



Meta Programming System: An Introduction

Contents

- 1 Introduction
- 2 Language Design
- 3 Language Editor
- 4 Language Generation
- 5 Hands-On

Contents

1 Introduction

2 Language Design

3 Language Editor

4 Language Generation

5 Hands-On

Meta Programming System

The **Meta Programming System** (MPS) ¹ is a language workbench to create **Domain Specific Languages** (DSL).

MPS uses/provides:

- Code storage in an **Abstract Syntax Tree** (AST)
- Projectional editing
- Code generation
- Language extension possibilities

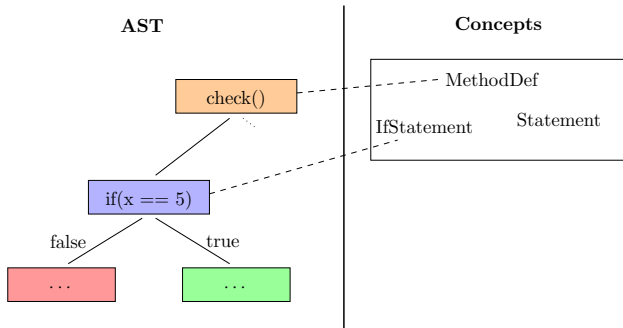
¹<https://www.jetbrains.com/mps/>

DSL Landschaft beschreiben (Grafik auf VuC) -> Basis auf AST ohne Parser

Abstract Syntax Tree

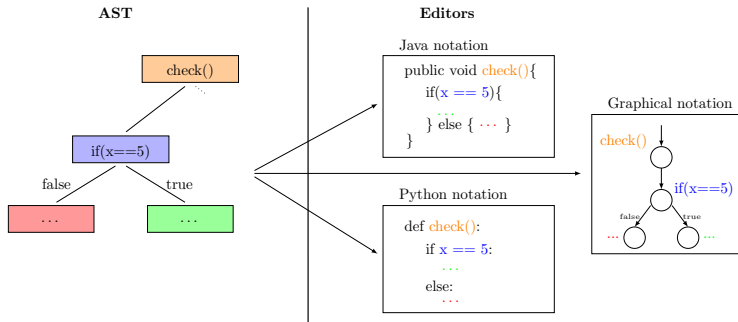
MPS is using an AST as its underlying model, therefore no specific parser is necessary.

The language definition is based on AST-nodes, which build the abstract syntax tree.



Projectional Editing

The **Projectional Editor** of MPS is a visual representation of the current AST. It is possible to have multiple editors with different presentation aspects.



Contents

1 Introduction

2 Language Design

3 Language Editor

4 Language Generation

5 Hands-On

§ Structure²

The structure of a language is defined as a **Concept** in MPS

Concept:

- Inheritance
- Implementation of Interface
- Properties:
 - Enumeration
 - Primitive Datatype
 - Constrained Datatype
- Children:
 - Any concept
 - Multiplicities ([1], [1..n], [0..n], [0..1])
- References:
 - Reference to another node

```
concept IfStatement extends Statement
    implements <none>

instance can be root: false
alias: if
short description: <no short description>

properties:
<< ... >>

children:
condition      : Expression[1]
trueBranch     : Statement[1]
falseBranch    : Statement[0..1]

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

²<https://www.jetbrains.com/help/mps/structure.html>

Contents

1 Introduction

2 Language Design

3 Language Editor

4 Language Generation

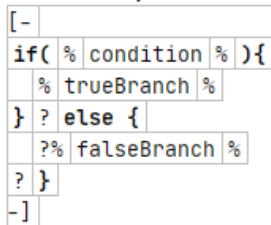
5 Hands-On

E Editor³

Types of Cell Models:

- **Constant cell:** <constant>
- **Property cell:** {property}
- **Child cell:** %child%
- **Referent cell:** (%reference%->{name})
- **Child list cell:** (>%child%/empty cell: <default><)
- **Model access:** *model access*
- **Collection cell:** [- -] (indent layout) or
[> <] (horizontal) or
[/ /] (vertical)

<default> editor for concept **IfStatement**
node cell layout:






inspected cell layout:
<choose cell model>

³<https://www.jetbrains.com/help/mps/editor.html>

💡 Intentions⁴

Intentions:

- Provide  Intention menu by pressing  + 
- Execute predefined actions
- Can be used to correct errors (**error intention**)

Variants:

- Intention
- Universal Intention
- Surround With Intention
- Parameterized Intention

```
intention AddElseClause for concept IfStatement {
    error intention : false
    available in child nodes : false

    description(node, editorContext)->string {
        return "Add Else-Clause";
    }

    isApplicable(editorContext, node)->boolean {
        return node.falseBranch.isNotNull;
    }

    execute(node, editorContext)->void {
        node.falseBranch = new node<Statement>();
    }
}
```

⁴<https://www.jetbrains.com/help/mps/mps-intentions.html>

◉ Checking Rules⁵

Checks:

- Inspect the model for known error patterns
- Static code analysis
- Reports found errors/warnings/infos
- Can provide quick fixes for errors/warnings

```
checking rule UniqueVariables {  
    applicable for concept = MethodDef as methodDef  
    overrides <none>  
  
    do {  
        set<string> names = new hashset<string>;  
        methodDef.variables.forEach({~it =>  
            if (names.contains(it.name)) {  
                error "Duplicate variable: " + it.name -> it;  
            } else {  
                names.add(it.name);  
            }  
        });  
    }  
}
```

⁵<https://www.jetbrains.com/help/mps/typesystem.html>

Quick Fixes⁶

Quick Fixes can be attached to the error, warning or info call in a checking rule via the

 Inspector menu

```
quick fix RemoveDuplicateNames
```

```
arguments:
```

```
<< ... >>
```

```
fields:
```

```
<< ... >>
```

```
description(node)->string {  
    "Replace duplicate name";  
}
```

```
execute(node)->void {  
    node:Variable.name = node:Variable.name + "_d";  
}
```

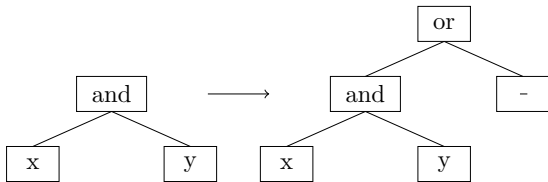
⁶<https://www.jetbrains.com/help/mps/typesystem.html>

Transformations⁷

Transformations let you edit the AST by replacing and moving AST nodes

Example:

(x and y) type `or` should yield
((x and y) or _)



```
transformation menu for concept BooleanExpression : default
```

```
section({ side transform : right }) {
  action
    text (editorContext, node, model, pattern)->string {
      "or";
    }
    can execute <always>
    execute (editorContext, node, model, pattern)->void {
      node<BooleanExpression> oldRoot = node;
      node<OrExpression> newRoot
        = node.replace with new(OrExpression);
      newRoot.left = oldRoot;
    }
    <no additional features>
}
```

⁷<https://www.jetbrains.com/help/mps/transformation-menu-language.html>

Contents

1 Introduction

2 Language Design

3 Language Editor

4 Language Generation

5 Hands-On

Language Generation

In MPS there are two possible ways of generating (here: transforming) code.

Model-To-Model Transformation

- Translate models into other models
- Predefined models exist:
 - Base Language (Java)
 - mbeddr (C and C extensions) ⁸
 - MPS CSharp (C#) ⁹
- Does not flush text into file
- Each predefined model has an associated Model-To-Text Transformation



Model-To-Text Transformation

- Convert a model into text
- Give the output some reasonable layout
- Lets you define a file ending (e.g. .java)
- Flush text into a file

⁸<http://mbeddr.com/>

⁹https://github.com/vaclav/MPS_CSharp

Model-To-Model Transformation ¹⁰

Each generator consists of ➡ **Mapping Configurations** that combines all templates

Some Generator Rules:

- **Root Mapping Rule:** Generates a root node in the output model
- **Reduction Rule:** Transforms a node based on a template
- **Mapping Label:** Helper for name consistency throughout generation

```
mapping configuration main
top-priority group false

mapping_labels:
<< ... >>

parameters:
<< ... >>

is_applicable:
<aAlways>

conditional_root_rules:
<< ... >>

root_mapping_rules:
[concept MethodDef --> MethodDef
inheritors false
condition <aAlways>
keep input root default]

weaving_rules:
<< ... >>

reduction_rules:
[concept IfStatement --> reduce_IfStatement
inheritors false
condition <aAlways>]
```

¹⁰<https://www.jetbrains.com/help/mps/generator-language.html>

Templates ¹¹

Template Macros used in a **Template Fragment** <TF TF>:

- **Property** \$[]: Computes value of a property
- **Reference** ->\$[]: Computes referent node
- **\$IF**\$[]: Conditional generation of template code
- **\$LOOP**\$[]: Applies template to set of nodes
- **\$CALL**\$[]: Calls another template with parameters
- **\$COPY_SRC**\$[]: Copies node
- **\$LABEL**\$[]: Registers generated name into generation context

```
template reduce_IfStatement
input IfStatement

parameters
<< ... >>

content node:
<TF [if ($COPY_SRC$[true]) {
    $COPY_SRC$[String x = ""; ]
}] TF>

Inspector
jetbrains.mps.lang.generator.structure.CopySrcNodeMacro

copy/reduce node

comment      : <none>
mapping label : <no label>
mapped node  : (genContext, node)->node<> {
                node.condition;
            }
```

¹¹<https://www.jetbrains.com/help/mps/generator-language.html>

Template Combination 1/4

```
concept ClassDef extends BaseConcept
                        implements INamedConcept
```

```
instance can be root: false
alias: <no alias>
short description: <no short description>
```

```
properties:
```

```
<< ... >>
```

```
children:
```

```
method : MethodDef[1]
```

```
references:
```

```
<< ... >>
```

Now the MethodDef concept is contained in a ClassDef

T

The ClassDef generator template contains a statically generated method `print()`

The ClassDef generator template contains the generated output of the MethodDef concept

```
template ClassDef
input      ClassDef

parameters

<< ... >>

content node:
<TF> public class $[ClassName] {
    public void print() {
        System.out.println("...");
    }

    $COPY_SRC$[public void methodDef() {
        <no statements>
    }
}
TF>
```

Template Combination 3/4

To generate the method, generate its children:

- Variables (e.g. `$COPY_SRCL$[String s = " ";]`)
- Statements (e.g. `$COPY_SRCL$[s = " ";]`)

What is now needed to always call the print-method of the ClassDef-concept generation output? The method is currently not available in this context.

```
template MethodDef
input      MethodDef
```

```
parameters
<< ... >>
```

content node:

```
<TF [ public void $[methodDef]() { TF>
    $COPY_SRCL$[String s = " "; ]
    $COPY_SRCL$[s = " "; ]
}
```

T Template Combination 4/4

To use the print-method, simulate the surrounding environment necessary for the generation (e.g. the surrounding class with its print()-method)

Mark the code that should be generated with the Template Fragment macro

```
template MethodDef
input MethodDef

parameters
<< ... >>

content node:
public class ClassDef {
    public void print() {
        <no statements>
    }
}

<TF public void $[methodDef]() { TF>
    $COPY_SRCL$[String s = ""; ]
    $COPY_SRCL$[s = ""; ]
    print();
}
```

TextGen⁸

The TextGen language operations:

- **append:** append text of the following kind:
 - **{string value}:** constant text
 - **\n:** line break
 - **\$list{node.list}:** list without separator
 - **\$list{node.list with ,}:** list with separator “,”
 - **#{node.child}**
- **with indent { code }:** increase indentation level for code
- **indent buffer:** apply indentation for current line
- **increase depth:** increase indentation level
- **decrease depth:** decrease indentation level

```
text gen component for concept ClassDef {
  file name : <Node.name>
  file path : <model/qualified/name>
  extension : (node)->string {
    "java";
  }
  encoding : utf-8
  text layout : <no layout>
  context objects : << ... >>

  (node)->void {
    append {public class } ${node.name} {{\n};
    with indent {
      indent buffer;
      append ${node.method};
    }
    append {\n}};
  }
}
```

⁸<https://www.jetbrains.com/help/mps/textgen.html>

Contents

- 1 Introduction
- 2 Language Design
- 3 Language Editor
- 4 Language Generation
- 5 Hands-On**

Hands-On

After this introduction into the world of MPS, there is a repository with all necessary information under <https://aqua-scm.cs.tu-dortmund.de/aqua/mps-workshop>