Model-Driven Development with openArchitectureWare

Markus Völter
voelter@acm.org
www.voelter.de

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

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

![Diagram showing relationships between Domain Concepts, mental work of developers, and Software Technology Concepts.](image)
MDSD Core Concepts

- several
- design expertise
- multi-step
- single-step
- no roundtrip

```
Model

- Metamodel
  - Metametamodel
    - composable
    - subdomains
      - partial
      - multiple
        - viewpoint

- Domain Specific Language
  - Domain
    - Ontology
  - Metamodel
    - graphical
    - textual

- target software architecture
  - several
  - design expertise
  - multi-step
  - single-step
  - no roundtrip

- transform
  - compile
  - interpret

- knowledge
  - precise/ executable

- Model
  - semantically
  - linguistic

- bounded area of knowledge/interest
```
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 & Challenges

• **Goals:**
  • We need an **end-to-end tool chain** that allows us to build models, verify them and generate various artifacts 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
openArchitectureWare

- Open Source
- Version 4.1.2 is current (and 4.2 in the making)
- Proven track record in various domains & project contexts
  - e.g., telcos, internet, enterprise, embedded real-time, finance, ...
- www.openarchitectureware.org
- IDE-portions based on Eclipse
- (Optional) Integration with Eclipse Modelling facilities (such as EMF)
Roadmap

- 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 notation.
- 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.
- 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 **state machine** 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 Modeling Framework**.
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.
Defining the Metamodel IV

- 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 built with the Eclipse GMF, the **Graphical Modelling Framework**.
Building the graphical Editor II

EMF

Metamodel
.ecore

EMF

EMF Genmodel
.genmodel

...edit
...editor

GMF

Mapping Model
.gmfmap

GMF Gen Model
.gmfgem

GMF

...diagram

Graphical Notation
.gmfgraph

Tool Definition
.gmftool

Metamodel

EMF

Graphical Editor

GMF

Constraints

oAW Check

Code Generator

oAW xPand

Recipes

oAW Recipes

Model Transformation

oAW xTend

Textual Editor

oAW xText
Building the graphical Editor III

- 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:
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.
Model-Driven Development with openArchitectureWare

Building the graphical Editor VI

- We map **nodes** and **links**.
- We **include all the other models** so they can be referenced.
- **Better editors** became available with GMF final.
- From that, we generate the editor plugins:

```
www.openarchitectureware.org
© 2006-7 Völter, Efftinge, Kolb
```
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 based on 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.

```java
importStateMachine;

context StateMachine ERROR "States must have unique Names":
  states.typeSelect(State).forAll(s1 | states.typeSelect(State).
    exists(s2 | s1 != s2 & & (s1.name == s2.name)));

canvas Named if !Transition.isInstance(this) ERROR this.metaType.name +" must be named":
  this.name != null;

canvas StartState ERROR "no incoming transitions allowed":
  this.inTransitions.size == 0;

canvas StartState ERROR "start state must have one out transition":
  this.outTransitions.size == 1;
```

- Error message in case Expression is false
- For which elements is the constraint applicable
- Note the code completion and error highlighting 😊

© 2006-7 Völter, Efftinge, Kolb
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.
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.

```
DEFINE SwitchBasedImpl FOR StateMachine

FOREACH 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 elegante 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** 😊
Code Generation IV

- 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 implBaseClassName(StateMachine this) : basePackage() + implBaseClassName();
String fqImplBaseClassName(StateMachine this) : basePackage() + "\"+implBaseClassName();
```

• 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.

```xml
<workflow>
  <component class="oaw.emf.XmiReader">
    <metaModelFile value="statemachine2.ecore"/>
    <modelFile value="?$(modelFile)"/>
    <outputSlot value="model"/>
    <firstElementOnly value="true"/>
  </component>

  <component class="oaw.check.CheckComponent">
    <metaModel id="hav" class="org.openarchitectureware.xtext.EmfMetaModel">
      <metaModelFile value="statemachine2.ecore"/>
    </metaModel>
    <checkFile value="statemachine2::construct:topLevel1::runtime::BatchErrors"/>
    <expression value="model.eAllContents().isEmpty()"/>
  </component>

  <component id="generator" class="oaw.xpand.Generator" skipOnErrors="true">
    <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>
```
There are various ways of integrating generated code with non-generated code:

- a) generated code
- b) generated code
- c) generated code
- d) generated code
- e) generated code

non-generated code
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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Class</th>
<th>Path</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>The type CbPlayer must implement the inherited abstract method CbPlayerActions.checkCD()</td>
<td>CbPlayer.java</td>
<td>open4-demo.gmf.statechart</td>
<td>line 3</td>
</tr>
<tr>
<td>The type CbPlayer must implement the inherited abstract method CbPlayerActions.dice Tray()</td>
<td>CbPlayer.java</td>
<td>open4-demo.gmf.statechart</td>
<td>line 3</td>
</tr>
<tr>
<td>The type CbPlayer must implement the inherited abstract method CbPlayerActions.pausePlaying()</td>
<td>CbPlayer.java</td>
<td>open4-demo.gmf.statechart</td>
<td>line 3</td>
</tr>
<tr>
<td>The type CbPlayer must implement the inherited abstract method CbPlayerActions.shutDown()</td>
<td>CbPlayer.java</td>
<td>open4-demo.gmf.statechart</td>
<td>line 3</td>
</tr>
<tr>
<td>The type CbPlayer must implement the inherited abstract method CbPlayerActions.startPlaying()</td>
<td>CbPlayer.java</td>
<td>open4-demo.gmf.statechart</td>
<td>line 3</td>
</tr>
<tr>
<td>The type CbPlayer must implement the inherited abstract method CbPlayerActions.stopPlaying()</td>
<td>CbPlayer.java</td>
<td>open4-demo.gmf.statechart</td>
<td>line 3</td>
</tr>
</tbody>
</table>

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 oaw4.demo.gmf.stateMachine2.recipe;

import java.util.ArrayList;

public class RecipeCreator extends AbstractExpressionRecipeCreator {

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

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

        String implClassName = (String) facade.evaluate("fqImplClassName");
        String implBaseClassName = (String) facade.evaluate("fqImplBaseClassName");
        JavaClassExistenceCheck extCheck = new JavaClassExistenceCheck("For the State Machine " + name + " you have to provide an implementation class named " + implClassName,
            project, implClassName);
        ecc.addChild(extCheck);
        JavaSuperTypeCheck superCheck = new JavaSuperTypeCheck("Your implementation class has to extend the generated base class " + 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 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;

extension stateMachine2::constraints::StateMachine;

StateMachine modify(StateMachine sm) {
    sm.transitions.add(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());
    }
```

Extensions can import other extensions. The main function "create extensions" guarantee that for each set of parameters the **identical** result will be returned. Therefore `createShutDown()` will always return the same element.
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.
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.eAllContents.typeSelect (Action).name.createAction()) ->
    events.addAll(sm.eAllContents.typeSelect (Transition).event.name.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.eAllActions(s,t.to.concreteState()).createAction()) ->
    setEvent(t.event.createEvent()) ->
    setTo(t.to.concreteState()).createState());
```
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
We use oAW’s textual DSL generator framework *xText*

Based on a BNF-like language it provides:
1. An **EMF-based metamodel** (representing the AST)
2. An **ANTLR parser** instantiating **dynamic** EMF-models
3. 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)

```plaintext
Literal:
stateState:
  "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;
```

• The **generated** eCore AST model

```plaintext
CompositeState -> AbstractState
  states : AbstractState

AbstractState
  name : EString
  transitions : Transition
  exitActions : Action
  entryActions : Action

State -> AbstractState
  Action
  name : EString
  Transition:
    state : EString
    event : EString

Assigns an identifier to a variable (here: state)

These variables will become attributes of the AST-class

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

Rule names will become the AST classes

The first rule describes the root element of the AST
```

© 2006-7 Völter, Efftinge, Kolb
• 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
The **generated editor** and its **outline view**

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

```plaintext
cdplayer.lsm

statemachine CdPlayer {
  // initial state
  state Off {
    @shutdown
    powerSwitchPressed -> Off
  }
  state Open {
    @openTray
    openClosePressed -> On
    powerSwitchPressed -> Off
    @closeTray [Unknown State Off]
  }
  state Stop {
    @stopPlaying
    playPressed -> Play
  }
  state Play {
    @playAfterPlaying
    stopPressed -> Stop
    pausePressed -> Pause
  }
  state Pause {
    @pausePlaying
    stopPressed -> Stop
    pausePressed -> Play
  }
}
```

www.openarchitectureware.org

© 2006-7 Völter, Efftinge, Kolb

www.openarchitectureware.org
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.