Model-based Software Engineering for/with Petri nets
From models to code

Ekkart Kindler

DTU Compute
Department of Applied Mathematics and Computer Science
Motivation (cf. Vision)

Analysis
Design
Implementation
Coding

Code is generated
Motivation

- If we want to get software automatically from models, we need to have a technology for transforming models into code.
- Programming this transformation manually is very error prone (and not in the spirit of our endeavour).
- In essence, we need a technology for transforming Models to Text: M2T-transformation.
Dear Mr. Miller,

we are pleased to inform you that you will be refunded $133.00 in income tax. The reason is that we had made a mistake.

Best regards,
C. Clerk

Dear Mrs. Jones,

we are pleased to inform you that you will be refunded $1,365.00 in income tax. The reason is that you had overpaid.

Best regards,
K. Kringle

Dear <Name>,

we are pleased to inform you that you will be refunded <amount> in income tax. The reason is that <reason>.

Best regards,
<clerkname>

Today, this is much better known from web programming: PHP or JSP, ASP, ...

Standard text in which some “specifics” will be filled in (attributes/parameters/fields).
There are different concrete template technologies (JET, XPand) that are made for transforming models into some form of text.

The ideas will be presented based on the Java Emitter Templates (JET).

JET is very similar to JSP and the control language is Java: we do not need to learn much new stuff!
1. Example: Overview report

Overview of component definition

The components name is "Move".

The automaton has 4 states:
  init (initial)
  walk
  running
  standing
Example: "template" it

Overview of component definition

The components name is "Move".

The automaton has 4 states:
  init (initial)
  walk
  running
  standing
Example: "template" it more

%--------------------------------
Overview of component definition
%--------------------------------

The components name is "<name>".

The automaton has <no of states> states:
<foreach state>
  <state.name> <initial>
<hcaerof>

%--------------------------------
Example: Java it

%--------------------------------

Overview of component definition
%--------------------------------

The components name is "< c.getName() >".

The automaton has
< c.getAutomaton().getState().size() > states:
< for (State s:c.getAutomaton().getState() ) { >
   < s.getName() > < s.isInitial() ? "(initial)" : "" >
< } >

%--------------------------------
Example: JET it

```java
<%@ jet package="translated" class="SimpleAutomaton"
        imports="dk.dtu.imm.se2e09.casetool.componentdefinition.*" %>
%
%-----------------------------------
Overview of component definition
%-----------------------------------
<%
    ComponentDefinition c = (ComponentDefinition) argument;
    Automaton a = c.getAutomaton();
%>

The components name is "<%= c.getName() %>".

The automaton has <%= a.getState().size() %> states:
<% for (State s:a.getState()) { %>
    <%= s.getName() %> <%= s.isInitial() ? "(initial)" : "" %>
<% } %>
%
%-----------------------------------
```

Not very readable anymore, but does what we want!
Example: Lessons learned

- Idea behind JET is simple
- It is not restricted to any syntax concerning the final output

- The final JET-template is not very readable
- But it is not difficult to "work through" it (see later: class that does transformation)

- Always start from a concrete example, that you turn into a template

As long as you do not know where you are heading it, DON’T even try to make a template.
2. Concepts

---

Overview of component definition

---

JET-directive.

Actually from this JET-template, a Java class will be generated that does the actual transformation.

These directives tell the name, package and imports for this class.

If you want to see this class, go to your runtime workbench and have a look into the hidden JET-project (class SimplePN2Java in the .JETEmitters project) in the navigator view.
<%@ jet package="translated" class="SimpleAutomaton"
    imports="dk.dtu.imm.se2e09.casetool.componentdefinition.*"%>

%--------------------------------

Overview of component definition
%--------------------------------

<%
    ComponentDefinition c = (ComponentDefinition) argument;
    Automaton a = c.getAutomaton();
%>

The components name is "<%= c.getName() %>

The automaton has <%= a.getState().size() %> states:
<% for (State s:a.getState()) { %>
    <%= s.getName() %> <%= s.isInitial() ? "(initial)" : "" %>
<% } %>

%--------------------------------

Text snippet directly going to the text-output (including all spaces, tabs, and line feeds!)
Ekkart Kindler
14MBSE f/w Petri Nets: From models to code

Overview of component definition

In principle, we can use all Java commands and all classes we want; don’t forget to import them!

We also need to configure the class-path of the generated class! This is done when the template is called (see later).

The Object passed to the template as an argument when the template is started.

JET Scriptlet: Must altogether give a legal Java method body!

```java
%@ jet package="translated" class="SimpleAutomaton"
imports=":dk.dtu.imm.se2e09.casetool.componentdefinition.*"
%
%
%--------------------------------
Overview of component definition
%--------------------------------
<%
ComponentDefinition c = (ComponentDefinition) argument;
Automaton a = c.getAutomaton();
%
The components name is "<%= c.getName() %>".
The automaton has <%= a.getState().size() %> states:
<% for (State s:a.getState()) { %>
<%= s.getName() %> <%= s.isInitial() ? "(initial)" : "" %>
<% } %>
%
```

JET Scriptlet: Must altogether give a legal Java method body!
JET/Java expression: must return a String (or something that can be “used” as a String).

More technically, it must be possible to append the returned value to a StringBuffer (see later).

The components name is "<%= c.getName() %>".

The automaton has <%= a.getState().size() %> states:
<% for (State s : a.getState()) { %>
  <%= s.getName() %> <%= s.isInitial() ? "(initial)" : "" %>
<% } %>

Overview of component definition

The components name is "<%= c.getName() %>".

The automaton has <%= a.getState().size() %> states:
<% for (State s : a.getState()) { %>
  <%= s.getName() %> <%= s.isInitial() ? "(initial)" : "" %>
<% } %>

--------------------------------

More technically, it must be possible to append the returned value to a StringBuffer (see later).
More details

Other JET-directives:

- include:
  ```
  < %@ include file="anotherTemplate.jet" %>
  ```

- skeleton:
  Defines the skeleton class to be used (later)

- startTag, endTag:
  used to replace `<%` and `%>` as start and end tags.

- nlString:
  defines the String serving as a newline
3. Applying the template

Up to now:
- What is a template
- What does it do/mean

For more details, have a look into the code of the generator project in our basic project.

In particular, have a look into the automatically generated project .JETEmitters in your runtime workbench.

Question now:
- How do we start (and configure) a transformation for such a template?
- What happens behind the scenes?
import org.eclipse.emf.codegen.jet.JETEmitter;
import org.eclipse.emf.codegen.jet.JETException;
...

ComponentDefinition c = ...

JETEmitter emitter = new JETEmitter(uriOfTemplate,
    getClass().getClassLoader());

emitter.addVariable(
    "CASETOOL", "dk.dtu.imm.se2e09.casetool" );
emitter.addVariable(
    "EMF_COMMON", "org.eclipse.emf.common" );

String result = emitter.generate(
    monitor, new Object[] { c });
Don’t worry! You can typically copy that from other projects (we can look at some practical details later on).
package translated;

import dk.dtu.imm.se2e09.casetool.componentdefinition.*;

public class SimpleAutomaton {

    protected final String TEXT_1 =
            "%-----…" + NL +
            "  Overview of component definition" + NL + "%------...";

    protected final String TEXT_2 =
            NL + NL + "The components name is \"";

    protected final String TEXT_3 =
            "\"." + NL + "" + NL + "The automaton has ";

    protected final String TEXT_4 = " state(s):";

    protected final String TEXT_5 = NL + " ";

    protected final String TEXT_6 = " ";

    protected final String TEXT_7 = NL + " ..." + NL + "%------...";
public String generate(Object argument) {
    final StringBuffer stringBuffer = new StringBuffer();
    stringBuffer.append(TEXT_1);
    ComponentDefinition component = (ComponentDefinition) argument;
    Automaton automaton = component.getAutomaton();
    stringBuffer.append(TEXT_2);
    stringBuffer.append(component.getName());
    stringBuffer.append(TEXT_3);
    stringBuffer.append(automaton.getState().size());
    stringBuffer.append(TEXT_4);
    for (State state : automaton.getState()) {
        stringBuffer.append(TEXT_5);
        stringBuffer.append(state.getName());
        stringBuffer.append(TEXT_6);
        stringBuffer.append(state.isInitial() ? "(initial)" : "");
    }
    stringBuffer.append(TEXT_7);
    return stringBuffer.toString();
}
Here, we used a setting, where the generator code is generated dynamically, which allows to change the JET templates while the software is running!

This is very convenient for development.

The generator code can also be generated once and for all at development time; which is faster and less prone to setup errors in deployed code.
Now:

- We know how to use the generator
- We know how the generated generator class looks (which defines a JET-template’s semantics)

Question now:

- Can we adapt change the generation of the generator class?
  This is interesting, once we call generators from other templates or to implement some helper functionality which should not be in the JET-template

The answer: Skeletons
public class CLASS
{
    public String generate(Object argument)
    {
        return "";
    }
}
public class **CLASS** extends MyClass
{
    private WhateverType attribute;

    public OtherType myMethod()
    {
        ... 
    }

    // Some comment.
    public String generate(Object argument)
    {
        return "";
    }
}
Ekkart Kindler

<%@ jet package="translated" class="SimpleAutomaton"
skeleton="my.skeleton"
imports="dk.dtu.imm.se2e09.casetool.componentdefinition.*" %>

Overview of component definition

ComponentDefinition c = (ComponentDefinition) argument;
Automaton a = c.getAutomaton();

The components name is "<%= c.getName() %>".

The automaton has <%= a.getState().size() %> states:
<% for (State s:a.getState()) { %>
    <%= s.getName() %><%= s.isInitial() ? "(initial)" : "" %>
<% } %>

----------------------------------
5. Summary

- JET is a simple way to define transformations

- Creating JETs is a bit tedious

- To ease working with JET (and getting them right)
  - start from an example
  - be disciplined
  - use helpers (in skeleton)
  - check them after changes
  - run runtime workbench in debug mode, so you do not need to restart runtime workbench after JET changes
Conclusion & Perspective

- Working with JET in practice is a bit tedious to work with (only limited syntactical support by IDE)
- But, I used it for larger generation projects (including Petri nets). When working backwards from an example, this is doable.

- Also important in MBSE: Bidirectional transformations between models (e.g. Triple Graph Grammars / TGGs): M2M

We have a look on how templates (and skeletons) are developed in practice in the hands-on slot.

If there is interest, we could have a look into some of the details.
Appendix
Overview of component definition

The components name is "<%= c.getName() %>".

The automaton has <%= a.getState().size() %> states:
<% for (State s:a.getState()) { %>
   <%= s.getName() %> <%= s.isInitial() ? "(initial)" : "" %>
<% } %>

--------------------------------