*11*

## Transformation and Generation

*In the case of both transformation and generation, another artifact is created from a program, often a program in a less abstract language. This is in contrast to interpretation, which executes programs directly without creating intermediate artifacts. Transformation refers to the case in which the created artifact is an AST, and code generation refers to the case in which textual concrete syntax is created. In some systems, for example MPS, the two are unified into a common approach.*

Transformation of models is an essential step in working with DSLs. We typically distinguish between two different cases: if models are transformed into other models, we call this *model transformation*. If models are transformed into text (usually source code, XML or other configuration files), we refer to *code generation*1. However, as we will see in the examples below, de-

1. As we discuss in Part I, we do not

pending on the approach and tooling used, this distinction is not always easy to make, and the boundary becomes blurred.

A fundamentally different approach to processing models is *interpretation*. While in the case of transformation and generation the model is migrated to artifacts expressed in a different language, in the case of interpretation no such migration happens. Instead, an interpreter traverses an AST and *directly* performs actions depending on the contents of the AST. Strictly speaking, we have already seen examples of interpretation in the sections on constraints and type systems: constraint and type checks can be seen as an interpreter where the actions performed as the tree is traversed are checks of various kinds. However, the term interpretation is typically only used

cover generation of byte code or machine code. This is mainly for the following reason: by generating the source code of a GPL, we can reuse this GPL’s compiler or interpreter, including all its optimizations (or platform independence). We’d have to rebuild these optimizations in the DSL’s generator. This is a lot of work, and requires skills that are quite different from those most DSL developers (including me) posses.

|  |  |
| --- | --- |
| Note that when developing transformations and code generators, special care must be taken to preserve or record trace information that can be used for error reporting and debugging. In both cases, we have to be able to go back from the generated code to the higher-level abstractions it has been generated from, so we that can report errors in terms of the higher-level abstractions or show the higher-level source code during a debugging session. We discuss this challenge to some extent in Chapter 15.  *11.1 Overview of the approaches*  Classical code generation traverses a program’s AST and outputs programming language source code (or other text). In this context, a clear distinction is made between models and source code. Models are represented as an AST expressed with some preferred AS formalism (or meta meta model); an API exists for the transformation to interact with the AST. In contrast, the generated source code is treated as text, i.e. a sequence of characters. The tool of choice for transforming an AST into text are template languages. They support the syntactic mixing of model traversal code and to-be-generated text, | tween transformation and generation versus interpretation in the chapter on language design (Section 4.3.5). |
| separated by some escape character3. Since the generated code is treated merely as text, there is no language awareness (and corresponding tool support) for the target language while edit- | 3 Xpand and Xtend use «guillemets». |
| ing templates. Xtend4, the language used for code generation | 4 Xtend is also sometimes referred |
| in Xtext, is an example of this approach.  Classical model transformation is the other extreme, in that it works with ASTs only and does not consider the concrete syntax of either the source or the target languages. The source AST is transformed using the source language AS API and a suitable traversal language. As the tree is traversed, the API of | to as Xtend2, since it has evolved from the old oAW Xtend language. In this chapter we use Xtend to refer to Xtend2. It can be found at **www.eclipse.org/xtend/**. |
| the target language AS is used to assemble the target model5. | 5 Note that in this chapter we look at |

for cases in which the actions actually *execute* the model. Execution refers to performing the actions that are associated with the language concepts as defined by the execution semantics of the concepts. We discuss interpretation in the next chapter2.

1. We elaborate on the trade-offs be-

For this to work smoothly, most specialized transformation languages assume that the source and target models are build with the same AS formalism (e.g., EMF Ecore). Model transformation languages typically provide support for efficiently navigating source models, and for creating instances of AS of

transformation in the context of *refinement*, i.e. the target model is less abstract and more detailed than the source model. Model transformations can also be used for other purposes, including the creation of views, refactorings and reverse engineering.

In addition to the two classical cases described above, there are also hybrid approaches that blur the boundaries between these two clear-cut extremes. They are based on the support for language modularization and composition, in the sense that the template language and the target language can be composed. As a consequence, the tooling is aware of the syntactic structure and the static semantics of the template language *and* the target language. Both MPS and Spoofax support this approach to various extents.

In MPS, a program is projected and every editing operation directly modifies the AST, while using a typically textuallooking notation as the "user interface". Template code (the code that controls the transformation process) and target-language code (the code you want to generate) can be represented as nested ASTs, each using its own textual syntax. MPS uses a slightly different approach based on a concept called *annotations*. Projectional editors can store arbitrary information in an AST. Specifically, they can store information that does *not* correspond to the language underlying a particular ASTs. MPS code generation templates exploit this approach: template code is fundamentally an instance of the target language. This "example model" is then annotated with template annotations that define how the example model relates to the source model, and which example nodes must be replaced by (further transformed) nodes from the source model. This allows any language to be "templatized" without changing the language definition itself. The MPS example below will elaborate on this approach.

Spoofax, with its Stratego transformation language, uses a similar approach based on parser technology. As we have already seen, the underlying grammar formalism supports flexible composition of grammars. So the template language and the target language can be composed, retaining tool support for

|  |  |
| --- | --- |
| the target language (tree construction). Examples for this approach once again include Xtext’s Xtend, as well as QVT Oper- |  |
| ational6 and ATL7. MPS can also be used in this way. A slightly | 6 **en.wikipedia.org/wiki/QVT** |
| different approach just establishes relations between the source and target models instead of "imperatively" constructing a target tree as the source is traversed. While this is often less intuitive to write down, the approach has the advantage that it | 7 **www.eclipse.org/atl/** |
| supports transformation in both directions, and also supports model diff8. QVT relational9 is an example of this approach. | 8 It does so by "relating" two instances of the same language and marking both |

as **readoqnly**; the engine then points out the difference between the two.

both of these languages. Execution of the template directly constructs an AST of the target language, using the concrete syntax of the target language to specify its structure. The Spoofax example will provide details.

|  |  |
| --- | --- |
| Since Xtext is based on EMF, generators can be built using any tool that can generate code from EMF models, includ- |  |
| ing Acceleo10, Jet11 and of course Xtend. Xtend is a Java-like | **www.acceleo.org/ pages/home/en** |
| general-purpose language that removes some of Java’s syntactic noise (it has type inference, property access syntax, operator overloading) and adds syntactic sugar (extension methods, multiple-dispatch and closures). Xtend comes with an interpreter and a compiler, the latter generating Java source. Xtend is built with Xtext, so it comes with a powerful Xtext-based editor. One particularly interesting language feature in the context of code generators are Xtend’s *template expressions*. Inside these expressions, a complete template language is available (similar to the older Xpand language). Xtend also provides automatic | **www.eclipse.org/ modeling/m2t/?project=jet** |
| whitespace management12. The functional abstractions pro- | Indentation of template code is |
| vided by Xtend (higher-order functions in particular) make it very well suited for navigating and querying models. In the rest of this section we will use Xtend for writing code generators and model transformations.  *11.2.1 Generator*  We will now look at generating the C code that implements cooling programs. Fig. 11.1 shows a screenshot of a typical | traditionally a challenge, because it is not clear whether whitespace in templates is intended to go into the target file, or is just used for indenting the template itself. |
| generator. The generator is an Xtend class that implements  **IGenerator**, which requires the **doGenerate** method to be implemented. The method is called for each model file that has changed13 (represented by the **resource** argument), and it has | This is achieved by a Builder that comes with Xtext. Alternatively, Xtend generators can also be run from the command line, from another Java program or from ant and maven. Also, other strategies can be implemented in |

### 11.2 Xtext Example

to output the corresponding generated code via the **fsa** (file system access) object[[1]](#footnote-1).

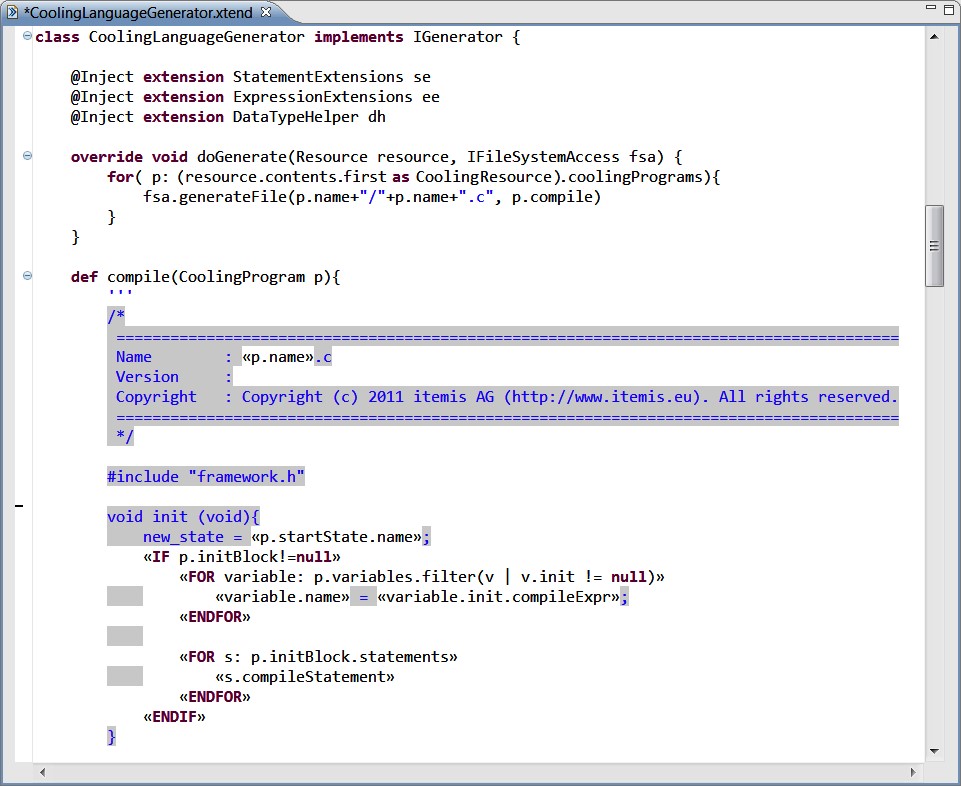
When generating code from models, there are two distinct cases. In the first case, the majority of the generated code is fixed; only some isolated parts of the code depend on the input model. In this case a template language is the right tool, because template control code can be "injected" into code that looks similar to what should be generated. In the other case there are fine-grained structures, such as expressions. Since

the Eclipse IDE itself, based on custom builder participants, buttons or menu entries.

these are basically trees, using template languages for these parts of programs seems unnatural and results in a lot of syntactic noise. A more functional approach is useful. Xtend can deal with both cases elegantly and we will illustrate both cases as part of this example.

We start with the high-level structure of the C code generated for a cooling program. The following piece of code illustrates Xtend’s power to navigate a model as well as the template syntax for text generation.

|  |
| --- |
| **def** compile(CoolingProgram program) {  ’’’  <<**FOR** appl: program.moduleImports.map(mi|mi.module).filter(typeof(  Appliance))>>  <<**FOR** c: appl.contents>>  #define <<c.name>> <<c.index>>  <<**ENDFOR**>>  <<**ENDFOR**>> |

Figure 11.1: The top-level structure of a generator written in Xtend is a class that implements the **IGenerator** interface, which requires the **doGenerate** method. Inside generator methods, template expressions (delimited with triple single quotes) are typically used. Inside those, guillemets (the small double angle brackets) are used to switch between to-be-generated code (gray background) and template control code. Note also the gray whitespace in the **init** function. Gray whitespace is whitespace that will end up in the generated code. White whitespace is used for indentation of template code; Xtend figures out which is which automatically.

|  |
| --- |
| // more ...  ’’’  } |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | the name of the respective content element and then its **index**. From within templates, the properties and references of model elements (such as the **name** or the **module** or the **contents**) can simply be accessed using the dot operator. **map** and **filter** | target file. | | are collection methods defined by the Xtend standard library16. | 16 **map** creates a new collection from | | We also have to generate an **enum** for the states in the cooling program. | an existing collection where, for each element in the existing collection, the expression after the **|** creates |  |  | | --- | | typedef enum states { null\_state,  <<**FOR** s : program.concreteStates **SEPARATOR** ",">> <<s.name>>  <<**ENDFOR**>> }; |   the corresponding value for the new collection. **filter** once again creates a new collection from an existing one where only those elements are included that are an instance of the type passed as an argument to **filter**.   |  |  | | --- | --- | | Here we embed a **FOR** loop inside the **enum** text. Note how we use the **SEPARATOR** keyword to put a comma *between* two subsequent states. In the **FOR** loop we access the **concreteStates** property of the cooling program. However, if you look at the grammar or the meta model, you will see that no **concreteStates** property is defined there. Instead, we call an extension |  | | method17; since it has no arguments, it looks like property ac- | 17 An extension method is an *additional* | |

The **FOR** loop iterates over the **moduleImports** collection of the **program**, follows the **module** reference of each of these, and then selects all **Appliance**s from the resulting collection. The nested loop then iterates over the contents of each **appl**iance and generates a **#define**15. After the **#define** we generate 15 Notice how the the first two and the last two lines are enclosed in guillemets. Since we are in template expression mode (inside the triple single quotes) the guillemets escape to template control code. The **#define** is *not* in guillemets, so it is generated into the

cess. The method is defined further down in the **CoolingLanguageGenerator** class and is essentially a shortcut for a complex expression:

|  |
| --- |
| **def** concreteStates(CoolingProgram p) {  p.states.filter(s | !(s **instanceof** BackgroundState) && !(s **instanceof** ShadowState))  } |

The following code is part of the generator that generates the code for a state transition. It first generates code to execute the exit actions of the current state, then performs the state change

(**current\_state = new\_state;**) and finally executes the entry actions of the new state (not shown):

|  |
| --- |
| if (new\_state != current\_state) {  <<**IF** program.concreteStatesWitExitActions.size > 0>> // execute exit action for state if necessary switch (current\_state) {  <<**FOR** s: p.concreteStatesWitExitActions>> case <<s.name>>:  <<**FOR** st: s.exitStatements>> |

method for an *existing* class, defined without invasively changing the definition of the class.

|  |
| --- |
| <<st.compileStatement>>  <<**ENDFOR**>> break; <<**ENDFOR**>> default:  break;  }  <<**ENDIF**>>  // The state change current\_state = new\_state;  // similar as above, but for entry actions } |

The code first uses an **IF** statement to check whether the program has any states with exit actions (by calling the **concreteStatesWitExitActions** extension method). The subsequent **switch** statement is only generated if we have such states. The **switch** switches over the **current\_state**, and then adds a **case** for each state with exit actions18. Inside the case we it-

18 The **s.name** expression in the **case** is

|  |  |
| --- | --- |
| The implementation of the overloaded methods simply returns the text string that represents the C implementation for the re- |  |
| spective language construct20. Notice how the implementation | 20 The two examples shown are simple because the language construct in the |

erate over all the **exitStatements** and call **compileStatement** for each of them. **compileStatement** is marked with **dispatch**, which makes it a multimethod: it is polymorphically overloaded based on its argument19. For each statement in the cooling language, represented by a subclass of **Statement**, there is an implementation of this method. The next piece of code shows some example implementations.

|  |  |
| --- | --- |
| |  | | --- | | class StatementExtensions {  **def dispatch** compileStatement(Statement s){  // raise error if the overload for the abstract class is called  }  **def dispatch** compileStatement(AssignmentStatement s){  s.left.compileExpr +" = " + s.right.compileExpr +";" }  **def dispatch** compileStatement(IfStatement s){  ’’’  if( <<s.expr.compileExpr>> ){  <<**FOR** st : s.statements>>  <<st.compileStatement>>  <<**ENDFOR**>>  }<<**IF** s.elseStatements.size > 0>> else {  <<**FOR** st : s.elseStatements>> <<st.compileStatement>>  <<**ENDFOR**>>  }<<**ENDIF**>>  ’’’  }  // more ... } |   19 Note that Java can only perform a polymorphic dispatch based on the **this** pointer. Xtend can dispatch polymorphically over the arguments of methods marked as **dispatch**. |

actually a reference to the enum literal generated earlier for the particular state. From the perspective of Xtend, we simply generate text: it is not obvious from the template that the name corresponds to an enum literal. Potential structural or type errors are only revealed upon compilation of the generated code.

for the **IfStatement** uses a template string, whereas the one for **AssignmentStatement** uses normal string concatenation.

DSL closely resembles the C code in the first place.

The **compileStatement** methods are implemented in the class **StatementExtensions**. However, from within the **CoolingLanguageGenerator** they are called using method syntax (**st. compileStatement**). This works because they are injected as extensions using the following statement:

@Inject **extension** StatementExtensions

Expressions are handled in the same way as statements. The injected class **ExpressionExtensions** defines a set of overloaded **dispatch** methods for **Expression** and all its subtypes. Since expressions are trees, a **compileExpr** method typically calls **compileExpr** recursively on the children of the expression, if it has any. This is the typical idiom to implement generators for expression languages21.

21 Earlier we distinguished between

|  |
| --- |
| **def dispatch** String compileExpr (Equals e){  e.left.compileExpr + " == " + e.right.compileExpr }  **def dispatch** String compileExpr (Greater e){  e.left.compileExpr + " > " + e.right.compileExpr }  **def dispatch** String compileExpr (Plus e){  e.left.compileExpr + " + " + e.right.compileExpr }  **def dispatch** String compileExpr (NotExpression e){ "!(" + e.expr.compileExpr + ")"  }  **def dispatch** String compileExpr (TrueExpr e){ "TRUE" }  **def dispatch** String compileExpr (ParenExpr pe){ "(" + pe.expr.compileExpr + ")"  }  **def dispatch** compileExpr (NumberLiteral nl){ nl.value  } |

generating a lot of code with only specific parts being model-dependent, and fine-grained tree structures in expressions: this is an example of the latter.

|  |  |
| --- | --- |
| *11.2.2 Model-to-Model Transformation*  For model-to-model transformations, the same argument can be made as for code generation: since Xtext is based on EMF, any EMF-based model-to-model transformation engine can be used with Xtext models. Examples include ATL, QVT-O, QVT- |  |
| R and Xtend22. | 22 Of course you could use any JVM- |

Model-to-model transformations are similar to code generators in the sense that they traverse over the model. But instead of producing a text string as the result, they produce another AST. So the general structure of a transformation is similar. In fact, the two can be mixed. Let us go back to the first code example of the generator:

based compatible programming language, including Java itself. However, Java is really not very well suited, because of its clumsy support for model navigation and object instantiation. Scala and Groovy are much more interesting in this respect.

|  |
| --- |
| **def** compile(CoolingProgram program) { **val** transformedProgram = program.transform ’’’  <<**FOR** appl : transformedProgram.modules.map(m|m.module).filter(typeof(  Appliance))>>  <<**FOR** c : appl.contents>>  #define <<c.name>> <<c.index>>  <<**ENDFOR**>>  <<**ENDFOR**>>  // more ...  ’’’  } |

We have added a call to a function **transform** at the beginning of the code generation process. This function creates a new **CoolingProgram** from the original one, and we store it in the **transformedProgram** variable. The code generator then uses the **transformedProgram** as the source from which it generates code. In effect, we have added a "preprocessor" model-tomodel transformation to the generator23.

23 As discussed in the design section

|  |
| --- |
| **class** Transformation {  @Inject **extension** CoolingBuilder  CoolingLanguageFactory factory = CoolingLanguageFactory::eINSTANCE  **def** CoolingProgram transform(CoolingProgram p ) {  p.states += emergencyState  p.events += emergencyEvent  **for** ( s: p.states.filter(typeof(CustomState)).filter(s|s !=  emergencyState) ) {  s.events += s.eventHandler [ symbolRef [ emergencyEvent()  ]  changeStateStatement(emergencyState())  ] } **return** p;  }  **def create** result: factory.createCustomState emergencyState() { result.name = "EMERGENCY\_STOP"  }  **def create** result: factory.createCustomEvent emergencyEvent() { result.name = "emergency\_stop\_button\_pressed" }  } |

|  |  |
| --- | --- |
| The two **create** methods create new objects, as the **create** prefix suggests24. However, simply creating objects could be | 24 The **factory** used in these methods is the way to create model elements in EMF. It is generated as part of the EMF |
| done with a regular method as well: | code generator |

The **transform** function (see below) enriches the existing model. It creates a new state (**EMERGENCY\_STOP**), creates a new event (**emergency\_button\_pressed**) and then adds a new transition to each existing state that checks whether the new event occurred, and if so, transitions to the new **EMERGENCY\_STOP** state. Essentially, this adds emergency stop behavior to any existing state machine. Let’s look at the implementation:

(Section 4.3), this is one of the most common uses of model-to-model transformations.

**def** emergencyState() { **val** result = factory.createCustomState result.name = "EMERGENCY\_STOP" result

}

What is different in **create** methods is that they can be called several times, and they still only ever create one object (for each combination of actual argument values). The result of the first invocation is cached, and all subsequent invocations return the object *created during the first invocation*. Such a behavior is very useful in transformations, because it removes the need to keep track of already created objects. For example, in the **transform** method, we have to establish references to the state created by **emergencyState** and the event created by **emergencyEvent**. To do that, we simply call the same **create** extension again. Since it returns the *same* object as during the first call in the first two lines of **transform**, this actually establishes references to those already created objects25.

25 This is a major difference between

We can now look at the implementation of **transform** itself.

It starts by adding the **emergencyState** and the **emergencyEvent** to the program26. We then iterate over all **CustomState**s except the emergency state we’ve just created. Notice how we just call the **emergencyState** function again: it returns the same object. We then use a builder to add the following code to each of the existing states.

**on** emergency\_button\_pressed { **state** EMERGENCY\_STOP

}

This code could be constructed by procedurally calling the respective factory methods:

|  |
| --- |
| **val** eh = factory.createEventHandler **val** sr = factory.createSymbolRef sr.symbol = emergencyEvent  **val** css = factory.createChangeStateStatement css.targetState = emergencyState eh.statements += css s.events += eh |

text generation and model transformation. In text, two textual occurrences of a symbol are the same thing (in some sense, text strings are value objects). In model transformation the identity of elements does matter. It is not the same if we create *one* new state and then reference it, or if we create five new states. So a good transformation language helps keep track of identities. The **create** methods are a very nice way of doing this.

|  |
| --- |
| The notation used in the actual implementation is more concise and resembles the tree structure of the code much more closely. It uses the well-known *builder*. Builders are implemented in Xtend with a combination of closures and implicit arguments and a number of functions implemented in the **CoolingBuilder** class[[2]](#footnote-2). Here is the code:  **class** CoolingBuilder {  CoolingLanguageFactory factory = CoolingLanguageFactory::eINSTANCE |

26 These are the first calls to the respective functions, so the objects are actually created at this point.

|  |
| --- |
| **def** eventHandler( CustomState it, (EventHandler)=>**void** handler ) { **val** res = factory.createEventHandler res  }  **def** symbolRef( EventHandler it, (SymbolRef)=>**void** symref ) { **val** res = factory.createSymbolRef it.events += res  }  **def** symbol( SymbolRef it, CustomEvent event ) { it.symbol = event  }  **def** changeStateStatement( EventHandler it, CustomState target ) { **val** res = factory.createChangeStateStatement it.statements += res res.targetState = target  }  } |

This class is imported into the generator with the **@Inject extension** construct, so the methods can be used "just so".

### 11.3 MPS Example

MPS comes with a **textgen** language for text generation. It is typically just used at the end of the transformation chain where code expressed in GPLs (like Java or C) is generated to text so it can be passed to existing compilers. Fig. 11.2 shows the **textgen** component for mbeddr C’s **IfStatement**. MPS’ text generation language basically appends text to a buffer. We won’t discuss this aspect of MPS any further, since MPS **textgen** is basically a wrapper language around a **StringBuffer**. However, this is perfectly adequate for the task at hand, since it is only used in the last stage of generation where the AST is essentially structurally identical to the generated text28.

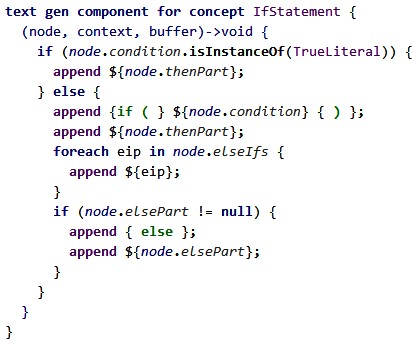
|  |
| --- |
| DSLs and language extensions typically use model-to-model transformations to "generate" code expressed in a low-level programming language[[3]](#footnote-3). Writing transformations in MPS in- |

28 If you want to generate text that is structurally different, then the textgen language is a bit of a pain to use; in this case, the MPS philosophy recommends that you build a suitable intermediate language (such as for XML, or even for a particular schema).

volves two ingredients. Templates define the actual transformation. Mapping configurations define which template to run when and where. Templates are valid sentences of the target language. *Macros* are used to express dependencies on and queries over the input model. For example, when the guard condition (a C expression) should be generated into an **if** statement in the target model, you first write an **if** statement with a dummy condition in the template. The following would work: **if (true) {}**. Then the nodes that should be replaced by the transformation with nodes from the input model are annotated

mation (we map one AST onto another) the transformations look very much like code generators, since the concrete syntax of the target language is used in the "template".

Figure 11.2: The AST-to-text generator for an **if** statement. If first checks if the condition happens to be a **true** literal, in which case the **if** statement is optimized away and only the **thenPart** is output. Otherwise we generate an **if** statement, the condition in parentheses, and then the **thenPart**. We then iterate over all the **elseIfs**; an **elseIf** has its own textgen, and we delegate to that one. We finally output the code for the **else** part.



with macros. In our example, this would look like this: **if (COPY\_SRC[true]){}**. Inside the **COPY\_SRC** macro you put an expression that describes which elements from the input model should replace the dummy node **true**: we use **node.guard** to replace it with the guard condition of the input node (expected to be of type **Transition** here). When the transformation is executed, the **true** node will be replaced by what the macro expression returns – in this case, the guard of the input transition. We will explain this process in detail below.

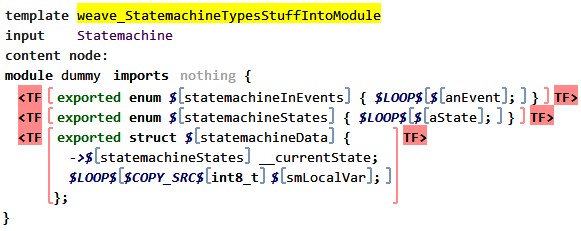
*Template-based Translation of the State Machine* State machines live inside modules. Just like **struct**s, they can be instantiated. The following code shows an example. Notice the two global variables **c1** and **c2**, which are instances of the same state machine **Counter**.

|  |
| --- |
| **module** Statemachine **imports** nothing {  **statemachine** Counter { **in events** start()  step(**int**[0..10] size)  **out events** started() resetted()  incremented(**int**[0..10] newVal)  **local variables int**[0..10] currentVal = 0 **int**[0..10] LIMIT = 10  **states** ( **initial** = start )  **state** start {  **on** start [ ] -> countState { send started(); }  } **state** countState {  **on** step [currentVal + size > LIMIT] -> start { send resetted(); } **on** step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size; send incremented(currentVal); |
| }  **on** start [ ] -> start { send resetted(); }  }  }  Counter c1; Counter c2;  **void** aFunction() { trigger(c1, start);  }  } |

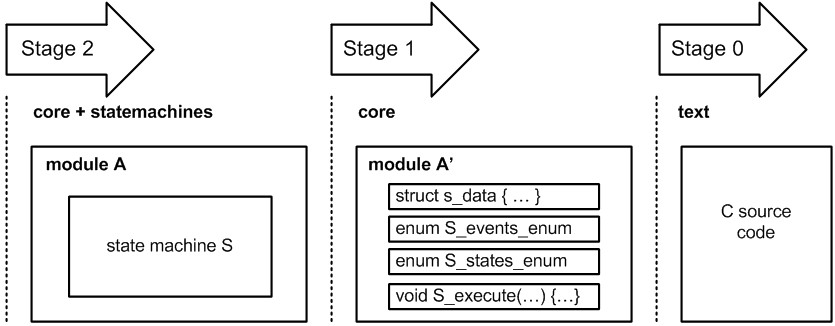
State machines are translated to the following lower-level C entities (this high level structure is clearly discernible from the two main templates shown in Fig. 11.3 and Fig. 11.7):

* An **enum** for the states (with a literal for each state).
* An **enum** for the events (with a literal for each event).
* A **struct** declaration that contains an attribute for the current state, as well as attributes for the local variables declared in the state machine.
* And finally, a function that implements the behavior of the state machine using a **switch** statement. The function takes two arguments: one named **instance**, typed with the **struct** mentioned in the previous item, and one named **event** that is typed to the event **enum** mentioned above. The function checks whether the instance’s current state can handle the event passed in, evaluates the guard, and if the guard is **true**, executes exit and entry actions and updates the current state.

Figure 11.3: The MPS generator that inserts two **enum** definitions and a **struct** into the module which contains the **StateMachine**.

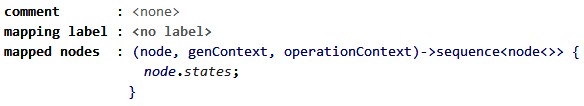


The MPS transformation engine works in phases. Each phase transforms models expressed in some languages to other models expressed in the same or other languages. Model elements for which no transformation rules are specified are copied from one phase to the next. Reduction rules are used to intercept program elements and transform them as generation progresses through the phases. Fig. 11.4 shows how this affects state machines. A reduction rule is defined that maps state machines to the various elements we mentioned above. Notice how the surrounding module remains unchanged, because no reduction rule is defined for it.

Figure 11.4: State machines are transformed (via a model-to-model transformation, if you will) into two **enum**s, a **struct** and a function. These are then transformed to text via the regular **com.mbeddr.core** textgen.

|  |  |
| --- | --- |
| Let us look in more detail at the template in Fig. 11.3. It reduces a **Statemachine**, the input node, to two **enum**s and a **struct**. We use template fragments (marked with **<TF ... TF>**) to highlight those parts of the template that should actually be used to replace the input node as the transformation executes. The surrounding **module dummy** is scaffolding: it is only needed because **enum**s and **struct**s *must* live in **Implementa-** |  |
| **tionModule**s in any valid instance of the mbeddr C language30. | 30 Since the templates are projected ex- |
| We have to create an **enum** literal for each state and each event. To achieve this, we iterate over all states (and events, respectively). This is expressed with the **LOOP** macros in the template in Fig. 11.3. The expression that determines what we | ample instances of the target language, the template *has to be a valid instance* of any MPS-implemented language. |
| iterate over is entered in the Inspector, MPS’ equivalent to a properties window; Fig. 11.5 shows the code for iterating over the states31. For the literals of the events **enum** we use a similar | 31 Note that the only really interesting part of Fig. 11.5 is the body of the  anonymous function (**node.states;**), which is why from now on we will only |
| expression (**node.events;**). | show this part. |

Figure 11.5: The inspector is used to provide the implementation details for the macros used in the templates. This one belongs to a **LOOP** macro, so we provide an expression that returns a collection, over which the **LOOP** macro iterates.



The **LOOP** macro iterates over collections and then creates an instance of the concept it is attached to for each iteration. In the case of the two **enum**s, the **LOOP** macro is attached to an **EnumLiteral**, so we create an **EnumLiteral** for each event and state we iterate over. However, these various **EnumLiteral**s must all have different names. In fact, the name of each literal should be the name of the state/event for which it is created. We can use a property macro, denoted by the **$** sign, to achieve this. A property macro is used to replace values of properties32. In this case we use it to replace the **name** property of the

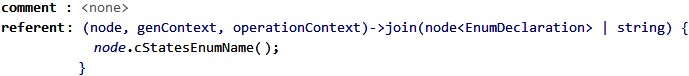
32 The node macros used above

|  |  |
| --- | --- |
| generated **EnumLiteral** with the name of the state/event over which we loop. Here is the implementation expression of the property macro: | (**COPY SRC**) replace whole nodes, as opposed to primitive *properties* of nodes. |

**\_**

node.cEnumLiteralName();

|  |  |  |  |
| --- | --- | --- | --- |
| In the code above, **cEnumLiteralName** is a behavior method33. | | 33 Behavior methods are defined as part | |
| It concatenates the **name** of the parent **Statemachine** with the string **\_\_state\_** and the name of the current state (in order to get a unique name for each state): | | of the behavior aspect of the concepts, such as **State**. | |
| **concept behavior** State {  **public string** cEnumLiteralName() { **return this**.parent : Statemachine.name + "\_\_state\_" + **this**.name;  }  } |
| The first of the **struct** attributes is also interesting. It is used to store the current state. It has to be typed to the state **enum** that is generated from this particular state machine. The type of the attribute is an **EnumType**; **EnumType**s extend **Type** and reference the **EnumDeclaration** whose type they represent. How can we | |  |
| establish the reference to the correct **EnumDeclaration**? We use a reference macro (**->$**) to retarget the reference. Fig. 11.6 shows the macro’s expression. | | Figure 11.6: A reference macro has to return either the target node, or the name of the target node. The name is then resolved using the scoping rules for the particular reference concept. |



Note how a reference macro expects either the target node

34

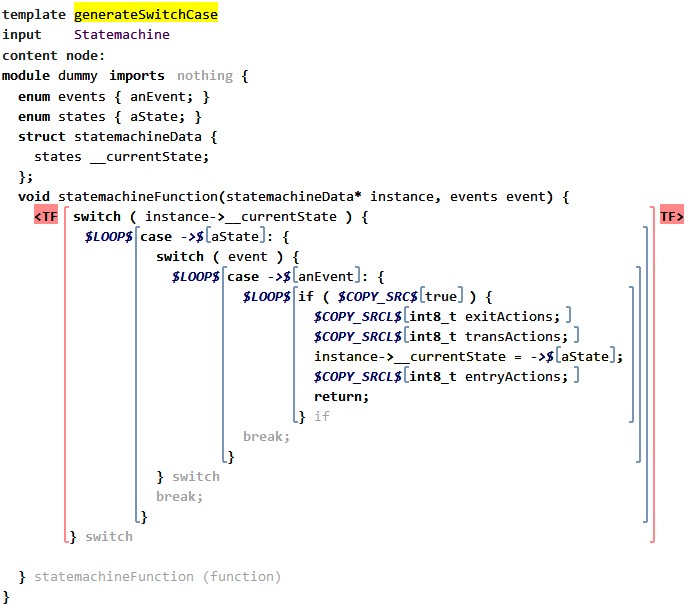
This is not a global name lookup!

|  |  |
| --- | --- |
| (here: an **EnumDeclaration**) as the return value, or a **string**. That **string** would be the name of the target element. Our implementation returns the name of the states **enum** generated in the same template. MPS then uses the target language’s scoping rules to find and link to the correct target element34. | Since MPS knows the reference is an  **EnumLiteralRef** expression, the scope of that concept is used. As long as the name is unique within the scope, this is completely deterministic. Alternatively, the actual node can be identified and returned from the reference macro |
| Let us now address the second main template, Fig. 11.7, which generates the execute function. The **switch** expression is interesting. It switches over the current state of the cur- | using mapping labels. However, using names is much more convenient and works also for cross-model references, where mapping labels don’t work. |

rent state machine instance. That instance is represented by the **instance** parameter passed into the function. It has a **\_\_currentState** field. Notice how the function that contains the **switch** statement *in the template* has to have the **instance** argument, and how its type, the **struct**, has to have the **\_\_currentState** attribute *in the template*. If the respective elements were not there in the template, we couldn’t write the template code! Since there is a convention that in the *resulting* function the argument will also be called **instance**, and the attribute will also be called **\_\_currentState**, we don’t have to use a reference macro to retarget the two.

Inside the **switch** we **LOOP** over all the states of the state machine and generate a **case**, using the state’s corresponding **enum** literal. Inside the **case**, we embed another **switch** that switches over the **event** argument. Inside this inner **switch** we iterate over all transitions that are triggered by the event we currently iterate over:

Figure 11.7: The transformation template for generating the **switch**-based implementation of a **StateMachine**. Looking at the template fragment markers (**<TF ... TF>**) reveals that we only generate the **switch** statement, *not* the function that contains it. The reason is that we need to be able to embed the state machine **switch** into other function-like concepts as well (e.g., operations in components defined in the mbeddr components extension), so we have separated the generation of the function from the generation of the actual state machine behavior. See the text for more details.



context state.transitions.where({~it => it.trigger.event == node; });

We then generate an **if** statement that checks at runtime whether the guard condition for this transition is **true**. We copy in the guard condition using the **COPY\_SRC** macro attached to the **true** dummy node. The **COPY\_SRC** macro copies the original node, but it also applies additional reduction rules for this node (and all its descendants), if there are any. For example, in a guard condition it is possible to reference event arguments. The reference to **size** in the **step** transition is an example:

|  |
| --- |
| **statemachine** Counter { **in events** step(**int**[0..10] size) ... **states** ( **initial** = start ) ...  **state** countState {  **on** step [currentVal + size > LIMIT] -> start { send resetted(); }  }  } |

|  |
| --- |
| (node, genContext, operationContext)->sequence<node<>> { **if** (node.parent : State.exitAction != **null**) { **return** node.parent : State.exitAction.statements;  }  **new** sequence<node<>>(empty); } |

Event arguments are mapped to the resulting C function via a **void\*** array. A reference to an event argument (**EventArgRef**) hence has to be reduced to accessing the **n**-th element in the array (where **n** is the index of the event argument in the list of arguments). Fig. 11.8 shows the reduction rule. It accesses the array, casts the element to a pointer to the type of the argument,

and then dereferences everything35. 35 The reduction rule creates code

that looks like this (for an **int** event attribute): **\*((int\*)arguments[0])**.

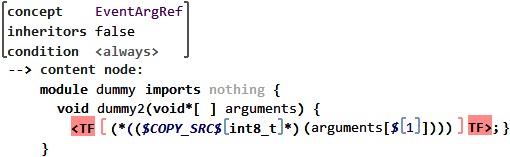


Figure 11.8: The reduction rule for references to event arguments (to be used inside guard conditions of transitions).

Inside the **if** statement, we have to generate the code that has to be executed if a transition fires. We first copy in all the exit actions of the current state. Once again, the **int8 exitActions;** is just an arbitrary dummy statement that will be replaced by the statements in the exit actions (**COPY\_SRCL** replaces a node with a *list* of nodes). The respective expression is this:

We then do the same for the transition actions of the current transition, set the **instance->\_\_currentState** to the target state of the transition using a reference macro (**node.targetState. cEnumLiteralName();**), and then we handle the entry actions of the target state. Finally we return, because at most one transition can fire as a consequence of calling the state machine execute function.

As the last example I want to show how the **TriggerSM-**

**Statement** is translated. It injects an event into a state machine:

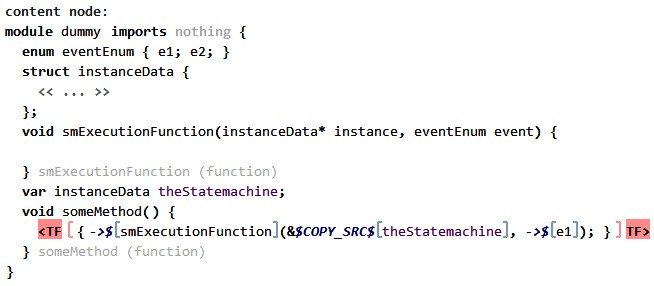
Counter c1;

**void** aFunction() {

trigger(c1, start);

}

It must be translated to a call to the generated state machine execute function that we have discussed above. For simplicity, we explain a version of the **TriggerSMStatement** that does not include event arguments. Fig. 11.9 shows the template.



We use a dummy function **someMethod** so that we can embed a function call, because we have to generate a function call to the execute function generated from the state machine. Only the function call is surrounded with the template fragment markers (and will be generated). The function we call *in the template code* is the **smExecuteFunction**. It has the same signature as the real, generated state machine execute function. We use a reference macro to retarget the **function** reference in the function call. It uses the following expression, which returns the name of the function generated for the state machine referenced in the **statemachine** expression of the trigger statement:

Figure 11.9: The reduction rule for a **trigger** statement. It is transformed to a call to the function that implements the behavior of the state machine that is referenced in the first argument of the **trigger** statement.

StatemachineType.machine.cFunctionName();

Note how the first argument to the **trigger** statement can be *any* expression (local variable reference, global variable reference, a function call). However, we know (and enforce via the type system) that the expression’s type must be a **StatemachineType**, which has a reference to the **Statemachine** whose instance the expression represents. So we can cast the expression’s type to **StatemachineType**, access the **machine** reference, and get the name of the execute function generated from that state machine36.

1. This is an example of where we use the type system in the code generator, and not just for checking a program for type correctness.

The second argument of the **trigger** statement is a reference to the event we want to trigger. We can use another reference macro to find the **enum** literal generated for this event. The macro code is straight forward:

node.event.cEnumLiteralName();

*Procedural Transformation of a Test Case* Instead of using the template-based approach shown above, transformations can also be written manually against the MPS API. To do this, a **mapping script** is called from the mapping configuration (instead of the mapping configuration containing rules). Such a mapping script can contain arbitrary BaseLanguage code that operates on the output model of the transformation37.

1. This is essentially similar to the

As part of the mbeddr project, we have built a Builder extension to BaseLanguage. In the example below, we will build the following code:

|  |
| --- |
| **module** SomeModule **imports** nothing { **exported test case** testCase1 { }  **exported** int32 main(int32 argc, string\*[] argv) { **return test** testCase1;  }  } |

The code below builds the code above. Notice how by default we work with concepts directly (as when we mention

**StatementList** or **ExecuteTestExpression**). However, we can also embed expression in the builder using the **#(..)** expression. Nodes created by the builder can be named (as in **tc:TestCase**) so they can be used as a reference target later (**testcase -> tc**).

node<ImplementationModule> immo = build ImplementationModule

name = #(aNamePassedInFromAUserDialog) contents += tc:TestCase

implementation code for intentions or

refactorings (Section 13.6); those also modify a model using the MPS Node

API.

name = "testCase1" type = VoidType contents += #(MainFunctionHelper.createMainFunction())

body = StatementList statements += ReturnStatement expression = ExecuteTestExpression tests += TestCaseRef testcase -> tc

Builders in MPS are a first-class extension to BaseLanguage, which means that the IDE can provide support. For example, if a concept has a mandatory child (e.g. the **body** in a **Function**), the IDE will report an error if no node is put into this child slot. Code completion can be provided as well38.

38 Users do not have to build the helper functions we have seen for Xtext/Xtend above. On the other hand, the MPS builder extension is specific to building MPS node trees, whereas the approach taken by Xtext/Xtend is generic, as long as users define the helper functions.

### 11.4 Spoofax Example

|  |  |
| --- | --- |
| In Spoofax, model-to-model transformations and code genera- |  |
| tion are both specified by rewrite rules39. This allows for the | 39 Rewrite rules were introduced in |
| seamless integration of model-to-model transformation steps into the code generation process; the clear distinction between | Section 9.3.1. |
| model-to-model transformation and code generation vanishes40. | 40 Similar to MPS, it is also possible |
| We look at the various approaches supported by Spoofax in this chapter.  *Code Generation by String Interpolation* Pure code generation from abstract syntax trees to text can be realized by rewriting to strings. The following simple rules rewrite types to their corresponding representation in Java. For entity types, we use their name as the Java representation: | to express model-to-model transformations using the concrete syntax of the target language, even though this requires a bit more setup and care. |

to-java: NumType() -> "int" to-java: BoolType() -> "boolean" to-java: StringType() -> "String" to-java: EntType(name) -> name

|  |  |
| --- | --- |
| Typically, more complex rules are recursive and use string in- |  |
| terpolation41 to construct strings from fixed and variable parts. | 41 We used string interpolation already |

For example, the following two rewrite rules generate Java code for entities and their properties:

|  |
| --- |
| to-java:  Entity(x, ps) ->  $[ class [x] {  [ps’]  } ] **with** ps’ := <map(to-java)> ps  to-java:  Property(x, t) ->  $[ private [t’] [x];  public [t’] get\_[x] { return [x]; |

before to compose error messages in Section 9.3.

|  |
| --- |
| }  public void set\_[x] ([t’] [x]) { this.[x] = [x];  } ] **with**  t’ := <to-java> t |

String interpolation takes place inside **$[...]** brackets and allows us to combine fixed text with variables that are bound to strings. Variables can be inserted using brackets **[...]** without a dollar sign42. Instead of variables, we can also directly use

42 You can also use any other kind of

the results from other rewrite rules that yield strings or lists of strings:

|  |
| --- |
| to-java:  Entity(x, ps) ->  $[ class [x] {  [<map(to-java)> ps]  }  ] |

Indentation is important, both for the readability of rewrite rules and of the generated code: the indentation leading up to the **$[...]** brackets is removed, but any other indentation beyond the bracket level is preserved in the generated output. In this way we can indent the generated code, as well as our rewrite rules. Applying **to-java** to the initial **shopping** entity will yield the following Java code:

|  |
| --- |
| **class** Item { **private** String name;  **public** String get\_name { **return** name;  }  **public void** set\_name (String name) { **this**.name = name;  } **private boolean** checked;  **public boolean** get\_checked { **return** checked;  }  **public void** set\_checked (**boolean** checked) { **this**.checked = checked;  } **private** Num order;  **public** Num get\_order { **return** order;  }  **public void** set\_order (Num order) { **this**.order = order;  }  } |

bracket: **{...}**, **<...>**, and **(...)** are allowed as well.

When we prefer camelcase in method names, we need to slightly change our code generation rules, replacing **get\_[x]** and **set\_[x]** by **get[<to-upper-first>x]** and **set[<to-upper-first>x]**. We also need to specify the following rewrite rule:

|  |
| --- |
| to-upper-first: s -> s’ **where**  [first|chars] := <explode-string> s ;  upper := <to-upper> first ; s’ := <implode-string> [upper|chars] |

**explode-string** turns a string into a list of characters, **to-upper** upper-cases the first character, and **implode-string** turns the characters back into a string43.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Editor Integration* To integrate the code generation into our editor, we first have to define the following rewrite rule:   |  | | --- | | generate-java:  (selected, position, ast, path, project-path) -> (filename, result) **with** filename := <guarantee-extension(|"java")> path; result := <to-java> ast | | While we are free to choose the name of this rule44, the patterns | | |  | | on the left- and right-hand side need to follow Spoofax’ convention for editor integration. On the left-hand side, it matches the current **selection** in the editor, its **position** in the abstract syntax tree, the abstract syntax tree itself (**ast**), the **path** of the | | |  | | source file in the editor45, and the path of the project this file | | |  | |

belongs to. As the right-hand side shows, the rule produces the name of the generated file and its content as a string. The file name is derived from the source file’s path, while the file content is generated from the abstract syntax tree.

Once we have defined this rule, we can register it as a **builder** in **editor/Lang-Builders.esv**. Here, we add the following rule:

**builder**: "Generate Java code (selection)" = **generate**-**java** (**openeditor**) ( **realtime**)

This defines a label for our transformation, which is added to the editor’s *Transform* menu. Additional options, such as **(openeditor)** and **(realtime)**, can be used to customize the behaviour of the transformation. The following table illustrates the available options.

per-file basis. We can also just generate code for the current selection: to do so, we can replace the last line by **result := <to-java> selected**.

|  |  |
| --- | --- |
| **Option** | **Description** |
| **(openeditor)** | Opens the generated file in an editor. |
| **(realtime)** | Re-generates the file as the source is edited. |
| **(meta)** | Excludes the transformation from the deployed plugin. |
| **(cursor)** | Transforms always the tree node at the cursor. |

*Code Generation by Model Transformation* Rewrite rules with string interpolation support a template-based approach to code generation. Thus, they share two typical problems of template languages. First, they are not *syntax safe*, that is, they do not guarantee the syntactical correctness of the generated code: we might accidently generate Java code which can not be parsed by a Java compiler. Such errors can only be detected by testing the code generator. Second, they inhibit subsequent transformation steps. For example, we might want to optimize the generated Java code, generate Java bytecode from it, and finally optimize the generate Java Bytecode. At each step, we would first need to parse the generated code from the previous step before we can apply the actual transformation.

Both problems can be avoided by generating abstract syntax trees instead of concrete syntax, i.e. by using model-to-model transformations instead of code (text) generation. This can be achieved by constructing terms on the right-hand side of rewrite rules:

|  |
| --- |
| to-java: NumType() -> IntBaseType() to-java: BoolType() -> BooleanBaseType()  to-java: StringType() -> ClassType("java.lang.String") to-java: EntType(t) -> ClassType(t)  to-java:  Entity(x, ps) -> Class([], x, ps’) ps’ := <mapconcat(to-java)> ps  to-java:  Property(x, t) -> [field, getter, setter] **with**  t’ := <to-java> t ;  field := Field([Private()], t’, x) ;  getter := Method([Public()], t’, $[get\_[x]], [],  [Return(VarRef(x))]) ;  setter := Method([Public()], Void(), $[set\_[x]],  [Param(t’, x)], [assign]) ;  assign := Assign(FieldRef(This(), x), VarRef(x)) |

When we generate ASTs instead of concrete syntax, we can easily compose transformation steps into a transformation chain by using the output of transformation *n* as the input for transformation *n*+1. But this chain will still result in abstract syntax trees. To turn them back into text, it has to be pretty-printed (or serialized). Spoofax generates a language-specific rewrite

rule **pp-<LangName>-string** which rewrites an abstract syntax tree into a string according to a pretty-printer definition46.

|  |  |  |  |
| --- | --- | --- | --- |
| *Concrete Object Syntax* Both template-based and term-based approaches to code generation have distinctive benefits. While template-based generation with string interpolation allows for concrete syntax in code generation rules, AST generation guarantees syntactical correctness of the generated code and enables transformation chains. To combine the benefits of both approaches, Spoofax can parse user-defined concrete syntax quotations at compile-time, checking their syntax and replacing them with equivalent abstract syntax fragments47. | |  | |
| For example, a Java return statement can be expressed as **|[ return |[x]|; ]|**, rather than the abstract syntax form **Return(VarRef(x))**. Here, **|[...]|** surrounds Java syntax. It quotes Java fragments inside Stratego code. Furthermore, **|[x]|** refers to a Stratego variable **x**, matching the expression in the return statement. In this case, **|[...]|** is an antiquote, switching back to Stratego syntax in a Java fragment.  To enable this functionality, we have to customize Stratego, Spoofax’ transformation language. This requires four steps. First, we need to combine Stratego’s syntax definition with the syntax definitions of the source and target languages. There, it is important to keep the sorts of the languages disjunct. This can be achieved by renaming sorts in an imported module, which we do in the following example for the **Java** and the **Stratego** module: | |  | |
| **module** Stratego-Mobl-Java **imports** Mobl  **imports** Java [ ID => JavaId ]  **imports** Stratego [ Id => StrategoId  Var => StrategoVar  Term => StrategoTerm ] |
| Second, we need to define quotations, which will enclose concrete syntax fragments of the target language in Stratego rewrite | |  |
| rules48. We add a syntax rule for every sort of concrete syntax | |  |
| fragments that we want to use in our rewrite rules: | |  |
| **exports context**-**free syntax**  "|[" Module "]|" -> StrategoTerm {"ToTerm"}  "|[" Import "]|" -> StrategoTerm {"ToTerm"}  "|[" Entity "]|" -> StrategoTerm {"ToTerm"}  "|[" EntityBodyDecl "]|" -> StrategoTerm {"ToTerm"}  "|[" JClass "]|" -> StrategoTerm {"ToTerm"}  "|[" JField "]|" -> StrategoTerm {"ToTerm"} "|[" JMethod "]|" -> StrategoTerm {"ToTerm"}  "|[" JFeature\* "]|" -> StrategoTerm {"ToTerm"} |

With these rules, we allow quoted Mobl and Java fragments wherever an ordinary Stratego term is allowed. The first four rules concern Mobl, our example source language49. As quotes,

49 We like to use concrete syntax for

we use **|[...]|**. The second set of rules work similarly for Java, our example target language. All syntax rules extend **Term** from the Stratego grammar, which we renamed to **StrategoTerm** during import. We use **ToTerm** as a constructor label. This allows Stratego to recognize places where we use concrete object syntax inside Stratego code. It will then lift the subtrees at these places into Stratego code. For example, the abstract syntax of **|[ return |[x]|; ]|** would be **ToTerm(Return(...))**. Stratego lifts this to **NoAnnoList(Op( "Return", [...]))**, which is the abstract syntax tree for the term **Return(x)**.

Third, we need to define antiquotations, which will enclose Stratego code in target language concrete syntax fragments. Here, we add a syntax rule for every sort where we want to inject Stratego code into concrete syntax fragments:

|  |  |
| --- | --- |
| **exports context**-**free syntax**  "|[" StrategoTerm "]|" -> JavaId | {"FromTerm"} |
| This rule allows antiquoted Stratego terms to be used wherever | | | |  |
| a Java identifier can be used50. We use **FromTerm** as a construc- | | | | 50 We renamed **Id** from the Java gram- |
| tor in the abstract syntax tree. Like **ToTerm**, Stratego uses this | | | | mar to **JavaID** during import. |

to recognize places where we switch between concrete object syntax and Stratego code. For example, the abstract syntax of **|[ return |[x]|; ]|** would be

ToTerm(Return(FromTerm(Var("x"))))

Stratego lifts this to the following, which is the abstract syntax tree for the term **Return(x)**:

NoAnnoList(Op("Return", [Var("x")]))

Finally, we need to create a **<filename>.meta** file for every transformation file **<filename>.str** with concrete syntax fragments. In this file, we tell Spoofax to use our customized Stratego syntax definition:

Meta([ Syntax("Stratego-Mobl-Java") ])

Now, we can use concrete syntax fragments in our rewrite rules[[4]](#footnote-4):

|  |
| --- |
| to-java:  |[ entity |[x]| { |[ps]| } ]| ->  |[ class |[x]| { |[<mapconcat(to-java)> ps]| } ]| to-java: |
| |[ |[x]| : |[t]| ]| ->  |[ private |[t’]| |[x]|; public |[t’]| |[x]| { return |[x]|; }  public void |[x]| (|[t’]| |[x]|) { this.|[x]| = |[x]|; } ]|  **with**  t’ := <to-java> t |

Using concrete object syntax in Stratego code combines the benefits of string interpolation and code generation by model transformation. With string interpolation, we can use the syntax of the target language in code generation rules, which makes them easy to write. However, it is also easy to make syntactic errors, which are only detected when the generated code is compiled. With code generation by model transformation, we can check if the generated abstract syntax tree corresponds to the grammar of the target language. Actually we can check each transformation rule and detect errors early. With concrete object syntax, we can now use the syntax of the target language in code generation. This syntax is checked by the parser which is derived from the combined grammars of the target language and Stratego.

This comes at the price of adding quotations and antiquotation rules manually. These rules might be generated from a declarative, more concise embedding definition in the future. However, we cannot expect full automation here, since choices for the syntactic sorts involved in the embedding and of quotation and antiquotation symbols require an understanding of Stratego, the target language, and the transformation we want to write. These choices have to be made carefully in order to avoid ambiguities[[5]](#footnote-5).

In general, there is room for more improvements of the embedding of the target language into Stratego. When the target language comes with a Spoofax editor, we want to get editor services like code completion, hover help, and content folding in the embedded editor as well. Until now, only syntax highlighting has been supported, using the Stratego coloring rules. Keywords of the target language will be highlighted like Stratego keywords and embedded code fragments will be given a gray background color.

1. Like any other Xtext language aspect, the generator has to be registered with the runtime module, Xtext’s main configuration data structure. Once this is done, the generator is automatically called for each changed resource associated with the respective language. [↑](#footnote-ref-1)
2. Of course, if you add the line count and effort for implementing the builder, then using this alternative over the plain procedural one might not look so interesting. However, if you just create these builder functions once, and then create many different transformations, this approach makes a lot of sense. [↑](#footnote-ref-2)
3. The distinction between code generators and model-to-model transformations is much less clear in this case. While it is a model-to-model transfor- [↑](#footnote-ref-3)
4. . [↑](#footnote-ref-4)
5. ’ [↑](#footnote-ref-5)