Template-Based Code Generation

The slides of this lecture are reused from the Model Engineering course at TU Vienna with the kind permission of Prof. Gerti Kappel (head of the Business Informatics Group)
Code Generation / Model-to-Text Transformation

VHDL language

VHDL specification (abstract syntax)

model-to-text

VHDL specification (concrete syntax)

code generation

Java repository for metamodel

general term used in lecture: model-to-platform
Simplified Example (Blocks instead of VHDL)

// Block Specification for NOTNOT
start[NOTNOT]

// Ports
<= Port.0
=> Port.1

// Contained Blocks
// Block Specification for NOT
start[NOT]
stop[NOT]

// Block Specification for NOT
start[NOT]
stop[NOT]

stop[NOTNOT]
// Block Specification for NOTNOT
start[NOTNOT]

// Ports
<= Port.0
=> Port.1

// Contained Blocks
// Block Specification for NOT
start[NOT]
stop[NOT]

// Block Specification for NOT
start[NOT]
stop[NOT]

// Block Specification for NOTNOT
stop[NOTNOT]
Model-to-Platform: Pro

1. Might be the only possibility to get your models running on a certain platform (C/C++ for embedded systems)

2. DSLs and persistence formats are very often textual

3. Separation of application modelling and technical code to increase maintainability, extensibility, and portability to new hardware, operating systems and platforms

4. Used to define semantics of modeling languages

5. Easy to combine existing framework/library code with application specific code generated from models

6. Potentially as efficient as platform allows

7. Certification (TÜV) is still only possible for generated code and not for an interpreter or the code generator itself
1. Having the same information in two places, code and models, is a recipe for trouble [Juha-Pekka Tolvanen]

2. Often not incremental and therefore time consuming for very large models (disturbs development workflow)

3. Debugging usually on code level and difficult to map to model level

4. Mixing in handwritten code can be very problematic!
Model-to-Platform with Templates

String code = "// Block Specification for" + block.getName() + "\n" + "start["" + block.getName() + "]\n" + ... + "stop["" + block.getName() + "]";

return code;

String code = blockTemplate(block);

blockTemplate(b) ::= <<
  // Block Specification for <b.name>
  start[<b.name>]
  ...
  stop[<b.name>]
>>

inversion of static text and program logic

static text with holes

separation of static output code and model data
Templates: Pro

1. If the amount of static text is much more than program logic, then templates are very readable.

2. Templates can be replaced or adjusted to change the generated code (advantage of separation).

3. Template languages typically provide extra support for common tasks such as file/folder creation, string operations, and declarative model access.

4. Templates can be created and maintained by non-programmers (especially relevant for web-based applications).
Templates: Contra

1. Introduces a further language to be mastered and an additional tool dependency

2. Templates grow ugly with an increasing percentage of program logic

3. Some template languages only support certain platforms and standards
Components of a Template-Based Approach

- model access
- static text
- model traversal
- control flow elements
- template invocation
- template engine

mandatory components

model

text
State-of-the-Art Template Languages/Engines

- **StringTemplate + Tree Grammar**
  - pure DSL, no GPL code allowed

- **Velocity**
  - very flexible, allows invocation of arbitrary GPL (e.g., Java) code
  - clear separation between model traversal and static text

- **XPand**
  - model traversal and static text are merged
State-of-the-Art Template Languages/Engines

- Simple, minimal language with very few features
  - StringTemplate + Tree Grammar

- Dedicated support for a certain modeling standard
  - XPand

- Complex language with advanced features
  - Velocity

Independent of any modeling standard
Velocity


model access

model access is provided via a context which is filled before evaluating templates

Velocity.init();

VelocityContext context = new VelocityContext();
context.put("block", new Block());

Template template = Velocity.getTemplate("block.vm");
StringWriter sw = new StringWriter();
template.merge(context, sw);
// Block Specification for $block.Name
start[$block.Name]
  // Ports
  #foreach( $port in $block.retrievePorts() )
    #if ($port.isOutPort)
      => $port.id
    #else
      <= $port.id
    #end
  #end
  // Contained Blocks
  #foreach( $sub in $block.subBlocks )
    #parse("block.vm")
  #end
stop[$block.Name]

block.vm
«IMPORT BlockLanguage»
«DEFINE blockSpec FOR Block»

«FILE block + ".spec"»
// Block Specification for «name»
start[«name»]
// Ports
«FOREACH this.retrievePorts() AS port»
  «IF port.isOutPort»
    => «port.id»
  «ELSE»
    <= «port.id»
  «ENDIF»
«ENDFOREACH»
// Contained Blocks
«EXPAND blockSpec FOREACH subBlocks»
stop[«name»]
«ENDFILE»

«ENDDEFINE»
a first model transformation (e.g. with SDMs) produces a tree (acyclic hierarchical structure)

model traversal for a tree simplifies to a standard depth-first search and does not need to be specified explicitly
StringTemplate and Tree Grammars
http://www.stringtemplate.org/

Tree Grammar (Controller)
- model access
- control flow elements
- template invocation

Template (View)
- static text
- model access
- control flow elements
- template invocation
Depth-First Traversal with Tree Grammar

```plaintext
block : ^(BLOCK n=STRING p+=port* sb+=block*)
   -> blockTemplate(name={$n},ports={$p},subBlocks={$sb})

port : ^(INPORT id=STRING)
   -> inportTemplate(portId={$id})
   |  ^(OUTPORT id=STRING)
   -> outportTemplate(portId={$id})
```
Decorate Tree Elements with Templates

\[
\text{blockTemplate}(name, ports, subBlocks) ::= \langle<
// Block Specification for \langle name\rangle
\text{start}[<name>]
// Ports
<ports; separator="\n">
// Contained Blocks
<subBlocks; separator="\n">
\text{stop}[<name>]
\rangle>
\]

\[
\text{inportTemplate}(portId) ::= \langle<
<= <portId>
\rangle>
\]

\[
\text{outportTemplate}(portId) ::= \langle<
=> <portId>
\rangle>
\]
Summary: Velocity

+ Very versatile, not tied to any modelling standard, very flexible, allows arbitrary method calls
+ Simple language, easy to learn, flat learning curve

- Can easily lead to unmaintainable templates
- Methods to extract and traverse model must be implemented for each modelling standard
Summary: XPand

+ Explicit support for Ecore/EMF, model traversal is greatly simplified via built in model parser
+ Extremely powerful language with polymorphism, a dedicated expression and validation language, aspect orientation, extra workflow language, …

- Steep learning curve, adds additional complexity
- Templates are not easily reusable in another context, e.g. another modelling standard, same model but different concrete syntax, …
Summary: StringTemplate + TreeGrammars

+ Enforces separation between model traversal and static text (MVC) with usual advantages of modularity and separation of concerns (e.g., view can be swapped)
+ If EBNF is known tree grammars are trivial, templates are extremely simple (complexity is shifted to model-to-tree transformation)

  - Requires a model-to-tree transformation, which might be annoying if no model transformation language is already in use (introduces an extra language to the chain)
  - Model-to-Platform transformation is split into different parts (overkill for simple transformations?)
Summary

- Model-to-platform transformation is of crucial importance for MDE

- Template-based code generation is advantageous

- There are many (many) template languages with different strengths and weaknesses:
  - JET
  - XSLT
  - MOFScript
  - Acceleo
  - FreeMarker
  - ...

14. Juni 2012 | Real-Time Systems Lab | Prof. Dr. Andy Schürr | Dr. Gergely Varró | 23