantiated from metamodels.

A metamodel defines the types of elements from which models are composed and the rules for their composition.

For metamodels, the Object Management Group has defined the MOF standard , a subset of which is implemented as Ecore in the Eclipse Modeling Framework .

Models may be represented in a variety of different ways, including diagrams, trees, tables, or humanreadable text.

Different kinds of editors may be employed to create and modify models.

In the case of a textual representation, a syntax-based editor may be used which persists the text and derives the underlying model by an incremental parsing process.

In the EMF ecosystem, the Xtext1 framework is frequently used to generate syntax-based editors from language descriptions.

In contrast, projectional editors provide for commands operating directly on the model and project the model onto a suitable representation .

A projectional editor may ensure syntactic correctness of models and enjoys further advantages concerning tool integration.

In particular, since models are stored as instances of metamodels, unique identifiers may be assigned to model elements such that they may be referenced in a reliable way.

Software product line engineering is a discipline which is concerned with the systematic development of families of software systems from reusable assets.

To this end, common and discriminating features of family members are captured in a variability model, e.g., a feature model .

In domain engineering, a variability model is developed along with a set of reusable assets.

In application engineering, product variants are developed from reusable assets.

Product variants may be constructed in different ways.

In case of positive variability, they are composed from reusable modules.

In case of transformational variability, product variants are constructed by applying a sequence of transformations.

In case of negative variability, multi-variant artifacts are represented as superimpositions of annotated elements.

An annotation constitutes a presence condition over features.

A product variant is defined by a feature configuration, stating which features have to be included and excluded, respectively.

To construct a single-variant artifact, all elements are removed from a multi-variant artifact whose annotations evaluate to false.

Model-driven software product line engineering combines MDSE with SPLE.

Thus, SPLE is applied to models.

While most SPLE approaches focus on source code rather than models, a number of MDSPLE tools have been developed,e.g., FeatureMapper , FAMILE , and SuperMod ¨ all of which are ¨ based on EMF.

Online education and specifically the purely online systems could significantly benefit from well designed and developed adaptive VLE systems.

Current education research lacks an integrated environment that will enable quick experimentation by multiple-domain users such as educators, data scientists, software developers.

Our proposed solution takes advantage of domain-related software engineering research and applies it in the area of adaptive VLEs.

The initial feedback by educators and data scientists was very positive.

Especially, the elements of isolating the domain from the implementation and the quick turn-around time for different experimentations were noted.

Especially, the language composability feature that is provided by MPS JetBrains is very powerful for future extensions of our framework towards a more advanced learning model, more automation, more application domains.

Therefore, our plans consist of enhancing the current framework with more usability testing, more education-related models, student centred views that will enable them to personalise their learning path etc.

Last but not least, we plan to apply it to other applications areas such as BT’s business analytics for customer churn, health applications with machine learning such as hospital readmission, health IoT applications, etc towards the development of a generic framework for adaptive systems.

We presented a generic framework for building projectional multi-variant editors which are based on feature models for defining variability and support negative variability by annotating domain model elements with feature expressions.

Human-readable textual notation is employed at the user interface.

In particular, the notation provides for a clear separation between domain model elements and annotations, and offers a variety of commands for flexible filtering of variability information .

Projectional multi-variant editors have not only been designed for model-driven engineering; they have also been realized with model-driven engineering.

Thus, projectional multi-variant editors constitute a complex use case for the application of modeldriven engineering.

As described in Section 4, we devised a notation for megamodeling which we ap plied to describe the internal architecture of projectional multi-variant editors.

Furthermore, Section 5 illustrates the complexity of the models employed internally by means of the mapping between domain models and feature models.

The work presented in this paper is still ongoing.

Future work will include support for defining partial or total feature configurations and configuring multi-variant domain models accordingly.

Here, ensuring well-formedness of configured domain models constitutes an important challenge which may be addressed along the lines of our previous work on FAMILE .

Please note that configuration of feature models and domain models is essential not only for application engineering but should be supported in domain engineering, as well.

In domain engineering, configuration support enables previews of configured domain models which may be visualized by coloring and eliding.

In addition, such previews will be editable.

In this way, the projectional multi-variant editor will support automated management of annotations in a similar way as variation control systems .

To experiment on whether a blended modelling environment could provide beneficial improvements to modelling activities and decrease modelling time, we implemented a prototype based on Papyrus and Xtext in the Eclipse environment.

There are alternative approaches to the development of a blended modelling environment for UML profiles, but any such tool must address some common issues such as the synchronisation of multiple representations by editors as well as their persistence, or the handling of cross-references between model elements.

For the purpose of synchronisation, some tools may choose to keep separate representations and use explicit transformations between them to keep them up-to-date.

An advantage of this approach is that representations could be persisted in the format preferred by the users or tools.

The disadvantage is that such transformations must be explicitly written and maintained, when the language evolves.

This entails the problem of keeping the transformations consistent with the language and with each other.

Furthermore, keeping sepa925 rate representations would be prone to error in a collaborative environment, as multiple copies of the same model could be modified separately in inconsistent ways.

A different approach is that of projectional editors which, as previously discussed, keep only one underlying representation of the model’s abstract syntax and different editors and viewers are responsible for the conversion between abstract and concrete syntax.

The obvious advantage here is that it avoids the problems described in the previous paragraph, as there is no need for transformations between representations.

A disadvantage is that it may not provide the same flexibility of storing multiple representations.

For the purpose of supporting UML-based profiles specifically in a blended environment, there are additional challenges, in particular, the support for multiple stereotype applications.

Our solution leverages Xtext’s own mechanisms which infer an appropriate Ecore meta-model upon which the generated environment operates.

To achieve the seamless synchronisation we override Xtext’s document provider behaviour, which maps domain elements to documents, and is used by editors to update the textual representation, the abstract syntax elements, notify listeners of changes, etc.

By overriding Xtext document provider, we are able to maintain only one common resource.

Our approach can be considered as semi-projectional, differing from existing projectional tools, such as MPS, in several ways: 1) it addresses support for DSMLs defined as UML profiles; 2) the textual representation is truly textual, as opposed to a form-based representation, and thus enables the use of textbased tools ; 3) it relies on existing mature frameworks for graphical modelling with UML and text-based IDEs .

In Section 4, we listed a set of five improvements to current practices brought by our blended modelling framework, which we discuss in the followings.

Grammar customisability Given a UML profile , Xtext grammars could be semi-automatically generated to define a textual concrete syntax for the profile .

Xtext provides an out-of-the-box features for generating grammars from UML profiles.

The generation process is not customisable nor parametric.

That is to say, generated grammars are in most cases unusable in practice without refactoring and manual tuning.

That was the case for the MARTE profile.

In our solution, we manually created and customised an Xtext grammar from a sub-set of MARTE.

Doing so, we were able to provide a customised and convenient solution to blended modelling for UML and MARTE.

Nevertheless, a blended modelling environment shall provide a specific feature for parametric and semi-automatic generation of Xtext grammars from profiles, so as to be customisable and refactorable to fit the stakeholder’s needs.

This would not jeopardise the serialisation mechanisms as long as it does not break the conformance of models to the UML profiles specification .

We are currently working on such a feature, which is expected to heavily simplify the job of a DSML developer.

Multiple stereotypes application in textual format The possibility to apply multiple stereotypes is crucial to provide a full-fledged modelling environment for UML and profiles.

Currently, since Xtext does not provide out-of-the-box features for generating grammars entailing concepts embodied in different profiles , this is not trivial to achieve.

In this work, we showed how we provided our framework with such a feature combining grammar rules representing stereotyped metaclass instances and runtime validation checks.

Stereotype applications correspond to boolean rule properties, literally indicating whether or not a given stereotype is applied.

A similar approach supports stereotype properties together with dedicated validation checks regulating their availability, i.e.

prevent editing the properties of non-applied stereotypes.

Each stereotype is managed independently, hence an indefinite number of stereotype applications can be handled on a single element.

Furthermore, users have the possibility to easily introduce additional constraints concerning one or more stereotypes in the form of validation checks, e.g.

HwProcessor instances should contain at least one packaged element stereotyped as HwCache.

When a stereotype is applicable to multiple UML base elements, there would be duplicates in the grammar .

Initially, we tried to avoid this by using Xtext’s fragment rules.

However, since fragments were not compatible with unordered sets in Xtext and we did not want to impose a specific order for stereotype tokens and stereotype properties, we decided to get rid of fragments and opt for better usability of the textual languages instead rather than a more condensed grammar.

Cross-profile modelling One of the main characteristics of our solution is that it does not entail complex profile-specific explicit synchronisation transformations between textual and graphical notations.

This makes most of the framework cross-profile.

The only transformations needed for propagating stereotype applications across the notations are mainly based on the grammar, hence generalisable.

The mechanism itself is cross-profile and profile-specific instances such as XMarte can be generated from its metamodel definition in a semi-automated manner, with the help of the blended DSML developer for more complex cases.

On-the-fly changes propagation Model changes done in one view are seamlessly reflected and visible in the other views .

On-the-fly propagation is achieved thanks to a single persistent resource shared among the views.

Although the propagation does not produce tangible delays in the rendering of the changed model across notations, the stakeholder may want to disable it for specific reasons .

This feature is currently not available in the framework, but we are working on it.

Cross-notation multi-view modelling We showed how an Xtext-based textual language , with related grammar and editor, representing only a sub-set of the HwLogical package of MARTE can seamlessly work on a UML resource containing other UML and MARTE concepts .

For instance, XMarte would be suitable for a platform modeller, who might not need or want to view functional details.

This is possible thanks to our enhanced Xtext resource management, which, instead of overwriting the in-memory model with plain text, propagates changes directly to the UML resource, the same used for editing and rendering UML models in the graphical and tree-based views by Papyrus.

As further enhancements of the multi-view nature of our framework, we plan to provide features for layered accessibility to shared information with multiple read/write access rights levels, enforcement of specific notations depending on the stakeholder roles, wizard-based customisability of perspectives, and inclusion of additional notations .

Expectations on software functionality and quality are increasing at a fast pace.

Additionally, the interconnected nature of software-intensive systems makes software grow exponentially in complexity.

The combination of high functional and extra-functional demands with ever-growing complexity leads to large increases in development time and costs.

To combat this threat, Model-Driven Engineering has been adopted in industry as a powerful means to effectively tame complexity of software, systems and their development, as shown by empirical research , by using domain-specific abstractions described in Domain Specific Modelling Languages .

DSMLs allow domain experts, who may or may not be software experts, to develop complex functions in a more domain-focused and humancentric way than if using traditional programming languages.

DSMLs formalise the communication language of engineers at the level of domain-specific concepts such as an engine and wheels for a car.

These concepts may not exist in another domain.

Moreover, DSMLs support more efficient integration of software with designs and implementations of other disciplines.

In this paper, we focus on DSMLs based on the Unified Modeling Language .

UML is the de-facto standard in industry and an ISO/IEC standard.

It is general-purpose, but it provides powerful profiling mechanisms to constrain and extend the language to achieve UML-based DSMLs .

Domain-specific modelling demands high level of customisation of MDE tools, typically involving combinations and extensions of DSMLs as well as customisations of the modelling tools for their respective development domains and contexts.

In addition, tools are expected to provide multiple modelling means, e.g.

textual and graphical, to satisfy the requirements set by development phases, different stakeholder roles, and application domains.

Nevertheless, domain-specific modelling tools traditionally focus on one specific editing notation .

This limits human communication, especially across stakeholders with varying roles and expertise.

Moreover, engineers may have different notation preferences; not supporting multiple notations negatively affects throughput of engineers.

Besides the limits to communication, choosing one particular kind of notation has the drawback of limiting the pool of available tools to develop and manipulate models that may be needed.

For example, choosing a graphical representation limits the usability of text manipulation tools such as text-based diff/merge, which is essential for team collaboration.

When tools provide support for both graphical and textual modelling, it is mostly done in a mutual exclusive manner.

Most off-the-shelf UML modelling tools, such as IBM Rational Software Architect or Sparx Systems Enterprise Architect , focus on graphical editing features and do not allow seamless graphical–textual editing.

This mutual exclusion suffices the needs of developing small scale applications with only very few stakeholder types.

For systems with heterogeneous components and entailing different domain specific aspects and different types of stakeholders, mutual exclusion is too restrictive and void many of the MDE benefits.

Therefore, modelling tools need to enable different stakeholders to work on overlapping parts of the models using different modelling notations .

Paper contribution.

In this paper we describe our work towards a full-fledged framework able to provide seamless blended graphical–textual modelling for UML profiles.

Differently from current practices, our framework is based on a lightweight form of blended modelling, where both graphical and textual editors operate on a common underlying model resource, rather than on separate persisting resources, thus heavily reducing the need for explicit synchronisation between the two.

To maximise the accessibility of our solutions, we leverage opensource platforms and technologies only.

We implemented a proof-of-concept framework, as well as designed and ran an experiment to assess potential benefits of blended multi-notation modelling as opposed to standard single-notation modelling.

Note that the area of so called action languages, such as for instance the UML actions or the VIDE1 action language, also for UML, has focused on how to integrate textual notations for description of algorithmic behaviours in graphical models.

In our work, we focus on a broader and more complex problem, namely the provision of a fully blended modelling environment for any portion of a UML-profile, being it structural or behavioural .

Projectional editing Projectional editing is another research area which investigates the use of various concrete syntaxes for editing models by displaying the different concrete syntaxes as projections.

JetBrains MPS and MelanEE apply this approach.

With projectional editing, the user edits the model through a syntax-specific view or editor, which itself updates the underlying abstract syntax model and these changes are automatically reflected in other views or editors for alternative concrete syntaxes.

The main advantage is that the model can be projected in various concrete syntaxes depending on what the user prefers.

However, it adds a considerable overhead for the DSML developer as the user actions have to be translated into change actions on the abstract syntax tree.

For parser-based textual DSMLs , a text editor in combination with a lexer/parser combination can be used.

There is a gap in AI research, specifically in providing AI in context and at the system-level.

Most of the research in the AI area has been focusing on algorithmic improvements and their performance.

However, AI will have to be adopted in several different application contexts from business processes to autonomous cars.

Domain modeling is a rising area in software and systems engineering, and in this paper, we presented its application for providing AI in context.

We developed a new domain-specific modeling language, using the JetBrains MPS development environment, to model a Classification Algorithms Framework ; this has as its main target to enable data scientists and other domain experts to perform quick, system-level and in-context experimentation of several algorithms.

Our future plans include applying CAF in several business-related application contexts within BT and other application areas.

In the longer term, and on more ambitious grounds, we will continue development of the language by ‘‘blurring’’ the limit between the algorithms and the domain, enabling optimal algorithm adoption in context.

We evaluated CAF according to the Quality characteristics defined in paper [23] and using the feedback we got from BT data scientists and MPS language developers .

Several tasks from basic to advanced were set as part of a CAF workshop.

The basic tasks involved using the language to choose and run a classification algorithm on a given data file.

The advanced tasks included extending the language with more classification algorithms.

The results from the two different perspectives were as follows: Data scientist perspective: The results from the data science perspective were as follows: Functional suitability: CAF scored very high according to its suitability for all the functionality required.

The domain was well specified, and it included processing of the data using different classification algorithms.

Usability: The feedback regarding the usability of the language was also quite good.

The one-page interface that presents the results as well as the output in one page was very appreciated by the data scientists.

It provided the option for quick experimentation of alternatives and in context.

However, the appearance of the one-page interface could be further improved by integrating web technologies.

Reliability: The language has precise semantics and the user is not ‘‘allowed’’ to make mistakes from the standard characteristics of the projectional editors.

This was the most attractive feature to the users although it required some initial extra time of getting used to the editor.

Productivity: The use of this language will result in increased productivity for the data-scientists as they will be able to implement a quick turn-around time and experiment with several classification algorithms.

Also, the system-level approach is expected to assist the adoption of the algorithms in several contexts.

Compatibility: The fact that java code is automatically generated from this language makes it very powerful for its deployment to application contexts.

Expressiveness: The language is very expressive as it has been developed through numerous meetings with BT’s engineers and therefore extensive domain analysis has taken place.

Language designer perspective: The results from the language designer perspective were as follows: Reusability: This language provides an excellent reusability element as it can be used as part of any system that requires data analysis.

Its use within the education context proves the plug-and-play strong element of the language.

Maintainability: The process of adding new algorithms is not easy and requires technical expertise to understand and maintain it.

Appropriate interfacing needs to be developed to enable easy extensibility of the language.

In Fig. 9 an Excel diagram depicts graphically the above results and shows that the average evaluation of the language is 8 out of 10 with strongest points the productivity, compatibility and expressiveness as is expected for domainspecific languages.

Reusability has been scored excellent as this language is anticipated to be at the core of many adaptive systems applications.

More evaluation and usage of the language inside BT’s extensive commercial application areas is planned.

I

INTRODUCTION It is increasingly recognized that intelligent and adaptive systems are the systems of the future [1]

The era of static software and systems development that were the same independent of the environment is coming to an end and all application areas would need to add some ‘‘intelligence’’ that will allow them to learn and adapt as they operate and collect data

Furthermore, data collection and processing, commonly called data analytics, are going to be part of all systems and software in the modern big data era

In order to add ‘‘intelligence’’ to systems the most commonly used algorithms are the classification algorithms

The current practice followed by the data scientists is to use libraries for classification algorithms such as scikit-learn, etc

that provide python implementation code for a wide range of algorithms

Most of the times, the data scientists are using directly the scikit-learn interfaces in order to run the algorithms and get the results

This is a detailed and not very user-friendly process

There is a need to abstract the use of classification algorithms at a system-level and provide an interface that would be more user-friendly and achievable for the non-data scientist user

This need has been recognized in environments such the weka library that provide a user-friendly web interface

However, in both the above approaches, an integrated environment where the classification algorithms would be considered at the system-level and in application context is missing

On the other hand, recent research in software and systems engineering focuses on domain modeling and providing tools for abstracting technical implementation details and leaving the domain expert-user to handle only domainrelated information

Domain-specific languages are a powerful tool for customized solutions that provide abstract and domain-specific interfaces

In this research, we used the JetBrains MPS DSL development environment in order to develop a Classification Algorithms Framework

This abstracted the detailed information normally required for using the classification algorithms by offering the user a one-page interface

Last but not least, the developed CAF can be reused in other application domains through the language modularization offered by MPS

This can be accomplished by developing a new domain language corresponding to the application, such as a language for interfacing with business managers, medical professionals, education professionals, etc

and then using CAF in order to perform data analysis

A background study on related work is presented in Section II

Specifically, we start with the definition of intelligent systems, existing research on frameworks for classification algorithms and research for system-level descriptions of intelligent systems

This article presented the current status of SLang development, with improvements done to the domain model, and a broader scope evaluation of SLang in a mocked practical scenario.

The effort to prototype and probe SLang as a model-driven approach to improve survey processes started with the goal of improving survey data collection systems through better communication between IT personnel and survey domain experts and through the improvement of software artifacts reuse, as well as elimination of unnecessary rework.

Questionnaire specification is the central point to achieving these goals.

The usage of a model-driven approach proved a good strategy to maintain questionnaire specification centralized and to help generate artifacts that improved systems integration.

Using the strategy of simulating a real survey scenario created an environment where stakeholders’ feedback was rich in ideas for improvements, showing that there is considerable potential in the adoption of a model-driven approach.

Further work has been planned to further investigate and evaluate some points, before adopting SLang at an industrial scale.

First, although SLang proved itself flexible enough for modeling complex questionnaires, usability evaluation is necessary to validate choices made in the concrete syntax and IDE approaches.

Second, the evaluation stressed the potential benefits of combining a questionnaire presentation language with SLang.

Third, survey IT infrastructure is diversified, and it is important to understand how far the usage of SLang can go.

Further testing with code generation and transformations to integrate survey models with existing systems, such as survey metadata repository, survey data distribution, and publication systems, and data collection management systems, are necessary to better understand and evaluate SLang’s full potential.

SLang was prototyped using the MPS language workbench.

This prototype provided the environment for experimenting and validating SLang as a model-driven approach, applied to the complex survey questionnaire domain.

The validation process included two stages: real-world questionnaire specification using SLang and the usage of questionnaire model transformations applied in a software called SInterviewer used by real stakeholders to perform mocked interviews.

In the first stage, two real-world questionnaire specifications were encoded in SLang, using this prototype implementation.

This part of the experiments validated the expressiveness of SLang.

Then, mock-up surveys were run, using the encoded questionnaires, on top of a survey environment, called SInterviewer.

This section describes some aspects of SLang prototype implementation, important aspects of SInterviewer, and the results of the prototype validation process.

The SLang prototype was built on top of JetBrains Meta Programming System .

Five workbenches were first considered for the implementation of SLang: Spoofax, XText, Rascal, MetaEdit + and MPS.

The decision to adopt MPS was influenced by the fact that only MPS had a projectional editor, that is, an editor that makes it possible to create, edit and interact with one or more ASTs, avoiding the need to use parser tools [8, 23].

Another aspect that influenced this decision was a broad Android-based data collection infrastructure used at IBGE and the fact that MPS provides Java compatibility, hence providing means for model transformations outputting Java code.

The process of prototyping SLang started with concepts mapping, which defined the AST creation rules and the SLang base structure.

Then, the concrete syntax was enforced using the MPS editors.

Finally, behavior and static semantics were added using the MPS Behavior, Constraint, and Type System aspects.

Behavior aspects made possible, for example, to attribute default values to questionnaire model properties, and to create and manipulate child nodes and references using MPS concept constructors and MPS concept methods.

Static semantics was established through MPS Constraint Aspects and Type System Aspects.

Constraint aspects provided, among other things, control of where concepts are allowed, validation for properties values, answer options control.

Type system aspects were used for semantic aspects that could not be modeled using MPS base concepts, behavior aspects, and constraint aspects.

For example, preventing nodes with the same name to exist in a specific scope could not be done using concept structure or constraint aspects.

Constraint and type systems aspects provided hooks used by MPS to implement context assistance and error reporting, in the final language IDE generated using MPS [2].

A survey is a systematic method for collecting data about entities to construct quantitative descriptors of the attributes of a larger population of which the entities are members.

The usage of a questionnaire is, by far, the most common data collection strategy [1].

Highly influenced by recent information technology advances, developing software to support a questionnaire-based survey seems to be an ordinary software engineering task.

After all, questionnaires are forms, for which a large number of different solutions and development strategies exist.

However, as survey scales in size or complexity, this ordinary task becomes daunting.

A questionnaire can comprise hundreds of variables intertwined in a complex web of data quality controls implemented to guarantee that each question is fully understood and adequately answered.

Hence, underestimating questionnaire design complexity is a common flaw that directly impacts survey quality [1].

In engineering, complexity is frequently handled by raising the level of abstraction.

In particular, model-driven software engineering aims at raising computer language abstraction further by making models first-class citizens in the software development process.

This article proposes the usage of a model-driven approach for designing complex questionnaires.

Specifically, it proposes a Domain Specific Language for modeling questionnaires, presents a prototype, and evaluates the use of the DSL as a strategy to reduce the gap between survey domain experts and software developers, improve reuse, eliminate redundancy and minimize rework.

In more detail, the article first describes a domain analysis that resulted in a model for the structure of questionnaires, including elements that allow the modeling of questionnaire data consistency and integrity rules, as well as the specification of behavioral aspects of the questionnaire required to capture navigation flow.

Next, based on the proposed model, the article describes the design of a prototype domain-specific language for modeling complex questionnaires, called SLang, and its implementation using the MPS projectional language workbench.

The SLang design process and the main decisions provide insights on how model-driven DSL approaches can be applied to real-world problems.

Finally, the article covers the evaluation of SLang, providing a good picture of the proposed solution in practice.

In this evaluation, real-world questionnaires modeled with SLang were deployed in a complete setting, including both the SInterviewer app and a backend responsible for centralizing collected data, with real end-users executing mock interviews.

SLang prototyping has been previously described in [2].

This article contains additional insights on the survey questionnaires domain and how the domain analysis was performed, additional information on the SLang design, with improved examples, and details of the language evaluation process, including an IT environment that is closer to reality.

The design of our editor will not perform better for simple edits in terms of number of keystrokes, but only for complex editing operations.

However, rather than looking at the number of keystrokes, the total time spent performing edits may be a more representative figure, as it allows to include the overhead of switching to the mouse to perform selection and navigation.

We plan to experiment with different designs to make the editor approachable and learnable, such that we can carry out controlled experiments to make stronger statements on the editor’s performance.

In this paper we described our work towards an open-source framework for blended graphical–textual UML modelling based on Eclipse, Papyrus and Xtext.

The framework aims at advancing the state of the practice of blended UML modelling by providing support for: textual grammar customisability, flexible application of multiple stereotypes on UML base elements using textual languages, cross-profile modelling based on blended notations, on-the-fly changes propagation across notations, and cross-notation multi-view modelling.

We ran two experiments to assess the potential impact of blended solutions on modelling effort, in terms of time.

Different notations are more suitable for different modelling tasks and, besides previous experience of the modeller with one or the other, enforcing the use of a task-optimal notation can decrease modelling time.

Overall, stakeholder’s free choice of notation does decrease modelling time and subjects used to a specific notation perform overall better using their preferred notation.

Nevertheless, for specific modelling tasks, enforcing a task-optimal notation has a better impact on modelling time.

To summarise, we can conclude that a balanced combination of freely chosen and fixed task-dependent notations may represent the optimal solution.

The goodness of such a combination depends on two factors: modelling tasks to be performed and stakeholder’s preferences.

For this reason, it is hard to identify a generic optimal combination.

In any case, we could observe that blended capabilities bring improvements in the modelling activities and decreases modelling time.

CONCLUSION

Linear text is the most widely embraced means for writing down programs.

But, we also know that in many contexts a picture is worth a thousand words.

Developers know this, which is why ASCII diagrams accompany many programs in comments and why type-set documentation comes with elaborate diagrams and graphics.

Developers and their support staff create these comments and documents because they accept the idea that code is a message to some future programmer who will have to understand and modify it.

If we wish to combine the productivity of text-oriented programming with the power of pictures, we must extend our textual programming languages with graphical syntax.

A fixed set of graphical syntaxes or static images do not suffice, however.

We must equip developers with the expressive power to create interactive graphics for the problems that they are working on and integrate these graphical pieces of program directly into the code.

Concisely put, turning comments into executable code is the only way to keep comments in sync with code.

When a developer invests energy into interactive GUI code, this effort must pay off.

Hence a developer should be able to exploit elements of the user-interface code in interactive-syntax extensions.

Conversely, any investment into GUI elements for an interactive-syntax extension must carry over to the actual user-interface code for a software system.

Finally, good developers build reusable abstractions.

In this spirit, an interactive-syntax extension mechanism must come with the power to abstract over interactive-syntax extensions with an interactive-syntax extension.

If this is available to developers, they may soon offer complete libraries of interactive-syntax building blocks.

Our paper presents the design, implementation, and evaluation of the first interactive-syntax extensions mechanism that mostly satisfies all of these criteria.

While the implementation is a prototype, it is robust enough to demonstrate the broad applicability of the idea with examples from algorithms, compilers, file systems, networking, as well as some narrow domains such as circuit simulation and game program development.

In terms of linguistics, the prototype can already accommodate interactive syntax for visual data objects, complex patterns, sophisticated templates, and meta forms.

We consider it a promising step towards a true synthesis of text and łmovingž pictures.

It has been recognized that the systems of the future in several application contexts will contain in some form or another big data gathering and analysis. A huge amount of data can be collected from Information Technology and the Internet of Things devices, creating the big data era.

However, appropriate analysis and processing of these data is a difficult and complicated task that requires specialized expertise.

Moreover, each application domain has its semantics and peculiarities in the interpretation of data making the problem even more difficult.

Online education and its personalization are no exception.

As an adaptive system application, it consists of three main stages: data collection, data processing and adaptation [1].

In this domain context, the whole adaptive system application is called in the literature learning analytics [2].

Data can be collected through several systems from Virtual Learning Environments to general IT systems with information for each student.

These data collected are pieces of learning evidences in our context.

However, the main IT systems utilized for online education are by far the VLEs.

Blended learning approaches are mainly used to combine online systems with face-to-face teaching in educational institutions.

Pure online education, on the other hand, relies solely on the VLE for the educator-student communication.

Data for learning evidence consist of information collected from these VLEs from student grades to the number of accesses to resources.

Utilizing these data to provide personalization and enhanced student experience is a promising area and especially for purely online education.

In this article, we have developed an integrated framework to mainly assist educators that develop purely online courses utilizing learning analytics in a systematic and ‘‘domain-specific’’ way using domain-specific languages [4].

Specifically, this framework consists of two main DSLs: AdaptiveVLE DSL and Classification Algorithms Framework DSL.

A range of data to be collected is given for configuration in the design of the AdaptiveVLE through the AdaptiveVLE DSL.

Then data clearing has to be processed manually.

Once the set of data to be used for the data processing is identified, the Classification Algorithms Framework DSL [5] is utilized and students are categorized according to their achievement levels.

According to this information, educators can alter the learning path for individual students to address their individual needs.

There are two types of additional resources to support weak and strong students and the AdaptiveVLE DSL is utilized to alter the learning path accordingly.

Also, in our proposed solution, additional external resources can be utilized to enrich the options available.

This is one of the most effective and simplest forms of adaptation that was chosen as less intrusive to the student and as conforming to differentiation approaches [6].

The life cycle of an application is bound to changes of domain and technical requirements.

Non functional requirements as scalability and availability may lead to a rewrite of the application as is for a new architecture or programming language.

DevOps and Microservices-based Applications appear to be an indivisible pair for organizations aiming at delivering applications and services at high velocity.

The philosophy may be introduced in the company with adequate training, but only if certain technological, organizational and cultural prerequisites are present [4–6].

If not, the prerequisites should be developed to guarantee adequate instruments to model and verify software systems and support developers all along the development process in order to deploy correct software.

Microservices allow developers to break up monolithic applications in a set of small and independent services where each of them represents a single business capability and can be delivered and updated autonomously without any impact on other services and on their releases.

In common practice, it is also expected that a single service can be developed and managed by a single team.

Microservices recently demonstrated to be an effective architectural paradigm to cope with scalability in a number of domains, however, the paradigm still misses a conceptual model able to support engineers starting from the early phases of development.

Several companies are evaluating pros and cons of a migrating to microservices.

Model-driven software development supports expressing domain requirements regarding contained data, function points, workflows, configurations, requirement tracking, test cases, etc.

by appropriate domain specific languages.

In this respect, this work discusses the provision of an model-driven approach for the automatic migration of monolithic applications to microservices.

In particular, domain-specific languages allow the definition and deployment of microservices, while model transformations are exploited to automatize the migration and the containerization phases.

The implementation of the migration framework is realised by means of JetBrains MPS .

MPS is a meta-programming framework that can be exploited as modelling languages workbench, it is text-based, and provides projectional editors.

The choice of MPS is due to the inherent characteristics of MSAs, which are by nature collections of small services that must interacts to satisfy the overall business goal.

In this respect, graphical languages do not scale with the complexity of the MSAs.

Moreover, MPS smoothly supports languages embedding, such that our definition of microservice mining, specification, and deployment phases for the migration that are easily implemented.

As a validation of our approach, we have migrated a simple Java application in the corresponding Jolie microservices deployed inside the Docker container.

In this paper we presented the experiences matured in the development of a Model-Driven approach for the migration of monolithic applications to microservice applications.

The proposed solution is based on the definition of two domain specific languages, one for the microservices specification and one for their deployment in a Docker container, and on a set of generators that make the migration approach automatic and with less manual intervention by developers.

The framework have been implemented by means of the MPS text-based language workbench and evaluated with an initial with the aim to demonstrate the feasibility of the approach, and calls for future research.

To make is scalable and usable in real contexts we are interested to test it using different industrial case studies to further investigate the soundness of the proposed methodology and eventually to extend the specification of the provided DSLs in MPS to make it more general.

The Eclipse Modeling Framework constitutes a popular ecosystem for model-driven development.

In the technological space of EMF, a wide variety of tools for modeldriven development have been implemented.

EMF has established itself as a de facto standard for data models upon which many technologies and frameworks are based, including server solutions, persistence frameworks, UI frameworks, and support for transformations.

Model editors which provide tool support for creating, modifying, analyzing, and displaying models, constitute key components of environments for model-driven development.

Probably the first EMFbased editor that has been provided is the tree editor belonging to the EMF core.

Since then, a number of frameworks for building model editors have been developed for different model representations.

For example, frameworks such as GMF2 and Sirius3 support the development of diagram editors while EMF Parsley4 focuses on visualizations as trees, forms, or tables.

While diagrams have been frequently used for representing models, human-readable textual syntax has become more and more popular recently.

The term “human-readable” excludes textual representations such as XML that have been designed for data exchange.

Rather, human-readable syntax for models resembles the textual syntax of programming languages.

The trend towards human-readable syntax may be exemplified by recent work on the Action Language for Foundational UML .

While the UML standard originally defined only the abstract syntax of models and their representation as diagrams, ALF provides a textual language for both structural and behavioral modeling of a subset of UML ) that features foundational execution semantics.

Textual editors may be divided roughly into two categories.

Syntax-based editors treat the text as the primary artifact that is stored persistently.

A command issued by the user results in the text being updated.

Subsequently, the changes are propagated to the model – i.e., to the abstract syntax tree that is represented by the plain text.

The model is maintained only transiently, during an editing session, and is used primarily for incremental syntactic and semantic analysis.

In the technological space of EMF, among several tools for building syntax-based editors, e.g., EMFText5 for textual languages that can be easily extended and integrated with other languages, the framework Xtext6 is the most prominent one.

The editors are generated from a grammar definition that refers to an underlying metamodel.

Grammar rules in Xtext are based on the Extended Backus-Naur Form .

Either the starting point is the grammar from which the metamodel is generated or the grammar is specified for an existing metamodel which has been built using any tool in the EMF context .

The generated editor artifacts can be augmented with Xtend7 classes describing the custom scoping rules, validation constraints, hovering information, highlighting, etc.

The editor plug-ins for the specific language may be used by arbitrary EMF-based projects.

Syntax-based editors are flexible since they allow the modeler to issue arbitrary text-based commands.

For the same reason, they are easy to learn .

On the other hand, they suffer from the following shortcomings:

There is a high risk of syntactic errors since the modeler may type arbitrary text.

This may be problematic for beginners who are not familiar with the respective modeling language.

Tool integration may be hampered by storing models as text files.

For example, models may be related by inter-model links, e.g., traceability links connecting models at different levels of abstraction or links between features and domain model elements in software product lines.

Since text files do not provide for reliable identifiers of model elements, inter-model links may be easily corrupted.

Projectional editors invert the syntax-based approach to model editing.

Rather than the text , the model is persisted.

In the context of product line engineering, a great importance is attached to projectional editors, e.g., the PEoPL approach combines different representations.

Commands issued by the modeler affect the model rather than the text.

After the model has been updated, the changes are propagated to the text which is updated accordingly in turn.

For experienced users, projectional editors may feel less natural and comfortable than syntax-based editors .

On the other hand, projectional editors solve the problems mentioned above:

A projectional editor guarantees syntactic correctness by offering only commands that perform correctness-preserving in-place model transformations.

For example, a command for inserting some syntactic unit is allowed only at locations where this unit is legal and ensures syntactic correctness of the inserted syntactic unit.

A projectional editor facilitates tool integration by providing reliable means for identifying model elements.

While line numbers in text files are subject to change, elements may be assigned universally unique identifiers that are immutable.

This paper fills a gap in the EMF tool landscape by providing a generic projectional editor for EMF models that is distinguished by the following key properties The projectional editor stores the abstract syntax of some model as an ordinary EMF model, enabling integration with any EMF-based tool for model transformations, code generation, etc.

The projectional editor is generic inasmuch as an EMF model may be instantiated from an arbitrary metamodel that provides universally unique identifiers for objects.

So far, the projectional editor supports textual representations.

However, its underlying design is extensible such that support for other representations may be added in the future.

Deviating from Figure 2, representations are persisted, as well – again as EMF models.

This approach allows to persist representation-specific information such as layout of text or diagrams .

The editor may be adapted to a specific modeling language by providing a declarative syntax definition which is used to map abstract to concrete syntax.

No programming is required to this end.

Projectional editors are not a new invention.

Rather, they were devised several decades ago as components of integrated programming environments; see for some early approaches.

In this context, they were called syntaxdirected editors.

Currently, the Meta Programming System 8 by JetBrains constitutes a contemporary framework for developing projectional editors – not just for text but also for other representations such as two-dimensional math notations, tables, or forms.

MPS also provides support for language modularization as well as composition .

In , experiences with teaching MPS in industry are outlined.

Recent research deals with support for incremental model transformations .

While this framework is powerful, it comes with a proprietary data model.

Instead of an open ecosystem, MPS provides a closed language workbench that requires its users to commit to the MPS data model and tool set.

For defining languages, MPS uses hierarchies of concepts and their implementations: While in the world of EMF, the abstract syntax of a language may be defined by metamodels using arbitrary editors , in MPS one concept is defined textually 8https://www.jetbrains.com/mps/ for each type in a separate file – similar to defining Java classes.

Furthermore, instead of specifying the concrete syntax similar to a grammar, each concept provides an additional text file to describe the notation of the respective element.

For defining custom scoping, validation, building a type system, etc.

also a special textual notation is used instead of providing artifacts in a common general-purpose language.

In this paper we presented our framework for generic projectional editors for arbitrary EMF models.

In contrast to syntax-based editors which are derived from the grammar of a modeling language and which persist the representation, e.g., text, instead of model files, our approach allows for persisting models including the preservation of inter-model references.

The main benefit of persisting models instead of their representations, e.g., plain text, is a much easier integration in the existing EMF modeling ecosystem as existing tools and technologies for processing the obtained models can be used out of the box.

In its current state, the framework allows developing customized textual editors for textual languages with minimal effort.

While the context-free syntax of the language is specified by means of an intuitive declarative language, static semantics can be customized by implementing high-level code stubs.

The feasibility of our approach has been demonstrated by a projectional editor for the textual modeling language ALF.

Future work comprises the integration of other model representations including diagrams and tables and support for user-specific validation, code highlighting, hovering, etc.

Concepts for language imports and compositions will be taken into consideration, as well.

Furthermore, we plan to remove dependencies to the Xtext and Xtend implementations.

Our editor uses nested blocks to denote inclusion structures within the tree, see Figure 1.

Colors become brighter to show deeper nesting and change to a new base color for each Smalltalk block.

By laying out the blocks similar to text, we have similar space efficiency to textual languages, however as the nesting increases our rows get higher, such that inclusions remains clear.

Similar to the Vim editor, we use a command mode for traversing the tree and issuing commands, and allow entering a separate input mode for typing names and literals.

A selection mode allows users to mark complex selections.

The commands are designed to be reusable: depending on the selected node the same command may perform a semantically similar operations, such as inserting a statement in a block, or adding an element to an array.

Commands mirror aspects of their functionality if modified via shift, reducing the number of distinct shortcuts to remember.

To allow for a natural editing flow, the input mode is contextaware: each node will recognize inputs that are not valid for its current context, but can be translated to a simple tree transformation.

For example, typing “+” while inside a number will wrap the number in a binary expression, and typing another number while inside the binary message’s selector will move the new number to the second operand.

Working in projectional editors can provide an editing experience that is as fast or faster than working in text editors [1].

In contrast to text editors, the editing metaphors of projectional editors typically require robust design of a large set of editing operations to reach the expressiveness of a text editor [10].

We distinguish between text-oriented and tree-oriented projectional editors in this paper.

A text-oriented editor uses similar layout and syntactical elements as a textual language, but constrains editing to maintain a valid abstract syntax tree .

While providing a familiar way of reading code, editing operations may as such not be possible in the way users would expect from the interface, unless special care is taken [9].

A tree-oriented projectional editor makes the structure of the AST apparent through user interface elements.

While often perceived as inefficient, the editing metaphors may appear clearer to users when compared to a text-oriented projectional editor, as boundaries between elements are immediately apparent.

We hypothesize that supporting users’ understanding of the AST through presentation in a tree-oriented projectional editor can increase editing efficiency, as opposed to making projectional editors feel more like text editors.

To test this hypothesis we present the design of a tree-oriented projectional Editor for Smalltalk [3].

The editor is available on GitHub

In this paper we have conducted a critical review of common approaches for producing graphical views from models, and then introduced a novel approach for producing transient graphical views using lazy model-to-text transformation.

We have also presented the open-source Picto tool, which implements the proposed approach, and evaluated the efficiency benefits it delivers compared to batch model-to-text transformation.

The proposed method has been shown to have a low upfront cost, to scale up linearly and to deliver substantial efficiency benefits when a modest number of views is accessed by users between changes in the underlying models ś which is often the case in practice.

In terms of visualisation capabilities, being based on M2T transformation and browser-based rendering, Picto can reuse any JavaScript-based visualisation library and can also be extended through dedicated Eclipse extension points with support for additional 3rd party tools beyond Graphviz and PlantUML.

Having said that, it is worth reiterating that Picto is not a replacement for graphical model editing frameworks such as Sirius but instead targets use-cases where read-only views are desirable/sufficient.

Future work on Picto includes view-based model differencing, and developing a bespoke rule-based language for view generation which will provide first-class support for core Picto concepts as opposed to piggy-backing on EGL’s parameters block .

4 EVALUATION

To measure the benefits of the lazy view generation strategy implemented by Picto, we have carried out performance evaluation experiments where we compared view generation times of Picto to those of a batch M2T transformation that produces identical views17.

The following sections describe the experiments and discuss the obtained results.

4.1 Comparison Method

We start by describing the visualisation scenarios, the compared approaches, and the measuring platforms and methods used during the comparison.

4.1.1 Visualisation scenarios.

Two scenarios were used during this evaluation.

The first one involved Ecore metamodels and generating views such as the one depicted in Figure 9.

More specifically, one view was generated for each EClass in the input metamodel .

The view contains the EClass itself, as well as the EClasses it refers to through EReferences, and its supertypes.

We used the BigQuery Github dataset18 to search for very large publicly available metamodels, from which we included the following four in the comparison: the UML2 metamodel; the Common Information Model 19, which is a standard for the definition of electrical networks; a metamodel used internally by the eMoflon solution of the 2017 Transformation Tool Contest 20; a reverse-engineered metamodel of the Sirius codebase21 that has been used by the developers of the EcoreTools diagramming tool to carry out performance tests.

The details of the selected metamodels are shown in Table 1.

For instance, RevEngSirius.ecore is the largest of these metamodels, with a size of ∼4.7 MiB, and around 5.2K EClasses.

The second scenario involves generating views from synthetic models conforming to a contrived component/connector metamodel.

In this metamodel, each component has input and output ports, and can contain other nested components, which are interconnected between them and with the available ports to represent a modular system.

For this visualisation, a view, like the 4.1.1 Visualisation scenarios.

Two scenarios were used during this evaluation.

The first one involved Ecore metamodels and generating views such as the one depicted in Figure 9.

More specifically, one view was generated for each EClass in the input metamodel .

The view contains the EClass itself, as well as the EClasses it refers to through EReferences, and its supertypes.

We used the BigQuery Github dataset18 to search for very large publicly available metamodels, from which we included the following four in the comparison: • the UML2 metamodel; • the Common Information Model 19, which is a standard for the definition of electrical networks; • a metamodel used internally by the eMoflon solution of the 2017 Transformation Tool Contest 20; • a reverse-engineered metamodel of the Sirius codebase21 that has been used by the developers of the EcoreTools diagramming tool to carry out performance tests.

The details of the selected metamodels are shown in Table 1.

For instance, RevEngSirius.ecore is the largest of these metamodels, with a size of ∼4.7 MiB, and around 5.2K EClasses.

The second scenario involves generating views from synthetic models conforming to a contrived component/connector metamodel.

In this metamodel, each component has input and output ports, and can contain other nested components, which are interconnected between them and with the available ports to represent a modular system.

For this visualisation, a view, like the 4.1.2 Compared approaches.

We measured the time it took to generate the views for the scenarios described above both using Picto and with standalone batch M2T transformations.

For the batch transformations, we used the same language as in Picto, this is, EGL .

This ensures that what we are measuring is the impact of the lazy generation strategy we devised for this work, as opposed to more fundamental differences in the performance of two M2T transformation languages.

Also, using EGL facilitated creating identical M2T transformations as those in Picto, with only minimal changes to make them work in batch/standalone mode.

In the two visualisation scenarios, the M2T transformations generate DOT graphs that are then translated to SVG/HTML for inbrowser rendering through the Graphviz program.

While Picto has facilities to do that transparently for the user , we need to provide the same in the batch M2T approaches.

Therefore, after the M2T batch transformation concludes, a post-processing step is carried out to, starting from DOT, generate the SVG and HTML files that would be rendered in a browser.

The time to perform this post-processing step is included in the results of the batch transformations.

One of the advantages of using a batch transformation instead of Picto is the possibility of parallelising the generation of views in different system cores/threads.

Therefore, we created two variants of the batch transformation approach: the first one uses sequential execution, while the second one employs multithreaded computation via a parallel EGL execution engine22 for the M2T transformation and the Java 8 Streams API for the postprocessing phase.

Summarising, three approaches were compared: Picto, a singlethreaded and a multi-threaded batch M2T transformation.

4.1.3 Measuring platforms.

The experiments were carried out on a desktop computer running Ubuntu on a 6-core, 12-thread AMD Ryzen 1600 CPU with 32GiB of ram and a PCIe NVM SSD.

As this powerful hardware might not be typical of a developer workstation yet, we also ran the transformations in a lower-spec laptop featuring the same Ubuntu system and a 2-core, 4-thread Intel Core i5 7200U CPU, 16GiB of RAM, and again a PCIe NVM SSD.

4.1.4 Measuring method.

For the batch M2T transformations, we measured the time it took to run the transformations against the target models.

On the other hand, Picto’s lazy computation strategy required some instrumentation for performing the measurements.

We included relevant code in a fork of Picto’s implementation that forces the generation of each individual view just as if a user has selected it from the user interface, and gathers these measurements in a results file.

For both types of approaches, generation times were measured 10 times, and then the results were averaged.

To ensure that average figures were not disproportionately affected by outliers, we also calculated the standard deviation of these times.

The coefficient of variation, this is, the ratio of the standard deviation to the mean, was not higher than 0.005 for the single-thread batch transformation, 0.127 for the multi-thread one, and 0.164 for the individual Picto views, which indicates a low spread in the obtained results.

The higher dispersion of the Picto times can be due to their measurement inside an Eclipse instance, as opposed to the batch transformations’ execution that happened through a standard Java process.

Also, to prevent any inconsistencies due to low CPU states during the initial measurements, we warmed up the measuring platforms by executing initial generations whose obtained times were discarded.

Being able to produce graphical views of software and system models is often desirable in model-driven engineering settings.

Such views are useful, for example, to explore reverse-engineered models [3, 12] from different viewpoints of interest, to visualise relationships between heterogeneous models, and to facilitate presentation of text-based models to a wider audience of stakeholders [13].

Common approaches for producing graphical representations of models include implementing a graphical editor using a framework such as Sirius, GMF, or Graphiti; implementing a bespoke graphical viewer using frameworks such as Zest, GEF or JavaFX; and using batch model-to-text transformation to generate textual artefacts , which can be subsequently rendered in a web browser.

In this paper, we discuss scenarios in which the model visualisation methods above are applicable, as well as their main strengths and weaknesses.

We then present a novel Eclipse-based framework, called Picto, for producing transient views from models conforming to different metamodels and modelling technologies, by lazily transforming them into textual formats such as Graphviz, PlantUML, SVG and HTML, which are subsequently rendered in an embedded browser.

We demonstrate the building blocks and capabilities of Picto through a running example and we showcase how it can be used to produce non-trivial views from heterogeneous models.

We also evaluate the performance and scalability benefits of the lazy transformation approach employed by Picto, compared to visualisation via batch model-to-text transformation.

4.2.1 Ecore metamodel visualisation.

Figure 14 shows the measured generation times of the three approaches for the four selected Ecore metamodels.

Our experiment simulates a scenario in which a user is accessing the generated views one by one, i.e., selecting the generated view for each EClass in the input metamodel, until all produced views have been accessed.

The y-axis represents the accumulated generation time of the accessed views, while the xaxis indicates the number of views that have been accessed up to that point.

The number of accessed views is irrelevant for the batch transformation approaches, as all views are generated upfront.

Because of that, batch approaches are represented by horizontal lines indicating the time they took to generate all views of each model, with the single-thread variant in dotted red, and the multi-thread one in dashed green.

As expected, the multi-threaded variant took less time to complete, providing savings of 82.7 to 84.7% compared to the single-threaded execution.

On the other hand, the number of accessed views is very relevant for Picto, whose execution time is represented with a solid blue line.

As the number of accessed views increases, so does Picto’s accumulated execution time .

For Picto, the y-axis value at the ł0 accessed viewsž point depicts the time it took to complete the upfront view tree computation phase .

This time is almost negligible, as it only amounts to 22.3, 39.2, 62 and 302 milliseconds for the UML, CIM, eMoflonTTC17 and RevEngSirius metamodels, respectively23.

Lastly, to improve presentation, the time it took Picto to generate each individual view has been averaged.

Showing the real time would have made relevant the order in which the views are accessed, i.e., if those views that took more time to get computed are accessed earlier, then the Picto accumulated time would increase quicker at first, and vice versa.

In any event, the generation times were fairly uniform across all EClass views of the metamodels, so this averaging only has a minor aesthetic impact.

Of particular interest in the graphs of Figure 14 are the crossing points at which the Picto time meets with the batch transformation times.

When that crossing happens, it means that the accumulated time it took Picto to generate the accessed views at that point has reached the time that took the crossed batch transformation to generate all views of the model.

So, the greater the number of accessed views required to reach those crossing points, the more substantial benefit the lazy generation of views is providing.

In contrast, if the number of accessed views increases past the crossing point with certain batch transformation, then the final generation time of Picto would be greater.

The first crossing point involves Picto and the multi-thread batch transformation times.

This crossing happens at 40, 100, 177, and 758 accessed views .

These numbers show that, when considering the generation of all views, parallelising this generation contributes to a great reduction of the computation times.

The second crossing takes place when Picto’s accumulated time reaches the execution time of the single-thread batch transformation.

This crossing happens in all experiments when almost all the views have been generated.

Precisely, it takes place when 235, 588, 1088, and 4972 views have been generated.

The extra time that Picto requires to generate the remaining views is an overhead of its lazy M2T functionality, which is avoided when generating all views at once in the batch transformation.

However, the measured overhead is very small, oscillating between 2 and 4% of the total generation time for Picto when compared with single-threaded batch figures.

4.2.2 Component model visualisation.

With respect to the component model visualisation scenario, Figure 15 includes the results for the two biggest models we generated.

The first model, gencomps-12.9K, is 9.3 MiB in size, contains around 12.9K components , and its visualisation included generating a total of 1221 views.

As for the bigger gencomps-29K model, its numbers go up to 23.8 MiB in size, 29K component elements, and 6888 views.

With respect to crossing points, Picto and the multi-thread batch execution crossed after accessing 158 and 883 views for the gencomps-12.9K and gencomps-29K models, while the crossing with the singlethread execution happens at 1081 and 6337 views .

If we compare the results of both visualisation scenarios, we can see that the obtained times for the component models are consistent with those shown for Ecore metamodels.

There is an increase in the total generation time in the case of the components scenario that we attribute to the larger size of these models, which translated into bigger view computing times.

For instance, the eMoflonTTC17 metamodel and the gencomps-12.9K model visualisations contain a similar number of views, with 1090 and 1221 views, respectively.

Nevertheless, the size of these models is 3.3 and 9.3 MiB which, summed to the difference of 121 total views between the visualisations, causes a noticeable difference in the single-thread and Picto total times .

For the multi-threaded batch execution, though, the times remained fairly similar for both models .

These times suggest that the parallel execution of Ecore metamodels was not able to deplete all the computing resources offered by the 6-core/12-thread computer CPU, so there were some resources available to cope with the generation of 121 extra component views in a very close total time.

A similar comparison can be carried out between the RevEngSirius metamodel and the gencomps-29K component model.

4.2.3 Laptop Platform results.

Lastly, Figure 16 shows the obtained times for the Ecore experiments when the generations were executed on a laptop.

In that case, the obtained times were very similar for the Picto and single-thread executions to those of the desktop machine.

On the other hand, the benefits of the multi-threaded version were not as significant, because the parallel execution in a 2-core/4-thread CPU could not provide the same performance as a more capable 6-core/12-thread CPU.

In that case, and for the Ecore experiments depicted in Figure 16, the crossing between the multi-thread execution and Picto happened at 102, 253, 458 and 2192 accessed views , which indicates that, for lower-spec platforms, the use of Picto is even more beneficial.

Implementing a software language—be it a modeling language or a programming language—requires specifying its syntax, as well as static and dynamic semantics.

While conceptually none of these present a particular challenge, developers often struggle to bring their languages to life.

Reasons for obstacles range from the lack of knowledge on parser implementation to absence of tool support for the language—most commonly, users expect an Integrated Development Environment to be shipped together with a language, which is vital for adoption of a language [10].

Recently, an array of tools designed to define software languages together with their IDEs have appeared under the name of language workbenches [3, 4, 10].

These tools allow specifying syntax, typing rules, and code generators for a language, and they output a tailored IDE with standard services such as syntax-aware editor, code completion, automatic code corrections, and others.

Despite this fact, many of current language workbenches have a very steep learning curve even for experienced software professionals [8], and their adoption rate still leaves much to be desired.

At the same time, a noticeable amount of domain modeling is done using “office software”, such as word processors [9] and spreadsheet calculators [1].

Our hypothesis is that meta-definitions—that are prevalent in language workbenches—is a key factor that complicates language engineering for beginner language engineers.

Consider the following definition of construct variable declaration: var x = 10.

This definition looks like an instance of a variable declaration, and no meta-definition along the lines of VarDecl → “var” ident “=” expr is explicitly mentioned.

We suggest an approach where a language is defined by giving examples of code written in it, which are then annotated to specify different concerns of language definition—such as abstract syntax, typing rules, validation rules, formatting rules, and dynamic semantics.

We consider three possible implementation strategies of this approach: within an existing rich text processor , within a notebook , and within a dedicated example-driven language workbench.

Gamification is increasingly gaining popularity as a tool to promote engagement in target human activities.

In this respect, gameful applications development is facing growing complexity that requires adequate design and deployment support.

In this chapter, we presented GDF, a framework for the design and deployment of gameful applications.

In particular, GDF is made up of domain-specific languages allowing for stepwise refinement of application definitions, from higher levels of abstraction to implementation code to be run on a gamification engine.

GDF has been engineered by using MPS due to three main reasons: the need to provide text-based DSLs, the availability of language extension mechanisms conveying consistency management between abstraction layers, and the provision of generators to automatically derive implementation code.

GDF has also been validated against multiple case studies in diverse domains, notably education, smart mobility, and training in modelling.

In this respect, MPS demonstrated powerful capabilities but also a steep learning curve that could be unacceptable for non-software engineers.

In this respect, one of the main future research directions we are pursuing is the integration of simplified user interfaces, e.g., dashboards, to alleviate the complexity of game definitions for GDF users.

The GDF is composed of a set of languages, each of which defined to cover the artifacts constituting the gamification stack.

A graphical representation of this stack is shown in Fig. 7 [6]: it is composed of a set of layers that will be referred to as game modeling layers, namely, GML, GaML, and GiML.

They represent incremental refinements/specializations of gamification concepts, from higher to lower levels of abstraction, respectively.

The remaining layers, i.e., GsML and GadML, are called utility layers and can be defined on top of any of the game modeling ones.

GDF conveys a gamification design process that reflects widely adopted practices in the state of the art and practice of the field [2, 25] .

Taking inspiration from this process, GDF provides different modeling languages for specifying the main game components, i.e., game elements, and how they interact to build up a gameful application, that is, mechanics.

Such components are progressively refined to reach implementation code for a target gamification engine that copes with game instances execution.

For this purpose, we selected a specific gamification engine [26] based on DROOLS rule engine.6 It is an opensource component and exposes its main functionalities as services that are used by GDF.

Notably, services include supporting the definition and deployment of games, accessing information about the game and player state, and supporting the configuration of notifications for communicating game results to the players.

The Gamification Modeling Language is used to represent the set of core elements essential to describe a gameful system .

Figure 8 shows an excerpt of the main GML concepts: a Game concept is composed of a set of properties that characterize a specific gameful solution, and a set of children concepts that allow to specify the main game elements, that is, the fundamental ingredients of a gamified application.

GML conforms to the MPS base language and provides the basic gamification building blocks.

Other languages are derived as lower abstraction levels.

In this respect, a game designer should extend/refine GML concepts every time there is a need to introduce new game elements or mechanics.

The Game Model Language extends GML with concepts used to define concrete game descriptions.

As shown in Fig. 9, through GaML, the designer can specify how the game components are assembled to create an application into a GameDefinition.

Notably, the concept of Point in GML is specialized in skillPoint and experiencePoint, to distinguish between points gained by means of specific activity goals and points gained due to the progression through the game, respectively.

Moreover, dataDrivenAction and evenDrivenAction are exploited to recognize activities based on data or on events .

In a similar manner, the Challenge concept coming from GML is refined through, e.g., PlayerChallenge and TeamChallenge, to distinguish between challenges intended to be completed individually and the ones to be accomplished as groups of players, respectively.

GaML is generic enough to enable the reuse of the defined gamification concepts into multiple development scenarios .

A game instance is a GameDefinition, as prescribed in GaML, opportunely instantiated to be run by the gamification engine.

In general, an instantiation consists of the specification of the players/teams involved in the game; hence, one or more instances of multiple games may run concurrently by means of the same engine.

The Game Instance Model Language binds game definitions coming from GaML with instantiation details, as depicted in Fig. 10.

In particular, the universityInstance defines teams and players that play in a certain instance of a game.

GiML also supports for single-player challenges.

singlePlayerChallenges demand to players the fulfillment of a specific goal, whose attainment requires a prolonged individual commitment, typically within a limited period.

In order to confer a feeling of progress and mastery, GDF supports the definition of Levels.

Levels are always defined in association with a specific Point Concept.

For this in GiML, we have identified two types of levels: skillLevels and experiencelevels.

The first are related to skillPoints, while the second to the experiencePoints point concepts.

GiML instantiates also the badgeCollections.

They are used in order to further reward, through a collectible visual representation, the results of a player in terms of specific achievements.

Apart from game modeling languages, GDF provides so-called utility languages.

One of them is the Game Simulation Language , which allows to simulate game scenarios.

In particular, a GameSimulation is composed of a GameDefinition and a set of SingleGameExecution elements, as depicted in Fig. 11.

In turn, each game execution is made up of a Team and/or a Player that can execute an actionInstance or a challengeInstance .

In this way, the designer can specify specific game situations and check what state changes are triggered.

In this respect, it is important to mention that the target gamification engine7 provides the necessary features to track the gamification rules triggered during the execution together with the corresponding state changes.

Another utility feature provided by GDF is adaptation.

This feature leverages specific capabilities of the target gamification engine and, in particular, a recommendation system for generating players’ tailored challenges based on game historical data and current status, a mechanism to “inject” new game contents on the fly.

With this premise, GadML allows to model those scenarios when a new game content has to be assigned to a specific player on the fly.

In particular, the GameAdaptation concept includes gameId and playerId parameters for a game adaptation, plus a set of children to specify the new challenge to be injected.

As Fig. 13 shows, a newChallenge refines a simple game adaptation by defining a ChallengeModel, ChallengeData , and ChallengeDate .

Playing is an activity humans do since their birth for learning, to meet others, to be part of communities, to relax, and so forth.

Indeed, we are so used to it that we spend a growing amount of our free time with some form of gaming even in adulthood, and people made a profession out of it [1].

Interestingly, psychologists also observed that by introducing gaming elements and mechanics into “normal”— non-gaming—tasks, it can be possible to promote engagement and even motivate people to achieve certain objectives.

As a matter of fact, an increasing number of activities include gamification elements, very often supported by software applications: Internet banking, sport/activity trackers, and shopping/traveling fidelity cards are all examples of application domains targeted by gamification [2].

A fundamental concern of gameful applications1 is their tailoring to the target domain and users: if a game is detached from the domain interests, the risk is to promote counterproductive/undesired behaviors; similarly, too easy or too complex games could fail engagement objectives due to loss of interest or discouragement, respectively [3].

A direct consequence of the mentioned tailoring needs is the critical contribution and cooperation of application domain and gamification experts: the former ones provide inputs about the engagement issues and desired outcomes, while the latter ones propose corresponding gamification strategies.

Such a cooperation conveys gameful application specifications to be implemented in an appropriate target platform.

In the current state of practice, one available implementation option is to pick up a prepackaged gamification application from a repository [4].

The advantage would be to have a quick development phase limited to configuration purposes, at the price of very limited customization possibilities, unless manually tuning the existing implementation.

Diametrically opposite, a completely new gamified application can be developed from scratch: this solution necessarily entails longer time to market, with the advantage of realizing a fully customized implementation.

Regardless of the choice, the realization and deployment phases introduce an abstraction gap between gamification stakeholders, namely, domain and gamification experts, and the gameful application itself.

In fact, the target application is typically implemented as a collection of rules matching incoming event notifications with corresponding game status updates.

Therefore, developers need to translate game mechanics and other elements into corresponding rules, while the other stakeholders are required to backtrack state changes into corresponding gaming events.

With the growing adoption of gamification in disparate application domains and its spread to a wider range of users, the complexity of gameful software is unavoidably increasing.

In this respect, the abstraction gap between design and realization becomes a critical issue: the implementation phase is more tedious and error-prone, due to the number of rules and the customization needs.

Moreover, maintenance and evolution activities are harder to manage, due to the disconnection between design and realization.

In order to close the gap between design and implementation of gameful applications, we proposed the Gamification Design Framework [5, 6].

GDF is a collection of domain-specific languages devoted to the specification, implementation, and deployment of gameful applications.

The framework has been developed by the following three key principles: Separation-of-concerns : a gamification approach can be described by means of several perspectives.

When the complexity grows, an effective way to alleviate it is to manage different perspectives as separate points of view that are later on fused into a complete solution; Correctness-by-construction : given the growth of gamification employment and range of its potential users, the specification of gameful applications becomes increasingly intricate.

In this respect, game rules shall be consistent with mechanisms and elements intended for the target application; Automation : in order to close the gap between design and implementation, the amount of manually written code shall be reduced as much as possible.

Or in the other way around, the degree of automation provided by the framework shall be maximized.

GDF actualizes the mentioned key principles by means of three DSLs that correspond to three abstraction layers any gamified application can be viewed through: the topmost layer defines general mechanics and elements a solution could include, e.g., the concept of point, bonus, challenge, etc.; the second layer instantiates a subset of the abstract concepts defined on the level above due to the specification of the gameful application under development, for example, number of steps, walker of the week, hundred thousand steps week, respectively; the third and bottom layer describes the implementation of the concepts above together with their deployment on a gamification engine.

Here, configuration parameters can be set, like thresholds to gain points, bonuses, and awards, the timing of challenges activation, and the assignment of players and teams to the defined tasks.

Moreover, the layers convey generators enabling the automated derivation of implementation code for the gamified application.

GDF is practically realized by means of Jetbrains MPS and is the result of a challenging language engineering process.

In particular, the DSLs included in GDF required a language workbench enabling meta-modeling, semantics specification through generators, and multi-view-based modeling support to ensure the consistency between the different points of view.

During the language engineering process, we soon faced the problem of modeling a system of constraints , which tended to be intractable by adopting diagrammatic approaches.

Moreover, we needed a mechanism enabling the introduction and refinement of high-level gamification solutions and concepts without requiring domain experts to modify the language specification, e.g., by adding new consistency checks.

As a consequence, the DSLs included in GDF convey a text-based concrete syntax that eases the definition of game rules.

Moreover, they exploit the language extension mechanisms provided by MPS to define the interconnections between the different abstraction layers, which implicitly ensures consistency through inheritance relationships.

As mentioned in Sect. 2.3, the language engineering process devoted to the creation of GDF constituted an endeavor due to diverse challenges, summarized as follows and discussed in detail after: – scalability of modeling techniques, and in particular of diagrammatic forms, in the case of systems of logical constraints; – multiple levels of concepts definitions and corresponding instantiations; – usability of available multilevel modeling workbenches.

The first problem can be regarded as general and not due to the specific application domain and the language workbench taken into account.

Indeed, using diagrammatic representations for complex systems of logical constraints tends to become quickly intractable: modeling sequences of events together with their preconditions and constraints, possibly also including negative statements, makes the size of the specifications to exponentially grow with the number of handled variables/events.

We already gained large experience in modeling smart mobility applications and incurred in the same scalability problem we faced for gamification [29].

As a consequence, after some preliminary attempts and the necessary investigation of the state of the art, we decided to opt for a text-based modeling solution.

Admittedly, at this point, we could have chosen a different language workbench, notably XText,10 for trying to implement GDF.

However, the language import feature embedded in MPS caught our attention, and we decided to give it a try, without a thorough comparison with possible text-based alternatives.

The second among the listed challenges can be reduced to a separation-ofconcerns need [30]: in order to cope with the growing complexity of gameful applications, it is desired to partition their design and development into smaller, typically simpler, sub-problems.

More precisely, sub-problems would be the definition of gaming elements and mechanics in general; the specification of selected elements and mechanics for a certain game; and the deployment of game elements and mechanics of a specific gamification engine, including the configuration of players and teams.

It is worth noting that in order to make this separation effective, consistency support shall also be provided, such that, e.g., it would not be possible to use a game element when not permitted by the mechanics used in the game itself.

In this context, the language import feature provided by MPS reveals its criticality: since MPS is based on projectional editing, it is simply not possible to produce ill-formed models.

As a direct consequence, since, e.g., models in originate from the definitions in , a mechanism or element defined in can only be used in as conforming to the language definition.

In the same way, a deployment specification in can be only done as conforming to the definition of the game provided in .

In the MDE literature, separation-of-concerns and consistency management have been widely investigated, and in fact, there exist solutions like multilevel modeling and megamodeling that would suit for GDF purposes.

Nonetheless, while from a theoretical perspective those approaches are valid, the same cannot be said about availability of tools.

As a matter of fact, in our experience, MPS has been the first workbench delivering language engineering support with a tractable level of complexity, even for persons not necessarily MDE experts.

Here it is important to remind that GDF is intended to be extensible to new game elements, mechanics, etc., and to be used by all gamification stakeholders.

Therefore, the usability of the language workbench is of critical importance.

As mentioned before, the language extension mechanisms provided by MPS allow to import existing languages and extend them by operating limited refinements, both for new concepts, new constraints, and so forth.

At the same time, consistency management is implicitly built by means of inheritance relationships.

As expected, also MPS requires a training period, and the learning curve might be steep.

In our experiences, people with programming backgrounds might have some advantage in the learning phase, while MDE experts might find some of the features as unexpected and/or counterintuitive.

Indeed, MPS exploits a precise set of tools that need to be carefully understood in order to be able to use the workbench effectively.

In other words, while MPS demonstrates to be fairly simple to use for entry levels, when users require advanced features, the workbench exposes a rather complex and extensive set of details that need to be taken care of.

In this respect, the vast availability of tutorials and examples falls short, because a more conceptual description about how different portions of the workbench are kept together would be required.

Moreover, the need for a language debugger increases with the size of the project; in fact, in most cases, a language engineer understands that a specification is wrong due to some unexpected behavior of the resulting artefacts, while the reason of the malfunctions remains completely hidden.

For the current version of GDF, we expect gamification stakeholders to optionally define new game elements and mechanics, while the typical use would be to instantiate existing mechanics and elements in corresponding game definitions.

In turn, game definitions would be exploited to define gameful application deployments on a selected gamification engine.

Moreover, GDF provides languages to define game monitoring and adaptations at runtime.

While in most cases it is enough to define game elements and state change rules, the specifications might grow in their complexity and require fine-tuning for which stakeholders would require MPS experts’ help.

In this respect, we are planning future refinements/extensions of GDF to make the specification of gameful applications simpler.

Notably, we would abstract from the MPS workbench itself by means of, e.g., a web interface for game definitions; moreover, we would like to investigate easier ways of specifying detailed game behaviors, especially to relieve stakeholders of the burden due to hand-tuning the generators embedded in GDF.

MPS features “projectional” editing, that is, developers are not editing simple text while providing inputs in MPS; on the contrary, their editing is bound to the abstract syntax tree inferred by the language definition.

In other words, the DSL concepts defined through MPS and used as inputs “activate” specific branches of the AST, and consequently, the editing proceeds by following the available alternatives as per language definition.

In this way, the code is always represented as an AST, conforming to the language by construction.

Programs in MPS are represented as instances of concepts, called nodes [27].

In this respect, it is possible to define how the different concepts of a language are visualized to the end user, and each projectional editor provides a representation of the AST with which the user interacts.

For each set of concepts in GDF, we have defined editors by means of specific projection rules used to define the desired concrete syntax.

These editors can be used by the game designer to define the gamification elements, mechanics, and dynamics.

At the same time, we added extra editors to define specific game simulations and adaptations.

Figure 14 shows the editor definition for the GameDeclaration concept.

At the top level, it consists of a collection cell [− ... −] which aligns a sequence of additional cells.

The sequence starts with the game name and its related id.

It continues with the sequence of core elements as point concepts, actions, and rules.

All these concepts are the key elements to define the points that each player can receive, the actions that lead to the accumulation of points, and the rules that define the overall game behavior.

Actions of a game can be defined at design time by a game designer using specific editors as the one illustrated in Fig. 15.

An action of a game, as the point, is defined by a name and by a set or properties that characterize them.

For example, an action can represent the interaction that a player does with the application together with information about the instance when it is executed, while a point represents a counter that is updated with a certain value every time a specific action is executed.

As mentioned before, to execute the game actions done by the player, GDF leverages an open-source Gamification Engine component.

This component embeds DROOLS, a state-of-the-art rule engine technology based on reactive computing models [28].

For this reason, GDF expresses the rules of a game using the DroolsRule concept that regulates the game behavior.

A game designer can use the editor depicted in Fig. 16 to specify when a certain rule is triggered and how the different points are accumulated .

Language Wheel is apparently the first workbench that focuses on purely example-driven definition of languages.

We find it particularly important to guide a language engineer by using language flavors, which can be thought of as schemata for language definition process.

The current state of implementation of Language Wheel is in the very early phase.

Many of the details of how various concerns of language definition can be defined using annotations remain to be elaborated.

Foremost, this applies to typing rules and scopes of visibility.

Identifying commonly used concepts that will constitute the standard library of Language Wheel is an interesting task in itself.

Many of those annotation perspectives will require some notion of a generic definition to be instantiated when specifying a particular language; for example, quickfixes can be considered as generic transformations

Engineering a software language, be it a programming language or a modeling language, requires versatile knowledge and software development skills from a language developer, as they have to define its syntax, static and dynamic semantics, as well as implement a tailored Integrated Development Environment , which is vital for successful adoption of the language [43].

Over a decade ago, tools to define software languages together with their IDEs have appeared under the name of language workbenches [13, 15]; many of them still have a very steep learning curve even for experienced software professionals [34].

Our hypothesis [4] is that one of the key factors that inherently makes language engineering intricate is the prevalence of meta-definitions, with language definition tools themselves being on the meta-meta-level.

We describe in this paper our vision of a web-based tool— Language Wheel—aimed at beginner language engineers, and list possible requirements for such a tool.

A language is defined by giving examples of code written in it using illustrative syntax definition.

These examples are then annotated to specify different concerns of language definition—abstract syntax, typing rules, validation rules, formatting rules, and dynamic semantics.

Such a definition mechanism can serve as a front-end language workbench, whose output is a language definition in another language workbench [13], such as, for example, Eclipse Xtext [5, 14], Monticore [21], Spoofax [23], or JetBrains MPS [8].

An alternative possibility is to output a tailored editor—be it textual or projectional [18, 46])—and a code generator for a language, or, depending on the language, a CRUD application.

Yet another possibility is to output a language server protocol instance [10] for the language defined in Language Wheel.

Projectional editing.

Similar to the “semi-projectional” editing as explained in Section 3, hybrid editors augment text-based programs with additional information [20].

MacGnome environment [29] had a special editing mode that allowed converting sections of code into plain text to perform editing; after that the sections were converted back to a structural representation.

In Greenfoot [6], a program is represented as frames, which are created using text- and mouse-based operations [26]; expressions can be entered in a textual mode and it is possible to convert them on the fly into structured expressions [6].

Barista [26] supports structure views that enable representing structural items in code in a visual way instead of textual.

Graphite [32] allows incorporating custom highlyspecialized interactive code generation interfaces directly into textual editors .

Many of these ideas can be reused in the implementation of the “semi-projectional” editor of Language Wheel.

In order to tackle the complexity of safety critical systems, increase trust and enable agile development we need semantically rich and deeply integrated models about different aspects of the system from requirements, design and safety engineering.

Currently, the industry is using ad-hoc and loosely coupled tool chains, featuring informal models, and they cannot cope with today’s challenges.

In this chapter we presented FASTEN, an extensible platform based on JetBrains MPS, developed and used by researchers from three organizations over the last three years, both from industry and academia.

We presented a set of requirements DSLs that enable the transition from informal requirements to formal models, DSLs for the formal specification of architecture and system-level behavior, and DSLs for safety engineering and assurance.

We successfully used the DSLs to model two industrial systems from the automotive domain.

MPS is a key enabling technology for the FASTEN system.

On tools building side, MPS empowered us to efficiently build extensible stacks of DSLs; to integrate independently developed modular DSLs; to provide most appropriate notations; to equip language constructs with rich and extensible sets of semantic rules; to implement advanced editing support for creating models; and, last but not least, to integrate external analyses tools and present the analyses results at the abstraction level of the DSL.

On research and technology transfer side, MPS enabled us to experiment with adequate abstractions, to prototype new ideas in closed-loop interactions with domain experts or fellow researchers, as well as to cooperate beyond the borders of a single company.

FASTEN is an open-source and open-innovation platform for research and technology transfer in the field of safety critical systems.

Future Work.

FASTEN can be easily extended with new DSLs in order to experiment with higher-level modeling abstractions.

FASTEN is still under development both at the platform level as well as regarding the higher-level modeling languages.

We are extending and fine-tuning the DSLs based on the feedback from domain experts such as requirements, system or safety engineers.

We plan future work along three directions: enabling new modeling abstractions; integration of existing abstractions to enable higher-level workflows, and better integration of tooling.

Examples for directions for modeling extensions are boilerplate patterns for the specification of timing/reliability aspects of requirements, integration of failure models and formal models for robustness analysis of the design, or enriching the semantics of safety argument structures to enable more automated checks.

Furthermore, we plan to integrate model-based fault injection approaches similar to [4].

Regarding directions for modeling integrations, we are looking at the integration of the modeling languages developed by Bosch with more functionality from FASTEN.Safe, especially the assurance case in GSN; or the integration of system models with the software or hardware aspects.

Finally, examples of better integration of tooling in modeling workflows is to further improve the lifting of analysis results, to enhance the existing interaction with analysis engines and to integrate new engines.

We started with the FASTEN project by focusing solely on extensions of SMV in October 2017 and then we gradually extended to other verification engines.

Bosch started their use in late 2018, and researchers from fortiss actively contributed to FASTEN since late 2019.

We estimate the total language development effort to roughly 3 person years, split mainly across the three organizations.

In addition to the language development, considerable effort was invested to come up with the present set of abstractions by continuously learning from our interactions with practitioners and the research community.

FASTEN is still actively developed and subject to significant extensions.

In the following, we present a discussion and summarize the lessons learned from developing and using FASTEN in our research and technology transfer projects.

7.1 Discussion.

Maturity.

FASTEN is a research tool.

Different functionalities are at various Technology Readiness Levels spanning from TRL4 to TRL6 .

FASTEN is currently not intended to be used in production.

Our main audience are tool builders, tech leads, technology scouts and people responsible for processes, methods and tools who are looking for ways to improve on the status quo and address new challenges caused by the complexity increase of the products.

Domain experts have the chance to test hands-on how different aspects of their systems could be modeled, tools builders get inspired about new tooling functions and how they can be integrated, and technology scouts get to know how DSLs can help increase the automation degree of the safety critical systems development.

Extensibility.

The FASTEN approach relies on development and integration of modeling languages, in a bottom-up fashion by using stacks of DSLs.

The multitude of DSLs presented in this chapter show that it is feasible to create domain specific environments on top of FASTEN.

As shown in Section 6, the FASTEN framework enables rapid prototyping for the integration of domain-specific languages with formal methods.

Its modularity and extensibility allows to quickly connect other DSLs such as SysML.

We could easily experiment with different language dialects for representing architecture and contracts.

Tool-driven Research Transfer.

The set of functionalities provided by FASTEN exceeds the state-of-practice technologies used in the industry: today’s practice is dominated by loosely coupled tool chains.

Each aspect from FASTEN is covered by one or more tools, each providing informal specification and modeling means .

The loosely coupling of tools lead to information loss at the boundaries and introduce accidental complexity; their informal content is preventing automation.

We have extensively used FASTEN for interacting with systems and safety tech leads to demonstrate advanced concepts related to model-driven engineering, modelbased safety engineering and formal methods.

FASTEN allowed us to create demos and verify the usefulness of different modeling and specification approaches for concrete industrial problems.

Being able to play with models and analyze them directly in the tool has shown to be extremely useful when presenting to respective business units and other stakeholders.

7.2 MPS Features Supporting Our Work

MPS is a key enabler for the development of FASTEN as we discuss in the following.

Language Development Productivity.

MPS enables highly efficient definition, extension and refactoring of languages.

The time taken between an idea and its implementation as DSLs and subsequent creation of user-level models is very short , which allows us to perform many iterations over a short timespan.

This, in turn, enabled us to evolve the languages based on practitioner feedback and to experiment with new modeling concepts or their combination.

In the end this stimulates co-creation of tooling hand-in-hand with domain experts or fellow researchers.

Support for Modular, Extensible and Stackable DSLs.

MPS’ mechanisms for language modularization proved to be essential for our project, because they allow independent creation of DSLs by different organizations.

For each of the integrated tools we have implemented its input language as an MPS language.

The extensions are "grown" as stacks of DSLs in a modular fashion with higher-level abstractions, similar to what mbeddr does for the C base language [37].

DSLs addressing different aspects of safety critical systems’ development are integrated with each other and seamless workflows beyond the boundaries of single disciplines are enabled.

Notation Freedom.

In FASTEN we heavily use combinations of notations: textual, diagrammatic, tree or tabular.

MPS allows easy definition of editors and provides multiple notations for the same language concepts, drastically improving usability.

We learned that domain-specific or even application-specific notations are key for tool adoption by domain experts, e.g., safety engineers [25].

Sometimes this might even require replicating the look-and-feel of established tools in shape and color to ease adoption.

Editor Automation and Auto-completion.

MPS provides several means for increasing the automation of model authoring – context-sensitive auto-completion being the most important.

This reduces to some extent the effort of learning new syntax of the verification tools – when auto-completion is used, many gotchas can be avoided.

Syntax-driven Editing and In-editor Errors.

The projectional editor of MPS guides the users to create meaningful models and prevents them up-front to make mistakes.

With MPS it is easily possible to define extensible sets of context sensitive constraints and display errors in the editor when they are violated.

Users get immediate feedback about errors and thereby many inconsistencies can be fixed right away, allowing domain experts to focus on essential things.

Model Annotations.

We have used nodes attributes to annotate design models with information of variables values .

This proves to be very useful when users need to debug their models.

7.3 Open Challenges with MPS-based Tooling Projectional Editing.

While MPS’ projectional editing allows maintaining different, domain- and/or stakeholder-specific notations, the projectional editor comes with its own challenges.

When used with the expectation of a classical text editor or graphical editor, the resulting editing experience might lead to a lot of frustration when editors are not designed with great care and significant effort.

Specialized DSLs like the "grammar cells"[38] ease the creation of editors and partially increase their usability by offering a behavior closer to the textual editors.

Despite this, the users still need to be aware that they are not working with a classical textual editor.

IDE Errors Hard to Understand.

There is a quite steep learning curve for nonprogrammers to get accustomed with MPS.

Many errors of the IDE seem cryptical to domain experts.

Several of these errors are not even meant to be seen by nonprogrammers and thereby they easily get confused.

We consider these situations to be bugs in MPS.

IDE Footprint.

The sheer size and resource-consumption of the IDE also tends to hinder adoption.

It is hard to argue why a >500 MB IDE is the right choice for working with small domain-specific artifacts that may sometimes look like simple text snippets.

Developing a lightweight MPS-based IDEs is still an open challenge.

Recently, there has been highly promising work done to deploy MPS on a server and access its models via web browsers in the modelix6 project.

We plan to leverage on this in order to make our DSLs more accessible by occasional users and thereby make experimentation by domain experts easier.

Deployment.

While MPS does provide support for deployment of languages as plugins or standalone IDE, this support is still fragile and requires patching jars for advanced customizations.

Furthermore, integration into CI pipelines requires handling of a significant technology stack – automation and maintenance of the builds remains challenging.

It is common knowledge that software-intensive systems in general are becoming larger, more complex and more relevant to crucial tasks in our society.

Due to the high impact of a malfunctioning, we must have a high degree of confidence that the systems cannot harm people or expensive equipment [14].

Testing is Limited.

Testing is a well-known approach to building trust.

Systems are “tried out”, unit tests and integration tests are written and automatically executed for the software parts, hardware-in-the-loop tests verify aspects of the hardware and red teams try to attack the system to uncover vectors for malicious attacks.

However, usually testing can only show the presence of bugs, and not prove their absence.

Phrased differently, testing suffers from the coverage problem, which means that you can only be sure that your system is “correct” if you test it completely.

“Completely” is a high bar that is often not reachable in practice for complex systems.

Formal Methods.

Formal methods can be an important ingredient in an engineer’s toolset to build trust in critical systems.

Depending on the particular formalism, formal methods can either help with systematically improving the coverage of tests or can even proof the absence of certain classes of errors such as runtime errors or conformance of a client’s implementation with a API.

Some formal verification tools work directly on source code, however, most require a model expressed in a particular language on which to operate.

While this can be seen as a disadvantage , it has the important advantage that models cannot just represent software – they can also represent aspects of the system implemented in hardware, or even aspects of the environment.

Models – for example of interfaces, protocols or state-based behavior – can also be defined in earlier stages of development where hardware or source code are still elusive.

This way, engineers can experiment with various design alternatives early in the development, building trust in their work early, and avoid expensive rework during later stages of development.

Bringing Formal Methods Closer to Practitioners.

However, formal methods are hard to use by practitioners for several reasons.

First, some of the formalisms are conceptually hard to understand [21]; they often encode non-trivial mathematical ideas that are not familiar to engineers [31].

The input languages of verification tools contain low-level abstractions that are targeted towards verification, which forces engineers to bridge a large abstraction gap when they encode system-level concepts.

Second, these formalisms are by necessity general – they are not specific to the engineer’s domain, which makes the transformation of the engineering model to the tool’s input and the lifting of the results even harder.

Third, there is often no robust tool support for verification engines; and common software IDE services such as auto-completion, refactorings or debugging the models is non-existent.

Fourth, using real-world verification requires the use of multiple formalisms for the definitions of state-transitions or constraints, requiring multiple encodings of the engineering model and/or fusion of the results.

Fifth, the interpretation of the results of the verification tools, such as understanding the witnesses for verification failures in terms of the engineering model, is often not trivial either.

In safety-critical contexts, formal verification results may be used as evidence supporting assurance arguments that demonstrate that the system meets critical goals.

Finally, not everything that is needed to make an argumentation for the system’s safety can be formalized [34, 13].

In these cases, unstructured or semi-structured artifacts must be integrated with formalized models, both conceptually and technically.

Our Vision.

We envision an integrated modeling and verification platform, that deeply integrates models for requirements, design, verification and assurance at increasing levels of formality as illustrated in Figure 1.

Our platform has the following characteristics: The user interacts with a limited number of models whose structure and notation is meaningful to the user’s engineering domain.

Informal parts of the system such as textual requirements, or safety arguments can be incrementally formalized and combined with other formal specifications.

The languages used to define these models allow the user to express properties that they want to verify; again, these properties are expressed with a language that is close to the user’s domain.

These models, together with the properties they must satisfy, are then automatically translated into one or more verification formalisms, and existing verification engines are executed to verify the properties.

The low-level verification results are lifted back to the level of the engineering model; potentially, the results from multiple verification tools are semantically integrated.

Using references and other model-level mechanisms, the formal models can be connected to informal or semi-formal content, integrating system and safety engineering models in a semantically rich assurance case to ensure consistency between design and safety models.

Last but not least, the tool should be built as an open platform to make it extensible with new formalisms or user-facing languages, and its user experience should be on par with modern IDEs in terms of editor features, type checking and error reporting.

How to get there.

We rely on language workbenches [12], tools that support the efficient implementation of languages, type systems, model transformations, and IDEs.

We use a layered approach that delivers early benefits even while only a part of our overall vision is implemented.

As foundational language workbench we chose JetBrains MPS1 due to its powerful support for language engineering .

We start with the implementation of several input languages of verification tools in our language workbench – e.g.

we implement the language SMV, the input language of the NuSMV [7] model checker; or Promela, the input language of Spin [15] model checker.

This step does not give us improvements in terms of semantic abstraction, but it results in a robust IDE for writing models in the notation of the formalism that has the usual modern front-end features such as syntax coloring, code completion, type checking and reporting of the verification results.

Based on JetBrains MPS’ support for modular language extension, we incrementally add discrete extensions to these low-level input languages to make idiomatic use simpler.

These extensions are still generic, but useful for less mathematically-minded users.

Next, we implement an integrating language based on the component-instanceconnector paradigm; such languages are well known to many engineers, provide good support for hierarchical breakdown of systems and are reasonably generic .

Furthermore, we develop extensions to the component language that allow the user to annotate properties relevant for verification.

A chain of model-to-model transformations convert this model, including the properties to verify, into the input language of one of the integrated verification tools.

After the verification is run, we provide lifting of verification results back to the users such that they can easily understand what went wrong and perform fixes.

This integrated language for modeling and verification is the first major goal of our vision.

To enable a transition from textual requirements to formal models, we have developed a set of DSLs for specifying requirements by using increasingly semantically rich models.

The requirements models range from plain natural language text to requirements templates or formal models written e.g. using temporal-logics.

The richer the models the more rigorous verification is possible.

The results of verification can be further used in safety assurance arguments that we integrate via another set of DSLs specialized for safety engineering.

Are we there yet? FASTEN2 is an open-source3 platform that enables experimentation with modeling abstractions amenable for verification on the way to our vision.

It supports exploration of the idea, in a bottom-up manner by combining informal and increasingly formal models, and verifies the degree to which it is realistic.

FASTEN is built on JetBrains MPS, which has been used successfully in a safety-critical context [35].

While FASTEN is not a production-ready tool, it has been used to verify realistic systems in industrial settings.

To validate the extensibility with regards to verification formalisms, we present various extensions shipped with FASTEN itself.

In addition to these open-source extensions, we also discuss closed-source extension, developed independently at Bosch.

To validate the extensibility for a particular domain, we demonstrate a more extensive case study developed at the Corporate Research department of Bosch.

Contributions.

The main contribution of this chapter is FASTEN, an open-source platform based on JetBrains MPS, for safety critical systems development.

FASTEN allows experimentation with adequate and domain specific modeling abstractions to capture different aspects of safety critical systems from requirements, design, verification to safety assurance.

The parts focused on safety assurance is referred as FASTEN.Safe.

For each of these aspects, we provide DSLs that enable the transition from informal to formal descriptions.

We present a novel architecture for building model driven engineering tooling around modular and extensible stacks of DSLs that leverage on the language engineering capabilities of the Jetbrains MPS language workbench.

FASTEN has been built over the last three years by industrial researchers from three companies and a research institute .

Last but not least, this chapter presents our experiences and lessons learnt with developing and using FASTEN in research transfer projects inside our organizations.

To enable an efficient implementation of our vision, we have modularized recurring functionality in a set of DSLs and libraries that make up the FASTEN Platform.

Figure 2 shows an overview of the FASTEN architecture.

FASTEN integrates external analysis engines as black boxes – NuSMV [7], Spin [15] and Prism [22] are integrated as external binaries, the Z3 [10] is integrated via its Java API.

Foundational Libraries.

All languages and functionalities are developed on top of JetBrains MPS plus languages and libraries provided by the MPS-extensions4 and mbeddr-platform5 projects.

We use these language libraries for diagrammatic, tree and tabular notations, for the improving editor usability and for generated code review.

From an implementation perspective this infrastructure proved to be crucial to our approach.

Our Infrastructure: the FASTEN Platform.

In order to facilitate the development of DSLs and the integration of formal analysis tools, FASTEN comes with a set of basic languages and functionalities that can be grouped as follows: 1.

1. Infrastructure: commonly used DSLs such as an expressions language; support for calling external tools, displaying analysis results in the IDE,

2. Basic modeling: base DSLs for the definition of modeling languages such as architecture, message-sequence-charts or tabular specifications,

3. Input languages of existing analyses tools: the implementation of input languages of the integrated analysis tools.

4. Standard languages: the GSN [18] modeling language for creating assurance cases, the STAMP [23] modeling language for performing hazard analysis and a base language for the specification of textual requirements.

On top of the FASTEN-Platform, we have created DSL extensions to enable more comfortable modeling and verification.

In the following sections we present selected examples of higher-level abstractions and functionalities that have been built so far.

We presented the central notions of the projectional editor, Gentleman.

The editor offers a rich GUI enabled by standard layouts and fields, and projections that can be applied to any part of the DSL concepts.

Ultimately, our goal is to integrate Gentleman in full-fledged language workbenches to offer a more adapted user experience to domain users.

Gentleman is an ongoing project with a roadmap filled with more novelties.

We would like to integrate a model explorer, investigate proper undo/redo functionalities for projectional editors, and enable collaborative modeling.

We are also working on providing a standard API based on the language server protocol to ease the integration with language workbenches.

Gentleman is designed to be lightweight and thus uses a minimalist approach to avoid any extraneous content that would otherwise distract and slow-down the user.

The base editor only comes with a single toolbar with a button to close the editor and a status bar.

Additional buttons can be added in the configuration of the editor.

We distinguish three usage scenarios of Gentleman: definition, edition, and reading.

In , the user defines the concepts of a model and the projections as presented in Section 2.

In this scenario, the target user is a language engineer or GUI designer involved in the DSL definition.

For greater flexibility and reuse, he defines projections separately from concepts, thus providing good separation of concern.

This is especially the case when the concepts are defined in an Ecore metamodel.

In , the user creates or edits an instance of the DSL.

The editor presents editable fields to add values to the AST of the model.

Figure 3 shows a snapshot in this scenario.

The purpose of is to simplify the projection for users to read the model rather than edit it.

It is a special case of where fields are made read-only and interactive actions are disabled.

For example, widgets to add, remove components, and empty optional attributes are not displayed.

The target users of and are the domain experts of the DSL.

To give users more flexibility during the editing activity, they can spawn as many editors as needed.

This way, they can edit different concepts each in a dedicated space or use different projections simultaneously.

The editors are juxtaposed next to each other and can be positioned as the user wants.

Users can attach a note to any part of the content of the editor.

It is not stored as part of the AST of the projection but as part of the editor.

Notes can be tagged in the form of anchors.

Users can search for tags to quickly navigate to specific locations of the model.

One use of notes is to add comments on a model.

Import and export allow the user to preserve the current state of the model and editor or load a saved one.

The storage medium is a JSON object representing the AST of the model and the projection configuration of each concept.

It also stores configurations specific to the editor, like the toolbar configuration and comments.

Recall that Gentleman is bootstrapped; therefore, it treats any instance being edited like a model.

Internally, Gentleman does not distinguish between a model definition , a projection definition , or an instance.

The export stores a reference to each concept and each projection.

It is also possible to save the model in plain text with no formatting or projection by using the print functionnality.

When the language engineer has defined a model and at least one projection for each concept, he can automatically synthesize a projectional editor for his DSL.

One attractive feature of Gentleman is its ability to preview a projection during the editing process.

This allows the designer to view the presentation of the projection associated with the concept and how it integrates with other projections.

It is also possible to edit the previewed projection to see how the design responds to different values entered and improve the user experience.

When the user interacts with a specific projection, a context in the status bar indicates the name and location of the current concept in terms of the structure of the model.

At the top of Figure 2, we see that the currently active field corresponds to the name of a concret concept.

The user can navigate through the model using the TAB key or mouse click.

As a projectional editor, he can only modify editable projections.

For example, in Figure 2, the user cannot remove the central topic; he can only set its name value or add/remove markers.

During the interaction with a field, the user may request the accepted values that can be assigned to the field by hitting the common CTRL+Space key combination.

The response depends on the state of the field and its concept: it pops a dialog showing information or a list of choices if an action is required.

In Mindmaps, the marker attached to a main topic must have been defined as a central topic component.

Therefore, at the level of the main topic, the context assistance lists the marker values defined at the central topic level.

Requesting context assistance for the name of the main topic displays the attribute’s meta-information, including constraints if they were set.

After the user edits a field, an orange, red, or green badge is displayed next to it.

They indicate that a value was modified since it was last in focus, that a constraint of the concept is violated, or that it is valid, respectively.

In Figure 3 a green badge is displayed over the text field to indicate that the value entered is valid.

When a constraint is not satisfied , an additional description is fed back to the user, in the form of a dialog if it is purely informational or a choice dialog if further actions can be taken.

As explained in Section 1, the model is always structurally valid.

However, the DSL may have semantical constraints, such as a maximum depth of sub-sub topics.

Gentleman supports reporting semantical constraints violations through its API.

Gentleman targets the web as its running platform and is implemented entirely in Javascript.

The application runs client-side, enabling offline work.

As with any web application, HTML and CSS are used to describe the content and its presentation.

The tool can be easily integrated into any web page with the Gentleman script loaded in it.

This can be achieved in one of two ways.

The developer can decorate an HTML Tag with the attribute data-gentleman, such as <div data-gentleman></div>.

Upon loading, every HTML element on the page found with this attribute will have a Gentleman instance attached to it with the editor rendered inside.

Alternatively, the developer can create a Gentleman instance dynamically in Javascript using the instruction editor = Environment.createEditor followed by editor.render to render the editor on the page.

This enables a web-based language workbench to have multiple projectional editors within a single modeling editor.

This is useful, for instance, to control and stylize the edition of attributes.

Gentleman is an open-source project available on GitHub.

The practice of model-driven engineering relies heavily on the use of models and domain-specific languages [23] which offer, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [21].

Over the years, many tools have been created to support this activity, giving birth to a new category of tools labeled as language workbench [7].

They support the efficient definition, reuse, and composition of languages and their IDEs [6].

However, the current state of tooling has some limitations that slow down the adoption of MDE and its derived paradigms [11, 24].

Many challenges revolve around the modeling languages [20] and the tools used in the process such as modeling editors.

We identify two limitations of modeling editors: the level of expressivity and flexibility induced by the tool and a deficiency in terms of usability, making their usage difficult for domain experts and practitioners alike.

Language workbenches offer a model editor that enables users to manipulate their models using the syntax of the DSL.

Most editors are parser-based and can be classified into two categories.

On the one hand, Free-form editors are typically used for textual DSL, like Xtext [4] and Spoofax [13].

On the other hand, syntaxdirected editors are typically for graphical DSLs, like MetaEdit+ [17] and AToMPM [19].

The difference between the two lies in their parsing technique as they both rely on a parser to build an abstract syntax tree with the given input and validate the syntax.

A projectional editor, however, does not rely on parsers.

As a user edits a program, the AST is modified directly.

Projection rules are used to create a representation of the AST with which the user interacts, reflecting the resulting changes [22].

Without a parser, it enables the support of notations that cannot be easily parsed, such as tables or mathematical formulas, and the composition of any language without introducing syntactic ambiguities.

As demonstrated in [3], this is much harder to achieve with parser-based tools.

The most promising projectional editor in the MDE community is currently Jetbrains MPS [5].

However, it is a heavy-weight editor that cannot be easily integrated in other tools.

In this paper, we present Gentleman, a lightweight web-based projectional editor generator, which aims to close the gap between models and domain experts.

Gentleman allows the user to define a model and projections for its concepts, and use the generated editor to create the model instances.

We demonstrate how to define a projectional editor for Mindmap modeling, covering model definition, text and table projection, multi-projection, and styling to showcase its main features.

The tool demonstration is available online.

2.2 Projections

A projection is a representation of a concept that can be visualized and interacted with in the graphical user interface .

It can be applied to any part of a concept, such as the concept as a whole or an attribute.

Note that the language engineer may define multiple projections for a concept.

By doing so, we can obtain the right combination of visuals in any given situation.

Gentleman offers predefined layouts found in modern GUI technologies [14].

It also provides data-specific controls in the form of fields, providing more structure to help users quickly scan and comprehend the information presented [12].

Both layouts and field can be customized with a style.

At the moment, Gentleman only supports relative and tabular positioning with limited support for graphical elements.

It does not provide the means to arbitrarily position elements, that would be possible with the use of HTML canvas, for example.

2.2.1 Layout.

A layout is concerned with the structure of a projection.

It organizes elements presented in the GUI by indicating the location of its child elements.

Gentleman defines layouts similar to those found in popular GUI frameworks such as Xamarin [10], SWT [9], and WPF [15].

For instance, the StackLayout piles elements horizontally or vertically, the WrapLayout group its elements in a block, and the TableLayout arranges them in a row or column-directed table.

Every layout presents a container that can be configured to be collapsible, draggable, or resizable.

A layout child elements can be a text content, a layout, or a field.

In the Mindmap instance presented in Figure 2, the organization of the elements is structured using the StackLayout to order them vertically.

We use the WrapLayout to group them together, such as the heading Mind Map <title="Planning">.

Each layout is further enriched with styling rules like color, border, and spacing values.

2.2.2 Field.

A field is concerned with manipulating the value of a concept; thus, it enables data input and output.

It provides an abstraction for the underlying widget to promote reusability and portability.

Gentleman uses a modular approach to select the right widget for the intent of the user.

For instance, it selects a Textarea if the user intends to write text on multiple line or a single-line Textbox otherwise.

Gentleman offers fields that cover the most fundamental widget components in GUIs [8].

Among others, a TextField allows the user to input characters for a String or Number.

A BinaryField enables the user to alternate between two states of a concept value, such as for a Boolean.

A LinkField allows the user to refer to another concept and bind its projection for a Reference.

A ChoiceField enables the user to select one item in a predefined list for a Prototype.

Each field offers specific customizations, such as the projection of each choice in a ChoiceField or the visual delimitation between items of a ListField.

More advanced fields, like a TableField allow the user to manipulate structured data and provide the ability to add, remove, sort, and filter data.

Fields also have generic properties to specify, for example, if they are read-only, disabled, or hidden.

In the Mindmap instance presented in Figure 2, the title, name and description attributes all target a String concept and, therefore are rendered as TextFields.

The marker added to the main topics are LinkFields, referencing the declared markers in the header.

The main topics and subtopics are rendered as ListFields stacked horizontally.

Additional main topics can be added to the central topic with the Add control action.

Note how we customized this action field to display a template of the main topic structure.

2.2.3 Style.

Any projection can be complemented with style rules to describe its presentation.

Styles can be defined directly in the Gentleman editor or imported and applied to a layout, text container, or field.

For example, users can set the font, color, and alignment of text and the border of a table.

To avoid repetition and encourage better integration, Gentleman leverages the browser technologies, offering full support for CSS class selectors.

Layouts and fields also expose class selectors for their HTML elements.

This enables the language engineer to declare global styles through CSS and specific context-based rules in Gentleman.

Note that images can be added through the background property offered in CSS and Gentleman.

In this paper, we presented an approach which allows for modeling structure as well as behavior of a software system using an integrated tool combining two OMG standards: UML and ALF.

While UML package diagrams and class diagrams are used for modeling the structure, method bodies – i.e., the behavior of a method – are specified using ALF.

Fully executable Java source code may be generated from the resulting model system which can be integrated seamlessly in existing software ecosystems.

The integrated user interface abstracts from the underlying set of models and provides a unified look and feel for the end user allowing for graphical as well as textual modeling.

The feasibility of the approach has been demonstrated using an example workflow which may occur in real-life iterative software development processes.

In this section, the approach presented in this paper is discussed.

First, the resulting benefits are given: Fully Executable Models.

This approach overcomes the code generation dilemma that occurs when model and hand-written code evolve independently.

Specifying both structure and behavior leads to a generation of fully executable source code that can be used by further programs and does not require any user interaction or code modifications afterwards; any information contained in the model is mapped adequately to the resulting source code automatically.

Convenient Notation.

Our approach combines modeling structure and behavior using two different modeling languages and two different paradigms of editing models.

On the one hand, the projectional diagram editor provides convenient graphical notation for modeling structural elements.

On the other hand, behavior is added textually by means of the parser-based editor instead of using another graphical editor; while graphical notation for behavioral model elements can result in very large and confusing diagrams, using textual syntax results in concise model representations that are easy to read and understand.

Visual Integration.

Modeling structure using UML diagrams and behavior using ALF text is not only combined technically but also visually: The different editors are combined in the modeling environment by means of appropriate Eclipse concepts.

The user gets the feeling of editing one model instead of dealing with a collection of models that are involved in the background.

Interlinked Model System.

Another conceivable approach to overcome the code generation dilemma could provide for modeling the structure and adding the behavior by means of code snippets in terms of plain text comments in the UML model.

By contrast, our approach contains all the information of the modeled system in several models within an interlinked system; hence, all the artifacts used for the code generation – in particular the ALF operations – are persisted in terms of models within the ALF model system.

In contrast to plain code snippets, cross links can be exploited to find model elements and text editor mechanisms as a content assist can be used.

Flexible Workflow.

The kernel transformation converting UML models to ALF model systems and vice versa is bidirectional and incremental.

Thus, a very flexible workflow is supported that allows for development processes consisting of several iterations of editing structural and behavioral model elements.

These aspects emphasize in particular the benefits of using ALF as the underlying language for expressing behavioral elements.

However, using ALF comes along with a significant drawback with respect to expressiveness: Although by means of ALF a quite large range of model elements can be expressed, only a proper subset of UML is supported; thus, some elements – e.g., interfaces – cannot be expressed exactly.

Nevertheless, the semantics of non-fUML elements often can be approximated pretty well using alternate components – e.g., abstract classes instead of interfaces – such that in practice, the limited expressiveness resulting from using ALF does not restrict the modeling process too hard.

This paper is an extended version of [20] and provides besides an extended example use case some more technical details.

Model-driven software engineering [24] aims at reducing effort for developing software by specifying higherlevel models, instead of lower-level hand-written source code.

An initial model capturing the requirements is often the starting point from which a number of models over multiple levels of abstraction is derived, until the system is eventually implemented.

In order to support model-driven software engineering in a full-fledged way, key enabling technologies are mandatory for defining modeling languages and specifying and executing model transformations.

Usually, modeling languages are defined with the help of metamodels in the context of object-oriented modeling.

To this end, the Object Management Group provides the Meta Object Facility standard [18].

Throughout the last two decades, UML [19] has been established as the de-facto standard modeling language for model-driven development.

In its current version, UML comprises seven kinds of diagrams dedicated to structural modeling and seven different diagrams addressing behavioral aspects of a software system.

In order to support model-driven software engineering in a full-fledged way, having models which allow for a generation of fully executable code is crucial.

However, generating executable code requires a precise and well-defined execution semantics of behavioral models.

Unfortunately, not all behavioral diagrams provided by UML are equipped with such a well-defined semantics.

Furthermore, some diagrams with a well-defined execution semantics, e.g., activity diagrams, are on a lower level of abstraction in terms of specifying control flow.

As a consequence, the state of the art in model-driven software engineering nowadays is specifying the static structure of the software system using models from which source code is generated.

This generated source code is then augmented with behavioral elements using regular programming languages.

This fact which we call the “code generation dilemma” [6] is problematic as the different fragments of the software system tend to evolve separately which quickly leads to inconsistencies between the model and the source code.

Round-trip engineering [7] may help to keep the structural parts consistent but unfortunately there is still no adequate representation of the manually supplied behavioral fragments.

The Action Language for Foundational UML [15] is also an OMG standard addressing a textual surface representation for a major part of UML model elements.

Furthermore, it provides an execution semantics via a mapping of the ALF concrete syntax to the abstract syntax of the OMG standard of a Foundational Subset for Executable UML Models, also known as Foundational UML or just fUML [16].

The primary goal is to provide a concrete textual syntax allowing software engineers to specify executable behavior within a wider model which is represented using the usual graphical notations of UML.

A simple use case is the specification of method bodies for operations contained in class diagrams.

To this end, it provides a language with a procedural character whose underlying data model is UML.

However, ALF also provides a concrete syntax for structural modeling within the limits of the fUML subset.

In the academic world, the Eclipse Modeling Framework [22] constitutes the platform for research dedicated to model-driven software engineering.

Its metamodel Ecore is based on a subset of MOF called Essential MOF .

Following a pragmatic approach, EMF strictly focuses on principles from object-oriented modeling only providing core concepts for defining classes, attributes, and relationships between classes.

Furthermore, it allows for Java code generation from these structural model definitions.

EMF provides an extensible platform for the development of MDSE applications.

In this paper, we present a tight integration of the OMG standards UML and ALF to realize an integrated modeling environment which allows for structural as well as behavioral modeling.

Fully executable Java source code is generated from the resulting models, allowing for “real” MDSE approaches.

One significant goal for the integrated modeling tool was that the integration is not reduced to a technical combination of both languages UML and ALF but also comprises the user interface constituting a visual integration of different editors such that an easy and fluent usage is feasible.

Although a pretty wide range of models are involved in the background processes, the user should get the feeling of editing one model instead of a collection of models where each of them represents a certain portion of the context.

This section describes the foundations of the implementation with respect to the user interface.

In order to facilitate an integrated user interface, an Eclipse view was created that provides the textual modifications of the ALF operations.

While the class diagram is visible within the graphical editor which constitutes the main editor where the user edits the structure, the behavior is modified textually within the additional view.

Xtext provides tool support to embed generated editors within SWT composites which is used for our tool to embed the text editor within the Eclipse view.

The view contents depend on the user’s actions in the main editor.

If the user clicks an operation or a derived property within the class diagram, the view is notified about the respective edit part and shows the embedded editor with the textual representation of the corresponding ALF operation.

The complete ALF operation is now visible and can be edited as in case of a usual text editor; apart from behavioral modeling concerning the operation body, also the structural model elements related to the operation – i.e., the name, the visibility, the parameter list, and documentation – can be modified textually.

The view provides a button for finishing and persisting the current modification of the respective ALF operation; when the button is clicked, the respective text file is saved, the parsing process of the ALF model is performed and eventually the ALF-to-UML transformation is induced such that the structural changes of the respective operation get visible within the class diagram.

Thus, the integrated tool provides round-trip engineering with respect to structural elements of operations; all other structural model elements are edited within the diagram editor while behavior can only be edited textually.

The main goal of our approach is to avoid the syntax barrier at early stages of introductory programming, so that novices are not obstructed by accidents, while focusing their attention on program semantics.

Javardise covers the syntax requirements of the first programming course taught at our institution.

Despite informal experiments where a few users were asked to try the editor, no controlled experiments were carried out, neither the editor has yet been in used in classrooms.

This section discusses the hypothetical benefits we aim at along with a forecast of drawbacks.

Malformed Code.

Those who teach lab classes know that a considerable number of times when they are asked for help is due to a purely syntactical error/typo.

For instance, a missing semicolon , unbalanced brackets, using illegal characters in identifiers, “reversed” assignment statement, and numerous creative forms of syntax guessing.

Even for an expert, finding the cause of the problem might not be immediate, as they are obfuscated in the code .

These issues contribute to slowdown the pace of a lab class.

On the other hand, while in a lab class a student may have the aid of a teaching assistant or a colleague, syntactical hurdles may constitute a harder barrier to progress when alone.

As syntax is not part of the essential difficulty of learning how to program, we argue that the associated hurdles should be minimized in favor of concentrating the effort on the essence.

Focus on Semantics.

Part of the difficulty on learning programming, often described as “overwhelming”, might be due to having to learn simultaneously a programming model and how to instantiate it using a particular syntax.

We believe that once a solid understanding of semantics is gained, learning a syntax becomes a minor issue.

The less time spent on syntax hurdles, the more time is spared to dedicate to semantics.

Switching to Conventional Editor.

The fact that when using a structured editor the user is not typing every character of the source code may require an adaptation phase when switching to a conventional editor.

This issue should be investigated.

In an editor like Javardise, the code is being presented exactly as the actual source code, and hence, we hypothesize that users will likely get accustomed to the syntax.

Nevertheless, it is worth noting that modern IDEs also perform various code insertions automatically and not every character is typed.

Therefore, we do not expect major issues with respect to editor switching.

Stepwise Acquaintance with Syntax.

We argue that exposing novices to more syntax than necessary to teach fundamental concepts is counter-productive.

Syntactic variations at early stages may confuse the learner, since different variations of expressing the same thing are presented.

For instance, in a language like Java, there are five basic ways to express a variable incrementation.

We believe that syntactical variations, such the possibility of using a for loop in place of a while, should be presented only when elementary constructs are well-understood .

By this time, the learner is likely to better realize why such syntactic possibilities were invented, while not confusing the essence.

An editor like Javardise, which supports syntax levels, is useful to this teaching strategy, as syntactical constructs may gradually be activated as learning progresses.

Unsatisfactory Usability.

We believe that the biggest threat to Javardise, as well as to structured editors in general, is the lack of usability that may be caused by the rigid mode of editing the source code.

Without a pleasant and productive means to manipulate the source code, the previous hypothetical benefits and the overall goal of easing introductory programming become hindered.

Hence, effective structured editors should require a serious investment on the usability of the interaction.

We plan to conduct qualitative studies to improve the usability of Javardise, as well as controlled experiments to evaluate its effectiveness.

In Software Engineering there is a usual distinction between essential and accidental difficulties [2].

Essential difficulties are those that are inherent to the complexity of the problem at hand, whereas accidental difficulties emerge from the form that is used to map the problem into a solution.

With respect to programming, languageindependent algorithms are mapped to a particular programming language syntax.

Understanding the algorithm and the general programming concepts are the essential difficulty, whereas the syntax is an accidental one.

Mastering a programming language requires understanding its programming model, as well as the syntax to instantiate it.

We may known well how a sorting algorithm works , but not being immediately able to implement it using a particular language that we are not acquainted with .

Although syntax is not the essence of a program, it may have a significant impact on the usability of a programming language.

Particular characteristics of certain syntaxes may consist of hurdles, especially to novices [3, 11].

Regarding experienced programmers, although they should be able to understand every kind of syntax, they usually have strong feelings in favor or against particular kinds of syntax .

Syntax-directed code editing was proposed back in the 1980’s [12], being the key idea the concept of a code editor that constrains the user editing activity so that program being written is always syntactically valid.

That is, the source code, while it may have semantic errors, it always parses .

We refer to this kind of tool as a structured editor.

Structured editors stemmed from a line of research at Cornell and Carnegie Mellon Universities, and were adopted therein for teaching, as well as in other universities in USA during the 1980’s [8].

Despite the promising idea, they have neither gained widespread adoption in programming education, nor in the software industry.

The reasons possibly relate to the usability of the editing experience, but this subject was not studied systematically.

Projectional editors, such as those that can be developed with MPS , is a closely related tool concept that in addition to AST-based editing is also capable of combining multiple syntaxes.

Nonetheless, the code writing experience of both structured editors and projectional ones is similar.

Projectional editors have a few usability issues that may hinder their wide adoption [13].

However, a usability study [1] revealed that projectional editors are efficient to use for basic editing tasks – the case with introductory programming.

The effectiveness of structured editors through controlled experiments has not been a subject of much research.

We believe in the potential of structured code editors, especially for introductory programming purposes.

We aim at rehashing the concept, perform user studies to investigate how to maximize the usability of structured editors for introductory programming, and further evaluate their benefits and drawbacks.

At this stage, we have developed Javardise, a structured editor for a subset of the Java language , covering the syntax that we expose to students during the first programming course taught at our institution.

7 Collaborative Domain-Specific Modeling

MPS comes with a high-end Integrated Development Environment that is well suited for the development and exploitation of Domain-Specific Languages in an industrial context.

This IDE is a stand-alone desktop application.

Its users include DSL engineers developing DSL models with accompanying functionalities and DSL users that create and exploit instances of such DSL models.

The appearance of the IDE can be customized to ease usage for DSL users .

This helps in reducing the learning curve that is often experienced and in only covering the specific functionality relevant for DSL users.

Exploitation of MPS at Canon Production Printing has, however, revealed the need for some additional capabilities: Collaborative modeling: DSLs at Canon Production Printing often represent domain-specific interfaces between models from different engineering disciplines, as also exemplified with the DSLs in Sects. 3 and 4.

Such interfaces would benefit from the ability to collaboratively update DSL instances in a similar fashion to Google Docs and Microsoft Office 365 support for office documents.

Collaborative modeling would benefit from a web-based front-end for, in particular, DSL users.

A server-based deployment would also ease version control for DSL users unfamiliar with traditional software technologies and terminology for version control.

In addition, it eases updating any involved tools in case of DSL evolution.

Integration in larger GUI applications: Canon Production Printing has various existing development environments that could benefit from exploiting DSL technology.

Such environments may rely on custom Graphical User Interfaces for which it is often not easy or even infeasible to replace them in a gradual or disruptive step by the IDE for MPS.

Such situations would benefit from the ability to integrate the DSL technologies provided by MPS into the existing development environments as DSL widgets.

Domain-specific customization of look-and-feel: Canon Production Printing positioned model-driven development as crucial for timely development of novel printer systems.

Engineers of all engineering disciplines are expected to exploit DSLs for which tooling is realized using MPS.

However, engineers with little or no affinity for software development have many difficulties in adopting the IDE .

It would be beneficial to provide DSL technology as part of highly customized GUIs with a domain-specific look-and-feel that is much closer to that of domain-specific tools currently used by such engineers.

This section7 peeks into the future direction Canon Production Printing is considering for a widespread exploitation of MPS throughout the organization, which covers not only research & development but also, for example, sales and service.

7.1 Blended Collaborative Domain-Specific Modeling

A fundamental concept underlying MPS is the use of a model-view architecture.

This is exposed by the DSL model instance being projected, possibly even with multiple different syntaxes at the same time, to a DSL user.

The DSL user can change the DSL model instance via any of these projections or views in traditional model-view architecture terminology.

This results in the DSL model instance changing, and hence all projections of the DSL model instance update consistently.

This principle is also suited for collaborative modeling since it is irrelevant whether the projections are to a single DSL user or to multiple DSL users.

Such projections may concern different syntaxes for the same DSL model instance.

Note that blended collaborative domain-specific modeling as described here requires the one DSL model instance to be accessible for all involved DSL users.

itemis is experimenting with generating all that is needed to support blended collaborative domain-specific modeling based on a DSL model defined in the IDE that comes with MPS [1].

Figure 18 shows a high-level overview of itemis’ initiative called Modelix [15].

It relies on the DSL model instance being accessible on a central server.

Based on a plugin, the DSL model instance can be accessed with the existing IDE of MPS.

The DSL model instance can, at the same time, also be accessed via a web-based front-end in a DSL user’s web browser.

The idea is that execution of DSL facilities such as model transformations and code generation runs on the server.

itemis’ Modelix approach shown in Fig. 18 has the major benefit that existing DSL models defined with MPS like those highlighted in previous sections can be used in a blended collaborative way without restarting their development in a different technology such as Javascript.

In addition, the existing IDE can still be used by DSL users that are already familiar with it as it integrates seamlessly.

Based on these advantages, Canon Production Printing intends to use itemis’ technology to extend the applicability of DSL technology throughout a larger part of the organization.

7.2 DSL Widgets in Custom GUIs

Next to the Graphical User Interfaces made available to print professionals as part of the Professional Digital Printer product families, several tools to develop and maintain these product families are also being created for use within Canon Production Printing.

Application of DSL technology to formalize the domain-specific knowledge underlying both such Graphical User Interfaces has major benefits.

However, existing DSL technology does not provide the customization flexibility that traditional software technology provides to develop GUIs.

With the introduction of itemis’ Modelix, Canon Production Printing envisions the use of Modelix to create DSL widgets as part of traditional GUIs applications realized with traditional web technology such as Google’s Angular [11].

MPS and Angular use a similar component-based approach to compose complex views on a collection of related data items from simple views on individual data items.

On the other hand, the concepts underlying a model-view architecture can be realized fairly easily in Angular.

In the Angular context, the controller concept of a model-view architecture, which is responsible for converting data into a form that can be displayed to users via a view, is often denoted as ViewModel.

Given the similarities, Canon Production Printing envisions that Angular applications can be partly DSLified with Modelix-based components, bringing together the strengths of MPS’s DSL technology and the customization flexibility of Angular.

7.3 Outlook for Collaborative Modeling Canon Production Printing intends to investigate how the combination of Modelix and Angular can serve the creation of blended collaborative domain-specific modeling as part of larger GUI applications with domain-specific customization and look-and-feel.

This future direction is expected to vastly increase adoption of MPS’s DSL technology at Canon Production Printing.

Engineers can spend countless hours discussing a design aspect of a printer system, both within and across engineering disciplines.

It is hard to overestimate the time spent on discussing printer-specific details while speaking different languages to explain such details to each other.

Commodity tools used for capturing printer-specific knowledge in models do not necessarily allow for domain-specific customization.

Hence, when sharing knowledge, engineers still spend a lot of time on interpreting the models in such tools.

At Canon Production Printing, technology for domain-specific languages has proven to provide a suitable means to bridge this domain-specific interpretation gap between models in commodity tools.

Moving to a model-based way-of-working is mostly a no-brainer at Canon Production Printing.

However, choosing MPS as core technology to bridge domainspecific interpretation gaps certainly is not.

It is also not straightforward to introduce it as core technology for formalizing printer knowledge to enable automated processing.

As highlighted in previous sections, the main challenges with MPS have been as follows: Steep learning curve of MPS, both for DSL engineers and DSL users: The projectional editing experience is unlike any commodity tool, which therefore requires adapting to a different way of user interaction.

In addition, the vast amount of already available languages and their interrelations is challenging to overview.

These foundations are, however, key to the strengths of MPS.

They enable language modularity and seamless editing of different views with one and the same underlying model.

We believe that the direction of itemis’ Modelix can substantially reduce the steepness of the learning curve for DSL users.

Lack of full-fledged DSL models in MPS for commodity languages such as C++, C# and VHDL: As exemplified in Sects. 4 and 5, such languages are generation targets from specification models at Canon Production Printing.

Instead of being able to exploit model-to-model transformations to full-fledged DSL models that rely on fully engineered model-to-text transformations in a final code generation step, we resorted to creating our own model-to-text transformations.

This approach hampers maintainability, also in view of remaining compatibility with libraries for such target languages.

Hence, we intend to exploit future DSL model solutions that the MPS community may develop.

Existing parsers or grammar rules commonly available in alternative technologies are not immediately reusable in MPS.

Integrating existing textual languages, for example, tends to require more effort than just integrating an existing parser technology.

We believe that extending MPS with a means to also allow traditional text-based parsing would ease adoption considerably.

We experienced that the performance of transformations can be undesirably low.

This is, however, not only experienced for MPS but also for alternative DSL technologies, including Eclipse Xtext/Sirius.

Despite the above, more and more people within Canon Production Printing are taking up the challenge to learn MPS or will be empowered by user-friendly interfaces toward a consistent and coherent set of domain-specific models .

The key advantage over other DSL technologies to pursue this direction are the architectural foundations of MPS easing supporting multiple syntaxes for a DSL model and composability or modularity of DSL models .

This enables DSL engineers to focus on the DSL model itself instead of on resolving low-level, often tool-related, challenges that tend to arise in other DSL technologies.

DSL users benefit primarily from the ease of using multiple syntaxes and of course the well-known general benefits of using DSL technology .

Canon Production Printing develops high-end professional digital printers in three categories: industry-leading continuous-feed printers for massive print volumes and fast, high-quality results in full color or black & white; highly efficient, highvolume printers for in-house printing or publishing; and large-format printers for stunning display graphics and high-quality CAD/GIS applications.

These products serve the professional print market with suitable trade-offs between multiple system Key Performance Indicators such as productivity, perceived image quality , print robustness , and cost.

Figure 1 highlights some example printer families with an indication of their productivity capabilities and physical sizes.

The printers in Fig. 1 exploit an inkjet print process, which covers steps such as image processing, positioning the jetted ink on media, and spreading & solidification of the ink.

How each print process step is realized has an impact on the various system KPIs.

A multitude of engineering disciplines is involved in the development of print processes and printers, and hence in realizing competitive values for all the system KPIs.

Development within Canon Production Printing exploits several model-based approaches in all engineering disciplines to cope with the ever-increasing complexity.

The complexity of professional digital printers is nowadays fairly comparable to that of high-end cars.

At Canon Production Printing, model-based approaches have already proven to be essential for efficiently performing continuous innovation with sustainable quality in a highly competitive market.

An important ingredient of model-based development at Canon Production Printing is to have good tool support to do the actual modeling.

Architects, designers, and engineers often rely on tools that are to some extent targeted for addressing specific aspects of a printer.

For example, the mechanical design relies on using a CAD tool, which is, however, an unfamiliar development environment for a software designer.

Nevertheless, the mechanical and software designer are working on the same printer, and hence part of the knowledge captured in their models must be the same to ensure that together they come to a design that will actually work.

Exposing models to a wider audience than the original context from which they originated is an essential ingredient of achieving consistent designs within and across engineering disciplines.

Instead of teaching the wider audience to use the tool in which a model was originally created, it can be more appropriate to introduce a tool that allows for automated exchange of the knowledge captured in various domain-specific models .

This allows for much better reuse of models within and across engineering disciplines.

In summary, the benefits of using a tool include: Users don’t have to learn the quirks of numerous tools.

Within one tool, multiple models can be integrated much more intimately.

Integrating and maintaining one tool in development environments is less work than integrating and maintaining multiple tools.

MPS is a core tool that is used extensively within Canon Production Printing.

However, it is not only used for realizing consistent exchange of knowledge captured in models developed in more domain-specific tools .

It is also being used as the main tool for creating models to capture knowledge that has not yet been formalized.

This allows computer-based processing for more automated printer development as essential extensions to the capabilities of consistent exchange of knowledge between domain-specific tools.

This chapter highlights several modeling efforts that emerged within Canon Production Printing and how MPS played a role in these efforts.

Figure 2 gives a high-level impression of the main relations between the different DSLs discussed in this chapter and knowledge captured in domain models for the involved engineering disciplines.

Our feature model is implemented in MPS, based on a generic feature modeling language provided by itemis, who integrated a constraint solver with their feature model language.

This constraint solver provides instant feedback on inconsistent models or contradicting constraints within the model.

On our request, itemis implemented an accompanying language to support the feature flags.

We further leveraged MPS’s language extensibility to add custom validations on top of the existing feature model language.

We exploit this, for example, to ensure unique feature names and that feature names adhere to the rules of the target language.

As MPS allows additional generators without changing the source model’s language, we have implemented our own generator based on the language stack developed by itemis.

The MPS projectional editor integrates seamlessly both the graphical feature tree and the textual constraint expressions.

Modeling and reasoning about all features and their dependencies is quite challenging.

However, the integrated rendering and seamless editing allow the focus to be placed on the inherent complexity, rather than dealing with inadequate or split up editors .

As we are replacing the logic with variation points throughout our code base, team members regularly encounter MPS for the first time when they adopt our feature model.

Being developers, this user group is not discouraged by MPS’s IDE appearance.

However, they still have difficulties in working with the tool without close guidance.

The main usability issues include the separation between context menu and intention menu, the unfamiliar keyboard navigation and selection scheme.

The feature modeling language consists of two main aspects: the feature model and configurations.

After every change to the feature model, the user has to explicitly adapt each configuration to the changes by triggering an intention.

This could be avoided if MPS would provide language developers with an easy mechanism to propagate changes to other parts of the model, while keeping manual changes on the propagation target untouched.

MPS supports generic language extension through node annotations and accompanying amendments to the projected main editor.

However, a language extension cannot easily amend the projected inspector editor.

The target programming language of our custom generator is not available as MPS language, and is too complex to easily implement.

MPS’s TextGen is not well suited to generate large amounts of text, especially if the generated structure differs from the input model.

We resolved this issue by leveraging the PlaintextGen extension [8].

MetaR exploited JetBrains MPS in many ways.

The generation of code in a target language simplified the implementation by removing the need to develop our own language runtime system for the DSL.

The possibility to extend the IDE with custom tools/plugins served the requirements of the project very well.

The Build Language provided a convenient way to package the software as MPS plugin and to manage its dependencies.

The MPS Plugins Repository, along with its automatic dependency resolution among plugins, saved us from creating our own distribution website and spared our users from numerous tedious installation instructions.

Above all, by coordinating and putting together the features offered by each aspect of the language definition, we achieved a homogeneous approach for many steps of the data analysis process.

Graphical elements with scripting, auto-completion, high-level abstractions over data and instructions, language composition, automatic installation of dependencies for each individual statement, extensions to the environment, seamless integration with external technologies, and solutions to package and distribute the software all smoothly combined in MetaR to serve the purpose of the project.

We were able to achieve everything we planned and beyond, and to create languages with a runtime support somehow unique in the DSL landscape.

The main lesson learned from this experience is that biologists and clinicians can use the tools of bioinformatics and get closer to data scientists.

And when different profiles can speak the same computational language, misunderstanding is reduced and the speed of a research project is greatly enhanced.

In our data-rich age, MetaR has proved to be an educational bridge between these two worlds.

Bioinformatics is an interdisciplinary field of study that combines biology with computer science to understand biological data.

The analysis of these data is an important part of most modern clinical and genomic studies.

However, while statistical and computational tools are available for statisticians and data scientists, providing them with both computational power and flexibility, they are often not suitable for biomedical researchers looking to perform data processing and simple analyses.

Following a long track record of success in providing tools and services to support the conduct, management, and evaluation of research, the Informatics Core at the Clinical & Translational Science Center 1 at Weill Cornell Medicine has put effort and funds to create new computational methods to facilitate data analysis.

For this purpose, the CTSC has developed MetaR [1, 2], a new kind of interactive tool providing high-level data abstraction, manipulation, and visualization.

MetaR is composed of a set of data analysis languages built with the Language Workbench Technology [3] offered by JetBrains Meta Programming System [4] to make data analysis easier for biologists and clinicians with minimal computational skills.

MetaR has been developed in a research community, where collaboration prevails over competition.

For this reason, MetaR offers a means to contribute to the project with new features that can be plugged in the DSL with minimal effort by advanced users.

This is possible thanks to the very particular way JetBrains MPS maintains and manipulates the language definition.

Differently from other language workbenches and development environments, MPS maintains the language definition in an Abstract Syntax Tree [12].

Working directly with the AST has the advantage of facilitating the creation of composable languages and making them seamlessly integrate.

Language composition has been a pillar of MetaR since its inception.

Two languages compose when elements of language B can be used or referred by elements of language A.

Since MPS works with ASTs, this translates to attaching nodes defined in B as children of a node defined in A.

Favoring composition makes it possible to create micro-languages [10] that integrate directly with the core DSL.

The design of the elements to compose in MetaR builds on top of a previous work [13] where we experimented and assembled our experience with language composition and MPS.

The structure of a language in MPS is defined by a set of concepts.

Mirroring what is available in object-oriented programming, a concept can extend another concept and implement multiple concept interfaces.

On the other hand, interfaces can extend several other interfaces.

All these relations permit the modeling of complex contracts among concepts, achieving language composition.

What can be primarily composed in MetaR are statements inside the analysis.

Or at least this is the type of composition most visible to the end user.

Several concepts and interfaces in the metar.tables language define the rules for code editing and rendering of new statements.

These rules mainly affect how statements are expected to behave, take their inputs, expose their outputs, and generate the R code to ultimately execute the analysis of the data.

Defining a new Statement concept compliant with these rules makes it immediately available to be used inside an analysis, no matter in which language it is defined: it just needs to be part of the lexical environment.

New statements simply and seamlessly compose with MetaR through language composition without any intervention in the host language.

With these techniques, several micro-languages have been created and incorporated in MetaR.

We call a language a micro-language when it has a very small structure offering extensions to MetaR core DSL.

Each of them provides abstractions designed for a specific purpose .

When combined through language composition, microlanguages provide complementary features and make it possible to write richer MetaR analyses.

One of the most prominent examples of a micro-language is the BioMart language .

7 Advanced Exploitation of MPS.

JetBrains MPS is not only a language workbench to create DSLs, it also provides a wide range of additional instruments to build a comprehensive working environment around the DSL itself.

MetaR leverages some of these MPS features to make its delivered functionalities and the user experience unique.

This section presents some notable capabilities of MetaR that would not have been achieved with any other language workbench than MPS.

7.1 Graphical Elements.

In MetaR, graphical elements are mixed with statements in the Analysis editor.

We have already encountered some of these elements in Sect. 6.

These inclusions are possible thanks to the capabilities of MPS to embed Java Swing components inside the editor’s cells.

Graphical elements in MetaR have two different functions:

1) Triggering actions—these actions typically support the statement’s purpose.

2) Displaying results—the results coming from one statement or multiple statements are shown in between the statements.

On a small scale, the second function can be compared to Jupyter Notebooks7 where snippets of code generate results inside a notebook’s cells.

7.1.1 Buttons.

MetaR comes with a language for adding cells with buttons to the projectional editor and associating actions with them.

We have already seen some buttons next to statements to show/hide/move cells in the editor .

The Table editor uses buttons differently.

In this case, the action associated with the button is to open the File Open Dialog box on the local system and allow for locating and selecting the file with the table .

7.1.2 HTML Tables.

Sometimes it is useful to visualize a snapshot of a table’s content directly inside the editor.

The preview table statement allows for the creation of a small customized preview by embedding an HTML table inside a cell.

Columns and rows of this preview can be resized as desired as shown in Fig. 23.

7.1.3 Auto-refreshable Images.

In Figs. 15 and 17, we saw how the multiplot statement displays images inside one or more cells in the editor.

The same images are also available as preview in the inspector of the concept.

Images in MetaR are auto-refreshable.

This means that whenever the image changes, it is automatically reloaded in the cell.

To achieve this result, MetaR uses a Project Plugin.

This type of MPS plugin provides a way to integrate Java code with the IDE functionalities.

They are created with the jetbrains.mps.lang.plugin.standalone language and held inside solutions.

The concept in charge of displaying a specific image also registers a listener of the image file in the plugin.

The plugin makes sure that the listener subscribes on the file exactly once and is triggered at the right time.

When the analysis is executed and a change is detected, the listener invokes the Java code inside another solution and the image is automatically refreshed in all of the cells where it is currently loaded.

7.2 Table Viewer Tool.

Tools are extensions of the original MPS IDE.

They provide customized views to open with the context menu.

The MPS plugin language supports the creation of plugin solutions that define new Tools.

The MetaR Table Viewer is a Tool associated with the Table concept and its descendants.

The viewer adds to the MPS interface the capabilities to load the table’s content and show it in a graphical context inside a view.

Wherever a table name appears , the tool can be opened to see the rows and columns of that table along with their values.

Rows are dynamically loaded as the user scrolls down the content .

7.3 Execution.

MPS provides Run configurations to define how to execute processes starting from selected nodes in the language.

MetaR uses these configurations to run the R scripts that it generates from the Analysis node.

7.3.1 RunR Configuration.

The RunR configuration is a plugin solution in MetaR for the execution of analyses.

By right-clicking on any descendant node of the Analysis node, it is possible to trigger the RunR configuration from the context menu.

A setting editor dialog for the configuration is opened to customize the execution.

When the configuration is started, it first builds the current model and then runs the generated R script for the analysis.

The configuration includes Commands to invoke the R runtime installed on the local system with the proper parameters .

7.3.2 Monitored Execution.

MetaR users are not experts at debugging problems in a program, however, and in any execution things can fail for several reasons.

Since the R script is not directly visible, MetaR offers a way to monitor the progress of the execution and understand which statement generates an error, if one occurs.

Each statement has an ID assigned, which is basically the identifier of the node.

As the execution advances, the statement IDs are sent to the Run view and printed.

Inside the view, the text is properly linked to the node in the AST with the same ID.

Whenever an error occurs , the user can click on the link and get to the root of the problem .

This is obviously not a debugging tool, yet it gives users an idea of where to investigate an execution problem.

7.3.3 Integration with Container Technology.

Reproducibility of results is a key point in research.

The scientific community does not accept or consider valid results that cannot be reproduced.

This also applies to data analysis: if the same analysis is executed with the same inputs several times, it must yield exactly the same results.

Any software application needs some sort of runtime support from other software.

This is critical in terms of reproducibility, because these external dependencies introduce a level of uncertainty that is difficult to control.

Several years ago, container technology was introduced to provide fully capable execution environments on any computer supporting the technology.

There is no need for the user to deal with installation of libraries and dependencies, downloading packages, messing with configuration files, etc.; everything is made available in a single package called image.

MetaR integrates with Docker [17], the most popular container technology.

Extensions to default MPS configuration settings have been created in MetaR to set the required information.

When options in Fig. 26 are set, each MetaR analysis becomes a containerized application; it is automatically executed inside the virtual environment created starting from the selected Docker image.

This small and lightweight environment, called container, guarantees that whenever and wherever the analysis runs, it will always use the same software packaged in the image.

Typically, the R runtime and common R packages are deployed in the image.

From the user point of view, once the checkbox in Fig. 26 is checked, it is handled transparently and seamlessly by MetaR.

5 Conclusion

5.1 Technical Advantages and Shortcomings.

MPS provided the means to go for a fully structured DSL while keeping the familiar document appearance .

Language composition allowed well-designed and fine-grained decisions on first-class language concepts vs. backward-compatible compromises or low-level constructs.

Intentions provided an easy way to integrate user-controlled pattern recognizers to higher-level language concepts.

We encountered performance issues if MPS showed big root nodes in the editor.

Even more useful editors would have been possible if MPS provided cached access to find all references results.

We did not succeed with implicit type inference, because we could not strictly separate scopes that should infer types from scopes that should only check type compliance.

Importing the existing code base posed the biggest challenge by far.

We succeeded with a multistep approach that ensured early feedback on import issues.

We strongly recommend short feedback loops for any large-scale model import, as processing time grows exponentially with every additional step, and the complexity even of simple transformations gets out of hand quickly.

Both non-incremental C source code import and insufficient model merge technologies forced us to only change the original C sources and reimport the complete code every time.

Improvements in both fields would considerably simplify similar challenges.

5.2 Project Results.

The users in the IT department are very satisfied with the MPS-based solution.

We consider their early involvement, continuous feedback during development, and early training for both MPS and Subversion vital for the success of the project.

The users profit from guaranteed consistency of the FuMo DSL code they are working with and the executed code.

The turnaround time between a change in a FuMo and testable code is reduced by several orders of magnitude—from days or weeks to seconds.

Complete version control of all artifacts through Subversion assures safe processing and storage of changes.

It streamlines collaboration, both within the IT department and with the external service provider.

Shorter turnaround time and reduced communication overhead lead to higher efficiency and fewer misunderstandings, ultimately speeding up time-to-market and lowering the defect rate.

Both the IT department and the external service provider can focus on their field of expertise: insurance domain knowledge and operational/nonfunctional aspects, respectively.

The MPS-based FuMos are in production for more than 2 years.

Zurich and itemis continue to improve the system, migrate it to new MPS versions, and adjust it to new requirements.

4 Evaluation and Lessons Learned.

MPS proved to be a good choice for this project.

Projectional editors combined the familiar look to the end-users with semantically structured models.

In combination with language extensibility, the editors enable new ways of dealing with imported legacy code.

Generators assured consistent output.

We could mitigate MPS-related issues regarding editor and typesystem performance.

We faced the biggest hurdles in the project with the import of the original code base.

They were not specific to the MPS platform, but a mixture of generic legacy transformation and code-to-model challenges.

The project’s total effort amounted to roughly 40 person months.

4.1 Language Implementation.

For both VADM and FuMo, projectional editors enabled a visual design very close to the original Word forms.

Their implementation did not pose considerable challenges.

MPS’ language composition features enabled clean language design without too many compromises for backward compatibility, as we could defer edge cases to special constructs not accessible to end-users, or even to plain C models.

Standard MPS features like technical references with forward and backward navigation, Subversion integration, and plain text intermixed with model elements contributed tremendously to the final result.

We experienced platform limitations in two areas: large editors tend to perform sluggishly, and we were not able to implement type inference as we planned.

The table-heavy VADM editors posed the biggest editor performance issue.

These editors barely interacted with the type-checking system, thus excluding it as potential performance issue.

The original VADM was kept in few and large Word files.

Thankfully, the top-level VADM structures provided semantic borders to break the documents into several root nodes.

The resulting editors performed reasonably.

Regarding the typesystem issue, we abandoned the idea of heavy type inference.

It might be possible to implement, but we decided not to spend the required effort.

The impact on the result was acceptable, as our end-users were familiar with typed languages.

4.2 Import and Generation.

Importing the original code was by far the most difficult part of the project.

Even our original approach took way longer than anticipated , before we concluded it to be infeasible because of too long feedback loops.

We eventually succeeded with the second approach , but again had to overcome unforeseen hurdles like running mainframe-targeted tests on a PC and semiautomatically analyzing test failures.

We suspected most of these issues to be typical of automated application modernization projects; the team had only limited experience in this field.

Using domain-specific languages presumably did not add huge additional effort to the fundamental problem.

On the contrary, the flexibility of language composition and interactive pattern recognizers opened up new possible ways to deal with application modernization issues.

4.2.1 Import Source.

We quickly concluded that we had to use the C code as base for the import: the C code was executed, so only this artifact was known to be correct.

The IT department members knew a multitude of examples where the Word FuMo was outdated with respect to the implementation in C.

Another argument was technical: Word is hard to read programmatically, especially as we would need to preserve formatting and indentation .

Zurich used advanced Word features like tables and track changes; this would require a very solid library to reliably access the document’s contents.

The original Word VADM/FuMos contained also a prose text description.

In some cases, this description was copied into the C source code as comments; in these cases, we could import the description.

For others, we had to copy them by hand from Word.

As a one-off action, this would take an acceptable effort of a couple of days.

4.2.2 Big Bang vs. Incremental Transformation.

A big bang transformation processes the complete source at one point in time in its entirety.

Before the transformation, only the source is used; afterwards, the complete source is discarded and the transformation outcome is the only usable artifact.

With an incremental transformation approach, parts of the source are transformed step by step.

The source artifacts are valid for non-transformed parts, whereas the outcome is the only valid artifact for already processed parts.

We opted for a big bang approach to import for several reasons.

The mbeddr C importer was responsible for importing the C source files into mbeddr C models.

It had to resolve all references prior to import.

The code base turned out to be highly coupled.

There were no simple ways to cut the code base in independent sub-slices that would not reference each other.

So we either had to import all at once or import overlapping sub-slices and merge them afterwards.

We deemed merging to be much harder than an all-at-once import.

Both the IT department and the external service provider kept working on the source code during the project.

This implies that in an incremental transformation approach, the C source files might have changed between the import of different sub-slices—rendering any kind of merge even harder.

Moreover, it would have been very hard to synchronize changes in the C source files to already imported mbeddr C models, let alone FuMo DSL code.

Applying some parts of a change in already imported models, and other parts in the C source files, does not seem feasible either.

The big bang approach implied we would never change imported models manually .

All the improvements were applied to the original C source code.

This provided some investment safeguard for Zurich: even if this project would have failed, they could still profit from the source code improvements.

4.2.3 Cleaning Up Sources.

The C source code had been developed over several decades.

Naturally, it accumulated technical debt like different implementation styles or workarounds that have never been fixed.

One particular area to clean up was memory management: the source code used three different APIs to allocate and free memory.

Zurich, itemis, and the external service provider unified them to one API.

This revealed memory management issues like use-after-free or duplicate memory usage.

Analyzing and resolving these issues took considerable effort.

The abovementioned cleanups are independent of targeting a DSL environment.

More specifically to this target, we had to unify different ways to implement the same FuMo DSL construct in order to automatically recognize it.

For example, the pseudo-code contained a foreach-loop concept.

This can be implemented in C with a for loop and index variable, or a while loop and pointer arithmetic.

If both patterns had been used a lot, we would recognize both.

However, if it had been implemented mostly with a for loop and the source had contained only a handful of while-loop variants, we would rewrite the latter.

4.2.4 Lifting from C to FuMo DSL.

Lifting[8] describes the process of transforming rather low-level C code to semantically richer, more domain-specific language.

Most of the concepts are very similar in C and FuMo DSL.

A simple tree walker would process the C AST and create the corresponding FuMo DSL model.

The tree walker would wrap any unrecognized element in an escape to C FuMo DSL concept.

Through manual inspection and interviews with IT department staff, we identified typical patterns in the C sources and how they mapped to FuMo DSL.

We were very keen on matching domain-specific patterns, like formulas typical to insurance math, mortality table lookups, or access to Zurich-specific subsystems.

We implemented pattern recognizers to find these patterns during the C to FuMo DSL transformation.

The tree walker incorporated the reliable recognizers.

Less reliable recognizers were available as MPS intentions on the FuMo DSL.

This combined the required manual assurance with easy application.

It allowed continuous improvement both during our project and future development.

Examples for pattern recognizers include foreach loops , VADM access, memory allocation, or pointer referencing.

The logical inverse of pattern recognizers are FuMo DSL to C generators.

We developed them alongside the pattern recognizers.

4.2.5 Handling VADM Access.

The actual VADM structures were expressed as C structs; importing them was straightforward.

Importing their usage, however, was much more difficult.

The runtime environment initialized one complete VADM structure .

We spent serious effort to identify these access patterns and provide good abstractions in FuMo DSL.

We did not want to expose the end-users to the intricacies of C pointer handling—they should be concerned about insurance business logic.

We found quite a few cases where we could not reliably recognize, lift, and generate the correct pointer access scheme; especially performance-optimized loop handling turned out to be problematic.

There were even too many of them to be treated as edge case with an escape to C.

We resorted to explicit FuMo DSL constructs for pointer handling for existing cases: We created these concepts in the importer, but did not provide the end-user with a way to instantiate them.

If future changes require performance optimizations, they need to be provided through support libraries.

This chapter reports on a project at Zurich’s [11] life insurance branch in Germany.

Zurich and itemis [3] started this project to shorten time-to-market by improving the product implementation process.

We achieved this goal by migrating technical product descriptions from Microsoft Word to an MPS-based domain-specific language .

To ease adoption, we leveraged MPS’ projectional editor by rendering the DSL very similar to the original Word documents.

Originally, the Word documents were implemented in C by an external service provider.

We superseded this process step by a generator from the DSL to C code, cutting down the turnaround time for each change from days or weeks to seconds.

3 Proposed Solution

We proposed to migrate VADM and all FuMos to MPS models and generate the C implementation.

The revised product development process is depicted in Fig. 4.

It removed the implement–check–adjust cycle between the IT department and the external service provider.2 The test creation process remained the same.

The IT department continued to define FuMos and VADM, but in MPS instead of Word.

The MPS models were stored in the Subversion [1, 7] version control system.

All non-domain aspects were moved to support libraries.

This approach removed most of the identified shortcomings Version Control FuMo models are versioned like any other development artifact.

The versioning system provides proven diff/merge support, completely integrated into MPS.

C implementations do not need to be versioned any more, as they can be regenerated at any time.

FuMo Structure By formalizing the FuMo DSL, we provide all the regular IDE tooling: entering invalid or inconsistent syntax becomes impossible by design, autocompletion supports the user with available choices, named references are replaced by technical references and guarantee consistency, and model validation provides instant feedback.

We have implemented a generator, so inherently each construct has defined semantics.

Of course, all technical meta-information listed in the FuMo header is implicit: we automatically collect the lists of parameters and used FuMos from the FuMo DSL.

Memory Allocation The FuMo DSL does not provide direct access to memory management; it is handled by the generator.

We use the same memory allocation API throughout the generator.

Global VADM Structure We do not change this fundamental software design choice.

For once, the approach has been chosen for a good reason: it minimizes data copying and enables high performance.

Also, such a change has been out of the scope of the project, and it would not have been wise to combine such a fundamental implementation change with the technology migration at hand.

However, due to the FuMo DSL, we know exactly which FuMo accesses which VADM attribute.

This offers a sound basis for future changes, if desired.

Test Execution Environment Although not initially planned, we had to enable test execution on PCs to finish the migration to MPS .

Based on this work, Zurich now runs tests in an automated nightly build.

Test Result Granularity For our migration efforts, we semiautomated more granular failure reports through trace logging .

However, this required invasive changes to the production code and would hamper performance seriously.

Thus, these changes were only temporary.

Enable Accurate Searches Most searches look up named references.

By providing first-class technical references, we replaced these searches by straightforward linking and find all references commands.

FuMo models become the definitive source, and generation guarantees consistency of FuMo and C implementation.

Thus, we cannot miss any reference from the model.

C Language Knowledge and Duplicated Implementation Effort IT department staff writes FuMos in FuMo DSL and can immediately generate, compile, and execute the resulting implementation—prototyping and implementing a FuMo uses the same tool.

This means searching and prototyping does not require detailed knowledge of the C language any more.

Programming skills are still required for debugging purposes.

Inconsistencies Between FuMo and C Implementation All C implementations are generated from the FuMo models.

Therefore, we cannot have any inconsistencies.

As we use the same generator for all FuMos, all C implementations for one FuMo DSL concept must be identical.

High Communication Effort With code generation, writing FuMos and implementing them in C happen at one place by one party, thus removing any communication overhead.

The external service provider stays accountable for nonfunctional aspects of the system.

They can focus on support libraries and operational aspects.

Of course, both parties need to stay in close alignment.

Long Turnaround Time Code generation delivers the C implementation of every FuMo within seconds, i.e., several orders of magnitude faster than the previous process.

This provides the IT department with immediate feedback and direct control of the outcome.

3.1 Solution Technologies.

We proposed MPS as implementation technology for several reasons.

To ease adoption, we kept a form-like user experience similar to the existing FuMos.

This is easily feasible with projectional editors.

Similarly, projectional editors support VADM’s tabular style.

Intermixing prose text in FuMos with formal math expressions and links to parameters would be hard or even impossible with parserbased systems.

MPS’ language extension mechanism supports clean language design decisions while providing an escape mechanism for edge cases in legacy code.

We could design the FuMo DSL on its existing level of abstraction.

Edge cases like irregular pointer access were handled by language concepts unavailable to the end-user, while performance optimizations like pointer arithmetic could be represented by embedded C code.

itemis knows MPS very well, rendering this technology the obvious choice.

We could leverage our experience with mbeddr [2, 4, 10] as MPS-based C implemen- tation and generator to C source code.

Our internal importer from C source code to mbeddr C models enabled the import of existing FuMo implementations.

Zurich tasked itemis with maintenance, migration, and further development, relieving them from MPS development.

We proposed Subversion as a version control system, as it was available within Zurich.

It is less complex than git, lowering the initial threshold especially for users unfamiliar with version control systems.

MPS supports Subversion out of the box.

The approach proposed in this work is, to the best of our knowledge, the first to explore the use of SysML together with component fault trees.

We formulate a set of drivers from an industrial context that motivate the design decisions detailed in this work.

Our contribution with this work is twofold: firstly, we propose a scheme to construct system fault trees for an FTA and support the FMEA by generating the structure tree, function network, and failure network from SysML internal block diagrams, which is close to schemes proposed for UML and CFTs.

We introduce a deployment port to explicitly model the error propagation between hardware and software.

Secondly, we propose a scheme to construct system fault trees from SysML activity diagrams, which is motivated by their usage in our industrial use cases.

We introduce two industrial case studies: a comparison of a manually created fault tree and a system fault generated from a SysML IBD of an electronic power steering system as well as a qualitative evaluation of the system fault tree generation from SysML activity diagrams of a boost recuperation system.

We document lessons learned from applying the proposed schemes in these case studies.

Lastly, we provide an outlook on remaining challenges to further raise the expressiveness of SysML models for use in model-based safety assessment and make use of dynamic aspects, e.g., as modeled in activity diagrams.

Our approach proved to be feasible in supporting the analysis of complex safety-critical systems, e.g., highly automated driving architectures, while several challenges still remain to be solved [5].

Apart from the challenge of compliance with relevant norms, e.g., the ISO 26262, further technical challenges remain that we want to discuss in the following.

Lots of SysML features still remain to be fully exploited for system fault tree generation, e.g., integrating not only flow ports but also service ports , as this type of interaction is typical not only for software systems, but also for open networked systems and systems with user interaction .

Also, an obvious further step is to exploit the dynamical architectural aspects covered by SysML activity diagrams, which currently get lost when generating the static system fault tree.

Approaches that exploit these dynamics were already proposed, e.g., by Kaiser et al. [21] and Kabir et al. [20], but need to be evaluated for use in an industrial automotive context.

To further integrate dynamic aspects into the proposed approach, enriching the fault model regarding timing aspects by techniques such as dynamic fault trees [20], combining CFTs with Markov chains [47], or formal safety contracts are on our roadmap as well.

The FMEA support has not yet been extended for models that mainly rely on activity diagrams.

Requirements often only provide an indication of the function of a block in the sense of an FMEA.

Additional research on the similarities and differences between the nature of requirements and FMEA functions might help to automatically relate these.

While importing general-purpose SysML models from Rhapsody and Enterprise Architect and the automatic generation of pessimistic fault trees already allowed fast application of the proposed approach on two industrial subsystems, legacy systems with missing or insufficient models currently cannot be supported by our tooling.

A potential path to circumvent this problem is complementing our approach with model mining approaches that mine system models as well as potentially even fault models from legacy code.

Further challenges include integration with model-based security analysis, which we started in [12], and analysis of the safety of the intended functionality .

The complexity of modern, software-intensive systems continues to increase due to their rising number of features and functionalities.

This trend exists in several domains, including automotive, robotics, and avionics.

Model-based systems engineering methods are suitable to tackle the complexity of such systems, since they support engineers with different, yet consistent perspectives on a holistic model of the system and provide different layers of abstraction.

Systems in the mentioned domains are often safety-critical since their malfunction might lead to damages or even loss of live.

Hence, these systems have to be developed according to mandatory safety standards such as the IEC 61508 and the ISO 26262 in the automotive domain.

These standards require safety analysis methods such as fault tree analysis or failure mode and effects analysis , which inherently include a model of the system as well.

However, as of today, these safety analysis methods are typically not linked with MBSE methods and artifacts.

Thus, without strict sequential processes, there is a great risk of inconsistency between the evolving system models and their safety analyses.

Several existing approaches combine MBSE with safety analyses methods, e.g., Hierarchically Performed Hazard Origin and Propagation Studies [33], modularized component fault trees [22,23], and the Safe Component Model [11].

Extensive overviews of these model-based safety analysis methods are given by Aizpurua and Muxika [2], Sharvia et al. [39], and Lisagor et al. [25].

The main benefits of these MBSA approaches are that the implications of changes in the system model on the system’s safety are directly visible and that they enable the divide and conquer paradigm, since the error propagation of each component is specified separately.

Höfig et al. [15] present two case studies that show why these benefits are especially useful for complex industrial use cases.

MBSA methods were introduced for various metamodels, e.g., hyper-graphs [13], MathWorks’ MATLAB/Simulink [3, 34], Architecture Analysis & Design Language [29], EAST-ADL [35,45], and the universal modeling language [1].

However, in our industrial experience, MBSE methods mainly use SysML [32] as metamodel.

While a recent activity of the Object Management Group to extend the SysML by a safety profile [7] is certainly beneficial, to the best of our knowledge this profile is not yet applied in the industry.

Clegg et al. [9] use a similar profile to extend SysML with the ability to model fault trees and link basic event and failure modes to SysML blocks.

The link between elements from safety analysis and system architecture improves a common understanding of the system and allows validation checks, especially on updates.

However, the approach of Clegg et al. does not leverage the system architecture and the error propagation information contained within it.

In [30], we present our MBSA approach that extends SysML with CFTs, we show how it supports safety experts when performing FTA and design FMEA, and we outline our design optimization approach with respect to safety.

In [24], we extended our MBSA approach to support Hazard and Operability studies, too.

In [31], we presented our lessons learned from MBSA with SysML and CFTs.

We showed how we extend SysML with CFTs.

Our approach is not limited to the static system topology, e.g., represented in SysML internal block diagrams , but also supports the architectural aspects of SysML activity diagrams by combining CFTs with call operations.

Please note that in contrast to existing MBSA approaches, we do not propose a profile that extends the SysML, but leverage JetBrains’ Meta Programming System to combine the languages of SysML and CFT in a prototypical tool.We discuss our conceptual considerations based on a set of design drivers and the suggestions of Kaiser et al. [23].

In this paper, we significantly extend our work [31] by combining it with our work [30] to support extending the methodology not only to support FTA but also FMEA and by providing a detailed case study with an electronic power steering unit that provides valuable insights of applying MBSA with industrial models.

The presented approach was realized in a tool for evaluation in two real-world automotive use cases.

This section introduces our tool and discusses the realization of particular features that correspond to our drivers –.

The modeling tool is based on the language workbench JetBrains Meta Programming System 1 that provides rich features for language design, language modularization, textual and graphical editors, as well as artifact generation.

Usage of language workbenches such as MPS in safetycritical environments have been proposed by Völter et al. [42], who justify the use of DSLs in order to introduce rigor, consistency, and traceability into development processes.

Particularly, the strong language modularization support provides the necessary means to allow integration of all relevant aspects into one language family and allow modeling these aspects in a single-source-of-truth model .

Only a very limited number of approaches we found in the literature are model-based or leverage the benefits of a DSL.

Projectional editing in MPS allows tailoring of our languages and editors to the needs of our stakeholders, e.g., different textual and graphical editors .

The generalpurpose language Java as base language in MPS allows us to integrate with the domain tools that our stakeholders are used to or required to use for creation of assurance cases .

4.1 SysML.

A complex, software-intensive system usually does not have one single notation that fits all relevant aspects.

Particularly, since different aspects may be specified by different roles at different times.

Often you want to define separate DSLs for each viewpoint, or provide different notations for different viewpoints [41].

We implemented the subset of the SysML 1.4 metamodel that is relevant for our stakeholders and use cases in MPS’ meta-metamodel, i.e., essentially blocks [32, Ch.8], ports and flows [32, Ch.8], activities [32, Ch.11], allocations [32, Ch.15], and requirements [32, Ch.16].

MPS provides features that allow to provide different DSLs for different aspects of the system.

The projectional editing feature of MPS allows to provide different concrete notations by providing different projections of the underlying abstract syntax tree.

Since all projections directly manipulate the same abstract syntax tree, no conversion needs to be done between the different notations as it would be necessary in parser-based approaches [43].

For certain viewpoints, a graphical notation is better suited, especially when dealing with relationships between system entities or some kind of data flow as often specified with SysML.

In these cases, graphical notations make it easier to derive a mental model of a system structure [36].

Textual models, on the other hand, integrate more easily with source code management and build infrastructures, e.g., merging of models.

We provide one textual DSL and two graphical DSLs for editing of blocks, ports, and connectors, corresponding to the SysML block definition diagram and internal block diagram.

Component fault trees can be edited in a textual or graphical DSL as well, see Fig. 8.

For editing of activities, we provide one textual and one graphical DSL corresponding to the SysML activity diagram.

From our experience, textual notation is faster and more efficient in the initial creation of a system, while graphical notation is better suited for understanding and maintaining a system.

Additionally, different stakeholders prefer a textual or a graphical notation.

In our use case, software developers tend to prefer textual notation for its efficiency, while safety experts tend to prefer graphical notation that is closer to the notation they are used from domain-typical tools, e.g., Isograph’s Reliability Workbench.

For our approach, we want to provide a graphical and a textual notation for the component fault trees language.

While the graphical notation is more accustomed to the safety expert reviewing it, the textual notation is closer to the component developer, in the future potentially providing the component fault tree together with the component.

An additional view that proved extremely efficient to discuss about error propagation is an integrated view that shows the graphical CFTs inside the blocks of the SysML IBDs, easing the comprehension of the error propagation through the system.

4.2 Automated analysis.

We decided for use existing, potentially certified tools for performing the fault tree analysis, instead of performing it on our own.

Advantages are the use of certified tools as well as providing the domain experts the possibility to review the analysis results in the tools of their choice and expertise, potentially integrated in established certification processes.

4.2.1 Integration of third-party tools In Sects. 3.2.2 and 3.3.2, respectively, we detail how the system fault tree and the FMEA artifacts are generated from the SysML model and the CFTs.

In order to analyze the generated fault tree, we transform it into the OpenPSA3 format, an XML-based open format for fault trees, as well as the respective configuration files for an FTA call to the open-source command-line tool XFTA4.

After the FTA was performed, we read the generated results file and display the computed reliability value at the respective output error.

The experience with our business units shows that the resulting reliability is often not as important as the minimum cut sets or the generated fault tree itself.

Arbre-Analyste is a free graphical viewer for the OpenPSA format that allows to compute the minimum cut sets on the generated system fault tree.

However, in order to use the resulting fault tree in a safety case of a product, the respective safety standards demand for qualified software tools.

For this reason, we extended our tool to not only generate the system fault tree as OpenPSA files, but additionally transform the fault tree in CSV files that can be imported by Isograph’s Reliability Workbench, which is a qualified and established FTA tool.

The automatically generated FMEA structure tree, function network, and failure network are transformed into the XML-based format of APIS IQ-RM, which is a qualified and established FMEA tool.

Figure 9 shows a screenshot of the generated FMEA artifacts in IQ-RM.

4.2.2 Lifting analysis results to the model While the usage of external tools for the analysis has its advantages, we loose the immediate feedback inside our modeling environment, which is one of our main drivers.

As a countermeasure, we want to lift the analysis results from the domain-specific tools up to the level of our models.

In case of the quantitative FTA, this means annotating the error probabilities calculated by the domain-specific FTA tools to the respective errors in the CFTs.

CFTs of blocks and activities may be instantiated multiple times in a system fault tree, though, so that we potentially get different probabilities for the output errors for each instantiation of the blocks or activities, respectively.

When reading back the results we therefore annotate a table of probabilities to the errors that shows the probability of the error for each instantiation of the CFT.

Figure 10 shows the CFT of the operation “ADCTrigger” that is called by four different call operation actions.

Thus, it is instantiated four times in the system fault tree, potentially leading to four different probabilities of the “flgADCMeasurement” output error.

The probabilities table attached to the output shows the probabilities in the context of the four call operation actions.

4.3 Usability features We have implemented several features that ease the creation and handling of SysML models and CFTs.

4.3.1 Import and export Most Bosch business units use either IBM’s Rhapsody6 or Sparx’s Enterprise Architect7 to model their systems in SysML.

In order to not manually regenerate the models in our MPS-based prototypical tool, we created importers that are able to read the exported XML Metadata Interchange file from Rhapsody or use the Java API of Enterprise Architect.

Due to the different modeling ways, we found it necessary to adapt and extend these importers for each use case we considered .

We also provide exporters to XMI in order to export modifications back to general-purpose SysML modeling tools.

However, while we are able to import a model itself, its graphical aspects and layout information are not yet considered by our importers.

As a result, the graphical representation often differs considerably from the representation in Rhapsody or Enterprise Architect.

Unless these aspects are covered properly by the importers and exporters as well, including the FTA and FMEA tools in the loop, fluent roundtrip engineering is not possible, which hinders adoption by end-users.

Additionally, the current version of our graphical representations only shows one layer of hierarchy.

We found this to be hindering discussions with the domain experts, since additional time is needed to check and adapt to the novel representation.

As long as these issues remain, we assume changes of the model are done within the general-purpose modeling tool in order to ensure the single-source-of-truth principle as well as the graphics and layouting decisions are kept.

4.3.2 Pessimistic CFT generation Since the imported SysML models often consist of a large number of blocks without an explicit failure model, we created a CFT generator that adds a pessimistic CFT to each block.

The generator assumes that blocks without any input ports contain one basic event that is forwarded to each output port.

Blocks with input ports have one OR gate that collects all input errors and propagates the result to each output port.

A first analysis of the output events of top level blocks often results in quick insights into the system.

It is especially helpful in detecting unconnected input ports or missing connections.

Since this feature allows automatic generation of the error model from a SysML IBD or activity diagram, it allows together with the analysis introduced in Sect. 4.2 a pessimistic system fault tree generation and FTA from a SysML IBD or activity diagram without additional modeling required, cf. [27].

Note, however, that this feature is only intended to accelerate the manual generation of CFTs.

It might not be helpful without manual adaptation of the CFTs and must not be used without manual review of the result.

4.3.3 Design optimization patterns In addition to the analysis of the system, we started developing a library of model transformations that optimize the system design with respect to dependability.

We follow existing approaches discussed in Sect. 2 and leverage the possibility of generating fault trees to automatically optimize the cost of the system with respect to the specified safety requirements.

In the following, we summarize this design optimization approach that was originally published by Munk et al. [30].

We propose to automatically improve an architecture model such that the probability of the top events remains below a given upper bound while the costs of that system are minimized.

As costs we consider not only the monetary costs of the hardware components but also the number of components, the size of required memory, the runtime load, the chip area, etc.; costs are annotated to the components of the architecture model.

Note that for functional architecture models the probability of basic events might be hard to define.

In this case, the functional architecture can still be automatically improved by removing minimum cut sets with only one basic event.

Our approach uses a catalog of safety-aware design optimization patterns of which each pattern is described as a model transformation on the architecture model.

For example, one optimization pattern adds dual modular redundancy or triple modular redundancy , respectively, to the system.

Another pattern replaces a component with a more reliable but more expensive, or a less reliable but cheaper variant.

The catalog of safety-aware design optimization patterns can then be shared among different projects and used for different products.

Due to the multi-objective nature of the underlying optimization problem, we use metaheuristics such as evolutionary algorithms as most suitable to find good architecture models [6].

An EA evaluates the fitness of a set of solutions .

For the next generation of that population, only a specific number of the fittest solutions are kept while the rest of the population is replaced by mutations of solutions or crossovers between two solutions.

In our case, the fitness of a specific solution is derived from the costs annotated to the architecture model as well as the probability of the top events.

Mutations are represented by model transformations while the minimum cut sets give a good indication which components of the architecture model are best to transform.

So far, we have not implemented the crossover operator.

Since different model transformations affect different cost metrics and can lower the probability of one top event but increase the probability of another top event, their result is a set of Pareto-optimal architecture models.

Thus, the presented cost optimization approach can merely indicate a set of potential architectures to a human expert who then manually chooses the final architecture from that set.

While modeling is now validated as effective for solving complex problems or developing complex systems, its widespread and systematic application has not yet lived up to expectations.

Several studies have been led to analyse the reasons of this situation and highlighted several issues, including the complexity of the modeling languages and tools [1–3] and the lack of required skills in abstraction [4].

Indeed, in many studies around Model-Based Systems Engineering adoption, usability of tools is very often stressed as one of the key issues for the adoption of MBSE paradigms.

We do agree of course that user experience of tools can be improved, and lot of people are working hard to improve including as for example new HCI or even AI-empowered assistants.

However, we are definitively also convinced that it is fundamental to assist users to face the complexity - accidental or essential - of MBSE related artefacts by augmenting the MBSE tools with self-training facilities.

In this way, it could be possible to reduce tools learning curve and encourage their users to persist with them despite the required initial efforts.

Indeed, it is to be expected that tools, languages, etc. used to solve complex problems will remain complex, even if their user experience is optimised.

As a consequence, it is important to provide additional features supporting the learning.

In this demo, we illustrate the role gamification could play to lower the entry barrier of modeling and modeling tools.

Gamification is the exploitation of gaming mechanisms for serious purposes, like promoting behavioral changes, soliciting participation, and engagement in activities.

Gamification is gaining popularity in all those domains that would benefit from the increased engagement of their target users [5].

Therefore, disparate contexts use gamification applications, such as education [6–8], health and environment [9, 10], e-banking [11], and even software engineering [12].

We present a gamified software modeling environment realized to design games for specific training/learning goals.

In particular we present how using it, the Papyrus1 modeling tool has been gamified with the objective to help students learning specific modeling aspects using both UML and Papyrus for UML.

Through a concrete case study, we demonstrate that our approach is applicable with limited effort, does not require any major modifications in the existing modeling environment, neither introduces complex dependencies.

In Section 2 we detail the main components of the gamified software modeling environment and in Section 3 we illustrate its application in a concrete software design courses where a “Hangman Game” is used to improve the learning curve.

In this chapter, we presented an approach to bridge the gap between textual and projectional LWB.

We defined a mapping between textual grammars and projectional meta models; this mapping produces the structure and editor aspects of a projectional language.

Moreover, our approach allows users to reuse textual programs by means of translating them to equivalent MPS models .

To validate our solution, we used as a case study a Rascal grammar of JavaScript .

Based on the grammar definition, we generated a projectional version of JavaScript.

To verify the correct mapping of the generated language, we successfully imported existing valid textual JavaScript programs into MPS.

In Sect. 6, we discussed some of the limitations of the current approach.

Language evolution is a crucial aspect to look at in the future.

Since the current approach assumes that the generation is done only once, we ignore the fact that the textual language and the projectional generated version might change.

Then we consider that keeping track of these changes and transferring/applyingthese changes to the other is essential.

If there are changes in the grammar after the projectional language generation, developers must regenerate the whole language, which may lead to losing information .

Similarly, this applies to programs written in such languages.

We consider that a mechanism for maintaining both versions is worth investigating as future work to keep a bidirectional mapping.

Language engineers can switch from one platform to another without losing information.

Our approach offers support for a unidirectional mapping from textual to projectional.

We believe that a bidirectional communication is required.

Because depending on the language, one may benefit more from having a textual or a projectional version of the language.

Therefore, to support both sides’ changes, we require a bridge to create a textual language from a projectional language.

Moreover, to complete the circle, a way of keeping track and propagating changes in both worlds will be required.

To avoid losing or reimplementing existing features.

As we described in Sect. 5.4, the usability of generated editors is one of the critical aspects that should be addressed in future research.

We found that we can generate editors with limited capabilities .

Therefore, we consider as future work exploring artificial intelligence techniques to improve the existing editor , maybe by identifying patterns in existing programs or commonalities in the grammar’s structure to guide or to customize the generation of the editor aspect.

Language workbenches [1] are IDEs that support engineers in the design and development of software languages [2].

These tools are aimed to improve and increase the adoption of Language-Oriented Programming .

LOP is a technique for solving software engineering problems through the use of multiple domain-specific languages [3].

DSLs are small and simple languages tailored to solve problems in a particular application domain [4].

There are two types of DSLs, internal and external [3].

The first one reuses the concrete syntax of the host language and its parser, much like a stylized library.

An external DSL, however, typically requires the implementation of a parser and compiler.

Jetbrains MPS is a projectional language workbench that obviates the need for parsing and, as a result, allows the engineer to define DSLs with a multiplicity of notations, varying from textual, and tabular, to diagrammatic, or prose-like.

MPS provides editor support that allows users to directly edit the abstract syntax structures of a language rather than reconstructing such structure from the linear sequences of characters entered in text editors.

Nevertheless, many existing languages are defined purely textually.

For instance, all mainstream programming languages are textual .

But many DSLs, like GNU Make, Graphviz, SQL, etc., are strictly textual languages too.

To make such existing languages available for use from within MPS, language engineers have to redefine the syntax of such languages using the concepts and editor features of MPS, which is a tedious and error-prone endeavor.

In this chapter, we detail an approach to take an existing context-free grammar of a textual language and convert it automatically to MPS concept definitions.

As a result, such languages can be imported into MPS without having to write abstract syntax definitions by hand.

Furthermore, the approach supports loading parse trees of existing programs into automatically generated MPS editors, so that they become available for reuse immediately.

Companies in the Eindhoven region have been using DSLs for several years [5].

Some of these companies use textual LWBs, projectional LWBs, or both, such as Canon Production Printing.

When companies are using both types of LWBs, it is often desired to reuse existing textual languages within a projectional LWB and vice versa.

If such a reuse facility exists, companies will avoid the costs of reimplementing features and maintaining the same functionality in different platforms.

Likewise, developers can be more productive from the engineering point of view and invest more time in developing new features or improving existing ones.

Finally, the reuse strategy could reduce time to market for new products.

In this chapter, we present an approach toward bridging the gap between textual and projectional LWBs, which has been implemented in the context of the Rascal and MPS language workbenches.

Our Rascal2MPS [6] takes a Rascal grammar and converts it to equivalent concept hierarchies and editor definitions in MPS.

The contributions of this chapter can be summarized as follows: A generic bridge between textual and projectional LWBs.

Employing this bridge, developers can obtain a projectional language in JetBrains MPS from a contextfree grammar written in Rascal.

A mechanism to generate projectional editors from a context-free grammar.

This mechanism uses a set of pretty-printing heuristics that takes into account the production rules’ structure.

A tool to import existing programs written in a textual language as projectional models of the generated language.

This section discusses the limitations of the approach, the rationale behind them, and possible solutions to overcome them.

These limitations are based on assumptions and constraints in the grammar.

Besides, there is also a technical limitation related to how the mapping is implemented.

1. The names of the nonterminal symbols in a grammar must be unique.

In other words, the current approach does not support the definition of two concepts with the same name.

The rationale behind this is that the name of a nonterminal symbol is used to define an interface concept in the generated MPS language, and the production labels are used to create concepts.

One way to avoid this constraint could be defining a renaming scheme that can detect and fix name conflicts.

However, this solution might introduce a side effect on the language’s usability; projectional editors use these names for IDE services such as tab completion, so they must be descriptive enough for end users.

Also, other language components must be refactored according to the renaming mechanism.

Therefore, we did not implement an automatic renaming scheme, and we preferred to include it as a limitation of the current approach.

2. In the mapping between a Rascal grammar and an MPS language, symbol labels are used as variable names, either for children or references in MPS concepts.

These names should be unique within the same concept, yet not for the whole language.

For instance, if we define concepts A and B, both can contain a reference of a child named name; however, A cannot have more than one child or reference called name.

In other words, symbol labels can be reused across concepts but not within the same concept.

3. Lexicals are a challenging concept to deal with because there is no standard way of defining them.

However, it is possible to make some assumptions on regularity and define a set of constraints to translate lexical between platforms in an automatic way, but this requires considerable effort.

As a result, we did not want to restrict regular expressions, so we included lexicals that represent MPS built-in types to the lexical library.

The current approach does not limit users from defining custom lexicals.

However, users must manually define a mapping between the custom lexical defined in Rascal and the right translation for MPS.

Section 4.1 describes the details on how to support customdefined lexicals.

4. It is required to label all the production rules and symbols within a production rule because the approach uses the labels for naming concepts or children reference fields.

A solution could be to generate placeholder names, yet this introduces other issues such as nondescriptive names and name matching issues when importing existing textual programs.

5. The current approach does not take advantage of name resolution, especially for code completion, which is a keystone for projectional LWBs.

For instance, in MPS, concept hierarchies do not rely on trees’ definition; instead, they use graphs.

6. The current implementation supports the mapping of lists and separated lists of symbols into MPS language concepts .

However, the mapping for separated lists is partially implemented.

The current approach treats separated lists just as a list.

As a result, the separator symbol is ignored for the generation of the editor.

The current approach does not support language nor program evolution.

In other words, the current approach considers languages as stand-alone units.

It does not consider that changes might happen to the language.

For example, if a developer uses a textual language A and generates a projectional language A\* inside MPS, the current approach only accepts valid programs according to A.

If there are changes to the original language A, those changes cannot be patched in the generated versions.

This forces to regenerate the whole language from scratch or make changes by hand.

Some changes do not break the importing of programs: Addition of language constructs to the grammar and then using them in a program.

This means that the plug-in for importing programs, ImportProgram , will not find such elements.

As a result, the plug-in notifies the user.

Modification of existing language constructs .

As expected, this type of change often ends up in a failure.

In sum, language engineers and users, in general, should be aware of the language’s version and the version used to define programs.

We see this problem as an opportunity for future extensions of the current approach to supporting languages and programs’ evolution.

3.2 Syntax of Textual and Projectional Languages.

As mentioned before, a software language’s syntax is a set of rules that describe valid programs [2].

Usually, it is divided into two, namely, concrete syntax and abstract syntax.

In this subsection, we describe how different LWBs represent both types of syntaxes.

In textual LWBs, a language’s concrete syntax is usually specified using ContextFree Grammars , while in projectional LWBs, the concrete syntax is expressed as AST projections.

Below we explain both approaches and highlight their main differences.

To clarify the differences between textual and projectional LWBs, we will use Rascal and MPS.

Table 1 shows a comparison of the notations used by these two platforms to define language’s syntax.

Context-Free Grammars A CFG is a formalism for describing languages using recursive definitions of string categories.

A CFG C is a quadruple:

C →

in which S is the start symbol , NT is a set of syntactic categories also known as nonterminals, T is a set of terminal symbols, and P are production rules that transform expressions of the form V → w.

V is a nonterminal , and w could be zero or more nonterminal or terminal symbols ).

For example, a CFG that describes the addition of natural numbers N is shown

below:

G =

The production rules P are defined as follows:

start → Exp

Exp → Number

Exp → Exp + Exp

Number → i

By applying the previous production rules, we can write the arithmetic expression

a + b as:

start → Exp

Exp → Exp + Exp

Exp + Exp → a + Exp

a + Exp → a + b

a + b

Once there are no more nonterminals , we cannot rewrite the expression a + b because there are no production rules that can be applied.

We say that a program is syntactically valid if there is a derivation tree from the start symbol to the string that represents the program.

For instance, the concrete and the abstract syntax of the language described above can be implemented in Rascal, as shown in Listings 1 and 3, respectively.

The first one defines two nonterminals, namely, Exp and Nat.

The Exp rule contains two productions, for literal numbers and addition.

The Nat nonterminal defines natural numbers.

AST Listing 3 defines an Algebraic Data Type that captures the structure of the language with two constructors: nat and add.

The terminals of the expression grammar are represented using built-in primitive types of Rascal .

Syntax in Projectional LWBs In a projectional LWB, the syntax is also divided into its concrete and abstract representation.

The concrete syntax corresponds to an editor definition, whereas the abstract syntax is defined in a concept hierarchy.

Projectional editors do not share a standard formalism for defining abstract syntax; therefore, each platform provides its own formalism.

MPS uses a node concept hierarchy [14].

For instance, the AST representing a language for describing the addition of natural numbers is shown in Fig. 1.

The MPS implementation uses an Expression interface and two concepts, namely, Addition and Number.

To represent integer numbers, we use the built-in integer data type.

How the users will edit expressions of this kind is defined by an editor definition.

However, MPS also offers a generic reflective editor, so that every concept in MPS comes with a default editor.

A reflective editor is a projectional representation of an AST that developers can use out of the box.

An example of an arithmetic expression program using the reflective editor is shown in Fig. 2.

SpecEdit provides a solution that unifies the two existing syntaxes of TLA+ and eases the work of engineers.

To do so, not only was it necessary to formalize a new model of the language but also was it crucial to work on the input modes provided to the users.

Not yet a full-fledged IDE, SpecEdit nonetheless is a viable TLA+ specification editor, as illustrated on the Elasticsearch case-study.

It is meant to epitomize what can be achieved through projectional technology for improv ing the experience of systems engineers using specification languages such as, but not limited to, TLA+.

Some future research directions include the use of modelfederations [19] to ensure the traceability of textual requirements translated into TLA+ specifications via SpecEdit.

We are also considering implementing various projections for TLA+ specifications, based on tabular/graphical Editors.

To reach the goal of automatically analyzed requirements, a formalization is required and the resulting expressions must be closer to the expertise domain of the engineers.

As noticed by formal methods users on industrial cases [1], formal specification based on simple discrete math with basic set theory and predicates notation is quite familiar to engineers.

Numerous research efforts address this challenge through mathematical formalism equipped with powerful tools for reasoning and handling specifications [2].

Temporal Logic of Actions is a formal specification language created by Leslie Lamport [3].

TLA+ defines a temporal logic based on set theory that facilitates the specification of dynamic systems.

TLA+ specifications are amendable to formal verification either through model-checking via TLC [4] or theorem proving using TLA+ Proof System [5].

Particularly well adapted for the specification of distributed systems, TLA+ has been successfully used in both academia and industry.

Amongst its industrial uses, we can cite its usage in production, for capturing the design requirements of todays most influential cloud infrastructures, S3 from Amazon [1], Azure from Microsoft [6] and Elasticsearch [7].

The transition from natural language requirements to formal specifications is not always as smooth as end-users might want it to be, even when provided with adequate analysis tools.

As observed by Green [8], the alignment of the language with the domain greatly influences the ability to effectively express facts in that domain.

In the case of TLA+, the language concepts and semantics are directly mapped to mathematics, offering the premises for both formalization and automation.

However, at the syntactic level, TLA+ suffers from the duality of its syntax, which introduces a gap between the conceptual view of the specification and its encoding in ASCII.

According to Green, this duality entails an arduous cognitive dimension [8] for the specification designer.

Moreover, TLA+ is targeted at non-programmers.

Leslie Lamport answered in 2014 to some user’s feedback [9]: “As for the ”pretty-printed” version versus the ASCII, a TLA+ user at Intel wrote that one of the good things about TLA+ is that if he doesn’t understand what a TLA+ construct means, he can look it up in a math book.

Math books don’t write math in ASCII, they use standard mathematical symbols.

I want TLA+ users to think in terms of math, which means thinking in terms of its symbols.

You will soon get to be bilingual, reading math and its TLA+ ASCII versions equally well.” He indicated at the time that it is inconceivable to give up the mathematical affiliation of TLA+, and that it is therefore inevitable to keep the mathematical notation in parallel with the code written in ASCII.

This leads us to the following research question: Is it possible to hide TLA+’s syntax duality in a viable bilingual Integrated Development Environment to reduce the mental efforts of system engineers? Such an IDE would expose only the mathematical syntax to the user, translating it to the ASCII version for ensuring compatibility with the existing tools.

At first glance this does not seem so trivial because the input device we use, the keyboard, does not allow for the direct input of special characters, like ∈.

Furthermore, the use of the Unicode ID or of complex key combinations needed for writing the corresponding Unicode characters are no better solution.

This paper introduces SpecEdit, an IDE with a projectional editor for TLA+ that solves this problem.

SpecEdit lets the designer use standard mathematical symbols in the specifications.

This approach is meant to minimize the mental effort and streamline the formal system specification process.

We illustrate the benefits of our approach using the openly available specification of the cluster coordination of Elasticsearch.

We furthermore assess the complementarity of SpecEdit with respect to the existing TLA+ tools and emphasize some of the advantages of projectional editors for writing formal requirements and specifications.

The integration of plaintext support in SpecEdit ensures that the benefits of projectional editing are maximized while guaranteeing the compatibility with legacy tools.

The major advantages of the projectional approach and MPS are to be able to dissociate the model from the view and to have a high composability of the language definition modules.

Merging the projectional approach with parsing nevertheless bears its limits.

The concern is that each time the underlying structure of the language is modified both the projectional editor and the ANTLR modules are impacted.

TLA+ however, is a mature language with a stable syntax, thus, the underlying abstract syntax is considered stable.

Since the ANTLR grammar maps the parse-tree to the TLA+ Concepts, the Editors can be modified without creating conflicts with ANTLR modules.

The projectional approach is not the most widespread approach in code editors nowadays.

It however has the potential to play an important role in the future to map DSL definition to any syntax.

This approach efficiently solves the difficulties of translating requirements into formal specifications.

In the case of SpecEdit, the use of MPS as a backbone for the creation of a tool dedicated to TLA+ has proved particularly fruitful.

Not only does MPS offer a complete customizable architecture, but also provides access to non-trivial mechanisms for advanced users.

SpecEdit transcends the current duality of the syntax of TLA+ and promises improvements that will facilitate the daily work of systems engineers to bridge the gap between conceptual view and syntax.

The transition from one tool to another is not an easy matter in daily professional life.

The projectional editor, though self-sufficient, combined with a traditional parsing approach, addresses this concern in the specific context of TLA+ by providing a bridge between SpecEdit and the existing TLA+ tools.

This section overviews the architecture of SpecEdit discussing the creation of a basic projectional editor as well as the improvements needed for a better user experience and compatibility of the IDE with existing tools.

A. SpecEdit’s Architecture

SpecEdit solves the syntax duality problem of TLA+ by providing an IDE with a projectional editor integrating modern features such as syntax highlighting and autocompletion.

Furthermore, SpecEdit offers a flexible backend that preserves the compatibility with the existing tools.

An overview of the high-level architecture of SpecEdit is shown in Figure 1.

In the Figure, the core components of SpecEdit are emphasized by the green and blue boxes .

The Projectional Editor displays the TLA+ model that is instantiated based on user input.

As usual, the model conformsTo the metamodel.

For SpecEdit, we created a syntax-driven TLA+ metamodel, to finely capture user intentions.

SpecEdit supports Free editing and Reference resolution, besides other features inherited from the underlying language workbench, such as syntax highlighting and refactoring .

The backend is composed of the PasteHandler, the Writer, and the Reader.

The PasteHandler enables on-the-fly conversion from ASCII TLA+ specifications to instantiated TLA+ model nodes.

The Writer represents the serialization modules, which, based on the Persistence model, outputs different file-based representations of the model.

The Reader represents the file-input modules, which can instantiate the TLA+ Model according to the Persistence model .

SpecEdit supports copy-paste, both internally by exchanging AST nodes through the Writer-Clipboard-Reader path, and externally by processing plaintext coming from the system clipboard via the PasteHandler.

SpecEdit backend supports TLA+ ASCII files and an XML-based serialization format .

SpecEdit is implemented with the MPS language workbench, a mature, commercially supported technology for DSL design, which facilitates the creation of IDEs with a projectional editor.

The first step towards the creation of an IDE with a language workbench, such as MPS, consists in the definition of the language with the tools of the workbench.

Section III-B describes the process of defining TLA+ as a Language in MPS.

Projectional editors rely extensively on the abstract syntax, and allow direct editing of the underlying model.

Users do not write code, but they instantiate metamodel elements via numerous specialized editors.

MPS gives the possibility to use the completion menu to select a new node to be instantiated.

MPS will thus show, in the menu, only nodes that can be instantiated at the cursor position.

However, this process is slow and very frustrating when not combined with free editing.

To overcome this limitations, SpecEdit implements free editing support on top of the MPS specialized editors.

Free editing, combined with Reference Resolution , optimizes the user experience.

By default, MPS saves models as XML files, which are easy to parse and manipulate programmatically.

However, to ensure the compatibility with the existing TLA+ tools, two custom modules were created, namely a PasteHandler and a custom ASCII persistence model .

B. Basic Projectional Editor for TLA+

Since MPS frees programmers from defining a grammar for their languages, the MPS Structure Language is provided as an alternative.

Concepts in MPS, defined via the Structure Language, represent the abstract syntax and they reference children nodes, parent nodes, properties , etc.

To define the underlying language metamodel in MPS, the TLA+ grammar was converted into a set of MPS Concepts.

The extraction of the language Concepts from the concrete syntax resulted in a metamodel with 110 interconnected Concepts.

For the definition of the concrete syntax, each Concept was associated with a specific MPS Concept Editor.

From the model-view-controller design-pattern perspective, these Concept Editors play the role of “views” and “controllers” for the associated language Concepts .

An Editor is described by the cells it is composed of, like a template.

Figure 2 illustrates an Editor for a TLA+ Module.

In blue are represented read-only fields.

ModuleName designates a mandatory string that must be entered by the user, SetOfMod- uleNames optional strings and SetOfUnits a set of expressions and statements that may constitute the TLA+ Module.

Since Unicode characters are supported by MPS, they were used to customize the Editors and thus to obtain a rendering that integrates the mathematical notation.

In Figure 3, the case branch is automatically instantiated, by the customized Editor, with an → as a read-only field.

To allow users to insert symbols, in fields that accept inputs , MPS Enums were used.

Enumerations in MPS allow to define properties with values from pre-defined sets.

The possible values are shown, in a context-menu, if the given values can be inserted at the current cursor position according to the metamodel.

Figure 4 shows the mapping between the TLA+ grammar rule ”PrefixOp” and the corresponding MPS Enum node as well as the rendering in the completion menu when prompted to users .

Note for instance the suggestion of the <> symbol, representing an eventuality in temporal logic, instead of the <> string.

C. Customization of user experience

This section discusses the solutions implemented in SpecEdit for free editing and reference resolution support.

1) Free editing: In MPS, by default, free editing is prohibited.

This means, that, unless the user is given a predefined box, users cannot use the keyboard to write code.

To allow free editing, programmers have to make fields editable in MPS Editors.

Once editable, users will be able to write freely with their keyboard but MPS will point to syntax errors since writing with the keyboard does not instantiate new Concepts.

To solve this issue, predefined strings were mapped to the Concepts as aliases .

Not only do aliases appear in completion menus and context assistants but MPS also instantiates the corresponding Concepts in the current model the user is editing when the alias or a part of it is typed in.

[!!!!!!!!!!!!!!!!!!!!!!!!!! GOT HERE !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!]

Another editing issue is hiding optional model elements, when they are not needed.

For instance, in TLA+, a module can optionally extend other modules.

In Figure 2 the optional block is shown as a set of cells preceded by a question mark.

In the case of a standalone TLA+ module, the ”EXTENDS” clause should be hidden.

However, if it is hidden, there is no mechanism defined to allow users to make it appear and thus to import modules.

In MPS this user intention is captured through ”display conditions” and ”side tranform actions”, which allow to implement specific mechanisms in the editor.

Figure 5 illustrates this principle.

A Transformation Menu can be invoked by a user typing “EXTENDS” right to the cell to which it has been associated.

It leads to the instantiation of a new element in the list of extended module names.

The condition in the “show if” property is then verified, and the hidden field is unhidden.

2) Reference Resolution: Syntax predictions are an asset to be used in parallel with free editing.

Predictions are not only meant to suggest AST nodes to instantiate but also strings to fill in editable fields.

Reference resolution is one of the functionalities meant to enhance the predictions provided by the IDE.

In MPS, a reference creates a link between two nodes of the AST outside the tree containment hierarchy.

For instance, a “VariableDeclaration” node is contained in a ”Module”, but can be referred to, using a “VariableReference” node, from any following definition.

The principle is thus to use pointers in Editors targeting the variable declaration identifier.

The main goal is to improve user experience by automatically providing the identifiers defined in the specification scope.

Figure 6 shows that when editing the ”CanMove” definition, the user is presented with the previously defined variables A and B.

Furthermore, connecting the concepts through references enables refactoring transformations, such as renaming, which applies seamlessly to all occurrences of the reference.

D. Plaintext support with ANTLR

This section discusses pasting and file-loading and saving in SpecEdit; three simple actions which need special attention in projectional editors.

1) Paste handler: When a user tries to paste an element into the IDE, a background routine is in charge of retrieving the clipboard content and inserting it in the code editor.

This happens transparently when working with text.

In a projectional editor, however, the text from the clipboard should be first parsed and interpreted to instantiate the corresponding nodes in the model.

SpecEdit uses the ANTLR parser-generator framework and a TLA+ grammar definition to process plaintext.

ANTLR builds a parse tree and generates a skeleton visitor class containing methods for traversing parse trees.

In the case of SpecEdit, the corresponding visitor class is subclassed and each visit method overridden to instantiate MPS Concepts.

The TLA+ Concepts defined are programmatically accessible within MPS.

As opposed to regular transpiling, in the case of SpecEdit, the input and output language are the same.

Their respective definitions are however different .

To integrate the modules generated with ANTLR into MPS and consequently into SpecEdit, a Java archive containing the compiled code and the various dependencies was created and imported under a new solution in MPS .

Based on this process, we created an MPS plugin inserting a new entry in the context menu.

Thanks to this plugin, when the user clicks on the new entry in the menu, a method retrieving the content of the clipboard is called and checks that the retrieved text verifies a given pattern.

The text retrieved from the clipboard, which\ is tokenized, parsed, visited and mapped with MPS Concepts allows the instantiation of an AST.

2) Custom persistence model: By default, models in MPS are saved in a proprietary XML-based format.

The idea we had was to create a custom persistence model allowing to remove any formatting specific to the IDE in order to save TLA+ source code files directly in plaintext and ensure the compatibility of SpecEdit with the existing TLA+ tools.

This implied modifying the reading and writing procedures of the IDE.

The approach followed is very similar to the one used for the realization of the paste handler.

It involved using the modules generated using ANTLR to ensure the transpiling.

As in the case of the paste handler, a new dedicated MPS plugin build solution was created that imports a custom persistence model.

Classes were created to override the persistence logic, encapsulate the parsing and visiting procedures and implement the different interfaces that are essential for dealing with the internal working mechanisms of MPS.

[16] Apart from the import source, which is a file and not the clipboard, the processing principle for loading ASCII TLA+ files is exactly the same as for the paste handler .

Note however that we created an explicit plugin descriptor for MPS to be aware that this plugin provides a model factory.

The writing procedure is delegated to text generators implemented via MPS TextGens.

The two import approaches selected, namely pasting and customizing the persistence model, were chosen because of their respective merits.

While importing unstructured models in MPS is essential from a user’s point of view, being able to insert pieces of TLA+ code from external editors into ongoing MPS projects is also useful.

Finally, the basic projectional editor, introduced in Section III-B, once tuned-up, became SpecEdit, a TLA+ IDE which solves the syntax duality problem of TLA+ without compromising either the user experience or the compatibility with the existing tools.

As language technology is complex, it is crucial to keep the incidental complexity of the tool used as low as possible.

MPS might not be the best tool in this regard, but it can be used in a way that lets students grasp the essential concepts.

This works out if the teaching setup is aligned with the features of MPS and introduces functionality step by step.

This approach is aided very well with the stability and adequate documentation of MPS such that students get all the information they need.

With this approach, students can understand the underlying concepts, and thereby they master the tool MPS.

Still, there are some serious shortcomings of MPS with regard to teaching.

Fixing them might even help the general applicability of MPS.

The work on appropriate meta-languages is a significant part of this improvement process.

5 Experiences and Evaluation. Having used Eclipse and Microsoft Studio and MPS in teaching language handling, there are several remarks to be made about the suitability of MPS for teaching and the pitfalls for the teaching situation.

Please note that we use MPS in the specific context of students, being novice programmers and not experienced in language design.

By running the language handling course, the following experiences were gathered.

Students are no experienced developers.

MPS may work well for experienced developers, who use many keyboard shortcuts regularly.

For those, MPS feels very natural.

Students are novice programmers, and they most often try to write some text and copy-paste existing specifications.

This is often tricky or impossible with MPS due to its projectional nature.

MPS shows best practice.

It is good to use a running example where aspects are added to complete a simple sample language.

It is also beneficial to cover all language aspects within one platform.

MPS has the advantage that the definitions of all MPS meta-languages are accessible in addition to several sample languages.

This allows copying from best practice examples.

The theory comes before tools.

The understanding of the concepts of language design is strengthened by showing their implementation in MPS.

However, students tend to drown in the tool details of MPS, which hampers their understanding.

It is often easier to start with the high-level theoretical concepts before showing the implementation.

MPS is heavy.

MPS is a heavy tool to use in teaching.

The learning curve is very steep, and students take a long time to get used to the tool.

There are very many details, and for a novice, it is not easy to see what is essential and what not and where to look for a place to change unwanted behavior.

We try to limit the complexity of the projects by focusing on the concepts mentioned in Sect. 3.

Distinguishing languages and specifications is tricky.

In MPS, both languages and meta-languages and even specifications are shown in the same way and in the same editor window; see Fig. 10.

They are also represented internally in the same way.

This is a challenge for students as they need to understand the difference between languages and specifications.

The teaching tool LanguageLab makes it easier to see this difference [14].

Learning MPS is possible.

Despite the heavy tool and the heavy tasks, the students consistently report that they learn a lot in this course and that they can use this in their future job.

This is visible in the results from the course, which are good grades and decent languages in most of the cases.

After the course, the students have a good understanding of language handling and how it can be used.

From our experiences, there are several possible improvements for MPS which are as follows.

Most of them relate to the complexity of the tool and the associated steep learning curve for new users.

There are plugins for some of the points mentioned, which should be included in the standard version of MPS.

Have a simpler starting user interface.

It would be a good idea to adapt the user interface to the experience of the user.

A set of essential features could be a starting point for a novice user, and then the user interface could grow in line with the new experiences of the user.

The concepts presented in Sect. 3 would be a good starting point for the essential features.

Restrict expressivity.

Currently, most of the MPS meta-languages have a procedural core that allows expressing everything computable in the sense of Java.

However, in many cases, a higher level of abstraction would restrict some functionality but would increase precision.

A good example is the handling of type systems.

A bad example is the behavior aspect which collects everything that does not fit somewhere else.

Have a web version of MPS.

It would be excellent if students could work in a version of MPS online, including shared documents.

This would also improve the teaching process a lot.

Besides, it also simplifies version handling and migration.

Provide an overview of the structure.

The individual definition of concepts in MPS gives a lot of freedom, but it is easy to lose the overview.

Class diagrams are an excellent way to present such an overview.

These diagrams could be generated from the structure definitions.

For large languages, an overview of the concepts is essential, and support for this is needed.

Improve meta-languages The current meta-languages are not always the most abstract languages to express the needed information.

It would be essential to have grammar cells in the core of MPS and even improve on this idea and introduce more high-level patterns.

The second area of improvement of metalanguages would be in the area of name binding, as described in [29].

It might be possible to create a version of MPS using simpler meta-languages by using bootstrapping as explored in [41, 42].

Provide a decent meta-language for execution.

This requirement might not be most pressing for practical application, but it is essential for teaching in the area of language processing.

As the examples of MPS show, interpreters are useful, and a good meta-language should be available.

Model-driven development has created high hopes for easier systems development and shorter development cycles [27].

The central idea still holds: If we can lift the level of abstraction, such that we see the relevant information of a problem and its solution, then the design of solutions becomes much easier.

Besides, it is possible to discuss the solutions with the experts.

In reality, however, the results were not too promising, because the standard modelling language was chosen to be UML [37]; because models were used as illustrations, and not as specifications; and because of missing or immature tool support.

Therefore, many programmers abandoned modelling.

Modelling can be connected to the expertise when the language used for the model is understandable for the experts, i.e., the language has to be a domainspecific language [10].

It is important to use full languages, not only notations without semantics.

This means DSLs need to be executable in order to be useful for modelling, such that they essentially are high-level programs [4, 30, 54].

In this spirit, languages like SDL [23] and executable UML [31] present a high level of abstraction together with executability.

There are attempts to add formality and executability to the OMG MDA framework [6].

This book presents examples of such DSLs, and this chapter looks into how DSLs can be included in computer science teaching.

For systems development, modelling is essential.

Modelling means to develop high-level descriptions of the problems and the solutions.

These descriptions have to translate into running systems that can be used to experiment with the problems and the solutions until a satisfactory result is achieved.

This is only possible if the languages used are formal, allowing to express the important information concisely and formally.

Out of such descriptions, programs can be derived—either manually or automatically.

In this chapter, we follow an MDD approach that values formality and complete automatic code generation.

The idea of changing the generated code afterward has been abandoned for compilers because it did not bring too good results.

Nowadays, developers rarely touch the code generated by compilers.

The same should be valid for code generated within MDD.

In this view, MDD is closely related to domain-specific languages , as it is easiest to write concise models using a concise language adapted to a domain [25].

This way, the complexity of the domain is reduced and captured in the concepts of the DSL.

A domain-specific language is a textual or graphical language with abstractions optimized for a domain and with well-defined semantics [53].

A DSL may be preprocessed, embedded, or transformed into other languages for execution, instead of being compiled to machine code using a traditional compiler.

Because the development of DSLs uses high-level descriptions, it is based on the same principles: the language handling tools are generated from high-level descriptions; see [3, 8].

This means MDD is used to define these types of languages.

An important aspect of this approach is to provide the language designer with support for rapid development and automatic prototyping of language support tools and allow for working on a high level of abstraction.

This way, the language designer can focus on the language and use the language definition to generate tools such as editors, validators, and code generators.

A related aspect is the ease of developing DSLs.

Ideally, languages should be put together in a plug-and-play fashion using best-practice language patterns.

This flexibility is achieved by language modularity and the ability to reuse existing languages, allowing language extension and language reuse [51].

Despite the importance of domain-specific languages and the tooling for them, many universities still teach language handling with the main focus on compiler theory.

For example, in Norwegian universities, there is a strong emphasis on compiler theory and little or no focus on meta-modelling in most of the available computer language handling courses [12, 13].

In contrast, we use an approach to teach DSL technology together with MDD technology under a framework of meta-models and generated code, still under the umbrella of computer language handling.

This allows for shifting focus from compiler development to meta-model-based language design and definition.

The primary purpose of this article is to share experiences from teaching metamodel-based language description and to discuss how the tool MPS [24] helps in teaching.

We will also discuss meta-languages for covering the different aspects of a language definition when teaching computer language handling.

The chapter presents a course run at the University of Agder the last 10 years and the experiences with the tools and the learning.

The article will also discuss the course content and design.

We present below a summary of lessons learned based on our experience in teaching language engineering with MPS in industry and academia: L1.

Low ceremony free courses attract the participants who only want to test the waters before fully committing to learning the technology and who would turn the technology down without trying otherwise.

Going online has increased the reach of the courses by an order of magnitude.

Paying attention to step-by-step guidance is crucial even at later stages, since the participants tend to skip unclear steps or forget things, if not repeated several times in different contexts.

Teaching MPS in a wide context of other language engineering tools, such as language workbenches Eclipse Xtext, Spoofax, and Rascal, gives students a fuller picture of this area.

Most importantly, by the time when MPS is introduced in the course, students have already seen what language implementation is comprised of and acquired some language engineering terminology.

This enables concentrating on implementation techniques that are distinctive in MPS.

Explaining projectional editing by discussing analogies among both nonprogramming and programming-related environments seems to facilitate perception of MPS projectional editor.

An analogy with an equation editor in a word processor enables introducing object-oriented view on language concepts and the notions of concept aliases and side transformations.

Explaining code generation with XML-based examples and XSL transformations allows the students to explore model-to-model transformations.

Availability of tools to run XSL transformations, textual representation of XML trees, and powerful mechanisms of XSLT facilitate students’ experience.

Using MPS to extend an existing general-purpose language seems to interest students as they can see an immediate quasipractical use of the acquired language engineering skills.

Covering at least the basics of domain engineering motivates the participants to learn the practicalities of the concrete tool.

It will also help them imagine how the principles and tooling could be applied to their projects and their infrastructure.

Language design concernsshould be explored as part of the initial discussions over nontrivial example projects.

The pros and cons of the alternative approaches should be presented as well as their consequences to other parts of the project.

Language-oriented programming is always concerned with two different artifacts: the language description and the solution description.

When we consider the OMG four-level architecture as shown in Fig. 1, then the language description is placed on level M2,7 while the solution description is placed on level M1.

There are also different roles connected to these two levels: a language designer works on level M2 and a solution designer works on level M1 The course on generative programming is mostly related to language designers and handles the tools and mindset needed to create languages and associated tools.

However, a good understanding of solution design and architecture is an essential precondition to becoming a good language designer.

Language designs are essentially also solution descriptions—in a very limited domain.

Here, we need language descriptions that lead to language tools.

Language descriptions describe languages completely with all their aspects.

Meta-modelling has often stopped at incomplete language descriptions consisting only of structure and constraints .

However, a language description has more aspects; in particular, concrete syntax and semantics have to be considered [26].

In [35], a language definition is said to consist of the following aspects: structure, syntax, and semantics .

Structure, also called abstract syntax, coincides with a narrow understanding of meta-model.

It consists of two sub-aspects, namely, the definition of possible concepts with their connections and the restriction of those using constraints.

Sometimes, a restriction could be expressed as a structure or the other way around.

Syntax stands for concrete syntax, and it defines how instances of the language are shown.

This can be the definition of a graphical or textual concrete language syntax, or something in between, like tables, diagrams, or formulas.

Behavior explains the semantics of the language.

This can be a transformation into another language , or it defines the execution of language instances .8 These aspects are not always as strictly separated as they seem in the illustration; constraints are shown as overlapping with structure since constraints interact closely with the structure-related technologies in building up the structure of the language.

However, constraints can also be used for defining restrictions for presentation as well as behavior.

The structure is the core of the language; it contains the concepts that should be part of the language and the relations between them.

A meta-model-based approach to language design focuses on the structure.

A welldefined language structure is the starting point to define one or more textual or graphical presentations for the language, as well as to define code generation into executable target languages such as Java.

MPS features a large set of meta-languages, and some of them match the aspects shown in Fig. 2.

Section 4 provides more details about the MPS concepts used in the course.

The definition of possible concepts in the structure aspect is handled by the structure meta-language.

At the same time, restrictions can be expressed using the constraints and the type system meta-languages.

The editor meta-language handles the syntax aspect covering both text syntax and diagrammatic syntax.

Transformation semantics is handled by the textgen meta-language for modelto-text transformations and by the generator meta-language for model-to-model transformations.

MPS does not support execution semantics; it can be captured using parts of the action meta-language.

In addition, MPS allows influencing the user appearance of the generated IDE using the intentions, the refactorings, and the usages meta-languages.

Moreover, MPS allows using low-level implementation details in the language design, for example, using the actions meta-language.

Finally, MPS provides means to describe tool-related information, for example, how the described IDE is built.

These parts are not essential in a language design course.

Domain-specific languages are always about the correct abstraction level.

They enable to express the knowledge of the domain.

A DSL can describe a complete solution on a suitable abstraction level.

Of course, it is necessary to know the domain well to come up with a good domain-specific language.

As meta-languages are also domain-specific languages, the same is true for them.

They have to be on the correct abstraction level.

Since the domain of language descriptions is relatively new, there are only a few good patterns of language description available.

Most often, the implementation is guiding the concepts provided because language designers are often solution designers knowing how to write a code.

An essential part of this course concerns finding a suitable abstraction level to facilitate code generation from models.

In this respect, tools for language description are used as an example.

Therefore, it is essential to have excellent highlevel abstractions available and explain how these are translated into low-level code by the tools.

However, it is a challenge to find tools and technologies that work on a high abstraction level for each language aspect.

If the abstraction level is too low, too many seemingly irrelevant details will create complications and complexities, making it more difficult for the students to get started with the tools.

On the other hand, if the abstraction level is too high, it may not be possible to generate working tools from the language specification.

For structure and textual syntax, some meta-languages provide a suitable level of abstraction, while it is more difficult to find the right abstractions for the other language aspects.

4 Using MPS Meta-Languages for Teaching.

MPS has many meta-languages, and not all of them are useful for novices.

Here, we look at how MPS handles the essential concepts introduced in Sect. 3.

4.1 Concepts for Teaching Structure.

Given the concepts for structure in the previous section, EMOF is a clear candidate to fulfil all the requirements.

Full MOF [38] could also be used, but it includes a lot of advanced concepts that are overkill for students.

The structure meta-language of MPS provides all the needed constructs and some more that are not needed for students and that may disturb the understanding.

MPS is missing an overview of the concepts of the language with their dependencies, as it is easily provided with EMOF class diagrams.

Such an overview is helpful in case students start to work on an existing project or in general need to get an overall understanding of the concepts involved.

Figure 3 shows how an MPS concept declaration covers the needed constructs in relation to an EMOF class diagram.

MPS also allows defining enumerations.

4.2 Concepts for Teaching Constraints.

Constraints are restrictions that are checked on the syntax tree after checking the restrictions of the syntax.

The traditional method to specify constraints is OCL [56], which allows formulating expressions over the abstract syntax.

This is not very domain-specific for the language definition.

A better approach exists for the handling of name resolution in [29].

The structure meta-language captures multiplicity and also allows to define lexical rules using constrained data types; see Fig. 4.

MPS has a general meta-language for constraints covering general constraints including uniqueness and scoping.

In this context, also the behavior meta-language is used.

The general MPS idea of the constraints language relates to a syntax check, where an input that does not match the constraints is not included in the model.

It is possible to define correctness in the sense of static semantics using the type system checking rules.

In the course, the use of checking rules is recommended for general constraints instead of using the constraints meta-language.

The type system meta-language gives a very high-level way to describe types and their connections using inference rules, subtyping rules, and checking rules as shown in Fig. 5.

The most significant abstraction gap is in the handling of name resolution.

MPS handles definitions using INamedConcept, but this is very rigid and cannot be defined after the structure is in place.

References are defined as reference constraints and are code-level, not high-level.

The same is valid for uniqueness.

4.3 Concepts for Teaching Syntax.

There are two main approaches for handling concrete syntax of a language: generation and recognition .

Editors would typically do both sides.

The editor meta-language of MPS allows the definition of projectional editors, i.e., editors built on the idea of model-view-controller .

This means that the tool always knows about the elements of the specification, and therefore ambiguity is considered unimportant.

However, as [21] argues, ambiguity might not be essential for the tool, but still for the user.

A check of the ambiguity of the notation could be useful.

handled in the course using separate tools .

Still, the handling of lexical constraints can be used as an example of recognition; see Fig. 4.

MPS editors support a two-way connection between the syntax and the corresponding structure, providing feedback from the syntax analysis in the form of syntax highlighting, error messages, code completion suggestions, etc.

The projectional nature of MPS is well aligned with teaching graphical and textual projectional approaches.

Even though MPS does not provide full graphical editors, the principles are clear enough as can be seen in Fig. 6.

MPS does not allow much notational freedom for the user, so this has to be explained separately as well.

The inspector view of MPS can be used as an example of user freedom, which is not visible in the main notation defined; see Fig. 7.

MPS allows defining editors automatically from the structure as proposed in [22].

This gives a quick win, even though the generated editors often have to be adapted.

Providing high-quality editors in MPS is very advanced and well beyond the capacities of ordinary students.

Here, a much higher level of abstraction would be needed, as shown in [5, 47, 55].

In particular, the division of syntax between the inspector and the regular editor is puzzling for the students and disturbs their understanding.

In the course, it is recommended to avoid using the inspector view when defining a textual representation.

4.4 Tools and Technologies for Teaching Transformation.

There are many tools available to express transformations, and MPS provides a very high abstraction level for transformations.

MPS allows both template-driven and data-driven definitions.

The meta-language generator allows the definition of model-to-model transformations, as shown in Fig. 8.

In contrast, the meta-language textgen provides means for the definition of model-to-text transformations, as shown in Fig. 9.

Model-to-model transformations are simple in MPS, while model-to-text is a bit less convenient.

In particular, the description capabilities are too different between the two kinds of transformations.

4.5 Tools and Technologies for Teaching Execution.

MPS does not provide dedicated meta-languages to handle execution.

There are some possibilities to define debuggers.

Moreover, it is possible to define simulators using the underlying base language .

High-level descriptions of operational semantics as proposed in [33, 40, 44–46, 48] are missing.

Besides, state transitions could be defined on a higher level using ASM [7] or QVT [36].

In the course, the concepts of executions are introduced, and possible implementations in MPS are discussed.

This course uses a tool to support learning.

However, immature or overly complex tools and technologies can demotivate students and, in some cases, even make them avoid meta-model-based projects.

Students need stable tools with good documentation and easily understandable meta-languages.

The tool has to be conceptually clear in its underlying platform.

Finally, it must be usable for novices, as our students are inexperienced developers; they want to copy and paste program text.

Currently, such a tool does not exist, and when the course started, the situation was even worse.

Tools are not designed for teaching, and it is very challenging to develop a neat tool with industrial strength at a university.

Moreover, at our university, we want to use a public-domain tool.

So we need to compromise and select an existing tool.

As MDD is very close to meta-modelling [3], in principle, the choice can be made related to MDD tools as well.

The Eclipse infrastructure [9] around xText [5] is the first choice in this area, in particular connected to EMF [50].

We also looked into Microsoft Studio [32] with its DSL package.

Another candidate would be Rascal [28] among many university-based tools.

In the evaluation of these tools, university-based tools usually are not stable enough and provide little documentation.

Microsoft Studio provided good integration, documentation, and ease of use but had a somewhat limited selection of meta-languages.

Therefore, with Microsoft Studio students had to work on a relatively low level of abstraction.

Eclipse with xText had good applicability and also a rich set of meta-languages.

However, the problem with Eclipse was its general stability and sparsity of documentation.

The meta-languages did not fit together; they changed in short cycles, and consistency between different packages was a nightmare.

Changes of plugins during the course were likely.

MPS somewhat combined the advantages of Eclipse and Microsoft Visual Studio, being both integrated and high-level, stable, and user-friendly; see also [15].

All of the other tools failed in being able to handle complete definitions as big languages, for example, SDL in structure, syntax, and semantics as described in [17] and implemented in [43].

MPS can define major languages, as evidenced by the definition of Java [18] in MPS [39].2 Moreover, MPS has been extended and used in industrial projects [52].

We presented our experiences in teaching MPS in two inherently different environments.

Courses and trainings given in an industrial setting are aimed at experienced developers and business experts, are often adjusted for a particular business domain, and are designed around acquiring practical language implementation skills by the participants.

The academic setting tends to give a broader yet less detailed overview of several language implementation techniques, with the goal that students grasp fundamental concepts and differences between approaches and tools.

Attracting students to taking any form of training in the MPS technology has constantly been the biggest challenge.

Numerous beginner-level questions on the MPS discussion forum indicate that a large number of professionals attempt to climb the steep learning curve on their own, which frequently leads to suboptimal solutions, shallow opinions, and a lot of frustration.

It remains to see whether joint efforts from both industry and academia in teaching MPS would benefit all the stakeholders.

Perhaps a first step already taken in this direction is the availability of teaching materials [3, 30] online.

We presented our experiences in teaching MPS in two inherently different environments.

Courses and trainings given in an industrial setting are aimed at experienced developers and business experts, are often adjusted for a particular business domain, and are designed around acquiring practical language implementation skills by the participants.

The academic setting tends to give a broader yet less detailed overview of several language implementation techniques, with the goal that students grasp fundamental concepts and differences between approaches and tools.

Attracting students to taking any form of training in the MPS technology has constantly been the biggest challenge.

Numerous beginner-level questions on the MPS discussion forum indicate that a large number of professionals attempt to climb the steep learning curve on their own, which frequently leads to suboptimal solutions, shallow opinions, and a lot of frustration.

It remains to see whether joint efforts from both industry and academia in teaching MPS would benefit all the stakeholders.

Perhaps a first step already taken in this direction is the availability of teaching materials [3, 30] online.

rapid application development studios and form builders that allow both visual and text-based editing of GUI forms; Scratch-like programming environments [27], where the code is constructed from graphical blocks corresponding to the programming language statements.

TSE generates structure editors based on the toString function for a value, with little to no further programmer effort required.

We envision value-centric programming systems that offer editable, domain-specific representations for custom data types, thus affording the programmer a more natural interface for specifying changes on the operation of their program.

At present, we implemented our TSE prototype independent of any of these possible settings.

While our independent implementation highlights TSE’s key techniques, applying TSE to a particular application requires a number of further design decisions, particularly surrounding the handling of actions.

For example, consider the set data structure in Figure 6.

The reference implementation [24] is based on a tree and maintains a number of invariants such as balancing, ordering, and non-duplication.

None of these invariants are expressible in a standard ADT definition alone, and the internal tree structure is not exposed in the toString output .

Therefore, only some of TSE’s selection regions are relevant—namely, the terminal items, as reported in Figure 6—and the structural transformations generated by TSE are not meaningful because they do not enforce the set invariants.

TSE does not yet provide an interface for specifying custom insert and remove functions, instead we imagine such an interface would be part of a larger, future IDE.

Beyond action handling for data with complex invariants, our prototype has a number of minor limitations.

First, systems that rely on string tracing provide custom implementations of string manipulation functions that correctly propagate dependencies.

We currently only support string concatenation and string length—supplementing our language with additional string functions remains future work.

Finally, our core language and TML do not support nested pattern matches.

How dependency semantics should work for nested patterns is an open question—although a language’s compiler will unnest the patterns [25], different unnestings can result in different dependency traces.

While not uncommon, such ambiguous cases did not occur in our examples.

Adapting TSE to the more common object-oriented setting will require different tracing semantics, because “variants” are handled by virtual method lookups rather than case splits.

Further details about TSE’s algorithm and heuristics will be available in an accompanying technical report.

Programmers often write toString functions to help interpret and debug code involving custom data types.

For example, for a type of values describing numeric intervals, the string ", Before)", which might be a default serialization provided by the language.

Custom toString functions are usually straightforward to write, but what if the programmer needs not only to display the value but also edit the value as well? One idea is for the programming environment to enrich default string representations with automatically-generated, type-directed GUI widgets.

For example, given the default representation "Interval, Before)", the system might render a slider for “scrubbing” 10 to different values and a widget to select NegInf and toggle it to After.

Ideally, however, the domain-specific representation " We design a system, called TSE, that given a toString function for a custom data type, automatically generates tiny structure editors for manipulating values of that type.

To do so, TSE instruments the execution of the toString function applied to a value, and then overlays UI widgets on top of appropriate locations in the output string .

To determine these locations, TSE employs two key technical ideas: a modified string concatenation operation that preserves information about substring locations and runtime dependency tracing to relate those substrings to parts of the input value.

We implement TSE for a simple functional language with algebraic data types , and we discuss the tiny structure editors that TSE produces.

Potential Applications TSE is currently a prototype, proof-of-concept tool.

However, we believe our approach would benefit a number of emerging techniques that allow programmers to specify code via direct manipulation of program values.

Literals in a Structure Editor.

In structure editors—such as the Cornell Program Synthesizer [4]—and block-based editors— such as Scratch [5]—tree transformations rather than raw text edits are used to manipulate data structures .

Liberated from raw text buffers, structure editors can use domain-specific representations for display.

For example, the Barista [6] editor for Java offers rich, custom, type-specific views for mathematical and logical expressions in code.

But display is easier than editing: for editing, Barista falls back to ordinary textual manipulation.

Programming by Examples .

Given input-output examples, these systems synthesize a small program.

Sometimes many examples are required: Myth [8] requires 20 examples to synthesize binary tree insertion.

Providing so many examples in text form can be cumbersome.

Direct-Manipulation Programming.

Several tools augment text-based coding with direct manipulation of output values.

Bidirectional programming systems allow users to edit numbers , strings , or lists in the output of a program to thereby change appropriate literals in the original code.

Compared to these BX systems, output-directed programming systems allow the user to make larger, structural changes to the program , performing refactorings or inserting chunks of new code.

To date, ODP systems carefully implement bespoke, domain-specific interfaces to enable selection and manipulation of the output.

Related Work Each of the programming interactions above would benefit from an easy way to create domain-specific interfaces for custom data types.

How do users currently input and edit program values in such systems? Parse Functions.

Programming is largely a text-based activity; entering values via text is thus a natural interface, but requires a parser.

Custom parsers can be integrated with a language pre-processor like Template Haskell [19] or typed literal macros [20].

But the difficulty of writing a parser may not be worth the gain in expressiveness over the language’s default value parser.

Our approach provides a structure editor on a domain-specific representation of a value, without the labor of writing a parser.

These approaches could be combined: a structure editor could offer an optional text interface for bulk input, although our prototype does not yet.

Handcrafted GUIs.

If interaction is important, the programmer may opt to manually craft a custom graphical user interface for their data type.

Although this effort is justifiable for common types, e.g. colors or regular expressions [21], writing a custom UI may not be worth the trouble for one-off data types.

String Tracing.

Some previous systems trace string operations, enabling developers to directly edit HTML output and thereby modify appropriate literal strings in the source PHP or Javascript.

TSE also relies on tracing, but uses a more generic mechanism [3], allowing TSE to track how substrings relate to any value of interest, rather than just string literals.

In this paper, we propose a DSL for IoT domain, called OntIoT, which is based on the SSN ontology.

The approach, used to build OntIoT, is generic that could be applied to build a DSL from any ontology.

OntIoT is based on projectional editing technique.

Our ongoing research is to evaluate the proposed DSL in a real scenario.

We plan to complete the mapping rules between OWL and DSL to cover the missing OWL constructs.

Furthermore, we plan to utilize OWL reasoning algorithms to enrich the OntIoT editor with more semantic features.

In addition, a graphical editor will be implemented for the proposed DSL.

In this section, we will describe the approach that is used to implement OntIoT.

Figure 4 describes the steps for building OntIoT and its editor from the SSN ontology.

OWL is used to represent the SSN ontology.

The transformation engine is responsible for transforming the OWL SSN ontology to the target language structure based on a set of general mapping rules .

The input for the engine is the SSN OWL file and the output is the abstract syntax tree for the OntIoT language.

The latter is used inside the language workbench to generate the final DSL and its editor by the language designer.

Our ontology-based approach for building OntIoT is a generic approach that can build a DSL from any OWL ontology.

SSN ontology is selected to build OntIoT since it is a W3C and OGC standard.

In addition, it is a high level ontology that could be used in a wide range of IoT applications.

Accordingly, OntIoT could be used in many IoT sub domains like smart home, smart healthcare, and agriculture monitoring.

SSN ontology is considered as a core ontology for a set of more specific ontologies like IoT-Lite4 and OpenIoT5 .

The extension of OntIoT to cover these ontologies is a straightforward using our generic approach without need to rebuild the language.

In our implementation, we used Meta Programming System 6 language workbench for building OntIoT.

MPS is an open source workbench that implements the projectional editing technique for DSL development.

The projectional editing technique is selected to implement OntIoT for many reasons: First, OntIoT is based on SSN ontology.

There are many similarities between the ontology structure and the AST structure.

The AST is the core unit in the projectional editing.

These similarities facilitate the transformation process.

Second, the projectional editing technique will allow to build different editors types for OntIoT .

Currently, we provide only a textual editor but we plan to build a graphical one.

Third, It allows OntIoT to contain mixed notations which is not applicable in other techniques.

Finally, projectional editing technique allows OntIoT to be easily extended to map the different ontologies that utilize or extend SSN ontology.

Internet of Things [1] is a term that refers to the interconnection of a set of things through a network.

These things are able to collect and exchange data with each other.

Things could be physical devices, virtual services, or anything.

Sensors and actuators are examples of the physical devices that could be connected to the IoT applications.

There are many successful IoT applications that promise great potential in enhancing the daily live like smart home and smart cities.

Although, IoT is a very promising area that could produce a new generation of software applications, the progress of the IoT development does not match the expected rate.

CISCO provided a survey1 in 2017 showing that only 26 percent of the companies could complete its IoT initiatives to be successful projects, and 60 percent of companies believe that the implementation of IoT applications is more complex than expected.

There are many reasons for the slow growth of IoT applications [2].

Different technical knowledge levels are required to build an IoT application.

The required knowledge is varied from the low level embedded development and network protocols to the high level user friendly UI design.

Across those levels, IoT development teams should focus on security, scalability, and other non-functional requirements that are mandatory for any IoT application.

As a result, more development time and special skills are required to build IoT Applications.

In addition, there is a lack of adequate languages and tool support for IoT programming [3].

Domain Specific Languages are languages that provide high abstractions and optimizations for a specific domain.

The problems in the given domain are represented more efficiently using the related DSL.

A DSL for the IoT domain could contribute in solving some challenges of IoT development.

The DSL will provide high abstractions related to the IoT domain like sensor, actuator, and observation concepts.

Accordingly, the mapping between the business model of the IoT, held by the domain expert, and the technical model, held by the developer, becomes faster and more straightforward.

As a result, the development time of the IoT applications will be reduced.

The DSL handles the generation of the low level code that decreases the required knowledge and increases the automated part.

The DSL also provides a special editor that facilitates the development process and ensures that output program is matching the domain constraints.

Finally, the involvement of domain expert becomes easier as the program is written with his terms.

The first phase for building a new DSL is the domain analysis phase which determines the concepts, relations, and constraints inside the concerned domain [4].

This phase is considered as a core phase in DSL development since a wrong analysis will lead to an invalid DSL even with a correct implementation.

On the other hand, the ontology provides a formal conceptualization for a given domain that determines the concepts and the relations of this domain [5].

A lot of effort has been done to develop valid ontologies for IoT domain [6-7].

Accordingly, building a DSL from these ontologies will generate a valid DSL and it will reduce the DSL development time.

In addition, the DSL could utilize existing ontology reasoning algorithms to provide semantic reasoning services for the IoT editor.

Finally, the existing documentation of the ontologies will guide the IoT developer to correctly model his problem using the corresponding DSL.

The main motivation of our work is to utilize the current standard IoT ontologies to build a DSL for IoT domain, this DSL benefits from the aforementioned advantages of ontology and DSL integration.

This paper proposes OntIoT DSL that models part of the IoT domain.

The structure of OntIoT is automatically generated from Semantic Sensor Network2 ontology [8].

The implementation approach of OntIoT is generic, it could be applied on any ontology or domain.

Furthermore, OntIoT is based on projectional editing technique [9], which allows the language to be extended to support different editors and mixing among textual, tabular, and graphical notations.

Moreover, the projectional editing technique allows OntIoT to support language extension and composition with other DSLs.

To fully realize the workflow described in Section 2, Deuce+ must comprise three distinct but interrelated components: a set of type-directed program transformations informed by common code authoring patterns of functional programmers; a syntax constraint language and engine to ensure these transformations are composable and produce stylistically consistent output; and a domain-specific language for specifying these transformations.

The first of these components will address Limitation A of Deuce by providing a sizeable set of transformations justifiably rooted in existing programmer behaviors, and the second and third of these components will address Limitation B of Deuce by providing a system for building and composing the transformations that is accessible to any user of the system.

Moreover, with such tools in place, a large library of automatically composable user-defined transformations will be made possible, further combating Limitation A.

3.1 Type-Directed Program Transformations With the strong guarantees of a rich type system, programmers can leverage expressive program transformations to alleviate some of the pains caused by the flexibility of text .

As part of the Deuce+ project, we would like to formalize some of the common, implicit techniques that functional programmers use to develop programs by crafting program transformations to perform these authoring patterns automatically.

From personal experience, we have identified a few such transformations, including the Refine Type, Make Progress on Hole, and Make Impossible transformations from Section 2.

However, we would like to design and conduct a user study to observe how functional programmers author code.

Do they start with a skeleton of a solution and later fill in the holes? Or do they, perhaps, write code from the top down? There are myriad other ways code can be written, too, and there may not even be a consensus among functional programmers on this issue; nevertheless, a formal user study is in order to even begin to answer these questions.

In the meantime, we investigate the three aforementioned program transformations and how they relate to some common functional programming authoring patterns, as summarized briefly in Figure 2.

Refine Type.

The Refine Type transformation mirrors the code authoring practice of maintaining code invariants.

After structurally selecting a type, users may activate the Refine Type transformation to narrow down the type’s set of possible values.

Inspired by refinement types [6, 7, 23], the Refine Type transformation prompts the user for an invariant that they wish to maintain about the program, and attempts to refactor the type to ensure that the invariant is satisfied.

For example, consider a functional queue type drawn from Chris Okasaki’s Purely Functional Data Structures [16].

Given the queue type on the left and the invariant that |front| >= |back|, Refine Type might suggest to transform the type to that on the right: The first |back| components of front and back are stored together in the list frontBack, and the remaining |front| - |back| elements are stored in the list remainingFront, so it is now impossible to represent a Queue in which |front| < |back|.

To maintain ease of use and increase backward compatibility, the Refine Type tool might also provide helper functions corresponding to the previous API of the queue: Make Progress on Hole.

The Make Progress on Hole transformation mirrors the code authoring practice of “following the types,” an oft-heard adage in the functional programming community suggesting that the type system can guide the user to implement the task at hand essentially automatically.

While such advice is clearly not universally applicable, the statement does hold some truth to it, as demonstrated by the practice of type-directed programming [1, 26, 27].

After structurally selecting a hole, users may activate the Make Progress on Hole transformation to fill a hole with an expression .

There are at least three type-based approaches that this transformation can rely on to fill holes with helpful expressions: expression templates, typedirected refinement, and program synthesis.

Expression templates are pre-written generic code snippets that can be suggested to the user to fill the hole at hand based solely on the bindings that are in scope and the type of the hole to be filled, as was done to introduce the List.map function in the implementation of the showEntries function in Section 2.

Type-directed refinement is the systematic destructuring of a type into its component types via pattern matching, as was done to pattern match on the entry variable in the implementation of the aforementioned showEntries function.

Another example of type-directed refinement would be, for instance, the filling of a hole of type with the expression , an expression whose holes have been refined to simpler types .

A final, more general approach to the task of filling holes lies in the practice of type-directed program synthesis, or, generating programs to match a specification , as in [5, 10, 12, 18, 19, 20, 21].

Make Impossible.

The Make Impossible transformation mirrors the code authoring practice of making illegal states unrepresentable [4, 15].

At the time of authoring a type, certain code decisions may seem like a good idea that only later reveal themselves to be cumbersome.

For example, a programmer might write a record type representing the state of an application window with the field content : Maybe String .

Some time later, the programmer may realize that windows should save their own position, and thus adds a field position : Maybe .

But, in enacting this change, the programmer has made representable two states that should be illegal: when either one of the fields is Just something and the other is Nothing.

By structurally selecting the record type and providing the patterns that should be unrepresentable , the programmer can use the Make Impossible tool to make values of the selected type that match the specified patterns impossible to represent.

In the windowing example from above, the Make Impossible tool would transform the record of Maybe types to Maybe { contents : String, position : }.

This transformation is made possible by the algebraic nature of datatypes, as used in [14].

While work is still underway to make the intuition rigorous, we may view the Make Impossible transformation on a datatype as operating on the polynomial functor of which the type is a fixed point, represented suggestively as τ − P, where τ is the datatype functor and P is the pattern we wish to eliminate.

For example, viewing record types as product types, we can determine the solution to the windowing example from above using algebraic laws related to this transformation :

As a medium for storing, transmitting, and interpreting information, text is as versatile as it is ubiquitous.

Countless programs and interfaces operate and rely on text files, from the UNIX command line to nearly every programming language compiler.

One particularly strong asset of text is its power to succinctly represent structured data in a way that is understandable to both humans and computers alike, as in CSV files, HTML documents, and – the focus of this paper – programming language source files.

Unfortunately, this flexibility comes with a price: manual editing of structured text can be tedious and error-prone.

On a basic level, one problem with manipulating structured text is that to do so requires knowledge of and adherence to rigid, static systems such as parsing and type checking.

A single missed comma in a CSV file or improperly annotated variable in program source code can cause a complete failure on the part of the computer to interpret the text as the author intended.

In the case of performing a nontrivial manual transformation on a program, the problem is exacerbated even further: programmers must also worry about the runtime behavior of their code and reason about changes in semantics that might be a result of their transformations.

Tools known as structure editors [9, 13, 24, 25] attempt to alleviate these difficulties by offering an interface that allows programmers to directly manipulate a projection of the underlying structure that is more faithful to the structure than is standard text.

A major drawback of these systems is their reliance on non-standard file formats, and, as a result, their incompatibility with the large set of existing tools that operate on programs.

One attempt to reconcile the flexibility of plain text with the power of projectional editing is Deuce [2, 8], a structure-aware code editor that operates on standard program source text, but augments the editing experience with direct manipulation capabilities for invoking relevant, automated program transformations.

In Deuce, the underlying structure of the program is revealed to the user via a set of overlaid polygons, as depicted in Figure 1a.

After structurally selecting various parts of the program by pressing the shift key, hovering, and clicking on the polygons that appear, a “Program Transformations” menu appears that is automatically populated with a set of relevant transformations for the selected polygons, as depicted in Figure 1b.

Hovering over the output of a program transformation previews it in the code panel, and clicking on the output updates the program with the transformed code.

While a good first step, Deuce falls short in several regards.

In particular, it has two main limitations:

Deuce only offers a relatively small number of ad-hoc program transformations;

and the implementation of these transformations is tedious and noncompositional, requiring manual munging of abstract syntax trees annotated with low-level syntactic details such as whitespace.

To address these limitations, we propose and present initial work on a vast expansion of the Deuce system – which we here call Deuce+ – with the goal of bringing expressive and extensible program transformations to the working programmer.

If the goal of Deuce+ is to make powerful program transformations backed by elegant mathematical ideas accessible to – and authorable by – the everyday programmer, then good usability is of utmost importance.

Every single operation described thus far should be influenced and rooted in not only sound mathematics, but proper usability studies.

Although the development of Deuce+ is currently very early in the design process, there are a few preliminary usability considerations that need to be addressed as Deuce+ develops.

With so many program transformations available to the user , it is critical that the user be able to find the correct transformation for the task at hand, even if the user does not know its name.

We will need to investigate what mental models users have of the transformations available to them and determine heuristics and display mechanisms for showing them relevant program transformations.

After a transformation has been selected, its output may be hard to decipher, so how does a user know when a program transformation is “correct?” We will need to investigate how to help users be confident in their output selections.

The languages underlying the program transformations themselves must also be streamlined and intuitive for end-user transformation authors.

Drawing from interdisciplinary programming language design [3], we will need to design and evaluate the style sheet and transformation specification languages holistically, aiming for providing an experience that supports and encourages expressive and extensible code.

The structured editing user interface introduced by Deuce will need to be improved to support usability improvements such as drag and drop, fluid and dynamic animations, and novel code overlays beyond simple polygons .

Completion of these tasks will ensure that – at every step of the way – the usability of Deuce+ is of the highest priority across all its components, from its graphical user interface to its program transformation authoring and end-user experience.

The top concerning challenge has been observed to be metamodeling tools’ steep learning curve required for defining the language syntax and semantics and the lack of support for training .

One of those participants is especially concerned about learning the projectional editing with MPS.

Some participants are concerned about the difficulties for keeping the language syntax and semantics complete and consistent as the new modeling elements are added — two participants herein are especially concerned about using the Eclipse-based meta-modeling tools.

Also, a few other participants are concerned about the lack of support for the integration with the version management tools for versioning meta-models and the collaborative meta-modeling.

Lastly, one participant stated to face with difficulties in using Enterprise Architect for defining the relationships between modeling elements.

In this study, we aimed to learn practitioners’ perspectives towards the meta-modeling tools and conducted a survey among 103 practitioners from 24 different countries.

The survey participants represent the different profiles of the population who differ in terms of the work industries, the problem domains, job positions, and years of experiences.

Our survey investigates three important research questions, which essentially focus on the usage frequencies of the existing metamodeling tools, practitioners’ expectations from the meta-modeling tools, and any challenges that practitioners face with.

According to the survey results, the top-used meta-modeling tools are Eclipse Sirius and GEMS , which are followed by Metaedit+ and Xtext .

The meta-modeling tool features have been categorized in the survey as the language definition, editor services, model transformation/code-generation, language validation, and language composability, and each category here led to very interesting results in the survey.

Concerning the language definition, while the diagrammatic and textual visualizations are highly popular among the participants, other types of visualizations and hybrid visualization are rarely preferred.

Many of the participants wish to define the language semantics using the translational semantics definition technique that can either be model-to-model or model-to-text.

The interpretative semantics definition is neglected.

The participants are also willing to communicate their language definitions via the importing/export facility and keep the record of the language definitions that can be managed in terms of versions.

The participants’ main challenges on the language definition are the lack of support for training and the steep learning curve.

Concerning the editor services, while 69% of the participants wish to use the free-form editing mode.

The projectional editing mode for the hybrid modeling with multiple visualizations is not popular.

The top features that are desired by the participants to be supported by the modeling editors are an error marker for semantical aspects , document generation , model re-use , a model comparison , and the automatic update of the models when meta-model changes .

The main challenge on the editor services is to do with the editor usability .

Concerning the model transformation/code-generation, the participants wish to use a transformation technology with the error detection and syntax highlighting features.

The challenges here include the lack of support for training and adapting the tools developed with the changing user requirements.

Concerning the language validation, many participants are willing to define both semantic and structural validation rules for DSMLs and the top-desired tool feature for the language validation is the support for integration with the external analysis and validation tools .

The challenges here are to do with the real-time checking of the validation rules and complexity management.

Concerning the language testing, many participants wish to test the language semantics.

The challenges on the language testing are to do with the lack of support for specifying and executing test cases, model-based testing, and integrating with test automation tools.

Concerning the language composability, most participants wish to define the language semantics via composition.

The challenges include the difficulties in using the composed languages and performing composition with textual languages.

We plan to validate the survey results via the international R&D projects that we are involved in.

To this end, we will consider the PANORAMA project,5 which is labeled by the European Union’s EUREKA Cluster program ITEA .

In PANORAMA, the Eclipse-based meta-modeling tools are being used to develop a scenario definition language for autonomous vehicles along with the supporting toolset .

So, we will survey and interview the practitioners who develop the language with some pre-determined list of questions that are derived from the survey results.

Moreover, we will consider our Ph.D.

students who use the Eclipse-based meta-modeling tools and Metaedit+ to develop DSMLs for different purposes and validate the survey results.

Lastly, we believe that the survey results will be highly useful for the tool vendors in determining the needs of practitioners from the meta-modeling tools.

So, in the next release of their tools, the tool vendors may consider the missing features in their tools that are crucial for practitioners.

Also, the survey results may be utilized by the practitioners who use or develop modeling languages.

Indeed, practitioners may learn about the top-used meta-modeling tools for the domain of their interests and the important features of the metamodeling tools from the practitioners’ perspectives.

So, practitioners may consider benefitting any particular meta-modeling tools for their domain and the particular features that they are interested in.

As Fig. 14 shows, 69% of the participants prefer the meta-modeling tools that support the free-form editing exclusively, which promotes the persistent storage of the editable models.

So, those participants essentially prefer to use any meta-modeling tool that supports developing a language with just one visualization only .

Another 21% of the participants prefer the meta-modeling tools that support developing a language with the projectional editing mode, which enables the models to be edited via different projections that each supports a different visualization .

Those participants may actually use the MPS meta-modeling tool for creating different projections for their language.

The rest of the participants do not seem to have a particular choice here and indicated to use the meta-modeling tools that support either of the editing modes depending on their language development requirements.

Fig. 15 gives the correlations between the participants’ domain and the editing modes that the participants prefer.

The participants from some domains do not prefer any meta-modeling tools with the projectional editing support.

On the other hand, the participants from the mobile and IOT domains do not prefer the free-form editing.

Also, the medical device development, enterprise solutions, and document engineering domains include many more participants who prefer the projectional editing mode.

Q12: The syntactic editor services that the participants prefer to use As Fig. 16 shows, reusing models is the top-popular syntactic editor service that has been selected by 75% of the participants.

Indeed, with model reuse, practitioners may keep the models created in a repository and create new models by re-using the existing models in the repository with the least effort possible.

Another highly preferred syntactic editor service is the comparison of models via a diff-like tool , which may enable to detect and review the differences between models and even merge them.

The other editor features that are also quite popular are the syntactic completion templates that provide incomplete models/code/graph to the users and the auto formatting, restructuring, aligning, and layouting of a model’s presentation .

Other syntactic features such as the customizable visual highlighting in models, navigation support , and model folding to hide part of a model are considered by relatively fewer participants .

Q13: The semantic editor services that the participants prefer to use Besides the modeling editor services that concern the syntax-related aspects, the modeling editors may offer services about the semantical aspects of languages.

As Fig. 17 shows, the top-popular semantic editor services is the need for an error marker for highlighting a model element and any associated error messages , which enables to detect the semantical errors at modeling time.

The automatic update of models upon any changes on the meta-model is the second top-popular semantic editor service , which enables the modelers to detect any errors due to the semantical changes automatically.

Also, the semantic completion for receiving automatic suggestions at modeling time and the refactoring of models without changing semantics are each desired by 60% of the participants.

The UML support for reusing and extending the UML language syntax and semantics and the live translation between the model and generated code are also desired by half of the participants .

The rest of the semantic editor services such as navigation to representations , advanced search , co-evolution of metamodels together with models, and quick fixes are relatively less popular among the participants.

Q14: The other editor services that the participants prefer to use Besides the syntax and semantics editor services discussed above, many other editor services may exist that are not given in the above questions’ lists.

So, in this question, we provided a list of possible general editor services and let the participants type any other services that are not given in our list.

As Fig. 18 shows, document generation is the top-popular editor service herein, which is desired by 77% of the participants.

Importing/exporting models for sharing models among collaborators and the version control system integration for managing the model versions are also the highly desired features for the modeling editors .

68% of the participants are interested in the usability of the modeling editors, which is essentially concerned with the minimum number of clicks for modeling/meta-modeling.

The editor services that are also quite popular and considered by approximately half of the participants are the traceability between different view models , the IDE integration and scalability .

The rest of the features given in Fig. 18 are relatively less popular among the participants.

Also, using the ‘‘other" option, one participant wishes for the offline access to the editor, where the changes can be reflected when connecting to the internet.

Another participant wishes for sharing the meta-model with any users who can use the language freely.

Q15: The challenges that the participants face with on developing/using editors The participants are most concerned about the usability of the editors they develop or use.

One of those participants stated to face with some usability issues in using Eclipse Xtext/Xtend and MPS metamodeling tools for developing editors.

The same participant is also concerned about Xtext’s support for scalability and traceability based on the OSLC2 linking.

Also, some of the participants are concerned about editors’ lack of support for model versioning, training , and integrating editors with other development technologies, such as .NET, to combine modeling and coding.

Lastly, one participant is concerned about the support for web-based editors, and another one is concerned about the syntax coloring.

Models are considered as the abstract descriptions of any real systems and may exist in multiple forms that each address a different aspect of systems [1–4].

Models are created for various purposes, including the facilitated communications among different stakeholders, analyzing the correctness and completeness of the abstract system descriptions, test-case generation, and code generation [5].

With the abstract models, practitioners can better understand complex systems, have the chance of analyzing the possible design decisions before implementation, and obtain high-quality code that is guaranteed to satisfy the design decisions.

As Selic suggests [2], for a model to be worthwhile, it needs to possess five important characteristics.

A model needs to be abstract by focussing on a particular problem of a system and suppressing the rest of the details; a model needs also to be understandable so that one can easily grasp the complex descriptions that are modeled abstractly; a model needs to represent a real system in an accurate way; a model may also promote the predictiveness of a real system, as models can be used to predict the quality properties of systems before building them; and, a model needs to be inexpensive and one should be able to create abstract models with the least cost and effort possible.

Models can be created via modeling languages, which can either be general-purpose or domain-specific [6].

While general-purpose modeling languages offer high-level notation sets that can be used to specify any types of systems, domain-specific modeling languages offer specialized notation sets on particular domains .

Modeling languages are based on meta-models, which state the language concepts and the syntactic and semantic rules that the models specified with those language concepts need to satisfy [1,9,10].

The language syntax describes the elements that can be used for creating models and the rules to be followed in using these elements.

On the other hand, the language semantics describes the meaning of the language elements, which can be formulated using different techniques such as operational [11], denotational [12], axiomatic [13], and algebraic semantics [14].

A meta-model for a modeling language can be defined using the meta-modeling tools [15].

With the metamodeling tools, one can specify the language definitions in terms of the language syntax and semantics, build a modeling editor according to the language definitions, and even generate tools for, e.g., model analysis and code generation purposes.

As presented in Table 1, there are various meta-modeling tools that can be used for defining the language meta-models and automatically producing the necessary modeling tools .

The existing meta-modeling tools may vary depending on their level of support for different kinds of requirements that are concerned with the language syntax and semantics definitions, editor services, model transformation, language extensibility, model analysis and validation, and being open-source or commercial.

1.1. Motivation and goal.

Software modeling is essentially considered by industry as highly crucial for developing large and complex software systems, given its support for such concepts as the abstraction, separation of concerns, and early analysability of design decisions.

Thanks to the existence of meta-modeling tools, practitioners may even develop their own domain-specific modeling languages and create models that are specific to their domain problems and develop the necessary toolset for processing the models according to their specific modeling goals.

However, it is not yet clear as to what extent the meta-modeling tools are adopted by practitioners in different industries; and, it is still difficult to understand practitioner’s expectations from the meta-modeling tools and any challenges that practitioners face with.

As discussed in Section 2, the literature includes several attempts at understanding practitioners’ perspectives towards modeling.

So, one can understand, e.g., the existing modeling languages, their usage frequencies in industry, their weak and strong points, practitioners’ challenges on modeling and modeling languages, the practical application of modeling in particular domains, and the analysis and comparison of a set of modeling and meta-modeling tools.

However, there is a gap in the existing literature to understand practitioners’ perspectives towards meta-modeling.

Although there are many metamodeling tools available in the market , there is no study which explores the attitudes of practitioners and when, how and why meta-modeling is used with possible challenges.

In this paper, the goal is to understand the practitioners’ preferences among different meta-modeling tools, their expectations, and any challenges faced with.

To achieve this, we designed and conducted a practitioner survey.

The survey results are expected to be useful for anyone who consider developing their own DSMLs in understanding the top-used meta-modeling tools for different domains.

Also, the tool vendors could use the survey results in learning the expectations of practitioners from the meta-modeling tools and any challenges encountered.

In our survey study, we addressed a number of important requirements for the meta-modeling tools and intended to learn practitioners perspectives towards the meta-modeling tools in terms of those requirements.

To determine the meta-modeling tool requirements herein, we considered Erdweg et al.’s comprehensive feature model for the meta-modeling tools [16] and extended that with further categories of requirements that we deem important after a series of pilot studies conducted with the area experts.

In the rest of this section, we discuss each category of requirements separately, which are concerned with the meta-modeling tools’ support for language definitions , modeling editors, model transformation, language validation, language testing, and language composability.

1.2. Categories for meta-modeling tool requirements

1.2.1. Language notation

The language notation is concerned with models’ appearances to users.

Languages may support such visualizations as diagrammatic, textual, tabular, trees, matrix, map, and hybrid.

The diagrammatic visualization enables the model elements to be specified using graphical symbols.

The textual visualization enables the model elements to be specified in terms of texts .

The tabular visualization enables the model elements to be specified using a table editor that can be displayed as a table and edited by simply specifying cell values.

The matrix-style visualization enables the model elements to be specified and edited in two axes where each cell in the matrix essentially indicates the relationships of the elements in the two axes.

The map visualization enables the model elements to be specified with their location data and the distances among the elements are of particular importance.

Lastly, the hybrid visualization supports multiple visualizations that can be used for editing the same model in a synchronized way.

1.2.2. Language semantics

The language semantics can be categorized as interpretative or translational.

The interpretative semantics promotes the execution of models without performing any translations into some intermediate formats.

The translational semantics promotes the definition of the model translations into an intermediate format that can be executed.

The translational semantics can be either model-to-text and model-tomodel.

The model-to-text translation is to do with defining the language semantics in terms of the rules for the translations into some structured text notation such as source-code .

The model-to-model translation is to do with defining the language semantics in terms of the rules for the translation into a model with a different notation set .

1.2.3. Editor services

The editor services are concerned with the capabilities of the modeling editors that users can create using the meta-modeling tools.

The editing mode that the meta-modeling tools support can be categorized as free-form and projectional.

In the free-form editing, users edit a textual or graphical model that is stored persistently, and a persistent model may then be transformed into an abstract representation that can further be transformed into an executable representation.

In the projectional editing, users may edit any projections of the model’s abstract representation that is stored persistently and transformed into an executable representation.

Each projection may be in different formats , and unlike the free-form editing, the projections that are edited by the users are not stored persistently.

Also, Erdweg et al. proposed in [16] many different syntactic and semantic editor services that may be interesting to the users.

The syntactic editor services include model highlighting, navigation support, folding models, syntactic completion templates, comparing models, and auto-formatting the model appearances.

The semantic editor services include semantic completion templates, model refactoring, error markers, live translation between model and generated code, and quick fix of the errors.

1.2.4. Model transformation/code-generation

Model transformation/code-generation is concerned with the transformation technologies that are supported by the meta-modeling tools and enable to develop code generators for DSMLs which can be integrated into the modeling editors for transforming models.

Model transformation/code-generation technologies may vary depending on the features that are supported for improving the development processes of the model transformers/code-generators.

These features include the syntactic and semantic error detections while developing the model transformers/code-generators, code templates, refactoring, integration with external programs/files, AI-based model transformation, support for scalability, debugging facilities, code folding, etc.

1.2.5. Language validation

The language validation is concerned with the meta-modeling tools’ support for defining validation rules for DSMLs.

The validation rules for a DSML may then be executed via the modeling editor produced by the meta-modeling tool so as to validate models created via the editor.

The validation rules can be categorized as the structural and semantic validation rules.

The structural validation rules are concerned with the structural aspects of the language definitions, such as the multiplicities of the language elements and containment relationships between them.

The semantic validation rules are concerned with the semantical aspects of the language definitions such as name/type analysis.

The meta-modeling tools may further offer such features as the integration with some external validation tools , model animation, model debugging, and automated model validations according to the user-defined or pre-defined rules.

1.2.6. Language testing

The language testing is concerned with testing different aspects of the language development, including the syntax & semantics definitions, editor services, code-generation, and the validation rules, with regard to any functional and quality requirements.

The syntax and semantics testing is to do with checking for the language definition requirements.

This may include checking if the language metamodel consists of the expected modeling elements and relationships and the syntax and semantics rules have been defined correctly and completely.

The editor testing is to do with checking if the editor meets such quality requirements as usability and performance.

Also, the editor testing may include checking if the editor enables to create models in accordance with the language syntax and semantics.

The code-generator testing is to do with checking if a code-generator developed via the meta-modeling tools performs the model transformation correctly and meets the quality expectations.

The validation rules testing is to do with checking if the user-defined or pre-defined validation rules can be defined in accordance with the language requirements and then used for validating models correctly in a way that also meets the language requirements.

1.2.7. Language composability

The language composability is concerned with the meta-modeling tools’ support for extending an existing language with some new features or unifying the parts of multiple languages for developing a new language.

Just like language testing, the language composability can be considered for different aspects of the language development.

The language syntax and semantics may be composed of the syntax and semantics of any existing languages that are stored in a repository for later re-use.

A modeling editor can be developed by composing multiple existing tools together such as the model versioning tool, collaboration tool, validation tool, and code-generation tool.

The model transformation/code-generation tool may be developed by re-using some transformation templates, patterns, or the existing code.

The validation rules can be defined by re-using and modifying the existing rules or composing multiple rules together under some conditions .

We learned from the survey results that modeling is essentially the concerns of several different industries where practitioners work on large and complex problems that can better be managed with the advantages of modeling such as abstraction, enhanced communication, separation of concerns, early analysis of design decisions, and generating code from models.

While the survey attracted the greatest number of participants from the defense/military & aviation industry, many other industries have been participated in the survey to indicate their perspectives towards the meta-modeling tools including IT & telecommunications, automotive & transportation, healthcare & biomedical, consumer electronics, finance, government, education, etc.

Also, modeling is considered by those industries for solving problems at various different domains that include embedded, automotive, control and automation systems, web, user interface design, medical device development, IOT, enterprise solutions, data analytics, railway systems, testing, robotic, mobile, chip development, document engineering, and real-time OSs.

The survey results revealed that most of the practitioners from different industries use either of the five different meta-modeling tools which are Sirius, GEMS, Metaedit+, Xtext, and Microsoft DSL tools.

Other meta-modeling tools such as ANTLR, ConceptBase, Melange, GEMOC, Graphiti, WebGME, Cameo, EVA, and JastEMF are rarely used.

Among the top-five meta-modeling tools, Sirius, Xtext, and GEMS are all actually Eclipse-based tools that are based on the Eclipse Modeling Framework and can be used as an Eclipse plug-in.

It should also be noted that Xtext is the only meta-modeling tool among the top-five, which offers a textual visualization.

Moreover, some countries seem to use particular meta-modeling tools.

Indeed, USA prefer Microsoft DSL tools, Finland prefer Metaedit+, and most European countries prefer Eclipse-based tools and Metaedit+.

Another lesson learned about the meta-modeling tools is that some participants seem to consider EMF as a meta-modeling tool.

Indeed, those participants used the ‘‘other’’ free-text area of the corresponding survey answer and typed ‘‘EMF’’ for their meta-modeling tool.

However, EMF is essentially not a meta-modeling tool but an Eclipse modeling project whose goal is to propose a modeling framework that can be used for building a modeling editor for a given data model [43].

EMF is used by many popular Eclipse-based meta-modeling tools, including Xtext, Sirius, GEMS, Graphiti.

So, these meta-modeling tools benefit the EMF utilities by enabling the users to develop textual/visual DSMLs with the EMF framework.

Concerning the lessons about the language definitions, while most practitioners prefer either diagrammatical or textual visualizations for their DSML notations, developing/using DSMLs with hybrid visualization that promote the use of multiple visualizations in a synchronized manner is neglected.

One of the underlying reasons is metamodeling tools’ lack of technology support for developing hybrid visualizations.

Indeed, most meta-modeling tools support textual or diagrammatic visualizations exclusively.

We also observed that practitioners wish to define the language semantics by means of translations into text or model and most of them ignore the interpretative semantics definition techniques .

This might be due to the fact that the meta-modeling tools essentially provide transformation technologies for defining and executing the translational semantics.

Defining the interpretative semantics for notations seems to be rather addressed by the researchers in academia.

Also, collaborative meta-modeling seems to be highly important for many practitioners, who indicated in the survey that importing/exporting the language definitions and versioning the definitions in a repository for later re-use are among their top-interests for meta-modeling.

Practitioners have been observed to be reluctant to develop/use DSMLs for hybrid modeling that could enable multiple stakeholders with different profiles to edit the same model using different visualizations.

Indeed, as the survey results reveal, most practitioners are not interested in hybrid modeling and they essentially prefer the metamodeling tools with the free-editing mode rather than projectional editing which promotes the same model to be edited by different projections .

This could be attributed to the fact that most practitioners are unfamiliar with the projectional editing, Indeed, the MPS meta-modeling tool that supports the projectional editing seems to be rarely used.

Another important lesson learned is about the model and metamodel re-usabilities.

According to the survey results, re-usability is one of the biggest concerns for practitioners for both modeling and meta-modeling.

Indeed, practitioners are highly interested in defining the language syntax and semantics by re-using the definitions of the existing languages that are kept in a repository without having to define the meta-models from scratch.

Also, practitioners wish the modeling editors produced by the meta-modeling tools to enable re-using the existing models for creating new, extended models.

Besides reusability, another important requirement for practitioners is observed to be the model and meta-model versioning that is concerned with keeping different states of the models and language definitions in a repository for later access and management.

Indeed, many practitioners wish the meta-modeling editors to be integrated with version control systems and provide support for creating different versions of their models that can even be compared and merged.

Concerning the language validation, we learned that defining validation rules is highly important for practitioners most of whom wish the meta-modeling tools to enable defining both the semantical and structural validation rules.

Practitioners also wish the meta-modeling tools to be integrated with some validation tools such as the formal verification and model simulation tools.

By doing so, the validation rules can be defined and executed on the models specified in such a way that the models can be proved for the validation rules.

We learned some lessons about the survey questions for understanding the challenges.

Indeed, despite that each category of the survey includes a question for learning the challenges which the participants face with in that category, many participants omitted the challenge questions.

We attribute this to the fact that the challenge questions are free-text and the participants have been prompt to type their own answers.

Given also the feedback we got from the participants that the survey is quite longer than they expect, those participants seemed reluctant to separate further time to state their challenges.

In the future, we will try to come up with a set of potential challenges and prompt the participants to choose among the pre-defined list of challenges.

Lastly, while we asked the participants a set of questions for each category of requirements and received their responses, we did not actually focus on associating the participants’ meta-modeling tool choice with their responses for the questions.

This is firstly because the questions have been intended for learning practitioners’ expectations regardless of their tool preferences and thus presented to the participants accordingly.

Also, most participants seem to have chosen multiple meta-modeling tools and therefore one may not accurately associate the responses with the tool usages.

Therefore, we could not discuss in our paper some potentially interesting outcomes about the relationships between the meta-modeling tools and the question answers for different tool requirements with web-access support).

Fig. 7 shows the participants’ usage frequencies of the metamodeling tools.

So, the top-used meta-modeling tools are Eclipse Sirius and GEMS .

Those are followed by Metaedit+ , and Xtext .

While some participants use the Microsoft DSL tools for meta-modeling, the rest of the meta-modeling tools are rarely used .

Note that 8% of the participants use their own meta-modeling tools that they build for their specific problems .

Fig. 8 shows the correlations between the domains for which the participants develop/use DSMLs and their choice of the metamodeling tools.

So apparently, some of the meta-modeling tools such as ANTLR, GEMOC, EVA, Cameo, JastEMF, and Graphiti are essentially used for a few particular domains.

While GME, Melange, and MPS are not among the frequently-used tools, those tools seem to be preferred for various domains.

In almost all the domains, the top-used metamodeling tool is Metaedit+ - except the embedded domain where the top-used meta-modeling tool is GEMS.

In Fig. 9, we give the correlations between the participants’ choice of meta-modeling tools and their work countries.

The tools that are used in the greatest number of countries are GEMS, Metaedit+, Sirius, and Xtext.

GEMS is top-used in Turkey, followed by the Netherlands.

The meta-modeling tools such as ANTLR, GEMOC, WebGME, Melange, JastEMF, Graphiti are each used in one or two countries.

Also, some countries seem to use particular meta-modeling tools more than other tools.

Indeed, USA’s top used tool is Microsoft DSL tools, Finland’s is Metaedit+, many European countries’ top-used tools are Eclipse-based tools , and the Netherlands and Turkey’s is GEMS.

Note that the Netherlands also use MPS quite a lot.

Another interesting finding here is that many countries include some practitioners who essentially prefer to use in-house solutions for modeling/meta-modeling without using any of the existing meta-modeling tools.

Summary

We have described the development of a domain-specific language for payroll applications at DATEV.

The DSL reduces complexity in terms of the core domain and infrastructure dependencies , increases quality by simplifying testing, immediate feedback, and a step-wise build process , is accessible to non-expert programmer end users and integrates with existing architectures and build pipelines, while keeping deployment options flexible .

This way, the language helps DATEV address core business challenges including keeping track with evolving law , the need to develop new and innovative products faster and running those on a wide variety of platforms .

While not everything was smooth sailing, the DSL is now in productive use.

Conclusions At a more general level, all involved parties agree that the goals set out for the DSL-based development process have largely been reached, as far as we can tell after a few years of development and use.

Both the business programmers and the infrastructure developers recognize the value of separating the domain and technical concerns.

Business programmers have been overheard saying that “we really don’t care about the technical implementation in the data center” are happy that they can implement, test and deploy new functionality reliably within a single sprint, something that was not feasible before.

The infrastructure developers are also happy because they “don’t have to care about the complexity of payroll calculation”, and that they are able to ship cross-cutting optimizations in the generated code with relative ease.

The simple migration from JEE to Spring also drove home this benefit.

The Future Based on the positive experience with the DSL, it will be expanded in the future.

Currently, around 15% of the overall domain functionality has been reimplemented with the DSL, and it will take a few more years to get to 100%.

As of now, two major extension of the scope of the DSL are currently being adressed.

The first is data transfer to external consumers where we are working on language features to specify the respective data mappings.

The second extension is the creation of reports and lists, where the DSL will support query and aggregation of data.

In addition, an MPS-based DSL is also being developed by a second department of DATEV, the one that provides the service of creating and optimizing the yearly tax declarations both for companies and individual citizens in Germany.

That language also relies on KernelF at its core and shares a few foundational extensions such as dates and temporal types, but uses different domain-specific abstractions.

6.1 RQ1 Is a suitably-designed DSL able to significantly reduce the perceived complexity in the payroll domain? Comparison to the old system Specific differences that led to accidental complexity have been pointed out in the chapter already using the LEGACY label.

We will not repeat them here.

Three-Layer Separation A naive look at the payroll domain suggests that it is mostly about complex decisions and calculations.

And indeed, these are relevant.

For example, Sec. 5 shows how we use decision tables and domainspecific data types to reduce complexity and increase readability.

However, most of the language features relate to versioning, temporal data, dependencies and the variability between different versions.

This this is where most of the domain complexity lies.

Addressing these complexities directly in the language allowed to reduce the perceived complexity.

We heard statements like “this looks simple – why do we need a DSL?” Of course it is only simple because of the DSL.

We have seen this three-layer structure – surface logic, hidden complexities, technical aspects – in other domains, too .

Debugging The ease of locating and understanding errors is a major factor for productivity and a major pain point in the LEGACY system.

The DSL brings three improvements: The execution of a calculation collects explanations, end-user relevant messages that explain a potentially non-intuitive result .

The tracer mentioned above that shows the complete calculation tree with values overlaid over the program.

Those two approaches allow the business developer to track down errors without considering technical aspects.

For case , this is different: if a calculation succeeds in the interpreter but fails in the generated Java code, then there is either an error in the interpreter or in the generator; debugging the interpreter implementation or the generated code, together with an engineer, is necessary.

But once the infrastructure is tested, this third step is rare and most of the debugging can be done with methods 1 and 2.

Post-Mortem Debugging If the calculation is correct in the interpreter but then fails in the generated Java, the error must lie in the generator, and the problem must be debugged by a technical developer.

However, sometimes a corner case might occur in a real-world calculation for which no test exists, leading to a faulty result.

To understand this, users can let the tracer create a test case which debugs the calculation in the IDE.

Depending on how often this will occur in practice , we will add functionality to collect the data at runtime and automatically construct a corresponding test case.

6.2 RQ2 Does the use of DSLs and the associated tools increase or decrease the quality of the final product? Reuse of a Mature Language Reuse is a proven means of reducing development effort and increasing quality.

There is lots of research into language modularity and composition , and it works robustly in MPS .

A low-level functional language is an good candidate for reuse because most DSLs require expressions to express arithmetics, comparison and logical operations.

KernelF is such as language, and the payroll DSL uses it as its core.

KernelF and its interpreter has been used in several other projects and it is therefore stable and mature.

In particular, its tests achive 100% branch coverage regarding the semantics definition in the interpreter.

The payroll DSL benefited significantly; we found only one major semantic bug in KernelF and fixed a few minor issues.

Redundant Execution The duplication of execution semantics in the interpreter and the generator adds complexity, and it took some time to align the semantics of the two by ensuring all tests succeed in both environments.

On the other hand, the relatively simpler interpreter acts as a kind of “executable specification” for the more complex generator.

Aligning the two was simplified by the fact that both are ultimately Java, so they could share runtime classes , avoiding discrepancies in smallstep semantics.

We are confident in the approach, because we have used it before in healthcare , where the redundancy was essential to the safety argument.

Generated Low-Level Code Because the mapping to the execution infrastructure is generated, it is very easy to achieve consistency in the implementation.

A change in the use of the infrastructure, a bug fix, or an optimization requires only a change in the generator to update the whole code base.

This approach increases agility for the technical aspects of the system.

Of course, the generator can also be a source of errors: a mistake in the generator replicates effectively into the code base as well.

However, such errors are often relatively easy to find, because lots of things break simultaneously.

Based on our experience in this and other projects, the trade off works: once the generator is tested reasonably well, overall stability increases, and the time to roll out improvements decreases.

Reuse of QA infrastructure We were able to reuse the KernelF infrastructure for testing, including the ability to run interpreted tests on the CI server as well as the facilities for measuring various aspects of coverage for the language implementation.

Multi-Step QA A goal of the DSL is to allow business programmers to express and test the payroll logic without caring about technical aspects .

To this end, we separate functional and technical concerns: models contain only business logic, the generators, runtimes and frameworks take care of the technical aspects.

Our development process adds concerns step by step, which means that a failure diagnoses precisely where a fault lies.

Step concerns and tests the functional correctness.

A failing test indicates an error in the business logic, or, initially, in the interpreter .

Step translates the business logic to Java and thus concerns performance.

We run the same set of tests, and if one fails, either the generator or the interpreter is faulty; likely it is the generator, because it is more complex.

Step adds the infrastructure to make the system scale.

A failure after this step indicates a problem with frameworks or the platform.

Documentation and Communication Because the DSL programs are free of technical concerns and use domain-relevant abstractions and notations, the need for documentation is greatly reduced.

This prevents the documentation from diverging from the code.

The language definition also serves as a formalized interface between the business programmers and the technical teams, which puts their communication and coordination efforts on a more solid foundation, reducing the risk of misunderstandings and inefficiencies.

6.3 RQ3 Can a DSL that reduces complexity be taught to domain-experts in a reasonable amount of time? IDE Support Users wanted tool support beyond the MPS defaults.

For example, they expected buttons to insert data, enum or calculation declarations into a module, intentions to selectively copy declarations inherited from a previous version into the current one for subsequent change, or menu items for creating a test case for a module.

While many of these make sense because they bundle repeated multi-step changes, others were exact duplicates as the default code completion.

For example, typing calc and then using code completion produces which is what our users wanted a button to do.

Once users got familiar with code completion , the requests for these fine-grained UI actions subsided.

Error Checking The quality of analyses and associated error messages is important for the acceptance of the DSL with its users.

We put a lot of effort into the wording of error messages and into making sure they are reported at the correct locations, and with accurate descriptions of what the problem is; many error messages come with quick fixes that automatically fix the problem when triggered by the user.

Liveness Short turnaround times help developers stay “in the flow”.

In addition, for people with limited experience with abstraction such as our users, it is very useful to be able to execute programs immediately and reduce the gap between the program and its execution – which is one of the motivations for live programming .

In our architecture, this rapid turnaround is facilitated by the in-IDE interpreter: users iteratively create models, play with them, and then write tests to verify the behavior in Fig.5).

The Big Picture Reuse between versions was a contested issue: a new version v4 selectively overwrites the declarations from previous versions, requiring the user to look through v1..v3 to understand the effective contents of v4.

End users did not appreciate this need to mentally assemble “everything” from parts to achieve reuse.

To resolve this tension, we exploit MPS’ projectional editor to optionally show inherited declarations in the new version: “everything” can be seen in one place, optionally.

In addition, we integrated automatically-rendered UML-style diagrams to show the relationships between the declarations in a module, as well as a tree view that shows the applicable versions and their effective declarations for a calculation that spans several business areas.

Since each business area can have a different set of versions that might start on different dates, it is not trivial to understand which versions of which business area are applicable for a calculation on some particular date.

End-User Involvement During initial development, involvement of domain experts was difficult.

The team worked on the core language abstractions without focussing on usability.

User feedback would have been negative for “superficial” reasons; we wanted to avoid such negative first impressions.

In addition, many future users struggle with formulating the requirements for the DSL because they are not aware of the design space for the language and IDE.

Instead, the DATEV language developers, themselves former payroll developers, acted as proxies for our users.

Once the language started to mature, future users were integrated more broadly through demo sessions, screencasts and workshops.

The feedback loops were shortend and we focused on more and more detailed aspects of the language and the IDE.

Teaching The best way to teach the DSL is to let future users experience the language.

We did this in four steps.

Language developers create sample models that address common problems in the domain; These samples form the basis for tutorials, demos, screencasts and howtos that illustrate language and tooling in a way that connects with future users.

User/developerpairs implement the examples; and Gradually, users try to independently implement further examples, supporting each other.

Language developers are available as 2nd level support.

Initially the last step was harder than expected; our users told us that routine work didn’t allow them to spend time ”playing around with the DSL”.

Now, after some time of learning, the approach works really well and the business programmers “experiment” with the language as the try to implement new requirements.

Git Most business programmers had not used a version control system before.

To keep the complexity of using Git low, we taught the users the basics using the built-in IDE features of MPS, avoiding the command-line.

In addition, developers try to avoid merge conflicts by using a development process that avoids parallel work on the same parts of the system by different developers in the first place.

Infrastructure A crucial ingredient to limiting the complexity for the end users is that they are not required to deal with any part of the deployment stack; once they get their tests running in the IDE and have pushed the changes into Git, they are done .

LEGACY Developers were required to deal with multiple components of the overall stack, increasing complexity How well does the DSL and its use for application development fit with established IT development processes and system architecture? Layered Architecture The DSL was specifically scoped to cover only the business logic of the domain; integration with the deployment infrastructure is done on the level of the generated code using agreed interfaces.

Before considering a DSL for the business logic, DATEV had already decided to use a microservice architecture and to apply domain-driven design .

Each service would be layered like an onion , with outside-in dependencies.

Figure Fig. 6 illustrates the current architecture of a microservice focusing on DSL integration.

The domain layer contains the generated business logic.

It relies on libraries that form the DSL runtime that are shared among services for the generated DSL code.

The api layer exposes the service functionality to the outside and, in our case, also contains the Driver that provides the current employee, the current date, access to reference data as well as to the other services.

Finally, the infrastructure layer contains technology adapters .

Generating the technology-independent domain layer from models was a natural integration point for the DSL.

The first test of this approach was to remodel, and then regenerate, a manually written domain layer for a prototype microservice.

Agreeing on the DSL runtime interfaces and those implemented by the generated domain layer took a couple of iterations.

In particular, building a common understanding of the relation between versions, their impact on deployment, and an the API that supports cross-version polymorphism for calculation versions took time.

Flexible Deployment From corporate architecture guidelines it was clear from the start that the calculations would run in a distributed, microservice architecture.

However, the allocation of functionality to services was open because of the different trade-offs regarding performance, scalability, stability and service management overhead: every version of every business area a separate service; all versions of a business area in one service; multiple business areas with all their version in one service; all business areas and versions in one service.

It was useful that the DSL can accomodate all four options by adapting generators or build scripts.

In addition, the development of business logic could proceed without deployment decision in the architecture team, which helped to “unblock” the teams.

Ultimately option was chosen for the initial deployment; for example, the two tax-related and the four social-insurancerelated business areas were deployed in joint services, respectively.

A different trade-off might lead to choosing different options in the future.

For now, the mapping of business areas to services is performed outside of the DSL, as part of the build process.

LEGACY The monolithic COBOL architecture could not be broken up easily into different deployment units, making the trade-offs harder to reevaluate.

Even during the development of the system, the execution infrastructure was changed from JEE to Spring; this required changes to method signatures and annotations in the generated POJOs and the persistence layer.

Those changes could be achieved by modifying the generators.

No modification of the business logic was necessary.

Overall, the integration effort into the new technology stack was low, in line with our expectations and the “promise” of model-driven development, DSLs and code generation.

Execution Paradigm A second technical aspect concerns the execution of the computation.

Initially it was not clear whether, when data changes, computations would recalculate everything for a particular employee and month, or whether they would store intermediate results and use the dependencies to incrementally recalculate the transitive closure of the changed data.

The functional nature of the language allows both, after generators and runtimes are adapted.

Currently, we use the simpler from-scratch approach.

More generally, future optimizations in terms of scalability or resource consumption will very likely be implementable in the generators and frameworks, without invasive changes to the DSL programs.

LEGACY The monolithic COBOL architecture relied on an hard-coded, imperative execution paradigm.

Technology-independent Testing A natural consequence of the onion architecture is that the domain layer can be run without infrastructure, by mocking the infrastructure interfaces.

This is an important ingredient of our QA approach, as illustrated by step 2 in Fig. 5.

Generator Complexity Developing the generator to Java was more effort than expected.

One reason was that the functional language had to be mapped to Java’s imperative style.

This led to excessive use of closures in the generated code as well as long, hard-to-debug chained dot expressions; we have implemented a transformation that splits the chains into sequences of variable declarations before generation.

The generated Java code will then also use a sequence of variable declaration statements, making it easier to read and debug.

DATEV initially wanted the generated code to look exactly as if it were hand-written, partly to simplify debugging, partly to preempt those developers who were be sceptical about code generation, and partly to make the integration with the existing infrastructure, frameworks and programming guidelines easier.

We were required to respect naming conventions and use strongly-typed APIs even behind the interfaces to the generated black box.

This led to larger, more complex generators as well as to a significantly bigger codebase compared to a solution that relied on more generic APIs inside the generated code.

Over time, as more and more of the microservices contain generated business logic and the trust in the generator-based approach increases, DATEV realized that the hard requirements for strongly-typed data structures and readable generated code decreases.

As of now, the first microservices process the data structures as JSON and do not rely on strongly typed Java-classes internally.

If this approach continues, this will reduce the complexity of the generator.

Another example is that necessay checks, if a new version of a business object still has a value for a deleted field, doesn’t lead to a ”compilation error” anymore – we now report this as an error during the validation phase, which is fully accepted by the users.

Build Process The automated build shown in Fig. 5 had to be integrated into DATEV’s CI infrastructure.

In principle this is not a problem with MPS – it can be used in headless mode to check, generate and test models.

However, the build infrastructure of KernelF relies on gradle and DATEV required the use of Maven.

Also, setting up an MPS headless build is generally tedious and error prone .

This led to a few weeks of additional effort.

MPS Distribution MPS is a Java application that runs on the desktop.

It does require infrastructure for deploying the tool to the PCs of the users.

The effort to set this up was higher than expected.

Language Updates Like most other IDEs, MPS relies on a plugin system; the languages and IDE customizations used by business programmers are such plugins.

The integration server builds these plugins for every commit, and at the end of each sprint, these are made available to the MPS installations via a CloudFoundry web server.

The MPS installations prompt the user to download the new plugins and potentially run model migrations.

Specific and Generic A well-designed general-purpose programming language has few orthogonal, and composable language concepts that allow users to define their own abstractions.

For DSLs, in contrast, it is less important that users can define their own abstractions; instead, users expect the DSL to come with prededefined abstractions for the use cases relevant to the domain .

However, if a DSL is designed in this rigid way, it cannot grow towards more expressive power over time without expensive structural refactorings.

An extensible functional language like KernelF, together with MPS’ capabilities, provides an elegant middle ground.

Structurally, everything is an Expression.

However, initial iterations of the language only ship with use-case specific, highlevel expressions that are easy for the end users to understand .

As users become more experienced, one can add more expressive constructs .isEmpty) without changing the fundamental architecture of the language.

A second example: in several cases, our end users asked us to remove genericity in favour of a more specific approach ; for example, when assigning to an enum-valued result variable in a calculation, users suggested code completion to propose only the enum literals , not realizing that they might want to compute the value and added them back in over time without any significant change to the language.

Potentially, this filtering can be user-specific.

In the case with the assignment to enum-valued variables we will customize the MPS code completion menu to show the enum literals at the top and in bold, and all the other expressions further down; this will highlight the “simple” approach while still allowing more expressive, generic expressions.

There’s a saying in the computer science community: “Every DSL will eventually evolve into a general-purpose language”.

We think this is wrong – this DSL and other similar ones are not a replacement for Java or C.

However, most DSLs, as they evolve, will need more features that make it Turing complete.

But these languages still have lots of domain-specific concepts in them as well, so they are not general-purpose.

However, when selecting the tooling to build the languge, make sure you chose one that is expressive enough to be able to handle this evolution.

Functional Programming As we have said earlier, a challenge for many business programmers is functional thinking and programming.

The functional approach is very useful for lots of technical reasons – such as easy extensibility and relatively easy analysability – and to provide lots of end-user-relevant features with acceptable implementation effort.

However, many business programmers, especially those who have extensive imperative experience, consider it a challenge.

We mitigate this by providing high-level declarative abstractions for things that are ubiquituous in the domain, so that “low-level functional algorithmic programming” is required rarely.

The approach is called “funclerative programming” in .

The Price of Reuse Language reuse comes at a price: an existing language concept might not be 100% what we need in a DSL.

For example, a keyword might be English instead of German, one might prefer a different default or one might prefer the first operation on a list<T> to be of type T instead of opt<T> because the reusing language does not use option types.

In practice, we usually start a new DSL by reusing language constructs from KernelF initially to get the project going quickly and proof its viability.

In later phases, once we know the investment will not be wasted, we replace them with more ideal, custom-developed constructs.

We have also reused the KernelF-to-Java generator; the low-level abstractions, such as the basic expressions, worked in the DATEV context without problems.

The higher-level the reused language construct, the more likely it is that the choices the original generator developer has made do not fit with the project-specific context.

For example, the generator for messages, a facility for collecting and reporting errors and warnings to the user, did not fit directly.

Luckily, MPS provides mechanisms to override the existing generator in such situations.

SMT It turns out that many analyses that are expected by our business programmers require abstract interpretation [9, 10] on an SMT domain.

An example is checking a set of Boolean expressions for completeness and overlap.

However, users do not necessarily understand why this is so much more complicated than some of the other error checking performed by the IDE.

We have observed the same in other DSL projects .

To make such analyses possible, we would have to translate all of KernelF to solvers like Z3 , and build this transformation in a way that is easily extensible towards constructs from DSLs that extend or embed KernelF.

This is a major task; itemis has been working on for the last few years, but has not yet finished.

This can be seen as a negative consequece of reusing KernelF: our language is now so expressive that it is prohibitively expensive to transate it into SMT.

However, the domain does require this expressiveness, so not reusing KernelF would not make it better.

However, what we can learn from this is that we should develop a successor to KernelF which is integrated with – meaning: translatable to – an SMT solver right from the start.

Attention to Detail There is different emphasis between end users and language engineers regarding detail.

Examples abound.

We had to allow leading zeros in date and month literals as well as German umlauts and § signs in identifiers.

We developed an infrastructure to manage abbreviations of name components .

We spent a lot of time on the exact rounding rules for currency types.

We worked on tool infrastructure to support internationalization for messages and integration with external translation tools.

And we were required to use German-language keywords for DATEV-specific language concepts, which leads to a curious mix of German and English, because the keywords of KernelF-concepts cannot easily be changed to German.

When initially estimating the overall effort for the project, we did not take such requirements into account.

Pros and Cons of the Projectional Editor A projectional editor is a good fit for DSLs like the current one because of its support for non-textual notations such as tables, the ability to use non-parseable, natural language like syntax and its support for more highly-structured, text-template-like notations such as the one for calculation shown in Sec. 6.2.

And since grammar cells have been available, the “feel” of the editor is close enough to a text editor for it to be acceptable to most users.

The projectional editor is also an important enabler for the versatile support in MPS for language extension and composition, because one never runs into parsing ambiguities.

However, we did run into a few limitations.

For example, insertion into the headers of decision tables took a while to get smooth.

And the /yyyy mm dd/ syntax for dates is a non-convincing compromise: dd.mm.yyyy is ambiguous with decimal number literals because a projectional editor has no look-ahead to be able to distinguish the two.

MPS Limitations More generally, we encountered a few limitations of MPS, including keywords in multiple languages, projecting nodes in places other than their location in the AST, execution of expensive global validations, and execution of a single set of tests both using the interpreter and the Java generator.

For , and we developed workarounds, is unresolved.

For a thorough discussion of the “good, bad and the ugly” of MPS regarding the development of large-scale DSLs, we refer to.

MPS, in General From an end user perspective, MPS looks and feels too much like an IDE .

The requirement to install it locally on the users’ PCs is also not a plus.

An ideal tool would run in the browser and feel more like a modern web app, while still supporting all the language engineering available in MPS.

However, as far as the authors know, such a tool is currently not available, even though various communities have started to develop prototypes.

MPS Learning Curve Learning to be a productive MPS language developer is hard, for many reasons: most developers do not have language development experience in the first place, MPS is a powerful tool and has many facets, not everything is as consistent within MPS as it could be, and the documentation is not as thorough and far-reaching as it should be.

While the focus of this chapter is not the mechanics of building the DSL , it is worth pointing out that it took longer than expected for the new language developers to become productive with MPS.

Resistance to Change? It is often said that domain experts and business programmers resist change, such as when moving to a DSL.

We have heard it too in this project.

Upon closer inquiry we have found that what they are really saying is something like: “last time we had to change, the new system was not ready yet, bugs were not fixed fast enough, we were not taught how it works, our feedback was not taken seriously, and, during the period of changing to the new system we were expected to be as productive as during the time before the change”.

Avoid these things, and you will encounter much less resistance.

Over the last three years, DATEV, a leading German payroll services provider, has been developing a domain-specific language for expressing the calculation logic at the core of their payroll systems. The goal is to allow the business programmers to express and test the calculations and their evolution over time in a way that is completely independent of the technical infrastructure that is used to execute them in the data center.

Business programmers are people who are experts in the intricacies of the payroll domain and its governing laws and regulations – but not in software development – which leads to interesting tradeoffs in the design of the DSL.

The specific set of challenges that motivated the development of the DSL are given in Sec. 3.2.

Payroll might seem dull and not too complicated .

However, the need to work on data that changes over time, to follow the evolution of the LaR, and to keep the language understandable for non-expert programmers makes it interesting from a language design perspective.

The need for execution independent of the deployment infrastructure in the data center and on other devices plus the required flexibility in terms of service granularity and packaging into user-facing applications add interesting non-functional challenges.