*16*

## Modularization, Reuse and Composition

*Language modularization, extension and composition is an important ingredient in the efficient use of DSLs, just as reuse in general is important to software development. We discuss the need for modularization, extension and composition in the context of DSL design in Section 4.6, where we introduce the four classes of modularization, extension and composition. In this chapter, we look at the implementation approaches taken by our example tools.*

### 16.1 Introduction

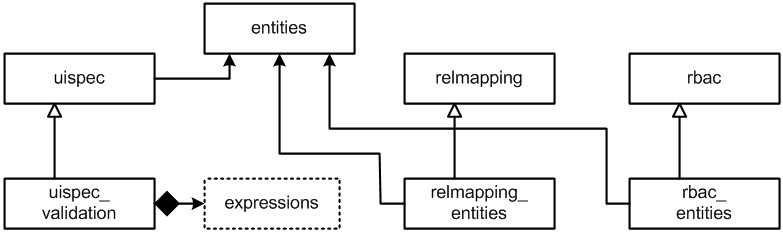
When modularizing and composing languages, the following challenges have to be addressed:

* The concrete and the abstract syntaxes of the languages have to be combined. Depending on the kind of composition, this requires the embedding of one syntax into another. This, in turn, requires modular grammars, or more generally, ways of specifying concrete syntax that avoids ambiguities.
* The static semantics, i.e. the constraints and the type system, have to be integrated. For example, in the case of language extension, new types have to be "made valid" for existing operators.
* The execution semantics have to be combined as well. In practice, this may mean mixing the code generated from the composed languages, or composing the generators or interpreters.
* Finally, the IDE services that provides code completion, syntax coloring, static checks and other relevant services have to be extended and composed as well.

In this chapter we show how each of those is addressed with the respective tools. We don’t discuss the general problems any further, since those have been discussed in Part II of the book on DSL design.

### 16.2 MPS Example

With MPS two of these challenges outlined above – composability of concrete syntax and modular IDEs – are solved to a large degree. Modular type systems are reasonably well supported. Semantic interactions are hard to solve in general, but can be handled reasonably in many relevant cases, as we show in this section as well. However, as we will see, in many cases languages have to be designed *explicitly for reuse* to make them reusable. After-the-fact reuse, without consideration during the design of the reusable language, is possible only in limited cases. However, this is true for reuse in software generally.

**ac**

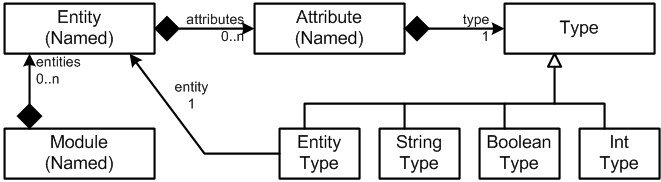
|  |  |
| --- | --- |
|  |  |
| We describe language modularization, extension and composition with MPS based on a set of examples1. At the center of | guage. |
| this section is a simple **entities** language – the usual entitywith-property-and-references "Hello World" example. We then build additional languages to illustrate extension and composition. Fig. 16.1 illustrates these additional languages. The **uispec** (user interface specification) language illustrates *referencing* with **entities**. **relmapping** (relational database mapping) is an example of *reuse* with separated generated code. **rbac** (role-based access control) illustrates reuse with intermixed generated code. **uispec\_validation** demonstrates *extension* (of the **uispec** language) and *embedding* with regard to the expressions language. |  |

#### 16.2.1 Implementing the Entities Language

Below is some example code expressed in the **entities** language. *Modules* are root nodes. They live as top-level elements

|  |
| --- |
| **module** company { **entity** Employee { id : **int** name : **string** role : **string** worksAt : Department freelancer : **boolean**  } **entity** Department {  id : **int** description : **string**  }  } |

*Structure and Syntax* Fig. 16.2 shows a class diagram of the structure of the **entities** language. Each box represents a language concept.

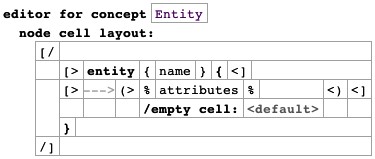
Fi

|  |  |  |  |
| --- | --- | --- | --- |
| The following code shows the definition of the **Entity** con- | | |  |
| cept3. **Entity** implements the **INamedConcept** interface to in- | | |  |
| herit a **name** property. It declares a list of children of type  **Attribute** in the **attributes** collection. Fig. 16.3 shows the definition of the editor for **Entity**. | | |  |
| **concept** Entity **extends** BaseConcept **implements** INamedConcept **can be root**: **true children**:  Attribute attributes 0..n |

*Type System* For the **entities** language, we specify two simple typing rules. The first one specifies that the type of the primitives (**int**, **string**) is a clone of themselves:

|  |
| --- |
| **rule** typeof\_Type { **applicable for concept** = Type **as** t **overrides false do** {  **typeof**(type) :==: type.copy; }  } |

.

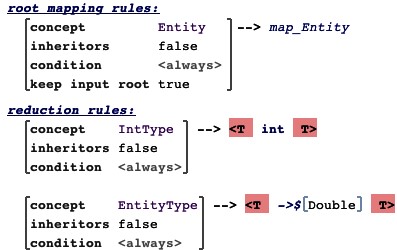


The only other typing rule is an equation that defines the type of the attribute as a whole to be the type of the attribute’s **type** property, defined as

**typeof**(attribute) :==: **typeof**(attribute.type);

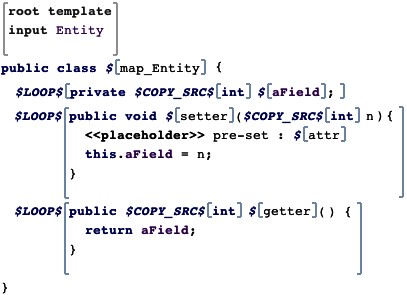
|  |  |
| --- | --- |
| *Generator* From **entities** models we generate Java Beans expressed in MPS’ BaseLanguage. For the **entities** language, |  |
| we need a *root mapping rule*4 and *reduction rules*5. The root map- |  |
| ping rule is used to generate a Java class from an **Entity**. The reduction rule transforms the various types (**int**, **string**, etc.) to their Java counterparts. Fig. 16.4 shows a part of the mapping configuration for the **entities** language. |  |

.



(**private int aField;**). Then we use macros to "transform" this prototype into an instance for each **Entity** attribute. We first attach a **LOOP** macro to the whole field. Based on its ex-

entity. We then use a **COPY\_SRC** macro to transform the type. **COPY\_SRC** copies the input node (the inspector specifies the current attribute’s type as the input here) and applies reduction rules. So instances of the types defined as part of the **entities** language are transformed into a Java type using the reduction rules defined in the mapping configuration above. Finally we use a property macro (the **$** sign) to change the **name** property of the field we generate from the dummy value **aField** to the name of the attribute we currently transform (once again via an expression in the inspector).



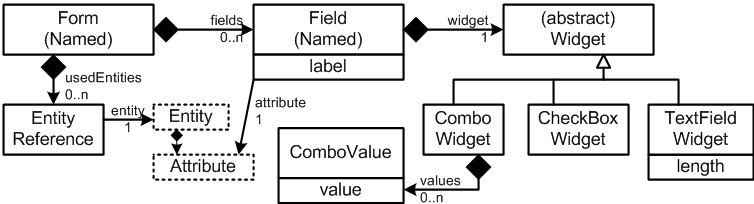
.

#### 16.2.2 Referencing

We define a language **uispec** for defining user interface forms based on the **entities**. Below is an example model. Note how the form is another, separate fragment. It is a *dependent* fragment, since it references elements from another fragment (expressed in the **entities** language). Both fragments are *homogeneous*, since they consist of sentences expressed in a single language.

|  |
| --- |
| **form** CompanyStructure **uses** Department **uses** Employee  **field** Name: **textfield**(30) -> Employee.name  **field** Role: **combobox**(Boss, TeamMember) -> Employee.**role field** Freelancer: **checkbox** -> Employee.freelancer **field** Office: **textfield**(20) -> Department.description |

*Structure and Syntax* The **uispec** language extends6 the **entities** language. This means that concepts from the **entities** language can be used in the definition of language concepts in the **uispec** language. Fig. 16.6 shows the abstract syntax as a UML diagram.



A **Form** owns a number of **EntityReferences**, which in turn reference the **Entity** concept. Also, **Field**s refer to the attribute that is intended to be edited via the field. Below is the definition of the **Field** concept. It owns a **Widget** and refers to an **Attribute**.

|  |
| --- |
| **concept** Field **extends** BaseConcept **implements** <none> **properties**: label : **string**  **children**:  Widget widget 1 **references**:  Attribute attribute 1 |
| *Type System* The language enforces limitations over which | | |  |
| widget can be used with which attribute type7. The necessary | | |  |
| typing rule is defined in the **uispec** language and references types from the **entities** language[[1]](#footnote-1). The following is the code | | |  |

for the type check.

|  |
| --- |
| **checking rule** checkTypes {  **applicable for concept** = Field **as** field **overrides false do** {  **if** (field.widget.isInstanceOf(CheckBoxWidget)  && !(field.attribute.type.isInstanceOf(BooleanType))) { **error** "only use checkbox with booleans" -> field.widget;  }  **if** (field.widget.isInstanceOf(ComboWidget)  && !(field.attribute.type.isInstanceOf(StringType))) { **error** "cannot use combobox with strings" -> field.widget;  }  }  } |

*Generation* The defining characteristic of language referencing is that the two languages only *reference* each other, and the instance fragments are dependent, but *homogeneous*. No syntactic integration is necessary in this case. In this example, the generated code exhibits the same separation. From the **Form** definition, we generate a Java class that uses Java Swing to build the UI form. It *uses* the beans generated from the entities: the classes are instantiated, and the getters and setters are called9. The generators are separate, but they are *dependent*

|  |
| --- |
| **concept behavior** Attribute { **public string** qname() { **this**.parent:Entity.name + "." + **this**.name;  } **public string** setterName() {  "**set**" + **this**.name.toFirstUpper();  } **public string** getterName() {  "get" + **this**.name.toFirstUpper(); }  } |

|  |  |
| --- | --- |
| The original **entities** fragment is still *sufficient* for the transformation that generates the Java Bean. The **uispec** fragment is not sufficient for generating the UI; it needs the **entities** fragment. This is not surprising, since *dependent* fragments can never be sufficient for a transformation: the transitive closure of all dependencies has to be made available. |  |
| *16.2.3 Extension*  We extend the MPS base language with block expressions and placeholders. These concepts make writing generators that generate base language code much simpler. Fig. 16.7 shows an example. |  |

|  |  |
| --- | --- |
| *Structure and Syntax* A block expression is a block that can be used where an **Expression** is expected10. The block can |  |
| contain any number of statements; **yield** can be used to "return" values from within the block11. An optional name prop- | 157–172, 2005 |
| erty of a block expression is used as the name of the generated method. The generator of the block expression in Fig. 16.7 transforms it into this structure: |  |

because they share information. Specifically, the **uispec** generator knows about the names of the generated entity classes, as well as the names of the setters and getters. This dependency is implemented by defining a couple of behavior methods on the **Attribute** concept that are called from both generators:



form is populated and to set the values into the bean if they have been changed in the form.

|  |
| --- |
| // the argument to setName is what was the block expression, // it is replaced by a method call to the generated method aEmployee.setName(retrieve\_name(aEmployee, widget0)); ...  // this is the method generated from the block expression  **public** String retrieve\_name(Employee aEmployee, JComponent widget0) { String newValue = ((JTextField) widget0).getText(); **return** newValue;  } |

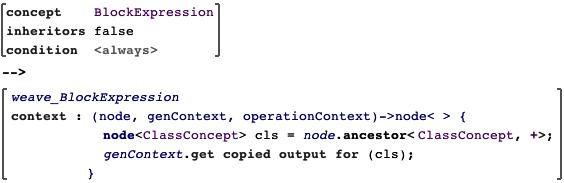
The **jetbrains.mps.baselanguage.exprblocks** language extends Ba- seLanguage. To make a block expression valid where BaseLanguage expects an **Expression**, **BlockExpression** ex-

tends **Expression**12.

|  |
| --- |
| **concept** BlockExpression **extends** Expression **implements** INamedConcept **children**:  StatementList body 1 |

|  |  |
| --- | --- |
| *Type System* The type of the **yield** statement is the type of |  |
| the expression that is yielded, specified by this equation13: |  |

**typeof**(yield) :==: **typeof**(yield.result);



|  |  |
| --- | --- |
| Since the **BlockExpression** is used as an **Expression**, it has to have a type as well. However, its type is not explicitly specified, so it has to be calculated as the common supertype of the types of all **yield**s. The following typing rule computes this type. We use the **:>=:** operator to express that the result type must |  |
| be the same or a supertype of the right argument. |  |

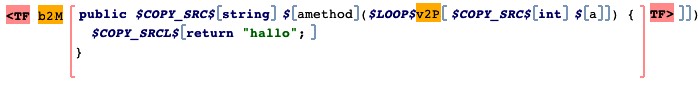
|  |
| --- |
| **typevar** resultType ; **for** (node<BlockExpressionYield> y : blockExpr.descendants<BlockExpressionYield>) {  resultType :>=: **typeof**(y.result);  } **typeof**(blockExpr) :==: resultType; |

*Generator* The generator for **BlockExpression**s reduces the new concept to pure BaseLanguage: it performs assimilation. It transforms a *heterogeneous* fragment (using BaseLanguage and **exprblocks**) to a *homogeneous* fragment (using only BaseLanguage). The first step is the creation of the additional method for the block expression. Fig. 16.8 shows the definition of the weaving rule; Fig. 16.9 shows the template used in that weaving rule.

The template shown in Fig. 16.9 shows the creation of the method. It assigns a mapping label14 to the created method.

14 In earlier MPS examples, we had used

The mapping label creates a mapping between the **BlockExpression** and the created method. We will use this label to refer to this generated method when we generate the method call that replaces the **BlockExpression** (shown in Fig. 16.10).



A second concept introduced by the **exprblocks** language is the **PlaceholderStatement**. This extends **Statement** so that it can be used inside method bodies. It also has a name. It is used to mark locations at which subsequent generators can add additional code. These subsequent generators will use a reduction rule to replace the placeholder with whatever they want to put at this location. It is a means of building extensible generators. Both **BlockExpression** and **PlaceholderStatement** will be used in subsequent examples of in this chapter.



|  |  |
| --- | --- |
|  |  |
| A particularly interesting feature of MPS is the ability to use several extensions of the same base language in a given pro- |  |
| gram *without defining a combining language*. For example, a user could decide to use the block expression language defined above together with the **dispatch** extension discussed in Section 12.2. This is a consequence of MPS’ projectional nature15. |  |
| Let us consider the potential cases for ambiguity: |  |

*Same Concept Name* The used languages may define concepts with the same name as the host language. This will not lead to ambiguity because concepts have a unique ID as well. A program element will use this ID to refer to the concept whose instance it represents.

*Same Concrete Syntax* The projection of a concept is not relevant to the functioning of the editor. The program would still be unambiguous to MPS even if *all elements had the same notation*. Of course, it would be confusing to the users.

*Same Alias* If two concepts that are valid at the same location use the same alias, then, as the user types the alias, it is not clear which of the two concepts should be instantiated. This problem is solved by MPS opening the code completion window and requiring the user to explicitly select which alternative to choose. Once the user has made the decision, the unique ID is used to create an unambiguous program tree.

#### 16.2.4 Reuse with Separated Generated Code

Language reuse covers the case in which a language that has been developed independently of the context in which it should be reused. The respective fragments remain *homogeneous*. In this section, we cover two alternative cases: the first case (in this subsection) addresses a persistence mapping language. The generated code is separate from the code generated from the entities language. The second case (discussed in the next subsection) describes a language for role-based access control. The generated code has to be "woven into" the **entities** code to check permissions when setters are called.

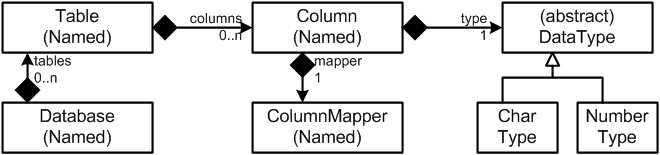
*Structure and Syntax* **relmapping** is a reusable language for mapping arbitrary data to relational tables. The **relmapping** language supports the definition of relational table structures, but leaves the actual mapping to the source data unspecified. As you adapt the language to a specific reuse context, you have to specify this mapping. The following code shows the reusable part: a database is defined that contains tables with columns. Columns have (database-specific) data types.

**database** CompanyDB

**table** Departments

**number** id **char** descr **table** People **number** id **char** name **char** role **char** isFreelancer

Fig. 16.11 shows the structure of the relmapping language. The abstract concept **ColumnMapper** serves as a hook: if we reuse this language in a different context, we extend this hook in a context-specific way.



|  |  |
| --- | --- |
| The **relmapping\_entities** language extends **relmapping** and |  |
| adapts it for reuse with the **entities** language16. To this end, |  |
| it provides a subconcept of **ColumnMapper**, the **AttributeColMapper**, which references an **Attribute** from the **entities** language as a means of expressing the mapping from the attribute to the column. The column mapper is projected on the right of the field definition, resulting in the following (heterogeneous) |  |
| code fragment17: |  |

|  |
| --- |
| **database** CompanyDB **table** Departments **number** id <- Department.id **char** descr <- Department.description  **table** People **number** id <- Employee.id **char** name <- Employee.name **char** role <- Employee.role **char** isFreelancer <- Employee.freelancer |

|  |  |
| --- | --- |
| *Type System* The type of a column is the type of its **type** property. In addition, the type of the column must also conform to the type of the column mapper, so the concrete **ColumnMapper** subtype must provide a type mapping as well. This "typing hook" is implemented as an abstract behavior method |  |
| **typeMappedToDB** on the **ColumnMapper**18. With this in mind, |  |

the typing rules of the **relmapping** language look as follows:

**typeof**(column) :==: **typeof**(column.type); **typeof**(column.type) :==: **typeof**(column.mapper); **typeof**(columnMapper) :==: columnMapper.typeMappedToDB();

The **AttributeColMapping** concept from the **relmapping\_entities** implements this method by mapping **int**s to **number**s and everything else to **char**s.

perspective to have this typing hook, since **relmapping** is designed to be reusable.

**public** node<> typeMappedToDB() **overrides** ColumnMapper.typeMappedToDB { node<> attrType = **this**.attribute.type.type; **if** (attrType.isInstanceOf(IntType)) { **return new** node<NumberType>(); } **return new** node<CharType>();

}

*Generator* The generated code is also separated into a reusable part (a class generated by the **relmapping** language’s generator) and a context-specific subclass of that class, generated by the **relmapping\_entities** language. The generic base class contains code for creating the tables and for storing data in those tables. It contains abstract methods that are used to access the data to be stored in the columns19.

|  |
| --- |
| **public abstract class** CompanyDBBaseAdapter {  **private void** createTableDepartments() {  // SQL to create the Departments table  }  **private void** createTablePeople() {  // SQL to create the People table  }  **public void** storeDepartments(Object applicationData) {  Insert i = **new** Insert("Departments");  i.add( "id", getValueForDepartments\_id(applicationData));  i.add( "descr", getValueForDepartments\_descr(applicationData)); i.execute();  }  **public void** storePeople(Object applicationData) { // like above  } **public abstract** String getValueForDepartments\_id(Object applicationData);  **public abstract** String getValueForDepartments\_descr( Object applicationData);  // abstract getValue methods for the People table } |

The subclass, generated by the generator in the **relmapping\_en-**

**tities** language, implements the abstract methods defined by the generic superclass. The interface, represented by the **applicationData** object, has to be kept generic so that any kind of user data can be passed in[[2]](#footnote-2).

|  |
| --- |
| **public class** CompanyDBAdapter **extends** CompanyDBBaseAdapter { **public** String getValueForDepartments\_id(Object applicationData) {  Object[] arr = (Object[]) applicationData;  Department o = (Department) arr[0]; String val = o.getId() + ""; **return** val;  }  **public** String getValueForDepartments\_descr(Object applicationData) {  Object[] arr = (Object[]) applicationData;  Department o = (Department) arr[0]; String val = o.getDescription() + ""; **return** val;  }  } |

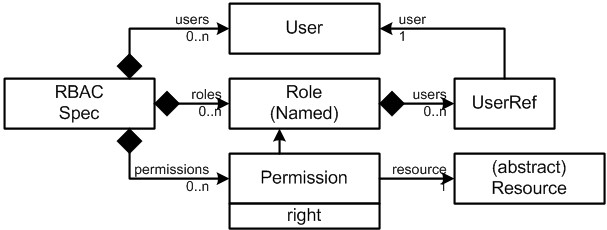
#### 16.2.5 Reuse with Interwoven generated code

**rbac** is a language for specifying role-based access control, to specify access permissions for entities defined with the **entities** language21. Here is some example code:

|  |
| --- |
| **RBAC**  **users**:  **user** mv : Markus Voelter **user** ag : Andreas Graf **user** ke : Kurt Ebert  **roles**:  **role** admin : ke **role** consulting : ag, mv  **permissions**:  admin, W : Department consulting, R : Employee.name |

*Structure and Syntax* The structure of **rbac** is shown in

Fig. 16.12. Like **relmapping**, it provides a hook, in this case, **Resource**, to adapt it to context languages: the sublanguage **rbac\_entities** provides two subconcepts of **Resource**, namely **AttributeResource** to reference to an attribute, and **EntityResource** to refer to an **Entity**, to define permissions for entities and their attributes.

F

*Type System* No type system rules apply here.

*Generator* What distinguishes this case from the **relmapping** case is that the code generated from the **rbac\_entities** language is *not* separated from the code generated from **entities**. Instead, inside the setters of the Java beans, a permission check is required.

|  |
| --- |
| **public void** setName(String newValue) { // check permissions (from rbac\_entities) **if** (!**new** RbacSpecEntities().currentUserHasWritePermission(  "Employee.name")) {  **throw new** RuntimeException("no permission");  }  **this**.name = newValue;  } |

The generated fragment is homogeneous (all Java code), but it is *multi-sourced*, since several generators contribute to the same fragment. To implement this, several approaches are possible:

### 22

|  |  |
| --- | --- |
| • We could use AspectJ . This would allow us to generate separate Java artifacts (all single-sourced) and then use the aspect weaver to "mix" them. However, we don’t want to introduce the complexity of yet another tool, AspectJ, here, so we will not use this approach. |  |
| • An interceptor23 framework could be added to the gener- |  |

ated Java Beans, with the generated code contributing spe-

cific interceptors (effectively building a custom AOP solution). We will not use this approach either, since it would require the addition of a whole interceptor framework to the **entities** implementation. This seems like overkill.

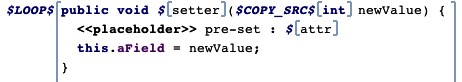
* We could "inject" additional code generation templates to the existing **entities** generator from the **rbac\_entities** generator. This would make the generators *woven*, as opposed to just dependent. Assuming that this would work in

MPS, it would be the most elegant solution – but it does not.

* We could define a hook in the generated Java Beans code and then have the **rbac\_entities** generator contribute code to this hook. This is the approach we will use. The generators remain dependent, they have to agree on the way the hook works.

Notice that only the AspectJ solution can work without any preplanning from the perspective of the **entities** language, because it avoids mixing the generated code artifacts (it is handled "magically" by AspectJ). All other solutions require the original **entities** generator to "expect" certain extensions.

In our case, we have modified the original generator in the **entities** language to contain a **PlaceholderStatement** (see Fig. 16.13). In every setter, the placeholder acts as a hook at which subsequent generators can add statements. So while we have to pre-plan *that* we want to extend the generator at this location, we don’t have to predefine *how*.

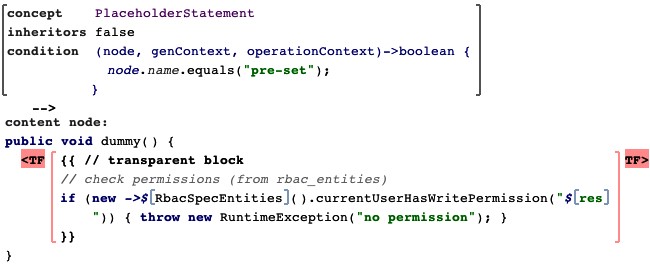


The **rbac\_entities** generator contains a reduction rule for **PlaceholderStatement**s. So when it encounters a placeholder (put there by the **entities** generator) it removes it and inserts the code that checks for the permission (Fig. 16.14). To make this work, we have to make sure that this generator runs *after*

the **entities** generator (since the **entities** generator has to 24 Generator priorities express a partial create the placeholder first) but *before* the BaseLanguage gen- ordering (run before, run after, etc.) between pairs of generators. Upon

erator (which transforms BaseLanguage code into Java text for generation, MPS computes an overcompilation). We use generator priorities to achieve this24. all schedule that determines which *16.2.6 Embedding* a permission check.

generators run in which order.



Figure

16

.

14

:

This reduction rule re

-

places

**PlaceholderStatement**

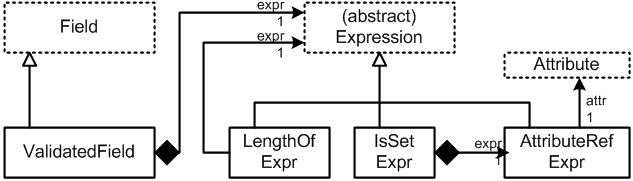
s with

*Structure and Syntax* **uispec\_validation** extends **uispec**:

a sublanguage of the **uispec** language. It supports writing another generated class that contains code such as the following in the UI form specifications:

|  |
| --- |
| **form** CompanyStructure **uses** Department **uses** Employee  **field** Name: **textfield**(30) -> Employee.name **validate** lengthOf(Employee.  name) < 30  **field** Role: **combobox**(Boss, TeamMember) -> Employee.role **field** Freelancer: **checkbox** -> Employee.freelancer **validate if** (isSet(Employee.worksAt)) Employee.freelancer == **false else**  Employee.freelancer == **true**  **field** Office: **textfield**(20) -> Department.description |

Writing the expressions is supported by embedding an **expressions** language. Fig. 16.15 shows the structure. To be able to use the expressions, the user has to use a **ValidatedField** instead of a **Field**. **ValidatedField** is also defined in **uispec\_validation**, and is a subconcept of **Field**.



|  |  |
| --- | --- |
| To support the migration of existing models that use **Field** instances, we provide an intention: the user can press **Alt-Enter** on a **Field** and select **Make Validated Field**. This transforms an existing **Field** into a **ValidatedField**, so that validation |  |
| expressions can be entered25. The core of the intention is the |  |
| following script, which performs the actual transformation: |  |

of a field definition to trigger the

execute(editorContext, node)->**void** { transformation code below. node<ValidatedField> vf = **new** node<ValidatedField>(); vf.widget = node.widget; vf.attribute = node.attribute; vf.label = node.label; node.replace with(vf);

}

|  |  |
| --- | --- |
| The **uispec\_validation** language extends the **uispec** language.  We also extend the existing, embeddable **expressions** language, |  |
| so we can use **Expressions** in the definition of our language26. |  |
| **ValidatedField** has a property **expr** that contains the validation expression.  As a consequence of polymorphism, we can use any existing subconcept of **Expression** in validations. So without doing anything else, we could write **20 + 40 > 10**, since integer literals and the **+** and **>** operators are defined as part of the composed **expressions** language. However, to write anything useful, we have to be able to reference entity attributes from within |  |
| expressions27. To achieve this, we create the **AttributeRefExpr**, | 27 We argued in the design part that, in |

as shown in Fig. 16.15. We also create **LengthOf** and **IsSetExpression** as further examples of how to adapt an embedded language to its new context – i.e. the **uispec** and **entities** languages.

The **AttributeRefExpr** references an **Attribute** from the **entities** language; however, it may only reference those attributes of those entities that are used in the form in which we define the validation expression. The following is the code for the search scope:

order to make an embedded language useful with its host language, it has to be extended: the following is an example of this.

|  |
| --- |
| (model, **scope**, referenceNode, linkTarget, enclosingNode)  ->**join**(ISearchScope | sequence<node< >>) {  nlist<Attribute> res = **new** nlist<Attribute>; node<Form> form = enclosingNode.ancestor<Form, +>; **if** ( form != **null** ) {  **for** (node<EntityReference> er : form.usedEntities) { res.addAll(er.entity.attributes);  } } res;  } |

Notice that the actual syntactic embedding of the **expressions** language in the **uispec\_validation** language is no problem at all as a consequence of how projectional editors work. We simply define **Expression** to be a child of the **ValidatedField**.

*Type System* The general challenge here is that primitive types such as **int** and **string** are defined in the **entities** language *and* in the embeddable **expressions** language. Although they have the same names, they are not the same types. So the two sets of types must be mapped. Here are a couple of examples. The type of the **IsSetExpression** is by definition **expressions.BooleanType**. The type of the **LengthOf**, which takes an **AttrRefExpression** as its argument, is **expressions. IntType**. The type of an attribute reference is the type of the attribute’s **type** property:

**typeof**(attrRef) :==: **typeof**(attrRef.attr.type);

However, consider the following code:

|  |
| --- |
| **field** Freelancer: **checkbox** -> Employee.freelancer **validate if** (isSet(Employee.worksAt)) Employee.freelancer == **false**  **else** Employee.freelancer == **true** |

|  |  |
| --- | --- |
| expressions language’s **==** operator. Here is how we do it.  In the **expressions** language, we define *overloaded operation rules*. We specify the resulting type for an **EqualsExpression** depending on its argument types. Here is the code in the | languages are developed by independent people, then it is hard to enforce a common base language. So the ability to have such mappings is useful. |
| **expressions** language that defines the resulting type to be **boolean** if the two arguments are **Equallable**29: | 29 In addition to this code, we have to specify that **expressions.BooleanType** is a subtype of **Equallable**, so this |

This code states that if the **worksAt** attribute of an employee is set, then its **freelancer** attribute must be **false** else it must be **true** (freelancers don’t **workAt** anything). It uses the **==** operator from the **expressions** language. However, that operator expects two arguments with **expressions.BooleanType**, but the type of the **Employee. freelancer** is **entities.BooleanType**[[3]](#footnote-3). In effect, we have to override the typing rules for the

**operation concepts**: EqualsExpression

**left operand type**: **new** node<Equallable>()

|  |
| --- |
| **right operand type**: **new** node<Equallable>()  **operation type**:  (**operation**, leftOperandType, rightOperandType)->node< > {  <**boolean**>; } |

We have to tie this overloaded operation specification into a regular type inference rule:

|  |
| --- |
| **rule** typeof\_BinaryExpression { **applicable for** BinaryExpression **as** binex **do** {  node<> opType = **operation type**( binex , left , right );  **if** (opType != **null**) { **typeof**(binex) :==: opType;  } **else** { **error** "operator " + binex.concept.name + " cannot apply **to** these argument types " + left.concept.name + "/" + right.concept.name -> binex; }  }  } |

To override these typing rules to work with **entities.BooleanType**, we simply provider another overloaded operation specification in the **uispec\_validation** language:

|  |
| --- |
| **operation concepts**: EqualsExpression **one operand type**: <**boolean**> // the entities.BooleanType!  **operation type**:  (op, leftOperandType, rightOperandType)->node< > {  <**boolean**>; // the expressions.BooleanType } |

*Generator* The generator has to create BaseLanguage code, which is then subsequently transformed into Java text. To deal with the transformation of the expressions language, we can do one of two things:

• Either we can use the **expressions** language’s existing totext generator and wrap the expressions in some kind of

**TextHolderStatement**[[4]](#footnote-4).

|  |  |
| --- | --- |
|  |  |
| • Alternatively, we can write a (reusable) transformation from expressions code to BaseLanguage code; these rules would get used as part of the transformation of **uispec** and **uispec\_validation** code to BaseLanguage. |  |
| Since many DSLs will probably transform code to BaseLanguage, it is worth the effort to write a reusable generator from **expressions** to BaseLanguage31. So we choose this second al- |  |

ternative. The generated Java code is multi-sourced, since it is generated by two independent code generators.

Expression constructs from the reusable **expressions** language and those of BaseLanguage are almost identical, so this

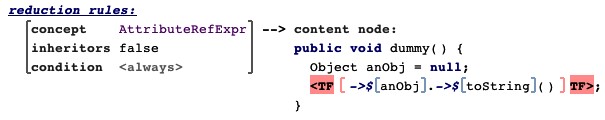
generator is trivial. We create a new language project **expressions.blgen** and add reduction rules32. Fig. 16.16 shows some

|  |
| --- |
| **expressions.blgen** only contains a  generator it is still a  *language*  in MPS  terminology.  of these reduction rules. |

32 In MPS all "meta stuff" is called a language. So even though

In addition, we also need reduction rules for the new expressions that we have added specifically in the **uispec\_validation** language (**AttrRefExpression, isSetExpression, LengthOf**). These transformations are defined in **uispec\_validation**, since this language is *not* reusable – it is specifically designed to integrate the **uispec** and the **expressions** languages. As an example, Fig. 16.17 shows the rule for handling the **AttrRefExpression**. The validation code itself is "injected" into the UI form via the same placeholder reduction as in the case of the language **rbac\_entities**.

reducFigure 16.17: References to entity



Language extension can also be used to prevent the use of specific base language concepts in the sublanguage, possibly in certain contexts. As an example, we restrict the use of some operators provided by the reusable expression language inside validation rules in **uispec\_validation**. This can be achieved

by implementing a **can be ancestor** constraint on **ValidatedField**.

|  |
| --- |
| **can be ancestor**:  (operationContext, scope, node, childConcept)->**boolean** { **return** !(childConcept == **concept**/GreaterEqualsExpression/ || childConcept == **concept**/LessEqualsExpression/); } |

#### 16.2.7 Annotations

|  |
| --- |
| **module** company **entity** Employee {  id : **int** -> People.id name : **string** -> People.name |

In a projectional editor, the CS of a program is projected from the AST. A projectional system always goes from AS to CS, never from CS to AS (as parsers do). This means that the CS does not have to contain all the data necessary to build the AST (which is necessary in the case of parsers). This has two consequences:

* A projection may be *partial*, in the sense that the AS contains data that is not shown in the CS. The information may, for example, only be changeable via intentions (discussed in Section 7.7), or the projection rule may project some parts of the program only in some cases, controlled by some kind of configuration data.
* It is also possible to project *additional* CS that is not part of the CS definition of the original language. Since the CS is never used as the information source, such additional syntax does not confuse the tool (in a parser-based tool the grammar would have to be changed to take into account this additional syntax to avoid derailing the parser).

|  |  |
| --- | --- |
| In this section we discuss the second alternative, since it constitutes a form of language composition: the additional CS is composed with the original CS defined for the language. The mechanism MPS uses for this is called *annotations*. We have |  |
| seen annotations when we discussed templates33: an annota- |  |
| tion is something that can be attached to arbitrary program elements and can be shown together with the CS of the annotated element. In this section we use this approach to implement an alternative approach for the entity-to-database mapping. Using this approach, we can store the mapping from entity attributes to database columns directly in the **Entity**, resulting in the following code: |  |

|  |
| --- |
| role : **string** -> People.role worksAt : Department -> People.departmentID freelancer : **boolean** -> People.isFreelancer  }  **entity** Department {  id : **int** -> Departments.id description : **string** -> Departments.descr } |

This is a heterogeneous fragment, consisting of code from the **entities** language, as well as the annotation code (e.g., **-> People.id**). From a CS perspective, the column mapping is "embedded" in the **Entity**. In the AST the mapping information is also actually stored in the **entities** model. However, the definition of the **entities** language does not know that this additional information is stored and projected "inside" entities. No modification to the **entities** language is necessary.

*Structure and Syntax* We define an additional language **relmapping\_annotations** that extends the **entities** language as well as the **relmapping** language. In this language we define the following concept:

|  |
| --- |
| **concept** AttrToColMapping **extends** NodeAnnotation **references**:  Column column 1 **properties**: role = colMapping  **concept** links:  annotated = Attribute |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **NodeAnnotation** have to provide a **role** property and an **annotated** concept link. Structurally, an annotation is a child of the node it annotates. So the **Attribute** has a new child of type |  | | **AttrToColMapping**, and the reference that contains the child is called **@colMapping** – the value of the **role** property prefixed by an **@**. The **annotated** concept link points to the concept *to which this annotation can be added*. **AttrToColMapping**s can be annotated to instances of **Attribute**. |  |   While structurally the annotation is a child of the annotated node, in the CS the relationship is reversed: the editor for  **AttrToColMapping** wraps the editor for **Attribute**, as Fig. 16.18 shows. |

**AttrToColMapping** concept extends **NodeAnnotation**, a concept predefined by MPS34. Concepts that extend the concept 34 In fact, this concept is called **NodeAttribute** in MPS. For historical reasons there a somewhat confused Since the annotation is not part of the original language, it cannot just be typed in: it must be attached to nodes via an intention. The annotation simply adds a new instance of

**AttrToCol- Mapping** to the **@colMapping** property of an **Attribute**, so we don’t show the code here.

*Type System* The same typing rules are necessary as in the **relmapping\_entities** language described previously. They reside in **relmapping\_annotations**.

*Generator* The generator is also broadly similar to the previous example with **relmapping\_entities**. It takes the **entities** model as the input, and then uses the column mappings in the annotations to create the entity-to-database mapping.

|  |
| --- |
| **grammar** org.xtext.example.lmrc.entity.EntityDsl **with** org.eclipse.xtext.common.Terminals  **generate** entityDsl "http://www.xtext.org/example/lmrc/entity/EntityDsl" |

The annotations introduced above were typed to be specific to certain target concepts (**Attribute** in this case). A particularly interesting use of annotations includes those that can be annotated to *any* language concept (formally targeting **BaseConcept**). In this case, there is no dependency between the language that contains the annotation and the language that is annotated. This is very useful for "meta data", as well as anything that can be processed generically.

#### 16.3 Xtext Example

In this section we look at an example roughly similar to the one for MPS discussed in the previous section. We start out with a DSL for entities. Here is an example program:

|  |
| --- |
| **module** company { **entity** Employee { id : **int** name : **string** role : **string** worksAt : Department freelancer : **boolean**  }  **entity** Department { id : **int** description : **string**  }  } |

The grammar is straightforward and should be clear if you have read the implementation part of this book so far.

This section of the book has been written together with Christian Dietrich. Contact him via **christian.dietrich@itemis.de**.

|  |
| --- |
| Module:  "module" name=**ID** "{" entities+=Entity\*  "}";  Entity:  "entity" name=**ID** "{" attributes+=Attribute\*  "}";  Attribute:  name=**ID** ":" type=AbstractType; Named: Module|Entity|Attribute;  AbstractType:  BooleanType|IntType|StringType|EntityReference;  BooleanType: {BooleanType} "boolean";  IntType: {IntType} "int";  StringType: {StringType} "string";  EntityReference: ref=[Entity|FQN];  FQN: **ID** ("." **ID**)\*; |

##### 16.3.1 Referencing

Referencing describes the case in which programs written in one DSL reference (by name) program elements written in another DSL35. The example we use is the UI specification lan-

|  |
| --- |
| **form** CompanyStructure **uses** Department // reference to Department Entity **uses** Employee // reference to Employee Entity  **field** Name: **textfield**(30) -> Employee.worksAt **field** Role: **combobox**(Boss, TeamMember) -> Employee.role **field** Freelancer: **checkbox** -> Employee.freelancer **field** Office: **textfield**(20) -> Department.description |

guage, in which a **Form** defined in the UI model refers to **Entities** from the language defined above, and **Field**s in a form refers to entity **Attribute**. Here is some example code:

*Structure* Referencing concepts defined in another language relies on importing the target meta model and then defining references to concepts defined in this meta model36. Here is

|  |
| --- |
| **grammar** org.xtext.example.lmrc.uispec.UispecDsl **with** org.eclipse.xtext.common.Terminals  **import** "http://www.xtext.org/example/lmrc/entity/EntityDsl" as entity **generate** uispecDsl "http://www.xtext.org/example/lmrc/uispec/UispecDsl" |

the header of the grammar of the **uispec** language:

Importing a meta model means that the respective *meta classes* can now be used. Note that the meta model import does not make the *grammar rules* visible, so the meta classes can only be

used in references and as base types (as we will see later). In the case of referencing, we use them in references:

|  |
| --- |
| EntityReference:  "uses" entity=[entity::Entity|FQN];  Field:  "field" label=**ID** ":" widget=Widget  "->" attribute=[entity::Attribute|FQN]; |

To make this work, no change is required in the **entities** language37. However, the workflow generating the **uispec** language has to be changed. The **genmodel** file for the meta model has to be registered in the **StandaloneSetup**38.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | bean = StandaloneSetup { ...  registerGenModelFile = "platform:/resource/org.xtext.example.lmrc.  entity/src-gen/org/xtext/example/lmrc/entity/EntityDsl.genmodel"  } |   This is an EMF technicality and we won’t discuss it in any further detail.  We have to do one more customization to make the language work smoothly. The only **Attribute**s that should be visible are those from the entities referenced in the current **Form**’s **uses** clauses, and they should be referenced with a qualified name  (**Employee.role** instead of just **role**). Scoping has to be customized to achieve this:   |  | | --- | | **public** IScope scope\_Field\_attribute(Field context, EReference ref) {  Form form = EcoreUtil2.getContainerOfType(context, Form.**class**); List<Attribute> visibleAttributes = **new** ArrayList<Attribute>(); **for** (EntityReference useClause : form.getUsedEntities()) { visibleAttributes.addAll(useClause.getEntity().getAttributes());  }  Function<Attribute, QualifiedName> nameComputation = **new** Function<Attribute, QualifiedName>() {  @Override **public** QualifiedName apply(Attribute a) { **return** QualifiedName.create(((Entity)a.eContainer()). getName(), a.getName()); }  }; **return** Scopes.scopeFor(visibleAttributes, nameComputation , IScope. NULLSCOPE);  } | | This scoping function performs two tasks: first, it finds all the **Attribute**s of all **used** entities. We collect them into a list **visibleAttributes**. The second part of the scoping func- | | |  | | tion defines a **Function** object39 that represents a function from | | |  | |

**Attribute** to **QualifiedName**. In the implementation method **apply** we create a qualified name made from two parts: the entity name and the attribute name (the dot between the two is default behavior for the **QualifiedName** class). When we create the scope itself in the last line we pass in the list of attributes, as well as the function object. Xtext’s scoping framework uses the function object to determine the name by which each of the attributes is referenceable from this particular context.

function object notation, because Java does not provide support for closures or lambdas at this point! Alternatively you could do this with Xtend, which does support closures.

*Type System* As we discussed in Section 20.2, dealing with type systems in the referencing case is not particularly challenging, since the type system of the referencing language can be built with knowledge of the type system of the referenced language.

*Generators* The same is true for generators. Typically they just share knowledge about the naming of generated code elements.

##### 16.3.2 Reuse

Referencing concerns the case in which the referencing language is built with knowledge about the referenced language, so it can have direct dependencies. In the example above, the **uispec** language uses **Entity** and **Attribute** from the **entities** language. It directly imports the meta model, so it has a direct dependency. In the case of reuse, such a direct dependency is not allowed. Our goal is to combine two *independent* languages. To illustrate this case, we again use the same example as in the MPS section.

*Structure* We first introduce a **db** language, a trivial DSL for defining relational table structures. These can optionally be mapped to a data source, but the language makes no assumption about how this data source looks (and which language is used to define it). Consequently, the grammar has no dependency on any other, and imports no other meta model:

|  |
| --- |
| **grammar** org.xtext.example.lmrc.db.DbDsl **with** org.eclipse.xtext.common. Terminals **generate** dbDsl "http://www.xtext.org/example/lmrc/db/DbDsl"  Root:  Database;  Database:  "database" name=**ID** tables+=Table\*;  Table:  "table" name=**ID** columns+=Column\* ;  Column:  type=AbstractDataType name=**ID** (mapper=AbstractColumnMapper)?;  AbstractColumnMapper:  {AbstractColumnMapper}"not mapped";  AbstractDataType: CharType | NumberType;  CharType: {CharType}"char";  NumberType: {NumberType}"number"; |

Just as in the MPS example, the **Column** rule has an optional **mapper** property of type **AbstractColumnMapper**. Since it is not possible to explicitly mark rules as generating abstract meta classes, we simply define the syntax to be **not mapped**40. This

|  |  |  |
| --- | --- | --- |
| language has been designed *for* reuse, because it has this hook  **AbstractColumnMapper**, which can be customized later. But the language is still independent. In the next step, we want to be able to reference **Attribute**s from the **entities** language: | | this. |
| **database** CompanyDB  **table** Departments  **number** id <- Department.id **char** descr <- Department.description  **table** People  **number** id <- Employee.id **char** name <- Employee.name **char** role <- Employee.role  **char** isFreelancer <- Employee.freelancer |
| To make this possible, we create a new language **db2entity** that *extends* the **db** language and *references* the **entities** lan- | |  | |
| guage41. This is reflected by the header of the **db2entity** lan- | |  | |
| guage (notice the **with** clause): | |  | |

|  |
| --- |
| **grammar** org.xtext.example.lmrc.db2entity.Db2EntityDsl **with** org.xtext.example.lmrc.db.DbDsl  **import** "http://www.xtext.org/example/lmrc/db/DbDsl" as db **import** "http://www.xtext.org/example/lmrc/entity/EntityDsl" as entity  **generate** db2EntityDsl  "http://www.xtext.org/example/lmrc/db2entity/Db2EntityDsl" |

We now have to overwrite the **AbstractColumnMapper** rule defined in the **db** language:

AbstractColumnMapper **returns** db::AbstractColumnMapper:

{EntityColumnMapper} "<-" entity=[entity::Attribute|FQN];

We create a rule that has the same name as the rule in the supergrammar. So when the new grammar calls the **AbstractColumnMapper** rule, our new definition is used. Inside, we define the new syntax we would like to use, and as part of it, we reference an **Attribute** from the imported **entity** meta model. We then use the **{EntityColumnMapper}** action to force instantiation of an **EntityColumnMapper** object: this also implicitly leads to the creation of an **EntityColumnMapper** class in the generated **db2entity** meta model. Since our new rule **returns** an **db::AbstractColumnMapper**, this new meta class extends **AbstractColumnMapper** from the **db** meta model – which is exactly what we need[[5]](#footnote-5).

*Type System* The primary task of the type system in this example would be mapping the primitive types used in the **entities** language to those used in the **db** language to make sure we only map those fields to a particular column that are type-compatible. Just as the column mapper itself, this code lives in the adapter language. It is essentially just a constraint that checks for type compatibility.

*Generator* Let us assume there is a generator that generates Java Beans from the **entities**. Further, we assume that there is a generator that generates all the persistence management code from **DbDsl** programs, except the part of the code that fetches the data from whatever the data source is – essentially we leave the same "hole" as we do with the **AbstractColumnMapper** in the grammar. And just in the same way as we define the

**EntityColumnMapper** in the adapter language, we have to adapt the executing code. We can use two strategies.

The first one uses the composition techniques of the target language, i.e. Java. The generated code of the **DbDsl** could for example generate an abstract class that has an abstract method **getColumnData** for each of the table columns. The generator for the adapter language would generate a concrete subclass that implements these methods to grab the data from entities43.

|  |  |
| --- | --- |
| This way the modularity (**entities**, **db**, **db2entity**) is propa- |  |
| gated into the generated artifacts as well. No *generator* composition is required44. | 44 In a Java/Enterprise world this would most likely be the way we’d do it in practice. The next alternative is a bit |
| However, consider a situation in which we have to gener- | constructed. |
| ate inlined code, for reasons of efficiency, e.g., in some kind of embedded system. In this case the **DbDsl** generator would have to be built in an extensible way. Assuming we use Xtend for generation, this can be done easily by using dependency injection45. Here is how you would do that: | 45 Sometimes people complain about the fact that Xtend is a general purpose language, and not some dedicated code generation language. However, the fact that one can use abstract classes, abstract methods and dependency injection is a nice example of how and |
| • In the generator that generates persistence code from a **DbDsl** program, the code that generates the inlined "get data for | why a general purpose language (with some dedicated support for templating) is useful for building generators. |
| column" code delegates to a class that is dependency-injected46.  The Xtend class we delegate to would be an abstract class | 46 This is a nice illustration of building a |

that has one abstract method **generateGetDataCodeFor( Column c)**.

**class** GetDataGenerator { **def void** generateGetDataCodeFor(Column c)

}

**class** DbDslGenerator implements IGenerator {

|  |
| --- |
| @Inject GetDataGenerator gdg  **def** someGenerationMethod(Column c) { // ...  String getDataCode = gdg.generateGetDataCodeFor(c)  // then embed getDataCode somewhere in the  // template that generates the DbDsl code }  } |

* The generator for the adapter language would contain a subclass of this abstract class that implements the **generateGetDataCodeFor** generator method in a way suitable to the **entities** language.
* The adapter language would also set up Google Guice dependency injection in such a way as to use this a singleton instance of this subclass when instances of the abstract class are expected.

##### 16.3.3 Extension

We have already seen the mechanics of extension in the previous example, since, as a way of building the reuse infrastructure, we have extended the **db** language. In this section we look at extension in more detail. Extension is defined as syntactic integration with explicit dependencies. However, as we discussed in Section 4.6.2 there are two use cases that feel different:

|  |  |
| --- | --- |
|  | above is an example of this. The |
| 2. The other case is where we create a completely new language, but reuse some of the syntax provided by the base language. This use case *feels* like embedding (we embed syntax from the base language in our new language), but with regard to the classification according to syntactic integration and dependencies, it is still extension. Embedding would prevent explicit dependencies. In this section we look at extension with an embedding flavor. |  |
| To illustrate an extension-with-embedding flavor, we will show how to embed Xbase expressions in a custom DSL. Xbase is a reusable expression language that provides primitive types, various unary and binary operators, functions and closures. As we will see, it is very tightly integrated with Java48. As an |  |

1. In one case we provide (small scale, local, fine grained) additional syntax to an otherwise unchanged language47.

example, we essentially create another entity language; thanks to Xbase, we will be able to write:

|  |
| --- |
| **entity** Person { lastname : **String** firstname : **String**  **String** fullName(**String** from) { **return** "Hello " + firstname + " " + lastname + " from " + from  }  } |

Below is the essential part of the grammar. Note how it extends the Xbase grammar (the **with** clause) and how it uses various elements from Xbase throughout the code (those whose names start with an **X**).

|  |
| --- |
| **grammar** org.xtext.example.lmrc.entityexpr.EntityWithExprDsl **with** org.eclipse.xtext.xbase.Xbase  **generate** entityWithExprDsl  "http://www.xtext.org/example/lmrc/entityexpr/EntityWithExprDsl"  Module:  "module" name=**ID** "{" entities+=Entity\*  "}";  Entity:  "entity" name=**ID** "{" attributes+=Attribute\* operations+=Operation\* "}";  Attribute: name=**ID** ":" type=JvmTypeReference;  Operation:  type=JvmTypeReference name=**ID** "(" (parameters+=FullJvmFormalParameter  (’,’ parameters+=FullJvmFormalParameter)\*)? ")" body=XBlockExpression; |

|  |  |
| --- | --- |
| the body of **Operation**, which essentially allows us to use the full Xbase language inside the body of the **Operation**. To make the **parameters** visible, we use the **FullJvmFormalParameter** |  |
| rule50. |  |

Let’s look at some of the details. First, the **type** properties of the **Attribute** and the **Operation** are not defined by our grammar; instead we use a **JvmTypeReference**. This makes all Java types legal at this location49. We use an **XBlockExpression** as

In addition to using Xbase language concepts in the definition of our grammar, we also tie the semantics of our language to Java and the JVM. To do this, the **JvmModelInferrer**, shown below, maps a model expressed with this language to a structurally equivalent Java "model". By doing this, we get a number of benefits "for free", including scoping, typing and a code generator. Let us look at this crucial step in some detail.

|  |
| --- |
| **class** EntityWithExprDslJvmModelInferrer **extends** AbstractModelInferrer {  @Inject **extension** IQualifiedNameProvider @Inject **extension** JvmTypesBuilder **def dispatch void** infer(Entity entity, |

|  |
| --- |
| IAcceptor<JvmDeclaredType> acceptor, **boolean** isPrelinkingPhase) { ...  }  } |

This Xtend class extends **AbstractModelInferrer** and implements its **infer** method to create structurally equivalent Java code as an EMF tree, and registers it with the **acceptor**. The method is marked as **dispatch**, so it can be polymorphically overwritten for various language concepts. We override it for the **Entity** concept. We have also injected the **IQualifiedNameProvider** and **JvmTypesBuilder**. The latter provides a builder API for creating all kinds of JVM objects, such as fields, setters, classes or operations. The next piece of code makes use of such a builder:

acceptor.accept( entity.toClass( entity.fullyQualifiedName ) [ documentation = entity.documentation ...

]

)

|  |  |
| --- | --- |
| At the top level, we map the **Entity** to a **Class**. **toClass** is one of the builder methods defined in the **JvmTypesBuilder**. The class we create should have the same name as the **entity**; the name of the class is passed into the constructor. The second argument, written conveniently behind the parentheses, is a closure. Inside the closure, we set the documentation of the |  |
| created class to be the documentation of the **entity**51. Next |  |
| we create a field, a getter and a setter for each of the attributes of the **Entity** and add them to the **Class**’ **members** collection: |  |

|  |
| --- |
| attr : entity.attributes ) { members += attr.toField(attr.name, attr.type) members +=  attr.toGetter(attr.name, attr.type) members += attr.toSetter(attr.name, attr.type)  } |

**documentation = ...** this actually means **it.documentation = ...**.

|  |  |
| --- | --- |
| **toField**, **toGetter** and **toSetter** are all builders contributed by the **JvmTypesBuilder**. To better understand what they do, |  |
| here is the implementation of **toSetter**52. |  |

|  |
| --- |
| **public** JvmOperation toSetter(EObject sourceElement, **final** String name, JvmTypeReference typeRef) {  JvmOperation res = TypesFactory.eINSTANCE.createJvmOperation(); res.setVisibility(JvmVisibility.PUBLIC);  res.setSimpleName("set" + nullSaveName(Strings.toFirstUpper(name))); res.getParameters().add(toParameter(sourceElement, nullSaveName(name), cloneWithProxies(typeRef)));  **if** (name != **null**) { setBody(res, **new** Functions.Function1<ImportManager, CharSequence>() { **public** CharSequence apply(ImportManager p) {  **return** "this." + name + " = " + name + ";";  }  });  } |

|  |
| --- |
| **return** associate(sourceElement, res);  } |

The method first creates a **JvmOperation** and sets the visibility and the name. It then creates a parameter that uses the **typeRef** passed in as the third argument as its type. As you can see, all of this happens via model-to-model transformation. This is important, because these created objects are used implicitly in scoping and typing. The body, however, is created textually; it is not needed for scoping or typing: it is used only in code generation53. The last line is important: it associates the source

element (the **Attribute** in our case) with the created element (the setter **Operation** we just created). As a consequence of

this association, the Xbase scoping and typing framework can work its magic of providing support for our DSL without any further customization!

Let’s now continue our look at the implementation of the

**Jvm- ModelInferrer** for the **Entity**. The last step before our detour was that we created fields, setters and getters for all attributes of our **Entity**. We have to deal with the operations of our **Entity** next.

|  |
| --- |
| **for** ( op : entity.operations ) { members += op.toMethod(op.name, op.type) [ **for** (p : op.parameters) { parameters += p.toParameter(p.name, p.parameterType)  } body = op.body  ]  } |

This code should be easy to understand. We create a method for each **Operation** using the respective builder method, pass in the name and type, create a parameter for each of the parameters of our source operation and then assign the body of the created method to be the body of the operation in our DSL program. The last step is particularly important. Notice that we don’t clone the body, we assign the object *directly*. Looking into the **setBody** method (the assignment is actually mapped to a setter in Xtend), we see the following:

**void** setBody(JvmExecutable logicalContainer, XExpression expr) { **if** (expr == **null**) **return**; associator.associateLogicalContainer(expr, logicalContainer);

}

The **associateLogicalContainer** method is what makes the automatic support for scoping and typing happen[[6]](#footnote-6):

* Because the operation is the container of the expression, the expression’s type and the operation’s type must be compatible
* Because the expression(s) live inside the operation, the parameters of the operation, as well as the current class’s fields, setters and getters are in scope automatically.

*Generator* The JVM mapping shown above already constitutes the full semantic mapping to Java. We map entities to Java classes and fields to members and getters/setters. We do not have to do anything else to get a generator: we can reuse the existing Xbase-to-Java code generator.

If we build a language that cannot easily be mapped to a JVM model, we can still reuse the Xbase expression compiler, by injecting the **JvmModelGenerator** and then delegating to it at the respective granularity. You can also change or extend the behavior of the default **JvmModelGenerator** by overriding its

**\_internalDoGenerate(EObject, IFileSystemAccess)** method

|  |  |  |  |
| --- | --- | --- | --- |
| for your particular language concept55. | | |  |
| *Extending Xbase* In the above example we embedded the (otherwise unchanged) Xbase language into a simple DSL. Let’s now look at how to extend Xbase itself by adding new literals and new operators. We start by defining a literal for dates: | | |  |
| XDateLiteral:  ’date’ ’:’ year=INT ’-’ month=INT ’-’ day=INT; |

|  |  |
| --- | --- |
| These new literals should be literals in terms of Xbase, so we have to make them subtypes of **XLiteral**. Notice how we override the **XLiteral** rule defined in Xbase. We have to repeat its |  |
| original contents; there is no way to "add" to the literals56. |  |

|  |
| --- |
| XLiteral **returns** xbase::XExpression:  XClosure |  XBooleanLiteral |  XIntLiteral |  XNullLiteral |  XStringLiteral |  XTypeLiteral | XDateLiteral; |

|  |  |  |  |
| --- | --- | --- | --- |
| We use the same approach to add an additional operator that | | |  |
| uses the **===** symbol57: | | |  |
| OpEquality:  ’==’ | ’!=’ | ’===’; |

The **===** operator does not yet exist in Xtend, so we have to specify the name of the method that should be called if the operator is used in a program58. The second line of the method

**initializeMapping** maps the new operator to a method named **operator\_identity**:

|  |
| --- |
| **public class** DomainModelOperatorMapping **extends** OperatorMapping { **public static final** QualifiedName IDENTITY = create("===");  @Override **protected void** initializeMapping() { **super**.initializeMapping(); map.put(IDENTITY, create("operator\_identity"));  }  } |
| We implement this method in a new class that we call **Object-** | |  |
| **Extensions2**59: | |  |

|  |
| --- |
| **public class** ObjectExtensions2 { **public static boolean** operator\_identity(Object a, Object b) { **return** a == b;  }  } |

|  |  |
| --- | --- |
| Through the **operator\_identity** operation, we have expressed all the semantics: the Xbase generator will generate a call to |  |
| that operation in the generated Java code60. We have also im- |  |
| plicitly specified the typing rules: through the mapping to the **operator\_identity** operation, the type system uses the types specified in this operation. The type of **===** is **boolean**, and there are no restrictions on the two arguments; they are typed |  |
| as **java.lang.Object**61. |  |

We also want to override the existing minus operator for the new date literals to calculate the time between two dates. We don’t have to specify the mapping to a method name, since the mapping for minus is already defined in Xbase. However, we have to provide an overloaded implementation of the **operator\_minus** method for dates:

|  |
| --- |
| **public class** DateExtensions { **public static long** operator\_minus(Date a, Date b) { **long** resInMilliSeconds = a.getTime() - b.getTime(); **return** millisecondsToDays( resInMilliSeconds );  }  } |

To make Xtend aware of these new classes, we have to register them. To do so, we extend the **ExtensionClassNameProvider**. It associates the classes that contain the operator implementation methods with the types to which these methods apply:

|  |
| --- |
| **public class** DomainModelExtensionClassNameProvider **extends**  ExtensionClassNameProvider { |

these could be done by overriding the **\_expectedType** operation in **XbaseTypeProvider**.

|  |
| --- |
| @Override **protected** Multimap<Class<?>, Class<?>> simpleComputeExtensionClasses()  {  Multimap<Class<?>, Class<?>> result = **super**.simpleComputeExtensionClasses();  result.put(Object.**class**, ObjectExtensions2.**class**); result.put(Date.**class**, DateExtensions.**class**); **return** result;  }  } |

We now have to extend the type system: it has to be able to derive the types for date literals. We create a type provider that extends the default **XbaseTypeProvider**62:

|  |
| --- |
| @Singleton **public class** DomainModelTypeProvider **extends** XbaseTypeProvider {  @Override  **protected** JvmTypeReference type(XExpression expression,  JvmTypeReference rawExpectation, **boolean** rawType) {  **if** (expression **instanceof** XDateLiteral) { **return** \_type((XDateLiteral) expression, rawExpectation, rawType);  } **return super**.type(expression, rawExpectation, rawType);  }  **protected** JvmTypeReference \_type(XDateLiteral literal,  JvmTypeReference rawExpectation, **boolean** rawType) {  **return** getTypeReferences().getTypeForName(Date.**class**, literal);  }  } |

|  |  |
| --- | --- |
| Finally we have to extend the Xbase compiler so that it can handle date literals: |  |

|  |
| --- |
| **public class** DomainModelCompiler **extends** XbaseCompiler { **protected void** \_toJavaExpression(XDateLiteral expr, IAppendable b) {  b.append("new java.text.SimpleDateFormat(\"yyyy-MM-dd\").parse(\"" +  expr.getYear() + "-" + expr.getMonth() + "-" + expr.getDay() + "\")"); }  } |

|  |  |
| --- | --- |
| *Active Annotations* Xtext’s Xtend language comes with Active Annotations. They use the same syntax as regular Java annotations63. However, they can influence the translation pro- |  |
| cess from Xtend to Java64. Each annotation is essentially asso- | processor). |
| ciated with a model-to-model transformation that creates the necessary Java code. This allows the execution semantics of the respective Xtend class to be influenced.  At the time of this writing, the most impressive active annotation (prototype) I have seen involves GWT programming[[7]](#footnote-7).  They implement the following two annotations: |  |

*Services* From a simple Xtend class that contains the serverside implementation methods, the annotation generates the necessary remote interface and the other boilerplate that enables the remote communication infrastructure.

*UI Forms* In GWT, a UI form is defined by an XML file that defines the structure, as well as by a Java class that implements the behavior. The behavior includes the event handlers for the UI elements defined in the XML file. To this end, the class has to have fields that correspond (in name and type) to the UI elements defined in the XML. By using an annotation, this duplication can be avoided: the annotation implementation inspects the associated XML and automatically introduces the necessary fields.

Active annotations will provide a number of additional features. First, they can implement custom validations and quick fixes for the IDE. Second, they can change the scope and the type system, with the IDE being aware of that66. Third, you can pass JVM types or expressions into annotations:

@Pre( b != 0 ) **def** divide(int a, int b) { run not just during code generation, return a / b

|  |  |
| --- | --- |
|  |  |
| It is possible to define whether the expression is passed in as an AST (**b != 0**), or whether the result of the evaluation of the expression is passed in (**true** or **false**).  While the syntactic limitations of annotations limit the kinds of language extensions that can be built in this way, the current prototypes show that nonetheless some quite interesting lan- |  |
| guage extensions are possible67. |  |
| *16.3.4 Embedding*  Embedding is not supported by Xtext. The reason is that, as we can see from Section 4.6.4, the adapter language would have to |  |

inherit from *two* base languages. However, Xtext only supports extending one base grammar.

We have shown above how to embed Xbase expressions into a custom DSL. However, as we have discussed, this is an example of extension with embedding flavor: we create a *new* DSL into which we embed the existing Xbase expressions. So we only have to extend from *one* base language – Xbase. An example of embedding would be to take an existing, independent SQL language and embed it into the **entity** DSL created above. This is not possible.

The same is true for the combination (in the same program) of several independently developed extensions to the same base language. In that case, too, the composite grammar would have to inherit from several base languages[[8]](#footnote-8).

#### 16.4 Spoofax Example

In this section we look at an example roughly similar to that for MPS and Xtext discussed in the previous sections. We start with Mobl’s data modeling language, which we have already seen in previous chapters.

To understand some of the discussions later, we first have to understand how Spoofax organizes languages. In Spoofax, language definitions are typically modularized (they declare their **module** at the top of a file). For example, Mobl’s syntax definition comes with a module for entities, which imports modules for statements and expressions:

|  |
| --- |
| **module** MoblEntities  **imports**  MoblStatements  MoblExpressions |

All syntax definition modules reside in the **syntax** directory of a Spoofax project. Typically, subdirectories are used to organize the modules of different sublanguages. For example, we can have subdirectories **entity** for Mobl’s entity definition language, **screen** for Mobl’s screen definition language, and **common** for definitions shared by both languages:

**module** entity/MoblEntities

**imports**

entities/MoblStatements entities/MoblExpressions

|  |
| --- |
| **module** screen/MoblScreens  **imports** common/Lexical |
| **module** common/MoblExpressions  **imports** common/Lexical |

As the example shows, the directory structure is reflected in module names. You can read them as relative paths from the **syntax** directory to the module.

Similarly to syntax definitions, rewrite rules for program analysis, editor services, program transformation, and code generation are organized in modules, which are imported from Mobl’s main module. The various modules for program analysis, editor services and program transformation are organized in subdirectories:

**module** mobl

**imports**

analysis/names analysis/types analysis/checks editor/complete editor/hover editor/refactor trans/desugar trans/normalize generate

##### 16.4.1 Referencing

We will illustrate references to elements written in another DSL with Mobl’s screen definition language. The following code uses Mobl sublanguage for data definition. It defines an entity

**Task** with some properties69.

|  |
| --- |
| **entity** Task {  name : String description : String  done : Bool date : DateTime  } |

The next piece of code shows a screen definition written in Mobl’s screen definition language. It defines a root screen for a list of tasks, using the **name** of a **Task** as a label for **list** elements.

|  |
| --- |
| **screen** root() { header("Tasks") group { list(t in Task.all()) {  item { label(t.name) }  }  }  } |

There are two references to the data model: **Task** refers to an **Entity**, and **name** refers to a property of that **Entity**. In general, a **Screen** defined in the UI model refers to **Entities** from Mobl’s entity language, and **Fields** in a screen refer to **Properties** in an entity.

*Structure* When referencing elements of another language, both languages typically share a definition of identifiers. For example, the screen definition language imports the same lexical module as does the data modeling language, via the expression module:

|  |
| --- |
| **module** entity/MoblEntities **imports**  ...  entity/MoblExpressions |
| **module** entity/MoblExpressions **imports**  ...  common/Lexical |
| **module** screen/MoblScreens **imports** common/Lexical **exports context**-**free syntax**  "list" ID "in" Collection "{" Item\* "}" -> List {"ScreenList"}  ID "." "all" "(" ")" -> Collection {"Collection"}  "item" "{" ItemPart\* "}" -> Item {"Item"}  "label" "(" ID "." ID ")" -> ItemPart {"LabelPart"} |

However, Spoofax also supports the use of different, typically overlapping identifier definitions70. In this case, the referenc-

ing language needs to import the identifier definition of the referenced language.

*Name Binding* Independent of the identifiers used in both languages, the reference has to be resolved. Definition sites are already defined by the referenced language. The corresponding references must be defined in the referencing language by using the namespaces from the referenced language. The previous syntax definition fragment of the screen definition language specifies lists and items in these lists. The following fragment shows the corresponding name binding specifications:

|  |
| --- |
| **module** screen/names **imports** entity/names **namespaces** Item **rules**  ScreenList(i, coll, i\*): **defines** Item i **of type** t **where** coll **has type** t  Collection(e): **refers to** Entity e  LabelPart(i, p): **refers to** Item i  **refers to** Property p **in** Entity e  **where** i **has type** e |

Here, the screen definition language declares its own namespace **Item** for items, which are declared in the list head, introducing a variable for the current item of a collection. For example, the screen definition we have seen earlier defines an item **t**[[9]](#footnote-9). When we describe the collection, we can refer to entities. The corresponding namespace **Entity** is defined in the data modeling language. The screen definition language uses the same namespace, to resolve the references into the referred language72.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | |  |
| *Type System* Similar to the name resolution, the type system of the referencing language needs to be defined with the knowledge of the type system of the referenced language. | | |  |
| constraint-error:  LabelPart(item, property) -> (property, "Label has to be a string.") **where** type := <index-type-of> property not (!type => !StringType()) |

To be able to check whether a label is a **string**, this constraint has to determine the type of the property used as a label.

*Generators* Generators of the referencing language also need to be defined with the knowledge of the generators of the referenced language. Typically, they just share knowledge about the naming and typing of generated code elements.

|  |
| --- |
| to-java:  LabelPart(item, property) ->  |[ new Label([java-item].[java-prop]);  ]| **where** label := <fresh-java-var-name> "label"; java-item := <to-java-var> item; java-prop := <to-java-getter> property |
| This rule generates Java code for a **LabelPart**. The generated code should create a new label with a text which should be determined from a **property** of an **item**. To generate the property access, the rule relies on the same scheme as rules from the generator of the entity definition part, by making calls to | | |  |
| rules from this generator73. | | |  |
| *16.4.2 Reuse*  As discussed in the previous sections, referencing concerns the case in which the referencing language is built with knowledge about the referenced language, so that it can have direct depen- | | |  |
| dencies74. In the case of reuse, such direct dependency is not | | |  |
| allowed: our goal is to combine two *independent* languages. | | |  |

*Structure* To illustrate this case, we again use the same example as in the MPS section. We first introduce a trivial DSL for defining relational table structures. These can optionally be mapped to a data source, but the language makes no assumption about what this data source looks like (and which language is used to define it). Consequently, the grammar has no dependency on any other one:

|  |
| --- |
| **module** DBTables  "database" ID Table\* -> Database {"DB"}  "table" ID Column\* -> Table {"DBTable"}  DataType ID ColumnMapper -> Column {"DBColumn"}  -> ColumnMapper {"DBMissingMapper"}  "char" -> DataType {"DBCharType"}  "number" -> DataType {"DBNumberType"} |

Again, the **Column** rule has an optional **ColumnMapper** which works as the hook for reuse. The reusable language only provides a rule for a missing column mapper75. In the next step, we want to be able to reference properties from Mobl’s data modeling language from a table definition:

|  |
| --- |
| **database** TaskDB **table** Tasks  **char** name <- Task.name  **char** description <- Task.description |

To do this, we define an adapter module, which imports the reusable table module and the data modeling language. So far, **ColumnMapper** is only an abstract concept, without a useful definition. The adapter module now defines a rule for **ColumnMapper**, which defines the concrete syntax of an actual mapper that can reference properties from the data modeling language:

|  |
| --- |
| **module** MoblDBAdapter  **imports**  DBTables  MoblEntities  **context**-**free syntax**  "<-" ID "." ID -> ColumnMapper {"PropertyMapper"} |

There is only one rule in this module, which defines a concrete mapper. On the right-hand side, it uses the same sort as the rule in the table module (**ColumnMapper**). On the left-hand side, it refers to a property (second ID) in an entity (first ID).

*NameBinding* The actual reference to entity and property names from the imported data modeling language needs to be specified in the name binding module of the adapter:

|  |
| --- |
| **module** adapter/names **imports** entity/names **rules**  PropertyMapper(e, p): **refers to** Entity e **refers to** Property p **in** Entity e |

*Type System* In our example, the types of the database language needs to be connected to the primitive types used in Mobl76. Constraints ensure we only map those fields to a par-

ticular column that are type-compatible:

|  |
| --- |
| **module** reuse/dbtables/analysis/types  **rules** constraint-error:  DBColumn(type, \_, mapper) -> (mapper, "Incompatible type") **where**  type’ := <type-of> mapper ;  <not(compatible-types)> (type, type’) compatible-types: \_ -> <fail> |

|  |
| --- |
| **rules** db-to-java: Column(t, n, mapper) ->  [Field([PRIVATE], t’, n),  Method([PROTECTED], BoolType, n’, params, stmts)] |

The code above is defined in the generic implementation of the database language. It assumes that a mapper has a type and checks if this type is compatible with the declared column type. It defines a default rule for type compatibility, which always fails. The connection to the type system of the entity language can now be made in an adapter module:

|  |
| --- |
| **module** analysis/types/adapter  **imports module** analysis/types  **module** reuse/dbtables/analysis/types  **rules** type-of: PropertyMapper(e, p) -> <index-type-of> p  compatible-types: (DBCharType(), StringType()) -> <id> compatible-types: (DBNumberType(), NumType()) -> <id> |
| The first rule defines the type of a **PropertyMapper** to be the type of the property. Then, two rules define type compatibility for Mobl’s **String** type with the **char** type in the table language, and Mobl’s **Num** type with table language’s **number** type. | | |  |
| *Generator* As in the Xtext example, we can use two strategies to reuse a generator for the database language. The first strategy relies on composition techniques of the target language, if that language provides such composition facilities77. | | |  |

The second strategy we discussed in the Xtext example addressed the generation of inlined code, which requires an extendable generator of the reusable language. With rewrite rules, this can be easily achieved in Spoofax. The reusable generator calls a dedicated rule for generating the inlined code, but defines only a failing implementation of this rule:

|  |
| --- |
| **where**  n’ := <to-fetch-method-name> n ; param := <to-fetch-method-parameters> mapper ; stmts := <to-fetch-statements(|n)> mapper  to-fetch-method-parameters: \_ -> <fail> to-fetch-statements(|n) : \_ -> <fail> |

This rule generates code for columns. It generates a private field and a protected method to fetch the content. This method needs a name, parameters and an implementation. We assume that the method name is provided by the generic generator. For the other parts (in particular, the implementation of the methods), the generic generator only provides failing placeholder rules. These have to be implemented in a concrete reuse setting by the adapter language generator:

|  |
| --- |
| **module** generate/java/adapter  **imports** generate/java reuse/table/generate  **rules**  to-fetch-method-parameters:  PropertyMapper(entity, property) -> [Param(type, "entity")] **where** type := <entity-to-java-type> entity  to-fetch-statements(|field-name):  PropertyMapper(entity, property) ->  [Assign(VarRef(field-name), MethodCall(VarRef("entity"), m, []),  Return(True())] **where** m := <property-to-getter-name> property |

This adapter code generates a single parameter for the fetch method. It is named **entity** and its type is provided by a rule from the entity language generator. The rule maps entity names to Java types. For the implementation body, the second rule generates an assignment and a return statement. The assignment calls the getter method for the property. Again, the name of this getter method is provided by the entity language generator.

##### 16.4.3 Extension

Because of Spoofax’ module system and rule-based nature, language extension feels like ordinary language development. When we want to add a new feature for a language, we simply create new modules for syntax, name binding, type system and code generation rules. These modules import the existing modules as needed. In the syntax definition, we can extend a syntactic sort with new definition rules. In the type system, we add additional **type-of** rules for the new language constructs and

define constraints for well-typedness. Finally, we add new generator rules, which can handle the new language constructs.

##### 16.4.4 Restriction

In the easiest case, restriction can be handled on the level of syntax rules. SDF’s import directives allow not only for renaming of sorts, but also for replacing complete syntax rules. To remove a rule completely, we can replace it with a dummy rule for a sort, which is not used anywhere. The following example restricts Mobl to a version without property access expressions78:

|  |  |  |  |
| --- | --- | --- | --- |
| *16.4.5 Embedding*  Embedding can be easily achieved in Spoofax. In general, the procedure is very similar to reuse. We will discuss the embed- | | |  |
| ding of HQL79 into Mobl as an example here80. | | |  |
| *Structure* Embedding requires an additional syntax definition module which imports the main modules of the host and guest language and defines additional syntax rules that realize the embedding. In target language embedding into Stratego, this was achieved with quotations and antiquotations. The following module is an initial attempt to embed HQL into Mobl: | | |  |
| **module** Mobl-HQL  **imports**  Mobl  Hql **context**-**free syntax**  QueryRule -> Exp {cons("HqlQuery")}  DeleteStatement ";" -> Statement {cons("HqlStatement")}  "~" Exp -> Expression {cons("DslExp")} |
| The module imports syntax definitions of host and guest languages. It embeds HQL queries as Mobl expressions and HQL’s delete statement as a Mobl statement without any quotations. Furthermore, it allows us to use quoted Mobl expressions inside HQL queries, using the tilde as a quotation symbol.  There are two issues in this module. First, we might accidentally merge sorts with the same name in host and guest lan- | | |  | |
| guage81. Since both languages are developed independently, | | |  | |
| we cannot assume mutually exclusive names in their syntax | | |  | |

definitions. One way to avoid name clashes is to rename sorts manually during import:

|  |
| --- |
| **module** Mobl-HQL  **imports**  Mobl  Hql [ QueryRule => HqlQueryRule  DeleteStatement => HqlDeleteStatement  Expression => HqlExpression  ...  ] |

This can be quite cumbersome, since we have to rename all sorts, not only the embedded ones. Alternatively, we can rely on Spoofax to generate a renamed version of a language definition. This *Mix* is a parameterized syntax definition, where

|  |
| --- |
| **module** HqLMix[Context]  **imports**  Hql [ QueryRule => QueryRule[[Context]]  DeleteStatement => DeleteStatement[[Context]]  Expression => Expression[[Context]] ...  ] |

Spoofax replaces each sort by a parameterized sort82:

The parameter allows us to distinguish sorts from the host and the target language. We can then import this module with an actual parameter and use the parameterized sorts in the embedding:

|  |
| --- |
| **module** Mobl-HQL **imports** Mobl HqlMix[HQL] **context**-**free syntax**  QueryRule[[HQL]] -> Exp {cons("HqlQuery")}  DeleteStatement[[HQL]] ";" -> Statement {cons("HqlStatement")}  "~" Exp -> Expression[[HQL]] {cons("MoblExp")} |
| The second issue is ambiguity: we have to integrate HQL queries into the precedence rules for Mobl expressions. To do this, we do not have to repeat all rules: preceding and succeeding rules | | |  |
| are sufficient83: | | |  |

|  |
| --- |
| **context**-**free priorities**  Assignment -> Exp  > QueryRule[[HQL]] -> Exp  > "if" "(" Exp ")" Exp "else" Exp -> Exp |

*Name Binding* The name bindings of host and embedded language are never connected. For example, only Mobl expressions can refer to Mobl variables. If an HQL query relies on a Mobl variable, it accesses it as an embedded Mobl expression. *Type System* The type system needs to connect the types from the host and guest languages. This can be achieved by adding typing rules for embedded and antiquoted constructs. For example, we need to connect the HQL type of a query to a Mobl type of the embedding expression:

|  |
| --- |
| **module** mobl-hql/types  **imports** mobl/types hql/types  type-of:  HqlQuery(query) -> mobl-type **where**  hql-type := <type-of> query mobl-type := <hql-to-mobl-type> hql-type  type-of:  MoblExp(exp) -> hql-type **where**  mobl-type := <type-of> exp hql-type := <mobl-to-hql-type> mobl-type  hql-to-mobl-type: JDBC\_Integer() -> NumType() hql-to-mobl-type: JDBC\_Float() -> NumType() hql-to-mobl-type: JDBC\_Bit() -> BoolType()  mobl-to-hql-type: NumType() -> JDBC\_Float() mobl-to-hql-type: BoolType() -> JDBC\_Bit() |
| The first rule determines the type of an embedded HQL query and maps it to a corresponding Mobl type. The second rule determines the type of an antiquoted Mobl expression and maps it to an corresponding HQL type. The remaining rules exem- |  |
| plify actual mappings between HQL and Mobl types84. |  |
| *Generator* There are two strategies for code generation for embedded languages. If the guest language provides a suitable code generator, we can combine it with the code generator of the host language. First, we need rules which generate code for embedded constructs. These rules have to extend the host generator by delegating to the guest generator. Next, we need rules which generate code for antiquoted constructs. These rules have to extend the guest generator by delegating to the host generator.  Another strategy is to define a model-to-model transformation which desugars (or "assimilates") embedded constructs to constructs of the host language. This transformation is then applied first, before the host generator is applied to generate code. The embedding of a target language into Stratego is an example of this approach. The embedded target language will be represented by abstract syntax trees for code generation fragments. These trees need to be desugared into Strat- |  |

ego pattern matching constructs. For example, the embedded

**|[return |[x]|; ]|** will yield the following abstract syntax tree:

|  |
| --- |
| ToJava(  Return(  FromJava(  Var("x")  )  )  ) |

In ordinary Stratego without an embedded target language, we would have written the pattern **Return(x)** instead. The corresponding abstract syntax tree looks like this:

|  |
| --- |
| NoAnnoList(  App(  Op("Result"),  [Var("x")]  )  ) |

The desugar transformation now needs to transform the first abstract syntax tree into the second one:

|  |
| --- |
| desugar-all: x -> <bottomup(try(desugar-embedded))> x desugar-embedded: ToJava(e) -> <ast-to-pattern> e  ast-to-pattern: ast -> pattern **where**  **if** !ast => FromJava(e) **then**  pattern := e  **else**  c := <constructor> ast ; args := <arguments> ast ; ps := <map(ast-to-pattern)> args ; pattern := NoAnnoList(App(Op(c), ps)) |

The first rule drives the desugaring of the overall tree. It tries to apply **desugar-embedded** in a bottom-up traversal. The only rule for desugaring embedded target language code matches the embedded code and applies **ast-to-pattern** to it. If this is applied to an antiquote, the contained subnode is already a regular Stratego pattern. Otherwise, the node has to be an abstract syntax tree of the target language. It is deconstructed into its constructor and subtrees, which are desugared into patterns as well. The resulting patterns and the constructor are then used to construct the overall pattern.

1. [↑](#footnote-ref-1)
2. [↑](#footnote-ref-2)
3. d [↑](#footnote-ref-3)
4. [↑](#footnote-ref-4)
5. . [↑](#footnote-ref-5)
6. . [↑](#footnote-ref-6)
7. . [↑](#footnote-ref-7)
8. . [↑](#footnote-ref-8)
9. . [↑](#footnote-ref-9)