Teaching Language Engineering Using MPS

Andreas Prinz

Abstract At universities, computer language handling is most often taught with a focus on compiler theory. However, in practical applications, domain-specific languages (DSLs) are much more important. DSLs implement model-driven technology in an understandable way, as models can be expressed easily using DSLs. One interesting domain for DSLs in this context is language handling itself, and many current tools for language handling are model-driven and based on meta-models. This chapter connects compiler theory and meta-modelling within a university course about language handling. The course features the relevant theory and uses MPS as a practical tool. We show how MPS is used in the course and discuss its suitability.

1 Introduction

Model-driven development (MDD) 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, (1) because the standard modelling language was chosen to be UML [37]; (2) because models were used as illustrations, and not as specifications; and (3) 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 domain-specific (modelling) language [10]. It is important to use full languages, not only

A. Prinz (✉)
Department of ICT, University of Agder, Grimstad, Norway
e-mail: Andreas.Prinz@UIA.no

A. Bucchiarone et al. (eds.), *Domain-Specific Languages in Practice*,
https://doi.org/10.1007/978-3-030-73758-0_11

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 (DSL), 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 meta-model-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.

The chapter is organized as follows: Sect. 2 describes the course formalities in order to set the scene. After that, Sect. 3 discusses meta-languages and concepts for simple language design. After that, Sect. 4 describes how MPS supports these meta-languages and concepts. Section 5 describes the experiences with the course and the possible improvements to MPS for the course. Finally, Sect. 6 concludes the chapter.

2 Course Requirements and Challenges

As usual at universities, the course description is given at a very high-level stating content and learning outcomes. As usual, the teacher (author) had a lot of influence on the course description. Please note that the course is a general language engineering course, not an MPS course.

2.1 IKT445 Generative Programming

The course "Generative Programming" started as a course on software engineering and compiler construction. After some years, it turned out that compiler construction was too far from the typical work tasks of our students. The relevant area for students would be domain-specific languages. In addition, the area of model-driven development (MDD) had to be taught to them. These two ideas led to a new version of the course covering MDD and DSL. The course IKT445 "Generative Programming"[1] has the following course description.

Level:	MSc
Duration:	1 semester
Prerequisites:	Object-oriented programming, UML modelling
Credits:	7.5 ECTS credits
Literature:	[1] [8]
Exam:	Multiple-choice tests (50%) and project (50%)

[1] See also the course home page https://www.uia.no/studieplaner/topic/IKT445-G.

Content:
- Model-driven development and meta-models
- Handling of structural, syntactical, and semantic language aspects
- Code generator theory and application
- Handling systems containing generated and manual code
- Grammars, languages, and automata

Learning outcomes:

On successful completion of the course, the student should:

- Know the concepts and terms of language description and use them correctly in arguments
- Be able to apply best practice of language engineering
- Be able to analyze and design high-level language descriptions capturing all language aspects
- Be able to translate between languages, grammars, and automata
- Be able to design code generation from high-level descriptions

2.2 Selecting a Tool for the Course

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] (called BaseLanguage) in MPS [39].[2] Moreover, MPS has been extended and used in industrial projects [52].

2.3 Content and Design of the Course

With a tool in place, we could focus on the course design. As meta-modelling and language handling tend to be challenging, the main focus has to be on *learning by doing*. This is achieved with a project that the students work with to get a deeper understanding of the theory; see Sect. 2.4. Also, the course starts with a small project that helps the students to get an overview early in the course. This small project is run as "compiler in a day," where students implement a tiny state chart language with four concepts[3] during one day. This event is usually run remotely by MPS with local support given by the course teachers.

As described in more detail in [13], we have investigated how a course that primarily focused on compiler theory could be updated to include meta-model-based approaches to language definition and a particular focus on determining the optimal abstraction level for each language aspect. Based on this, the course contains seven units that cover meta-model-based and compiler-based approaches to language definition. A small Petri net language with six concepts is used as sample language.

Unit 1 Introduction: Compilers, languages, meta-languages
Unit 2 Structure: MDA, meta-models, abstract syntax
Unit 3 Editors: Graphical and textual editors, connection to the structure
Unit 4 Constraints: Static and dynamic checks, type systems, constraints, lexic
Unit 5 Parsers: Grammars, top-down and bottom-up parsing, errors
Unit 6 Transformations: Code generation, M2M, M2T, templates, generators
Unit 7 Execution: Interpreters, runtime environments

Each of the seven units runs for 2 weeks with the following layout:

1. Short lecture of the main theory
2. Individual study: reading, videos, experiments, and test-RAT
3. RAT: IRAT, TRAT, Feedback
4. Project group work: students work with their project

[2] Please note that MPS does not define the semantics of BaseLanguage—it is just translated to Java.

[3] We use the number of concepts as an indication of language size for student projects.

In this layout, the readiness assurance test (RAT) is central. It is a multiple-choice test to ensure that the students have the necessary abilities and knowledge to work on more profound problems, typically in their project.

First, the students solve the individual RAT (IRAT) for themselves. Afterward, the same RAT is solved as a team RAT (TRAT), where students discuss in their team about the RAT and agree on a joint solution. Naturally, team results are much better than the individual results. Finally, students get feedback from the teacher in case there is disagreement or the answer is unclear. In the study period before the RAT, students have access to a test RAT that helps them to prepare for the RAT.

2.4 Projects

Students work with their project in the seven units and in the final exam period. The project is done in groups of students and is about implementing a small language or extension of a language. The teacher selects the languages in cooperation with the students. Often, one group of students starts the work on the language, and in the later years, other groups improve the implementation and focus on specific aspects of the language. There is also the chance of contributing to existing real-life languages.

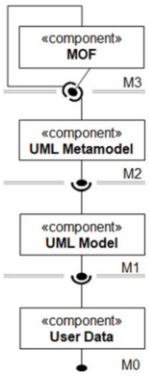
Typically, the project has to be small enough for novices to be managed. Most often, the languages are stored in the local git repository of our university, but some languages have wider visibility. The following languages are worth noting.

ODD implements the overview, design concepts, and details (ODD) protocol. The protocol has seven thematic sections: (1) purpose; (2) entities, state variables, and scales; (3) process overview and scheduling; (4) design concepts; (5) initialization; (6) input data, and (7) sub-models [19, 20]. Each section contains questions to guide modellers in the provision of related model details. ODD emerged as an effort to make model descriptions more understandable. Although ODD is a big step toward verification, validation, and reproducibility of simulation models, the informal character of the answers allows ambiguities in the model description. Our version of ODD[4] with 128 concepts is formal and can be used to generate and run NetLogo code.

COOL, the classroom object-oriented language [2] is a language to teach compiler construction. It is a small object-oriented language that is simple enough to be handled in a short time frame. COOL is also complex enough to have all the essential features of a best-practice modern language. The COOL MPS project[5] with 38 concepts provides an IDE for COOL and translates COOL to Java.

[4] https://github.com/uiano/odd2netlogo.

[5] https://github.com/uiano/COOL-MPS.



OMG Level	Examples	Java example	OCL example
3 = meta meta model	MOF	EBNF/ English	MOF
2 = meta model	UML MM	Java language	OCL language
1 = model	UML Model	a Java program	a formula
0 = instances	objects of UML classes	a run	a truth value

Fig. 1 Four-level modelling architecture according to OMG [27]

ACT ONE is a language for algebraic specifications [11], which is used in another course at our university. As there are no existing tools for ACT ONE, students have created an IDE for ACT ONE with 23 concepts for creating and running specifications.[6] The IDE is in its early phase, and its usability can be improved.

During the years, students have created several publications in this course, or in follow-up projects after the course, for example, in bachelor projects, master projects, or PhD projects. [12–16, 21, 34, 35, 57].

3 Meta-languages

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

[6] https://github.com/uiano/ACT-ONE.

[7] A meta-language description is placed on level M3.

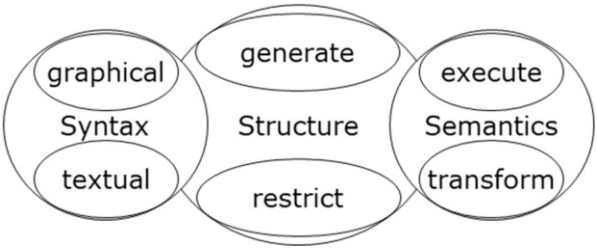


Fig. 2 Aspects of a computer language description

has often stopped at incomplete language descriptions consisting only of structure (defined using MOF [38]) and constraints (defined using OCL [56]). 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 (see Fig. 2).

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 (denotational or translational semantics), or it defines the execution of language instances (operational semantics).[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 (and restricting) 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 well-defined 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

[8]Another type of semantics is axiomatic semantics that gives meaning to phrases of a language by describing the logical axioms that apply to them. Axiomatic semantics is not relevant in this article.

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 model-to-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 high-level 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.

3.1 Concepts for Structure

The structure of a language specifies what the instances of the language are; it identifies the significant components of each language construct [8] and relates them to each other. Although there are proven methods for finding the concepts of a domain [25], most of these methods are not applicable for inexperienced students. Therefore, the course uses extensive feedback to make sure the concepts identified

are useful. There are several ways to express structure; grammars, meta-models, database schema descriptions, RDF schemata, and XML schemata are all examples of different ways to describe structure. The following concepts are considered essential for structure for an introduction to language design:

- Concept and abstract concept
- Enumeration
- Property, child, and reference

3.2 Concepts for Constraints

Constraints on a language can put limitations on the structure of a syntactically well-formed instance of the language. This aspect of a language definition mostly concerns logical rules or constraints on the structure that are difficult to express directly in the structure itself. Neither meta-models nor grammars provide all the expressiveness that is needed to define the set of (un)wanted language instances. The constraints could be first-order logical constraints or multiplicity constraints for elements of the structure. The following concepts are considered essential for constraints for an introduction to language design:

- Name binding with definition and use, uniqueness, identifier kind, and scope
- Lexical constraints
- Multiplicity constraints
- General constraints
- Type systems: definition and comparison of types

3.3 Concepts for Syntax

The concrete syntax of a language describes the possible forms of a statement of the language. In the case of a textual language, it tells what words are allowed to use in the language, which words have special meaning and are reserved, and what words are possible to use for names. It may also describe what sequence the elements of the language may occur in: the syntactic features of the language. This is often expressed in a grammar for textual languages.

For a graphical, diagrammatic, or tabular language, the situation is similar: the appearance of each concept has to be defined and the possible placements in relation to other concepts. The following concepts are considered essential for syntax for an introduction to language design:

- The appearance of concepts including keywords
- The appearance of properties, children, and references
- Placement of elements (over, under, left, right, in) and indentation

- Notational freedom for users
- Ambiguity
- Highlighting and context assistance

3.4 Concepts for Transformation

The behavior of a language describes what the meaning of a statement of the language is. Two main types of formal ways of defining semantics are called operational and denotational semantics [49]. Denotational semantics in the strict sense is a mapping of a source expression to an input–output function working on some mathematical entities. We use a form, called translational semantics, that is related to model transformations, thereby defining an "abstract" compiler. The following concepts are considered essential for transformation for an introduction to language design:

- Model-to-model and model-to-text transformations
- Template-driven and data-driven transformation descriptions
- Navigation in the input and the output

3.5 Concepts for Execution

Operational semantics [49] describes the execution of the language as a sequence of computational steps, thus defining an "abstract" interpreter. Operational semantics is described by state transitions for an abstract machine, for example, using abstract state machines (ASM) [7]. The state transitions are based on a runtime environment, describing the possible runtime states. The following concepts are considered essential for execution for an introduction to language design.

- Runtime environment with elements independent on the program (e.g., program counter) and elements dependent on the program (e.g., variable locations)
- State changes
- Application of state changes related to sequential order and concurrency

3.6 Tool-Related Concepts

There are many more aspects of languages that could be important when designing real-life languages. MPS has dedicated meta-languages for some of those, for example, languages to design actions, refactorings, intentions, usages, connections to source code control, data flow, stand-alone versions of a tool, and many more. For a compiler course, these aspects are disturbing, and they are not taken into account

in the teaching of the course. However, these meta-languages can be relevant for the concrete student projects.

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 (essential MOF) 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.

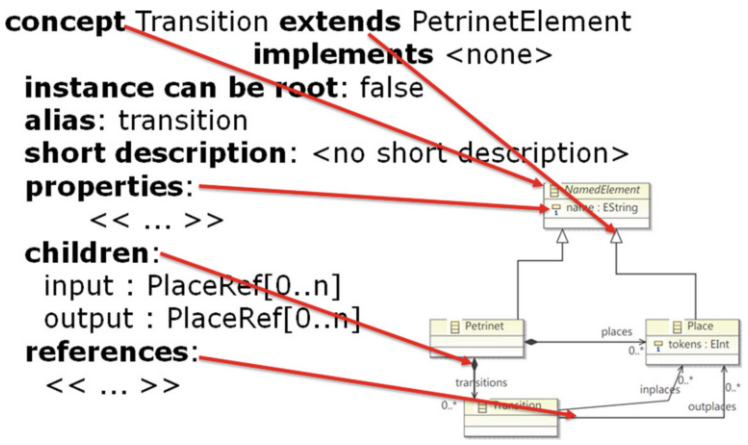


Fig. 3 Structure definition in MPS and EMOF

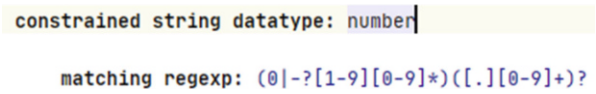


Fig. 4 Lexical constraints in MPS

4.2 Concepts for Teaching Constraints

Constraints (sometimes called static semantics) 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 (logical) 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 (pretty printer) and recognition (parser). 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 (MVC). 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.

```
checking rule ParameterNumberCheck {
  applicable for concept = OperatorReference as oR
  overrides <none>

  do {
    if (oR.parameters.size != oR.ref.parameters.size) {
      error "Wrong number of parameters! Expected: " + oR.ref.parameters.size + "
          oR;
    }
  }
}
inference rule typeof_SortReference {
  applicable for concept = SortReference as sR
  applicable always
  overrides false

  do {
    typeof(sR) :==: sR.reference;
  }
}
subtyping rule AnyRule {
  weak = false
  applicable for concept = Sort as sort

  supertypes {
    return node-ptr/ANY/.resolve(null).sorts.first;
  }
}
```

Fig. 5 Some type system rules of the ACT ONE language

As MPS does not support grammars and recognition-based text handling, this is handled in the course using separate tools (JavaCC). 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

```
<default> editor for concept Entity
  node cell layout:
    [-
    [/
      [> The entity { name } has colour % colour % and shape % shape % and it
      [> Entity { name } has the attributes <]
      (/ % userDefinedAttributes %                                    /)
        /empty cell: [> Press enter to add attribute to { name } <]
      <constant>
      ? Press enter to add another entity
      [> <constant> *context assistant menu placeholder* <]
    /]
    -]

  inspected cell layout:
    <choose cell model>
```

Fig. 6 An editor from the ODD language

```
Style:
hint {
  << ... >>
}

Common:
```

cell id	<default>
action map	<default>
keymap	<default>
menu	<none>
transformation menu	<none>
attracts focus	noAttraction
show if	(editorContext, node)->boolean { return *node*.next-sibling.index < 0; }

```
Constant cell:
```

text	Press enter to add another entity
text*	<none>

Fig. 7 The inspector view of the editor in Fig. 6

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 (Java). 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

```
template reduce_LetReference
input    LetReference

parameters
<< ... >>

content node:
public class xxx {
  public ITerm ccc() {
    final ITerm var = null;
    return <TF [($SWITCH$ switch_IStorable=>ITerm[Specification.Sort]) ->$[var] ] TF>;
  }
}
```

Fig. 8 Model generation from ACT ONE to Java

```
text gen component for concept Entity {
  (node)->void {
    append {breed[ ] ${node.name} { a-} ${node.name} { ]} \n;
    foreach e in node.userDefinedAttributes {
      append ${node.name} {-own} {[ ] ${e.name} { ]} \n;
    }
  }
}
```

Fig. 9 Text generation from ODD to netlogo

concepts of executions are introduced, and possible implementations in MPS are discussed.

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 (level M2) and meta-languages (level M3) and even specifications (level M1) 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.

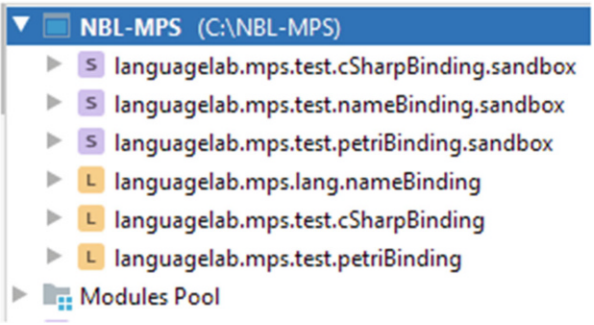


Fig. 10 Level confusion in MPS: lang.nameBinding is a meta-language (M3), test.petriBinding is a regular language (M2), and test.petriBinding.sandbox is a collection of specifications (M1)

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 (simplified) 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 meta-languages 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.

6 Conclusions

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.

Acknowledgments The course and the work on the teaching setup would not have been possible without my PhD students Themis Dimitra Xanthopoulou, Renée Schulz, Vimala Nunavath, Terje Gjøsæter, Trinh Hoang Nguyen, Liping Mu, and Merete Skjelten Tveit.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (1986)
2. Aiken, A.: Cool: a portable project for teaching compiler construction. SIGPLAN Not. **31**(7), 19–24 (1996). https://doi.org/10.1145/381841.381847
3. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. In: Software, IEEE (2003)
4. Bennedsen, J., Caspersen, M.E.: Model-Driven Programming, pp. 116–129. Springer, Berlin, Heidelberg (2008). http://link.springer.com/book/10.1007%2F978-3-540-77934-60
5. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham (2013)
6. Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: Proceedings of ASE'01, Automated Software Engineering (2001)
7. Börger, E., Stärk, R.F.: Abstract state machines: a method for high-level system design and analysis. Springer-Verlag New York, Inc., Secaucus, NJ (2003)
8. Clark, T., Sammut, P., Willans, J.S.: Applied Metamodelling: a Foundation for Language Driven Development, 3rd edn. (2015). CoRR abs/1505.00149. http://arxiv.org/abs/1505.00149

9. Dai, N., Mandel, L., Ryman, A.: Eclipse Web Tools Platform: Developing Java Web Applications. Eclipse Series. Addison-Wesley, Boston (2007)
10. Dmitriev, S.: Language oriented programming: the next programming paradigm. JetBrains onBoard **1**(2) (2004)
11. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, 1st edn. Springer Publishing Company, Incorporated, Berlin, Heidelberg (2011)
12. Gjøsæter, T., Prinz, A.: Teaching computer language handling - from compiler theory to metamodelling. In: Fernandes, J.M. , Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering (GTTSE2009). Revised Papers. Lecture Notes in Computer Science, vol. 6491, pp. 446–460. Springer, New York (2009). https://doi.org/10.1007/978-3-642-18023-1_14
13. Gjøsæter, T., Prinz, A.: Teaching model driven language handling. ECEASST **34** (2010). https://doi.org/10.14279/tuj.eceasst.34.591
14. Gjøsæter, T., Prinz, A.: Languagelab 1.1 user manual. Tech. rep., University of Agder (2013). http://brage.bibsys.no/xmlui/handle/11250/134943
15. Gjøsæter, T., Isfeldt, I.F., Prinz, A.: Sudoku - a language description case study. In: Gasevic, D., Lämmel, R., Wyk, E.V. (eds.) Software Language Engineering (SLE2008). Revised Selected Papers. Lecture Notes in Computer Science, vol. 5452, pp. 305–321. Springer, New York (2008). https://doi.org/10.1007/978-3-642-00434-6_19
16. Gjøsæter, T., Prinz, A., Nytun, J.P.: MOF-VM: instantiation revisited. In: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, pp. 137–144 (2016). https://doi.org/10.5220/0005606101370144
17. Glässer, U., Gotzhein, R., Prinz, A.: The formal semantics of SDL-2000: status and perspectives. Comput. Netw. **42**(3), 343–358 (2003). https://doi.org/10.1016/S1389-1286(03)00247-0
18. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification. The Java Series, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (2000)
19. Grimm, V., Berger, U., DeAngelis, D.L., Polhill, J.G., Giske, J., Railsback, S.F.: The ODD protocol: a review and first update. Ecol. Modell. **221**(23), 2760–2768 (2010). https://doi.org/10.1016/j.ecolmodel.2010.08.019
20. Grimm, V., Polhill, G., Touza, J.: Documenting social simulation models: The ODD protocol as a standard, pp. 117–133. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-540-93813-2_7
21. Guttormsen, S.M., Prinz, A., Gjøsæter, T.: Consistent projectional text editors. In: Pires, L.F., Hammoudi, S., Selic, B. (eds.) Proceedings of MODELSWARD 2017, pp. 515–522. SciTePress, Setúbal (2017). https://doi.org/10.5220/0006264505150522
22. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) Model Driven Architecture - Foundations and Applications, pp. 114–129. Springer, Berlin Heidelberg (2009)
23. International Telecommunication Union: Z.100 Series, Specification and Description Language SDL. Tech. rep., International Telecommunication Union (2011)
24. JetBrains: MPS meta programming system. https://www.jetbrains.com/mps/
25. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. Wiley & Sons, Inc., Hoboken, NJ (2007)
26. Kleppe, A.: A language description is more than a metamodel (2007). This paper is published through a website (megaplanet.org) only. No paper copy available.; 4th International Workshop on Software Language Engineering, ATEM 2007
27. Kleppe, A., Warmer, J.: MDA Explained. Addison-Wesley, Boston (2003)
28. Klint, P., van der Storm, T., Vinju, J.: Easy meta-programming with Rascal. In: Proceedings of GTTSE'09, pp. 222–289. Springer, Berlin, Heidelberg (2011)
29. Konat, G., Kats, L., Wachsmuth, G., Visser, E.: Declarative name binding and scope rules. In: Czarnecki, K., Hedin, G. (eds.) Software Language Engineering, pp. 311–331. Springer, Berlin, Heidelberg (2013)

30. Madsen, O.L., Møller-Pedersen, B.: A unified approach to modeling and programming. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I, MODELS'10, pp. 1–15. Springer, Berlin, Heidelberg (2010). http://dl.acm.org/citation.cfm?id=1926458.1926460
31. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (2002)
32. Microsoft: Getting started with domain-specific languages. https://docs.microsoft.com/de-de/visualstudio/modeling/about-domain-specific-languages?view=vs-2019
33. Mosses, P.D.: Structural operational semantics modular structural operational semantics. J. Log. Algebr. Program. **60**, 195–228 (2004). http://dx.doi.org/10.1016/j.jlap.2004.03.008
34. Mu, L., Gjøsæter, T., Prinz, A., Tveit, M.S.: Specification of modelling languages in a flexible meta-model architecture. In: Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, August 23–26, 2010. Companion Volume, pp. 302–308 (2010). https://doi.org/10.1145/1842752.1842807
35. Nytun, J.P., Prinz, A., Tveit, M.S.: Automatic generation of modelling tools. In: Rensink, A., Warmer, J. (eds.) Proceedings of ECMDA-FA 2006. Lecture Notes in Computer Science, vol. 4066, pp. 268–283. Springer, New York (2006). https://doi.org/10.1007/11787044_21
36. OMG Editor: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1. Tech. rep., Object Management Group (2011). http://www.omg.org/spec/QVT/1.1/
37. OMG Editor: Unified Modeling Language: Infrastructure version 2.4.1 (OMG Document formal/2011-08-05). OMG Document. Published by Object Management Group, http://www.omg.org (2011)
38. OMG Editor: Meta Object Facility (MOF). Tech. rep., Object Management Group (2016). https://www.omg.org/spec/MOF
39. Pech, V., Shatalin, A., Völter, M.: JetBrains MPS as a tool for extending Java. In: Proceedings of the Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '13, pp. 165–168. ACM, New York (2013). https://doi.org/10.1145/2500828.2500846
40. Plotkin, G.D.: A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Aarhus University (1981). http://opac.inria.fr/record=b1049300
41. Prinz, A., Mezei, G.: The art of bootstrapping. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) MODELSWARD 2019, Revised Selected Papers. Communications in Computer and Information Science, vol. 1161, pp. 182–200. Springer, New York (2019). https://doi.org/10.1007/978-3-030-37873-8_8
42. Prinz, A., Shatalin, A.: How to bootstrap a language workbench. In: Hammoudi, S., Pires, L.F., Selic, B. (eds.) Proceedings of MODELSWARD 2019, pp. 345–352. SciTePress, Setúbal (2019). https://doi.org/10.5220/0007398203470354
43. Prinz, A., Scheidgen, M., Tveit, M.S.: A model-based standard for SDL. In: Gaudin, E., Najm, E., Reed, R. (eds.) Proceedings of SDL 2007: Design for Dependable Systems. Lecture Notes in Computer Science, vol. 4745, pp. 1–18. Springer, New York (2007). https://doi.org/10.1007/978-3-540-74984-4_1
44. Prinz, A., Møller-Pedersen, B., Fischer, J.: Object-oriented operational semantics. In: Proceedings of SAM 2016, LNCS 9959. Springer, Berlin, Heidelberg (2016)
45. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
46. Sadilek, D.A., Wachsmuth, G.: Using grammarware languages to define operational semantics of modelled languages. In: Oriol, M., Meyer, B. (eds.) Objects, Components, Models and Patterns, TOOLS EUROPE 2009. Proceedings, Lecture Notes in Business Information Processing, vol. 33, pp. 348–356. Springer, New York (2009). https://doi.org/10.1007/978-3-642-02571-6_20
47. Scheidgen, M.: Textual Editing Framework (2008). Accessed 14 April 2020. http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/documentation.html

48. Scheidgen, M., Fischer, J.: Human Comprehensible and Machine Processable Specifications of Operational Semantics, pp. 157–171. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72901-3_12
49. Sethi, R.: Programming Languages: Concepts and Constructs. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
50. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional, Boston (2009)
51. Şutîi, A.M., van den Brand, M., Verhoeff, T.: Exploration of modularity and reusability of domain-specific languages: an expression DSL in metamod. Comput. Lang. Syst. Struct. **51**, 48–70 (2018). https://doi.org/10.1016/j.cl.2017.07.004. http://www.sciencedirect.com/science/article/pii/S1477842417300404
52. Szabó, T., Völter, M., Kolb, B., Ratiu, D., Schaetz, B.: mbeddr: extensible languages for embedded software development. In: Proceedings of the Conference on High Integrity Language Technology, HILT '14, pp. 13–16. ACM, New York (2014). https://doi.org/10.1145/2663171.2663186
53. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
54. Völter, M.: From programming to modeling - and back again. IEEE Software **28**(06), 20–25 (2011). http://dx.doi.org/10.1109/MS.2011.139
55. Völter, M., Szabó, T., Lisson, S., Kolb, B., Erdweg, S., Berger, T.: Efficient development of consistent projectional editors using grammar cells. In: Proceedings of Conference on Software Language Engineering, SLE 2016, pp. 28–40. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2997364.2997365
56. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
57. Xanthopoulou, T.D., Prinz, A., Shults, F.L.: Generating executable code from high-level social or socio-ecological model descriptions. In: i Casas, P.F., Sancho, M., Sherratt, E. (eds.) System Analysis and Modeling Conference, SAM 2019, Proceedings. Lecture Notes in Computer Science, vol. 11753, pp. 150–162. Springer, New York (2019). https://doi.org/10.1007/978-3-030-30690-8_9