Model-Driven Engineering Tools
Xtext and MPS

Bentley James Oakes
Bentley.Oakes@uantwerpen.be

University of Antwerp
Flanders Make
1 Production System

2 Xtext
   ■ Meta-model and Model
   ■ Abstract and Concrete Syntaxes
   ■ Constraints
   ■ Model-to-Text Generation
   ■ Model-to-Model Transformation

3 Meta-Programming System (MPS)
   ■ Meta-model and Model
   ■ Abstract and Concrete Syntaxes
   ■ Constraints
   ■ Model-to-Text Generation
   ■ Model-to-Model Transformation

4 Conclusion
This presentation will present examples of the *ProductionSystem* language within two model-driven engineering tools: **Xtext** and **MPS**. We'll move through six topics when creating the *ProductionSystem* language in each tool:

- Meta-models and models
- Abstract and concrete syntax
- Constraints
- Modularizing languages
- Model-to-text generation
- Model-to-model transformation (briefly)
1 **Production System**

2 **Xtext**
   - Meta-model and Model
   - Abstract and Concrete Syntaxes
   - Constraints
   - Model-to-Text Generation
   - Model-to-Model Transformation

3 **Meta-Programming System (MPS)**
   - Meta-model and Model
   - Abstract and Concrete Syntaxes
   - Constraints
   - Model-to-Text Generation
   - Model-to-Model Transformation

4 **Conclusion**
Production systems are composed of *Machines* connected by *Segments*. *Items* travel along these segments and are operated upon by different machines operated by *Operators*. A *Schedule* language specifies the order for operators to operate the machines in
Outline

1. Production System

2. Xtext
   - Meta-model and Model
   - Abstract and Concrete Syntaxes
   - Constraints
   - Model-to-Text Generation
   - Model-to-Model Transformation

3. Meta-Programming System (MPS)
   - Meta-model and Model
   - Abstract and Concrete Syntaxes
   - Constraints
   - Model-to-Text Generation
   - Model-to-Model Transformation

4. Conclusion
Xtext is a framework for development of programming languages and domain-specific languages (DSLs). With Xtext you define your language using a powerful grammar language. As a result you get a full infrastructure, including parser, linker, typechecker, compiler as well as editing support for Eclipse, any editor that supports the Language Server Protocol and your favorite web browser.

Website: https://www.eclipse.org/Xtext/
**Xtext Structure**

**Top:** A DSL is created by defining a grammar on the top level. Xtext then generates plugin code to define an editor (parser, generation code, etc.)

**Bottom:** This plugin runs in another instance of Eclipse. The user can then write their models in this custom editor.
Meta-model:

```
CylAssembler:
  'CylAssembler:' frag_SegmentData
  'linked=' linked=[CubeAssembler]
    ('currOp=' currOp = [Operator])?
  ','
  ;

; 
```

This grammar defines the abstract and concrete syntax for a cylinder assembler

Model:

```
CylAssembler: Cylassemble ins=s2 outs=J1 linked=Cubeassemb ;
```

This text is in the Production System editor, and defines a CylAssembler instance.
- The DSL meta-model is specified in Xtext using a textual grammar.
- The rules (generally) follow Extended Backus-Naur Form (EBNF).
- These rules define options for taking characters and producing data structures (the process of parsing).
- The model file is then parsed using this grammar to build the Abstract Syntax Tree (AST).
This is the definition of a rule `CylAssembler` and a fragment

Blue literals are the literal characters to find

name=ID means that the attribute name is given a value by the token that matches the built-in rule ID

Square braces are references, * is zero-or-more, ? is optional
This grammar defines both the abstract and concrete syntax of the language.

- Can define white-space aware languages (like Python) too.
Grammars are tricky to construct, very leaky abstraction
Have to learn syntax and then carefully predict how parsing will happen

**Issue:** Ambiguous grammars
- Left recursion - Example: Term: Term + Op

**Difficult to debug:**
Decision can match input such as "RULE_ID" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
**Reason:** Operator and another rule only had name=ID
Here, the Schedule language refers to an *Operator* from the Production System language

```plaintext
import "http://www.uantwerpen.be/ProductionSystem" as PS

Schedule:
' Sched' name=ID
  'operator:' operator=[PS::Operator|QualifiedName]
steps+=Step+;
```

In the Schedule model:

```plaintext
Sched sched1
operator: be.uantwerpen.Prod1.Alice
```

Can refer to any Operator in ProductionSystem models in the same folder (automatically)

**Trick:** Must add referencedResource in GenerateSchedExample.mwe2
Editor Features

From the grammar, Xtext is able to do:

- Auto-complete
  - Text and references
  - Can customize the scope
- Custom warnings and errors
Xtext also automatically generates an Ecore metamodel file (*.ecore)

Visualized using Ecore Diagram Editor plugin
Arbitrary Java code written in the ProductionSystemValidator.java file

Constraints validated while written code or on button press
Model-to-Text Generation

- Template-based code (or string concatenation) written in the `ProductionSystemGenerator.xtend` file

```java
var name = resource.allContents.filter(ProductionSystem).head.getName()
fsa.generateFile(name + '.dot',
);

digraph "name" {
  "FOR seg : resource.allContents.filter(Segment).toListable"
  "FOR out_seg : seg.outs"
    "seg.name" -> "out_seg.name";
  "ENDFOR"
  "ENDFOR"
  "FOR inspect : resource.allContents.filter(Inspection).toListable"
    "inspect.name" -> "inspect.accept.name" [color=green];
    "inspect.name" -> "inspect.fix.name" [color=blue];
    "inspect.name" -> "inspect.destroy.name" [color=red];
  "ENDFOR"
}
```

Generates DOT code for generating a graph
Can write Atlas Transformation Language (ATL) rules to transform a model

Uses .ecore files as metamodel and .xmi files as model

```java
module PStrans;  
-- @path PS=./ProductionSystem.ecore  
-- @path PN=./PN.ecore  
create OUT: PN from IN: PS;
```

```java
rule Arrival2Petri {  
  from  
  s: PS!Arrival  
  to  
  p1: PN!Place (  
    name <- 'P' + s.name  
  ),
```

Also see Epsilon - https://www.eclipse.org/epsilon/
Conclusion

Pros:
- Xtext is easy way to build up a language, editor, generator...
- Integrates well with Eclipse ecosystem
- Provides metamodel and models in plain files or Ecore files

Cons:
- Have to become familiar with parsing
- Very difficult to understand how to achieve something
- Lack of documentation, support is from 2-3 people on forums

Tutorials:
1 Production System

2 Xtext
   - Meta-model and Model
   - Abstract and Concrete Syntaxes
   - Constraints
   - Model-to-Text Generation
   - Model-to-Model Transformation

3 Meta-Programming System (MPS)
   - Meta-model and Model
   - Abstract and Concrete Syntaxes
   - Constraints
   - Model-to-Text Generation
   - Model-to-Model Transformation

4 Conclusion
The Meta-Programming System (MPS) is a language workbench to design domain-specific languages (DSLs). It uses projectional editing which allows users to overcome the limits of language parsers, and build DSL editors such as ones with tables and diagrams.

Website: http://www.jetbrains.com/mps/
MPS Structure

- MPS is very explicit about DSLs
- DSLs are defined in languages (the orange L in the figure) - the meta-model
- Multiple DSLs are then used in solutions (the purple S) - the models
A unique feature of MPS is that the user defines *aspects for each concept*:

- **Structure** - The abstract syntax
- **Editor** - Definition of concrete syntax
- **Constraints** - Constraints on attributes and references
- **Behavior** - Java-like utility code, concept constructor
- **Typesystem** - Define typeign rules (eg. MyString is a String)
- **textGen** - Simple text generation
- And a few more...
Structure Example

Language:

```plaintext
concept Inspection extends BaseConcept
    implements Machine

instance can be root: false
alias:
short description:

properties:
children:
references:
fixSeg    : Segment[1]
destroySeg : Segment[1]
acceptSeg : Segment[1]
```

The structure aspect defines the abstract syntax for an inspection machine

Model:

```
Inspection Inspec1 Ins: J2 Outs: Items: Fix fix_seg_in
Destroy Incin1
Accept recv_seg
```

This defines an Inspection machine in the model
The AS for the Production System is defined in the *structure*

- Has explicit inheritance, properties, children, references, and cardinality

```plaintext
<concept> Inspection </concept> extends BaseConcept
  implements Machine

instance can be root: false
alias:
short description:

properties:
children:
references:
  fixSeg   : Segment[1]
  destroySeg : Segment[1]
  acceptSeg : Segment[1]
```
Editor aspect:

```
<default> editor for concept Inspection

node cell layout:

| > Inspection # frag_Segment # [ / |
| > Fix ( % fixSeg % -&gt; { name } ) ] |
| > Destroy ( % destroySeg % -&gt; { name } ) ] |
| > Accept ( % acceptSeg % -&gt; { name } ) ] |

[/]
```

- The *editor* aspect defines the concrete syntax for the concept
- This is very flexible, and can offer tables, diagrams, images in the syntax
  - Spectrum from textual to graphical syntax

Model:

```
Inspection Inspect1 Ins: J2 Outs: Items:
Fix fix_seg_in
Destroy Incin1
Accept recv_seg
```
But how can the user create an Inspection machine, what’s the syntax?

MPS uses *projectional editing*, where the user edits the Abstract Syntax Tree (AST) directly.

That is, MPS only lets the user create what is valid at that time.

Creating a new machine with Ctrl-Space

After selecting the Inspection Machine:

```
Inspection <no name> Ins: << ... >> Outs: << ... >> Items: <no items> Fix <no fixSeg>
Destroy <no destroySeg>
Accept <no acceptSeg>
```
Two large pitfalls in MPS:

- Getting used to projectional editing
  - Not writing text like programming, but building up the model as a tree
- Languages and models are stored as MPS-specific XML
  - Can use version control inside MPS

```xml
<nodes concept="6EZK7" id="100s4CrBT3z">
<property role="TrG5h" value="sched-alice" />
<ref role="OT_FE" node="4t2UbpND4Ff" resolve="Alice" />
<node concept="6EZKv" id="4t2UbpNDbnB" role="6EZKo" />
<node concept="6EZKi" id="4t2UbpNDbnH" role="6EZKo">
<property role="6EZKh" value="4" />
<ref role="6EZKn" node="100s4CrBUhl" resolve="Cubearr" />
</node>
</nodes>
```
MPS makes it trivially easy to mix and extend languages

```plaintext
schedule sched-alice operator : Alice {
  steps :
  walking step
  machine step machine : Cubean
  duration : 4
}
```

Used Languages and DevKits

- SchedLang
- ProdSysLang
Constraints

- Arbitrary Java-like code written in the constraints aspect
- Feedback aspect used for pop-up errors/warnings

Constraint aspect for *Machine* concept:

```java
property {name}
    get <default>
    set <default>
    is valid (propertyValue, node)->boolean {
        propertyValue.matches("[A-Z][a-z]*\[0-9]*");
    }
```
MPS has a textGen aspect for simple text generation

```plaintext

text gen component for concept Inspection {

  (node)->void {

    append indent ${node.name} {->} ${node.acceptSeg.name} { [color=green]} \n;
    append indent ${node.name} {->} ${node.fixSeg.name} { [color=blue]} \n;
    append indent ${node.name} {->} ${node.destroySeg.name} { [color=red]} \n;

  }
}

Generates DOT code for generating a graph
```
MPS implements model transformation as reduction rules
Main purpose is to generate simpler and simpler models, then to generate code/text
Example: Petri Net with inhibitor arcs $\rightarrow$ PN w/o IA $\rightarrow$ LoLA net
Idea is to promote language “stacks”
Conclusion

Pros:
- Very easy to start building languages and models with different languages
- Variety of aspects, which are explicit for each concept
- Concrete syntax can be extended and flexible
- Good documentation and tutorials
- Can generate plugins for other Jetbrains IDEs, or whole language editors

Cons:
- Projectional editing can be difficult to get used to
- Languages and models are not stored as plain-text
- Doesn’t operate in standard ecosystem

Tutorials:
- [https://dev.to/antoine/creating-a-simple-language-using-jetbrains-mps-c7d](https://dev.to/antoine/creating-a-simple-language-using-jetbrains-mps-c7d)
1 Production System

2 Xtext
   - Meta-model and Model
   - Abstract and Concrete Syntaxes
   - Constraints
   - Model-to-Text Generation
   - Model-to-Model Transformation

3 Meta-Programming System (MPS)
   - Meta-model and Model
   - Abstract and Concrete Syntaxes
   - Constraints
   - Model-to-Text Generation
   - Model-to-Model Transformation

4 Conclusion
Two model-driven engineering tools have been presented by implementing the *Production System* language:

- **Xtext** - https://www.eclipse.org/Xtext/
- **MPS** - https://www.jetbrains.com/mps/

Questions or comments?