Developing a DSL in the MPS

Tim Kräuter

February 18, 2021

1 Introduction

This document summarizes creating and using a Domain Specific Language (DSL) in the Meta Programming System (MPS). The development of a DSL in the MPS consists of three steps, as depicted in Figure 1.

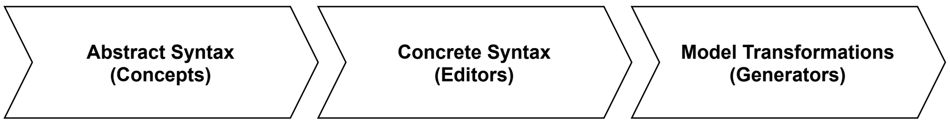


Figure 1: Workflow of developing a DSL in the MPS

First of all, one has to define the abstract syntax of the DSL, also known as the *metamodel*. We will use the term abstract syntax instead of metamodel in the rest of this document, but the terms can be seen as equivalent in the context of DSLs.

Secondly, we can define an *editor* for each concept of the abstract syntax, resulting in concrete syntax.

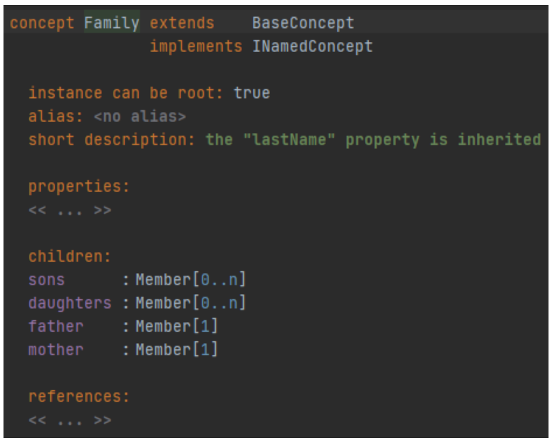
Finally, we can define model transformations between our languages using the *generator* in the MPS.

2 Abstract Syntax

Creating the abstract syntax is done using *concepts*, which are part of the structure aspect of a language in MPS. Concepts are defined textually and are similar to a class or interface in object-oriented modeling.

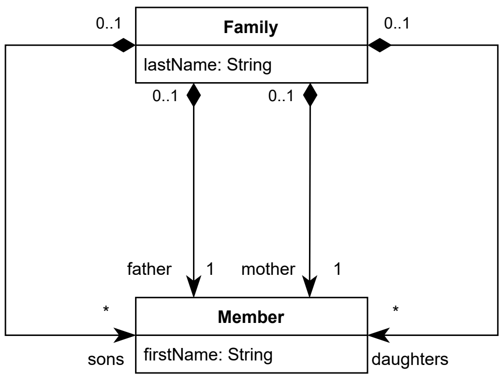
A concept defines the structure of a concept instance, a node of the Abstract Syntax Tree (AST) of the DSL. This means which properties the nodes might contain, which nodes may be referred to, and what children nodes are allowed [4].

For example the families metamodel (Figure 2b) and a concept for the class `Family` (Figure 2a) are depicted in Figure 2.



```
concept Family extends    BaseConcept
               implements INamedConcept

  instance can be root: true
  alias: <no alias>
  short description: the "lastName" property is inherited

  properties:
  << ... >>

  children:
  sons      : Member[0..n]
  daughters : Member[0..n]
  father    : Member[1]
  mother    : Member[1]

  references:
  << ... >>
```

(a) Family concept in the MPS



(b) Families metamodel

Figure 2: Example of the family concept for the families metamodel

3 Concrete Syntax

The term concrete syntax might be misleading regarding MPS because the resulting editor works directly on the AST. Consequently, the term projectional editor is used in MPS since each editor is a projection of the AST.

We can define as many editors as we need for one abstract syntax. For example, we can implement multiple projectional editors for a single concept, as seen for an if-statement in Figure 3.

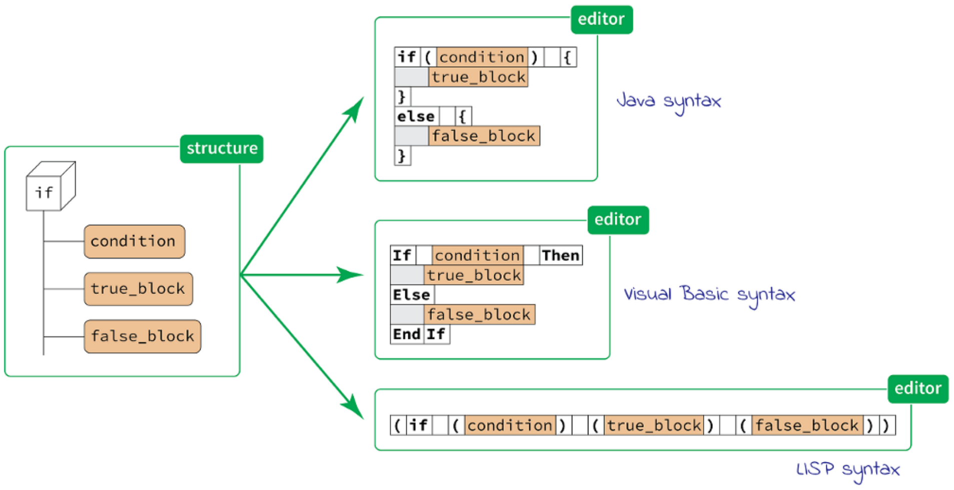


Figure 3: If-statement with multiple projectional editors [1]

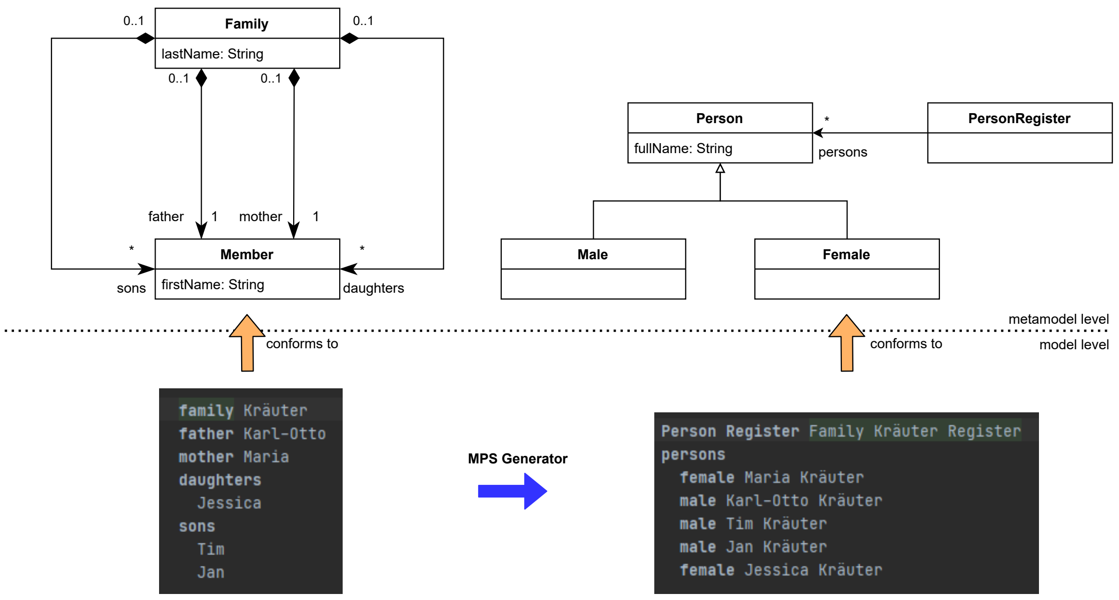
We have also defined an editor for the family concept described in section 2. However, editors cannot be captured sufficiently by a screenshot since plenty of information is contextual and shown in the MPS inspector. However, the project, including the editor and more, can be found in [5].

4 Model Transformations

Model transformations are implemented by using the MPS *generator* aspect. The generator is a hybrid model transformation language, i.e., a mix of imperative and declarative programming similar to Epsilon Transformation Language (ETL).

In MPS a model-to-model transformation may involve many intermediate models and ultimately results in an output model [3]. Both model-to-model and model-to-text transformations can be achieved with MPS.

The example model transformation depicted in Figure 4 was realized in [5]. It is inspired by the ATL transformation tutorial in [2].



```
family Kräuter
father Karl-Otto
mother Maria
daughters
  Jessica
sons
  Tim
  Jan
```

```
Person Register Family Kräuter Register
persons
  female Maria Kräuter
  male Karl-Otto Kräuter
  male Tim Kräuter
  male Jan Kräuter
  female Jessica Kräuter
```

Figure 4: Families2Persons model transformation

5 Conclusion

The MPS workbench is a capable tool for creating DSLs and worked well in the limited time we spent with it.

We have by far not described all the features of the MPS. For example, we did not describe defining constraints for the abstract syntax or test cases for our model transformations. Regarding model transformations, a trace model can be created by defining mapping labels in the transformation definition. Afterward, the target node corresponding to a source node can be retrieved. Since we have labels, we can also transform one source node into multiple target nodes and differentiate them by the given labels.

Working on the abstract syntax tree is different from traditional programming in both a positive and a negative way. One has to embrace the IDE functionalities offered by the MPS to become a successful language developer.

With projectional editing and working on the AST, the MPS takes a unique approach to DSL engineering. However, the ability to use a projectional editor in the browser would improve real-world use of the MPS.

References

[1] Mikhail Barash. Looking at code through the prism of jetbrains mps. `https://medium.com/@mikhail.barash.mikbar/looking-at-code-through-the-prism-of-jetbrains-mps-8e9b70e3257d`, 2018.

[2] Eclipse Foundation. ATL/Tutorials - Create a simple ATL transformation - Eclipsepedia. `https://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation`.

[3] JetBrains. Generator - Help - MPS. `https://www.jetbrains.com/help/mps/mps-generator.html`.

[4] JetBrains. Structure - Help - MPS. `https://www.jetbrains.com/help/mps/structure.html`.

[5] Tim Kräuter. Families2Persons MPS Project. `https://github.com/timKraeuter/PCS955-DAT355/tree/main/MPS%20-%20Projects`.

Glossary

AST Abstract Syntax Tree. 1, 2, 4

DSL Domain Specific Language. 1, 3, 4

ETL Epsilon Transformation Language. 3

MPS Meta Programming System. 1–4