Model-Driven Development - From Frontend to Code

Sven Efftinge  
sven@efftinge.de  
www.efftinge.de

Bernd Kolb  
bernd@kolbware.de  
www.kolbware.de

Markus Völter  
voelter@acm.org  
www.voelter.de
Model Driven Development is about making software development more **domain-related** as opposed to **computing related**. It is also about making software development in a certain domain **more efficient**.
MDSD Core Concepts

- Target software architecture
- Metametamodel
- Domain Specific Language
- Model
- Metamodel
- Domain
- Ontology
- Bounded area of knowledge/interest
- Composable
- Subdomains
- Partial
- Multiple
- Viewpoint
- Precise/executable
- Knowledge
- Semantics
- Compile
- Interpret
- Transform
- Compile
- Interpret
- Target software architecture
- Design expertise
- Multi-step
- Single-step
- No roundtrip
- Several
How does MDSD work?

- Developer develops **model(s)** based on certain metamodel(s).

- Using **code generation templates**, the model is transformed to executable code.

- Optionally, the **generated code is merged** with manually written code.

- One or more **model-to-model transformation steps** may precede code generation.
• **Goals:**
  • We need an **end-to-end tool chain** that allows us to build models, verify them and generate various artefacts from them.
  • All of this should happen in a homogeneous environment, namely Eclipse.

• **Challenges:**
  • **Good Editors** for your models
  • **Verifying** the models as you build them
  • **Transforming/Modifying** models
  • **Generating** Code
  • **Integrating** generated and non-generated code
Roadmap for the two Sessions

- **Session 1**
  - We will start by defining a **metamodel** for state machines, based on the UML metamodel
  - We will then build a **graphical editor** for state machines using the well-known UML-based notations.
  - We will then add additional **constraints** (e.g. That states must have different names)
  - Next up will be a **code generator** that creates a switch-based implementation of state machines in Java.
  - **Recipes** help developers with the implementation of the actions associated with states.

- **Session 2**
  - We will then cover **model-to-model transformations** and **model modifications**.
  - Finally, we will build a **textual editor** for rendering the state machines textually.
Defining the Metamodel

- A **statemachine** consists of a number of **states**.
- States can be start states, stop states and “normal” states.
- A **transition** connects two states. States know their outgoing and incoming transitions.
- We also support **composite states** that themselves contain sub state machines.
- A state machine is itself a composite state.
- A state has **actions**. Actions can either be entry or exit actions.

- The metamodel is defined using EMF, the **Eclipse Modelling Framework**.
Defining the Metamodel II
Defining the Metamodel III

- The metamodel is defined **using EMF**.
- EMF provides **tree-based editors** to define the metamodel.
- The metamodel has its own project called `oaw4.demo.gmf.statemachine2`
• Note that we have to create the `genmodel` as well as the `.edit` and `.editor` projects from the ecore model.

• This is necessary for the graphical editor to work.
### Building the graphical Editor

- The editor is **based on the metamodel** defined before.

- A number of additional models has to be defined:
  - A model defining the **graphical notation**
  - A model for the editor’s **palette** and other tooling
  - A **mapping model** that binds these two models to the domain metamodel

- A **generator** generates the concrete editor based on these models.

- The editor is build with the Eclipse GMF, the **Graphical Modelling Framework**.
Building the graphical Editor II

EMF

Metamodel .ecore

EMF Genmodel .genmodel

EMF...edit...editor

GMF

Mapping Model .gmfmap

Graphical Notation .gmfgraph

Tool Definition .gmftool

GMF Gen Model .gmfgen

GMF...diagram

Metamodal

EMF

Graphical Editor

GMF

Constraints

oAW Check

Code Generator

oAW xPand

Recipes

oAW Recipes

Model Transformation

oAW xTend

Textual Editor

oAW xText
• We use **another project** for the GMF models from which we’ll create the editor:  
  `oaw4.demo.gmf.statemachine2.gmf`

• This project contains **all the additional models** we talked about before:
Building the graphical Editor IV

- The gmftool model contains the definition of the **palette** that will be used in the editor.

- We have **creation tools** for all the relevant metamodel elements.

- Each of these tools has a **nice icon** associated.
Building the graphical Editor V

- The **Figure Gallery** contains the figures (as well as their associated labels)
  - Shapes
  - Line Style
  - Colors
  - Decorations
- **Diagram Nodes** represent the vertices in the graph that is being edited.
- **Compartments** can be defined as parts of Nodes.
- **Connections** play the role of the edges in the graph.
Building the graphical Editor VI

- We map **nodes** and **links**.
- We **include all the other models** so they can be referenced.
- **Better editors** will become available by GMF final.
- From that, we generate the editor plugins:
Here is the **editor**, started in the runtime workbench, with our CD Player example.
Constraints

• Constraints are **rules that models must conform to** in order to be valid. These are in addition to the structures that the metamodel defines.

• Formally, constraints are part of the metamodel.

• A constraint is a **boolean expression (a.k.a predicate)** that must be true for a model to conform to a metamodel.

• **Constraint Evaluation** should be available
  • in **batch mode** (when processing the model)
  • as well as **interactively**, during the modelling phase in the editor

  ... and **we don’t want to implement constraints twice** to have them available in both places!

• **Functional languages** are often used here.
  • UML’s OCL (Object Constraint Language) is a good example,
  • We use **oAW’s check language**, which is alike OCL
Constraints II

- Constraints are put into the `statemachine2` project, the same as the metamodel.
- **StatemachineBatchErrors** are used in batch validation mode (automatically evaluated every 2 seconds in the editor)
- **StatemachineLiveErrors** prevent erratic modellings in the first place.
Constraints III

- Here are some examples written in oAW’s Checks language.

For which elements is the constraint applicable

<table>
<thead>
<tr>
<th>Constraint Expression</th>
<th>Error message in case Expression is false</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR or WARNING</td>
<td></td>
</tr>
</tbody>
</table>

- Note the code completion and error highlighting 😊

Error message

Code Completion

Metamodel

Graphical Editor

Constraints

oAW Check

Code Generator

oAW xPand

Recipes

oAW Recipes

Model Transformation

oAW xTend

Textual Editor

oAW xText
Constraints IV

- To make the GMF generated editors evaluate our constraints, we needed to **tweak things a little bit**; most of this is in `oaw4.demo.gmf.statemachine2.etc`

- We wrote our own **ConstraintEvaluators** and plugged in the oAW CheckFacade.

- We used **AspectJ** to weave in Adapters into the EMF Factory

- We wrote a **watchdog** that does the batch evaluations whenever the model does not change for two seconds.

- Also, you have to make two important adjustments in the **gmfgen** model
Constraints V

- In this model there are **two errors**
  - There are two states with the same name (Off)
  - The start state has more than one out-transition
- The validation is executed automatically
- Clicking the error message **selects** the respective “broken” **model element** in the diagram.
Code Generation

- Code Generation is used to **generate executable code** from models.
- Code Generation is **based on the metamodel** and uses **templates** to attach to-be-generated source code.
- In openArchitectureWare, we use a **template language** called **xPand**.
- It provides a number of **advanced features** such as polymorphism, AO support and a powerful integrated expression language.
- Templates can access **metamodel properties** seamlessly.

```plaintext
«DEFINE SwitchBasedImpl FORStateMachine»
«FOR EACH states.typeSelect|State| AS s
public static final int «s.constant»
«ENDFOREACH»
```
What **kind of code** will be generated? How do you implement a state machine?

There are **many ways** of implementing a state machine:

- GoF’s State pattern
- If/Switch-based
- Decision Tables
- Pointers/Indexed Arrays

We will use the switch-based alternative. It is neither the most efficient nor the most elegant alternative, but it’s simple.

For more discussion of this topic, see *Practical State Charts in C/C++* by Miro Samek
Code Generation III: Pseudocode

- Generate an **enumeration** for the states
- Generate an **enumeration** for the events
- Have a **variable** that remembers the state in which the state machine is currently in.
- Implement a function **trigger(event)** which
  - First **switches over all states** to find out the current state
  - Check whether there’s a **transition for the event** passed into the function
  - If so,
    - execute **exit action** of current state,
    - Set **current state** to target of transition
    - Execute **entry action** of this new current state
    - Return
  - And also handle **nested states** 😊
• The **generator** is located in the `oaw4.demo.gmf.statemachine2.generator` project.

• There are a number of **code generation templates**.
  - Extensions are also defined.

• There are also workflow files (.oaw) that control the **workflow** of a generator run.

• Different workflow files contain different “parts” of the overall generator run and **call each other**.

• Workflow files are in some small way like ant files.
The **blue text** is generated into the target file.

- The **capitalized words** are xPand keywords
- **Black text** are metamodel properties
- **DEFINE...END-DEFINE** blocks are called templates.
- The whole thing is called a template file.
• One can **add behaviour to existing metaclasses** using oAW’s **Xtend** language.

```java
import SimpleSM;

String basePath() : basePackage()
String basePackage() : "de.jax";

String constantName(Named this) : name.toUpperCase();
String methodName(Action this) : name.toFirstLower();

String implBaseClass_Name(StateMachine this) : this;
String implClassName(StateMachine this) : name.toFirstLower();
String fqImplBaseClassName(StateMachine this) : basePackage()+"."+implBaseClass_Name();
String fqImplClassName(StateMachine this) : basePackage()+"."+implClassName();
```

• Extensions can be called using **member-style syntax**: `myAction.methodName()`

• Extensions can be used in **Xpand templates**, **Check files** as well as in other **Extension files**.

• They are imported into template files using the **EXTENSION** keyword
• Workflow **loads** the model, **checks** it (same constraints as in Editor!) and then **generates** code.

A component is a "step" in the workflow

```xml
<workflow>
  <component class="oaw.emf.XmlReader">
    <metaModelFile value="statemachine.ecore"/>
    <modelFile value="$(modelFile)"/>
    <outputSlot value="model"/>
    <firstElementOnly value="true"/>
  </component>

  <component class="oaw.check.CheckComponent">
    <metaModel id="mm" class="org.openarchitectureware.emf.Metamodel">EmfMetaModel</metaModel>
    <metaModelFile value="statemachine.ecore"/>
    <checkFile value="statemachine::constrainContent:_constrainBatchErrors"/>
    <expression value="model.eAllContents"/>
  </component>

  <component id="generator" class="oaw.xpand.GenGenerator">skipOnError="true"</component>
    <metaModel idRef="simpleSM"/>
    <expand value="templates::Root::root FOR $(slot)"/>
    <genPath value="$(src-gen)"/>
    <advises value="templates::aspects::Logging"/>
    <beautifier class="org.openarchitectureware.xpand2.output.JavaBeautifier"/>
  </component>
</workflow>
```

A number of parameters are passed in

We invoke the same check file as in the editor

This starts the first, "top level" template

Code is automatically beautified
There are various ways of integrating generated code with non-generated code:

- a)  
- b)  
- c)  
- d)  
- e)  

- generated code
- non-generated code

Metamodel | EMF
---|---
Graphical Editor | GMF
Constraints | oAW Check
Code Generator | oAW xPand
Recipes | oAW Recipes
Model Transformation | oAW xTend
Textual Editor | oAW xText
Recipes II

- To help developers to “do the right thing” after the generator has created base classes and the like, you can use a recipe framework.

- It provides a **task-based approach** to “completing” the generated code with manual parts.

- This works the following way:
  - As part of the generator run, you **instantiate checks** that you write to a file
  - After the generator finishes, the **IDE** (here: Eclipse) loads these checks and verifies them against the complete code base (i.e. Generated + manual)
  - If things don’t conform to the rules, **messages are output** helping the developer to fix things.

- For example, in the state machine case, actions must be implemented in subclasses.
Here’s an error that suggests that I **extend** my manually written class **from the generated base class**:

Recipes can be arranged hierarchically

This is a failed check

“Green” ones can also be hidden

Here you can see additional information about the selected recipe
I now **add the respective extends clause**, and the message goes away – automatically.
Now I get a number of compile errors because I have to implement the abstract methods defined in the super class:

I finally implement them sensibly, and everything is ok.

The Recipe Framework and the Compiler have guided me through the manual implementation steps.

If I didn’t like the compiler errors, we could also add recipe tasks for the individual operations.

oAW comes with a number of predefined recipe checks for Java. But you can also define your own checks, e.g. to verify C++ code.
Here’s the **implementation of the Recipes**. This workflow component must be added to the workflow.

```java
package org.demo.gmf.statemachine.recipe;

import java.util.ArrayList;

public class RecipeCreator extends AbstractExpressionRecipeCreator {

    @Override
    protected Collection internalCreateRecipes( ExprExpression expression, String project ) {
        List<Check> checks = new ArrayList<Check>();
        Object sm = facade.evaluate("this");
        String name = (String) facade.evaluate("name");

        ElementCompositeCheck ecc = new ElementCompositeCheck(sm, "stateemachine implementation must be completed.");
        checks.add( ecc );

        String implClassName = (String) facade.evaluate("fqImplClassName()");
        String implBaseClassName = (String) facade.evaluate("fqImplBaseClassName()");

        JavaClassExistenceCheck existCheck = new JavaClassExistenceCheck( "For the State Machine "+name+" you have to provide an implementation class named "+implClassName,
            project, implClassName );
        ecc.addChild( existCheck );

        JavaSuperTypeCheck superCheck = new JavaSuperTypeCheck("your implementation class has to extend the generated base "+implBaseClassName,
            project, implClassName + implBaseClassName );
        ecc.addChild( superCheck );

        return checks;
    }
}
```

You extend one of a number of suitable base classes...

...and override a suitable template method

You can then create any number of checks.

This one checks that a class extends another one

And return the checks to the framework
### Model Transformations I

- **Model Transformations** create one or more new models from one or more input models. The input models are left unchanged.
  - Often used for stepwise refinement of models and modularizing generators
  - Input/Output Metamodels are different

- **Model Modifications** are used to alter or complete an existing model

- For both kinds, we use the **xTend language**, an extension of the openArchitectureWare expression language.

- **Alternative languages** are available such as Wombat, ATL, MTF or Tefkat (soon: various QVT implementations)
The model modification shows how to add an additional state and some transitions to an existing state machine (emergency shutdown)

```java
import statemachine2::constraints::statemachine;

extension statemachine2::constraints::statemachine;

StateMachine modify(StateMachine sm) {
    sm.transitions.addAll(sm.allConcreteStates().createTransition()) ->
    sm.states.add(createShutDown()) ->
    sm;

    private create State this createShutDown() :
        setName("EmergencyShutDown");

    private create Transition this createTransition(State s) :
        setEvent("Error") ->
        setName("Aborting") ->
        setFrom(s) ->
        setTo(createShutDown());
}
```

Therefore createShutDown() will always return the same element.
Model Transformation III

- The generator is based on an **implementation-specific metamodel** without the concept of composite states.

- This makes the **templates simple**, because we don't have to bridge the whole abstraction gap (from model to code) in the templates.

- Additionally, the **generator is more reusable**, because the abstractions are more general.

- We will show a transformation which transforms models described with our GMF editor into models expected by the generator.
Model Transformation IV

- We want to transform from the editor’s metamodel `statemachine2` to the generator’s metamodel `simpleSM`

```java
import statemachine2;

extension statemachine2::constraints::Statemachine;
extension org::openarchitectureware::util::IO;

create simpleSM::StateMachine createStateMachine(StateMachine sm)
    setName(sm.name) ->
    setInitialState(sm.concreteState().createState()) ->
    states.addAll(sm.allConcreteStates().createState()) ->
    actions.addAll(sm.allStates().createState().createAction()) ->
    events.addAll(sm.allStates().createState().createEvent()) ->

private create simpleSM::State createState(State s) :
    setName(s.name) ->
    transitions.addAll(s.allOutTransitions().createTransition()) ->

private create simpleSM::Action createAction(String n) :
    setName(n);

private create simpleSM::Event createEvent(String n) :
    setName(n);

private create simpleSM::Transition createTransition(Transition t, State s) :
    actions.addAll(t.to.concreteState().name.createAction()) ->
    setEvent(t.event.createEvent()) ->
    setTo(t.to.concreteState().createState());
```

- We need to ‘normalize’ composite states.
- States inherit outgoing transitions from their parent states.
- For those transitions the exit actions are inherited, too.
- Unify action and event elements with the same name.
Textual Editor I

- A graphical notation is not always the best syntax for DSLs.
- So, while GMF provides a means to generate editors for graphical notations, we also need to be able to come up with **editors for textual syntaxes**.
- These **editors need to include** at least
  - Syntax highlighting
  - Syntax error checking
  - Semantic constraint checking
Textual Editor II

- We use oAW’s textual DSL generator framework xText
- Based on a BNF-like language it provides:
  - An **EMF-based metamodel** (representing the AST)
  - An **Antlr parser** instantiating **dynamic** EMF-models
  - An **Eclipse text editor plugin** providing
    - syntax highlighting
    - An **outline view**, 
    - syntax checking
    - as well as **constraints checking** based on a **Check** file, as always oAW
The **grammar** (shown in the boostrapped editor)

The first rule describes the root element of the AST

The **generated** eCore AST model

States contain a number of entry actions, transitions and exit actions

Assigns an identifier to a variable (here: state)

These variables will become attributes of the AST class

Rule names will become the AST classes

- The grammar (shown in the boostrapped editor)
- The generated eCore AST model

A literal

```
"testState":
  "stateMachine" name=ID "("
  (entryActions+=Action) *
  (transitions+=Transition) *
  (exitActions+=Action) *
  (states+=AbstractState) ""
"
```

Abstract AbstractState :
  CompositeState | State:

State :
  "state" name=ID "("
  (entryActions+=Action) *
  (transitions+=Transition) *
  (exitActions+=Action) ""
"

Action :
  "@" name=ID;

Transition :
  event=ID "->" state=ID;
Textual Editor IV

- You can define **additional constraints** that should be validated in the generated editor.

- This is based on oAW’s *Check* language
  - i.e. These are constraints like all the others you’ve already come across

```text
import textualism

extension org.openarchitectureware.xtext.xtext:

context State WARNING "The states name should begin with an upper case letter":
  name.matches("^[A-Z][a-z]*");

context State WARNING "The state \"name\" is never referenced":
  eContainer.eContents.get(0)==this ||
  allTransitions||.exists(t|t.state==name);

context CompositeState WARNING "The composite state \"name\" has no child states":
  states.size==0;

context AbstractState ERROR "Duplicate State \"name\":
  allAbstractStates||.select(e|e.name==name).size==1;

context Transition ERROR "Unicorn State \"this.state\":
  state()==null;

context Transition if state()==null ERROR
  "The state \"this.state\" is not visible from here."
  state().eContainer == containingState||.eContainer;
```
Textual Editor V

- The **generated editor and its outline view**

```plaintext
statemachine CdPlayer {
  // initial state
  state Off {
    @shutDown
    powerSwitchPressed -> Off
  }

  state Open {
    @openTray
    openClosePressed -> On
    powerSwitchPressed -> On
    @closeTray
    powerSwitchPressed -> Off
    @openTray
  }

  /*
   * composite state
   */

  statemachine On {
    @checkCD
    openClosePressed
    powerSwitchPressed
    // children
    state Stop {
      @stopPlaying
      playPressed -> Stop
    }

    state Play {
      @startPlaying
      stopPressed -> Stop
      pausePressed -> Pause
    }

    state Pause {
      @pausePlaying
      stopPressed -> Stop
      pausePressed -> Play
    }
  }
```

- **Literals have become keywords**
- **Constraints are evaluated in real time**
## Tooling Versions

Eclipse 3.1 or Eclipse 3.2, suitable EMF version

**Eclipse >= 3.2M6, GMF >= 1.0M6**

Eclipse >= 3.1, oAW >= 4.0

Eclipse >= 3.1, oAW >= 4.0

Eclipse >= 3.1, oAW >= 4.0

Eclipse 3.2, oAW >= 4.1

Eclipse 3.2, oAW >= 4.1
Summary

- The tool chain we’ve just shown provides an end-to-end solution for MDSD,
  - Completely Open Source
  - Using standards wherever worthwhile,
  - And pragmatic solutions wherever necessary.

- To get the tools, go to
  - www.eclipse.org/emf
  - www.eclipse.org/gmf

- THANK YOU.