Delft University of Technology
Software Engineering Research Group
Technical Report Series

Domain-Specific Language Engineering

A Case Study in Agile DSL Development (Mark I)

Eelco Visser

Report TUD-SERG-2007-017

TUD-SERG-2007-017

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Domain-Specific Language Engineering

A Case Study in Agile DSL Development

Eelco Visser

Software Engineering Research Group
Delft University of Technology
visser@acm.org

Abstract. The goal of domain-specific languages (DSLs) is to increase the productivity of software engineers by abstracting from low-level boilerplate code. Introduction of DSLs in the software development process requires a smooth workflow for the production of DSLs themselves. This tutorial gives an overview of all aspects of DSL engineering: domain analysis, language design, syntax definition, code generation, deployment, and evolution, discussing research challenges on the way. The concepts are illustrated with DSLs for web applications built using several DSLs for DSL engineering: SDF for syntax definition, Stratego/XT for code generation, and Nix for software deployment.

1 Introduction

In recent years there has been an increasing momentum (some call it hype) for approaches with names as domain-specific languages, model-driven architecture, software factories, language workbenches, and intentional programming. While there are differences between these approaches (mostly of a technological nature?), the common goal is to achieve a higher-level of abstraction in software development by abstracting from low-level boilerplate code. (Making *domain-specific languages* the approach of my choice, I'll use its terminology from now on.) The idea of domain-specific languages has been around for a long time, but what seems to be new in the current wave, is the requirement to use DSL design and implementation as a standard tool in the software development process. The challenge then is to develop a systematic method for designing new domain-specific languages.

This tutorial describes an experiment in DSL design and implementation. The experiment is simply to take a new domain (web applications), to develop a DSL (set of DSLs) for this domain, and observe the process to extract ingredients for a standard process. The target of the experiment are web applications with a rich domain model that can serve as content management system editable via the browser, but also allow querying and aggretation based on the structure of the data. The tutorial takes one particular combination of technologies. The DSL will be a textual language. The generator targets Java with a certain collection of frameworks for implementation of web applications. The DSL is implemented using Stratego/XT, SDF, and Nix.

Contributions The contributions of this tutorial are

- an experience report
- an introduction to a particular (meta) technology (Stratego/XT) from an application perspective
- guidelines; in particular, an emerging method for developing DSLs in a more systematic way

So this tutorial is not:

- a comparison of techniques and tools for DSL definition
- a comparison of visual vs textual languages
- a comparison of web programming languages

and therefore, does not claim any contributions in those areas.

1.1 Outline

This tutorial discusses the DSL engineering process by following the development of an actual DSL and discussing the considerations that played a role during the process. Thus, we follow the iterations in the design and implementation of the language. The techniques used for the implementation are introduced as we go.

2 Capturing Programming Patterns

The first step in the process of designing a DSL is to consider common programming patterns in the application domain. We will turn these patterns into templates, i.e. program fragments with holes. The holes in these templates can be filled with values to realize different instantiations of the programming pattern. Since the configuration data needed to fill the holes is typically an order of magnitude smaller than the programming patterns they denote, a radical decrease in programming effort is obtained. That is, when exactly these patterns are needed, of course. With some thought the configuration data can be turned into a proper domain-specific language. Instead of doing a 'big design up front' to consider all aspects a DSL for web applications should cover and the language constructs we would need for that, we develop the DSL in iterations. We start with relatively large patterns, i.e., complete classes.

As argued before, we take a particular technology stack as basis for our WebDSL. That is, this technology stack will be the platform on which code generated from DSL models will run. That way we have a concrete implementation platform when considering design and implementation issues and it provides a concrete code base to consider when searching for programming patterns. Hopefully, we will arrive at a design with abstractions that transcend this particular technology.

In this work we are using the EJB3/Seam architecture for web applications. That is, applications consist of three layers or tiers. The presentation layer is concerned with producing webpages and interpreting events generated by the user. For this layer we are using JavaServer Faces (JSF). The persistence layer is concerned with storing data in the database and retrieval of data from the database. This layer really consists of two parts. The database proper is a separate service implemented by a relational database (I have been using MySQL, but that is not really relevant). In the implementation of a web application, however, we approach the database via an object-relational mapping (ORM) framework, which takes care of the communication with the database and translates relational data into objects that can be used naturally in an object-oriented setting. Thus, after defining a proper mapping between objects and database tables, we need no longer worry about the database side. Finally, to connect the JSF pages defining the user-interface with the objects obtained from the database we use EJB3/Seam session beans [8, 10].

While it used to be customary for these types of frameworks to require a large portion of an application to be implemented in XML configuration files, this trend has been reversed in the EJB3/Seam architecture. Most of the configuration is now expressed as annotations in Java classes building on the concept of dependency injection. A little XML configuration remains, for instance, to define where the database is to be found and such. This configuration is mostly static and will not be a concern in this paper.

In this section, we start with considering the entity classes used with the Java Persistence API (JPA) [11] or Hibernate [2], and how to build a generator for such classes, from syntax definition, and abstract syntax to rewrite rules and

strategies. As such this section serves as an introduction to these techniques. In the next section we then consider the generation of basic web pages for view and editing the content of persisted objects.

2.1 Programming Patterns for Persistence

The Java Persistence API (JPA) is a standard proposed by Sun for object-relational mapping (ORM) for Java. The API is independent of vendor-specific ORM frameworks such as Hibernate; these frameworks are expected to implement JPA, which, Hibernate 3 indeed does [2]. While earlier versions of Hibernate used XML configuration files to define the mapping between database schemas and Java classes, the JPA approach is to express these mappings using Java 5 annotations in Java classes. Objects to be persisted in a database are represented using 'plain old Java objects (POJOs)'. Roughly, classes are mapped to database tables and properties (fields with getters and setters) are mapped to database columns. We will now inspect the ingredients of such classes as candidates for code generation.

Entity Class An *entity class* is a Java class annotated with the `@Entity` annotation and with an empty constructor, which guarantees that the persistence framework can always create new objects.

```
@Entity
public class Publication {
  public Publication () { }
  // properties
}
```

An entity class is mapped to a database table with the same name. (If desired an alternative name for the table can be specified, but we won't be concerned with that. In general, we will not necessarily try to cover all the variability in the target technology, but only use that what is necessary.)

Identity Entities should have an *identity* as primary key. This identity can be any value that is a unique property of the object. The annotation `@Id` is used to indicate the property that represents the identity. However, the advice is to use an identity that is not directly linked to the logic of the object, but rather to use a synthetic idenity, for which the database can generate unique values. This then takes the following pattern:

```
@Id @GeneratedValue
private Long id;

public Long getId() {
  return id;
}
private void setId(Long id) {
  this.id = id;
}
```

Properties The values of an object are represented by *properties*, class member fields with getters and setters. Such properties are mapped to columns in the database table for the enclosing class.

```
private String title;

public String getTitle() {
  return title;
}
public void setTitle(String title) {
  this.title = title;
}
```

Entity Associations No annotations are needed for properties with simple types. However, properties referring to other entities, or to collections of entities, require annotations. The following property defines an association to another entity:

```
@ManyToOne
@JoinColumn(name = "PublicationAuthor")
@Cascade({CascadeType.PERSIST,CascadeType.SAVE_UPDATE,CascadeType.MERGE})
private Person author = new Person();

public Person getAuthor() {
  return author;
}
public void setAuthor(Person author) {
  this.author = author;
}
```

The `@ManyToOne` annotation states that a `Person` may be used in *many* publications as author. The `@JoinColumn` annotation defines an explicit name for the auxiliary table that is used to implement the relation between `Publication` and `Author`, by storing the primary keys (identities) of the objects involved. The `@Cascade` annotation defines which persistence operations should be propagated along the association. In this case save operations should be propagated, but deletion should not; if a publication is deleted, its author should not be deleted since it may be used in other publications.

2.2 A Domain Model DSL

Entity classes with JPA annotations are conceptually simple enough. However, there is quite a bit of boilerplate involved. First of all, the setters and getters are completely redundant, and also the annotations can be become fairly complex. However, the essence of an entity class is simple, i.e., a class name, and a list of properties, i.e., (name, type) pairs. This information can be easily defined in a structure of the form `A{ prop* }` with `A` a name (identifier) and `prop*` a list of properties of the form `x : t`, i.e., a pair of an field name `x` and a type `t`. For example, the following entity declarations

```
Publication {                  Person {
  title    : String            fullname : String
  author   : Person            email    : String
  year     : Int               homepage : String
  abstract : String            }
  pdf      : String
}
```

define the entities `Publication` and `Person`, which in Java take up easily 100 lines of code.

2.3 Building a Generator

In the rest of this section we will examine how to build a generator for the simple domain modeling language sketched above. A generator typically consists of three main parts, a parser, reads in the model, the code generator proper, which transforms an abstract syntax representation of the model to a representation of the target program, and a pretty-printer, which formats the target program and writes it to a text file. Thus, we'll need the following ingredients: a definition of the concrete syntax of the DSL, for which we'll use the syntax definition formalism SDF2; a parser that reads model files and produces an abstract representation; a definition of that abstract representation; a transformation to the abstract representation of the Java program to be generated, for which we'll use term rewrite rules; and finally, a definition of a pretty-printer.

2.4 Syntax Definition

For syntax definition we use the syntax definition formalism SDF2 [13]. SDF2 integrates the definition of the lexical and context-free syntax. Furthermore, it is a modular formalism, which makes it easy to divide a language definition into reusable modules, but more importantly, it makes it possible to *combine* definitions for different languages. This is the basis for rewriting with concrete syntax and language embedding; we'll see examples of this later on.

The syntax of the basic domain modeling language sketched above is defined by the following module `DomainModel`. The module defines the lexical syntax of identifiers (`Id`), integer constants (`Int`), string constants (`String`)[1], whitespace and comments (`LAYOUT`). Next the context-free syntax of models, entities, properties, and sorts is defined. Note that in SDF productions have the non-terminal being defined on the right of the `->` and the body on the left-hand side.

```
module DomainModel
exports
  sorts Id Int String Definition Entity Property Sort
  lexical syntax
    [a-zA-Z][a-zA-Z0-9\_]* -> Id
```

[1] Integer and string constants are not used in this version of the language.

```
    [0-9]+                -> Int
    "\"" ~[\"\n]* "\""    -> String
    [\ \t\n\r]            -> LAYOUT
    "//" ~[\n\r]* [\n\r]  -> LAYOUT
  context-free syntax
    Definition*          -> Model      {cons("Model")}
    Entity               -> Definition
    Id "{" Property* "}" -> Entity     {cons("Entity")}
    Id ":" Sort          -> Property   {cons("Property")}
    Id                   -> Sort       {cons("SimpleSort")}
```

Abstract Syntax An SDF syntax definition defines the concrete syntax of strings in a language. For transformations we want an abstract representation, i.e. the tree structure underlying the grammar. This structure can be expressed concisely by means of an algebraic signature, which defines the constructors of abstract syntax trees. Such a signature can be derived automatically from a syntax definition (using `sdf2rtg` and `rtg2sig`). Each context-free production gives rise to a constructor definition using the name declared in the `cons` attribute of the production as constructor name, and the non-literal sorts as input arguments. Thus, for the `DomainModel` language defined above, the abstract syntax definition is the following:

```
signature
  constructors
    Model      : List(Definition) -> Model
               : Entity -> Definition
    Entity     : Id * List(Property) -> Entity
    Property   : Id * Sort -> Property
    SimpleSort : Id -> Sort
               : String -> Id
```

Parsing A parser reads a text representation of a model, checks it against the syntax definition of the language, and builds an abstract syntax representation of the underlying structure of the model text. Parse tables for driving the `sglr` parser can be generated automatically from a syntax definition (using `sdf2table`). The `sglr` parser produces an abstract syntax representation in the Annotated Term (ATerm) Format [3], as illustrated by the following parse of a domain model:

```
Person {
  fullname : String
  email    : String
  homepage : String
}
```

⇒

```
Entity("Person",
  [ Property("fullname", SimpleSort("String"))
  , Property("email",    SimpleSort("String"))
  , Property("homepage", SimpleSort("String"))
  ]
)
```

The main ATerm form is $c(t_1,\ldots,t_n)$, where c is a constructor name and the t_i are ATerms. Other forms are lists of terms ($[t_1,\ldots,t_n]$), strings (`"..."`), and integer constants.

2.5 Code Generation by Rewriting

Programs in the target language can also be represented as terms. For example, the following parse shows the abstract representation of the basic form of an entity class (as produced by the `parse-java` tool):

```
@Entity
public class Publication {
  public Publication () { }
}
```

⇓

```
ClassDec(
  ClassDecHead(
    [MarkerAnno(TypeName(Id("Entity"))), Public()]
    , Id("Publication")
    , None(), None(), None()),
  ClassBody(
    [ConstrDec(
       ConstrDecHead([Public()],None(),Id("Publication"),[],None()),
       ConstrBody(None(), []))
    ])
)
```

This entails that code generation can be expressed as a term-to-term transformation. Pretty-printing of the resulting term then produces the program text. The advantage over the direct generation of text is that (a) the structure can be checked for syntactic and type consistency, (b) a pretty-printer can ensure a consistent layout of the generated program text, and (c) further transformations can be applied to the generated code. For example, in the next section we'll see that an interface can be derived from the generated code of a class.

Term rewriting is a formalism for describing term transformations [1]. A *rewrite rule* p_1 `->` p_2 defines that a term matching the term pattern p_1 can be replaced with an instantiation of the term pattern p_2. A term pattern is a term with *variables*. In standard term rewriting, rewrite rules are applied exhaustively until a normal form is obtained. Term rewriting engines employ a built-in *rewriting strategy* to determine the order in which subterms are rewritten.

Stratego [15, 5] is a transformation language based on term rewriting. Rewrite rules are *named* and can be *conditional*, i.e., of the form l `:` p_1 `->` p_2 `where` s, with l the name and s the condition. Stratego extends basic term rewriting by providing *programmable rewriting strategies* that allow the developer to determine the order in which rules are applied.

The following Stratego rewrite rule defines the transformation of an `Entity` term in the domain model language to the basic Java class pattern that we saw above. Note that the rule generalizes over the particular class by using instead of the name `"Publication"`, a *variable* `x` for the class and the constructor. Thus, the rule generates for an arbitrary `Entity x`, a class `x`.

```
entity-to-class :
  Entity(x, prop*) ->
  ClassDec(
    ClassDecHead(
      [MarkerAnno(TypeName(Id("Entity"))), Public()]
      , Id(x)
      , None(), None(), None()),
    ClassBody(
      [ConstrDec(
         ConstrDecHead([Public()],None(),Id(x),[],None()),
         ConstrBody(None(), []))
      ])
  )
```

Concrete Syntax The `entity-to-class` rewrite rule defines a template for code generation. However, the term notation, despite its advantages for code generation as noted above, is not quite as easy to read as the corresponding program text. Therefore, Stratego supports the definition of rewrite rules using the *concrete syntax* of the subject language [14]. For example, the following rule is the concrete syntax equivalent of the rule above:

```
entity-to-class :
  Entity(x_Class, prop*) ->
  |[
     @Entity
     public class x_Class {
       public x_Class () { }
     }
  ]|
```

Note that the identifier `x_Class` is recognized by Stratego (the Stratego parser in fact) as a *meta-variable*, i.e. a pattern variable in the rule.

While rewrite rules using concrete syntax have the readability of textual templates, they have all the properties of term rewrite rules. The code fragment is parsed using the proper syntax definition for the language concerned (Java in this case) and thus syntax errors in the fragment are noticed at compile-time (of the generator). The transformation produces a term and not text; in fact, the rule is equivalent to the rule using a term above. And thus the advantages of term rewriting discussed above hold also for rewriting with concrete syntax.

2.6 Pretty-printing

Pretty-printing is the inverse of parsing, i.e. the conversion of an abstract syntax tree (in term representation) to a, hopefully readable, program text. While this can be done with any programmatic method that prints strings, it is useful to abstract from the details of formatting program texts by employing a specialized library. The GPP library [6] supports formatting through the *Box language*, which provides constructs for positioning text blocks. For pretty-printing Java

and XML the Stratego/XT toolbox provides custom built mappings to Box. For producing a pretty-printer for a new DSL that is still under development it is most convenient to use a pretty-printer *generator* (`ppgen`), which produces a pretty-print *table* with mappings from abstract syntax tree constructors to Box expressions. The following is a pretty-print table for our `DomainModel` language:

```
[
  Entity             -- V[V is=2[ H[_1 KW["{"]] _2] KW["}"]],
  Entity.2:iter-star -- _1,
  Property           -- H[_1 KW[":"] _2],
  SimpleSort         -- _1
]
```

Here `V` stands for *vertical* composition, `H` stands for *horizontal* composition, and `KW` stands for *keyword*. While a pretty-printer generator can produce a *correct* pretty-printer (such that parse(pp(parse(prog))) = parse(prog)), it is not possible to automatically generate pretty-printers that generate a *pretty* result (although heuristics may help). So it is usually necessary to tune the pretty print rules.

2.7 Generating Entity Classes

Now that we've seen the techniques to build the components of a generator we can concentrate on the rules for implementing the `DomainModel` language. Basically, the idea is to take the program patterns that we found during the analysis of the solution domain, and turn them into transformation rules, by abstracting out the application-specific identifiers. Thus, an entity declaration is mapped to an entity class as follows:

```
entity-to-class :
  Entity(x_Class, prop*) ->
  |[
     @Entity public class x_Class {
       public x_Class () { }

       @Id @GeneratedValue private Long id;
       public Long getId() { return id; }
       private void setId(Long id) { this.id = id; }

       ~*cbd*
     }
  ]|
  where cbd* := <mapconcat(property-to-gettersetter(|x_Class))> prop*
```

Since an entity class always has an identity (at least for now), we include it directly in the generated class. Furthermore, we include, through the antiquotation `~*`, a list of class body declarations `cbd*`, which are obtained by mapping the properties of the entity declaration. (Here `mapconcat` is a strategy that applies its argument strategy to each element of a list, concatenating the lists resulting from each application.)

Value types The mapping for value type properties simply produces a private field with a public getter and setter. This requires a bit of *name mangling*, i.e. from the name of the property, the names of the getter and setter are derived, here using the `property-getter` and `property-setter` strategies.

```
property-to-member-decs(|x_Class) :
  Property(x_prop, s) -> class-body-dec*|[
    private t x_prop;
    public t x_get() { return title; }
    public void x_set(t x) { this.x = x; }
  ]|
  where t := <builtin-java-type> s
      ; x_get := <property-getter> x_prop
      ; x_set := <property-setter> x_prop
```

The fact that the property is for a value type is determined using the strategy `builtin-java-type`, which defines mapping for the built-in types of the `DomainModel` language to types in Java that implement them. For example, for example, the `String` type is defined as follows:

```
builtin-java-type :
  SimpleSort("String") -> type|[ java.lang.String ]|
```

Reference types Properties with a referency to another type are translated to a private field with getters and setters with the `@ManyToOne` and corresponding annotations as discussed before:

```
property-to-member-decs(|x_Class) :
  Property(x_prop, s) -> class-body-dec* |[
    @ManyToOne
    @Cascade({
      CascadeType.PERSIST,CascadeType.SAVE_UPDATE,CascadeType.MERGE
    })
    private t x_prop;
    public t x_get() { return x_prop; }
    public void x_set(t x_prop) { this.x_prop = x_prop; }
  ]|
  where t      := <defined-java-type> s
      ; x_Prop := <capitalize-string> x_prop
      ; x_get  := <property-getter> x_prop
      ; x_set  := <property-setter> x_prop
      ; columnname := <concat-strings>[x_Class, x_Prop]
```

Propagating declared entities The previous rule decides that the property is an association to a reference type using the strategy `defined-java-type`, which maps entities declared in the domain model to the Java types that implement them. Since the collection of these entity types depends on the domain model, the `defined-java-type` mapping is defined *at run-time* during the transformation as a *dynamic rewrite rule* [5]. That is, before generating code for the

entity declarations, the following `declare-entity` strategy is applied to each declaration:

```
declare-entity =
  ?Entity(x_Class, prop*)
  ; rules(
      defined-java-type :
        SimpleSort(x_Class) -> type|[ x_Class ]|
    )
```

This strategy first matches (`?`p with p a term pattern) an entity declaration and then defines a rule `defined-java-type`, which inherits from the match the binding to the variable `x_Class`. Thus, for each declared entity a corresponding mapping is defined. As a result, the `property-to-member-decs` rule fails if it is applied to a property with an association to a non-existing type (and an error message might be generated to notify the user). In general, dynamic rewrite rules are used to add new rewrite rules at run-time to the transformation system. A dynamic rule inherits variable bindings from its definition context, which is typically used to propagate context-sensitive information.

2.8 Composing a Code Generator

Using the ingredients discussed above, the basic version of the code generator for WebDSL is defined as the following Stratego strategy:

```
webdsl-generator =
  xtc-io-wrap(webdsl-options,
    parse-webdsl
    ; alltd(declare-entity)
    ; collect(entity-to-class)
    ; output-generated-files
  )
```

The strategy invokes `xtc-io-wrap`, a library strategy for handling command-line options to control input, output, and other aspects of a transformation tool. The argument of `xtc-io-wrap` is a sequence of strategy applications (s_1`;`s_2 is the sequential composition of two strategies). `parse-webdsl` parses the input model using a parse table generated from the syntax definition, producing its abstract syntax representation. The `alltd` strategy is a generic traversal, which is used here to find all entity declarations and generate the `defined-java-type` mapping for each. The generic `collect` strategy is then used to create a set of Java entity classes, one for each entity declaration. Finally, the `output-generated-files` strategy uses a Java pretty-printer to map a class to a program text and write it to a file with the name of the class and put it in a directory corresponding to the package of the class.

3 Capturing More Programming Patterns: CRUD Pages

The next step towards full fledged web applications is to create pages for viewing and editing objects in our `DomainModel` language. That is, from a domain model we will generate a basic userinterface for creating, retrieving, updating and deleting (CRUD) objects. Well, we'll start with viewing and editing. For example, consider the following domain model of `Person`s with `Address`es, and `User`s.

```
Person {                Address {           User {
  fullname  : String      street : String     username : String
  email     : String      city   : String     password : String
  homepage  : String      phone  : String     person   : Person
  photo     : String    }                   }
  address   : Address
  user      : User
}
```

For such a domain model we want to generate view and edit pages as displayed in Figures 1 and 2. Implementing this simple user interface requires an understanding of the target architecture. Figure 3 sketches the architecture of a JSF/Seam application for the `editPerson` page in Figure 2. The `/editPerson.seam` client view of the page on the far left of Figure 3 is a plain web page implemented in HTML, possibly with some JavaScript code for effects and cascading stylesheets for styling. The rendered version of this code is what we see in Figure 2. The HTML is *rendered* on the server side from the JavaServer Faces (JSF) component model [9] defined in the `editPerson.xhtml` file. In addition to regular HTML layout elements, the JSF model has components that interact with a *session bean*. The `EditPersonBean` session bean retrieves data for the JSF model from the database (and from session and other contexts). For this purpose the session bean obtains an `EntityManager` object through which it approaches the

Fig. 1. viewPerson page

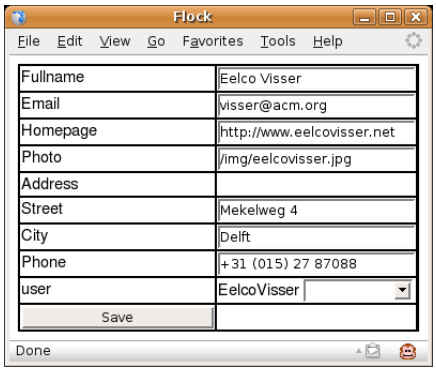

Fig. 2. editPerson page

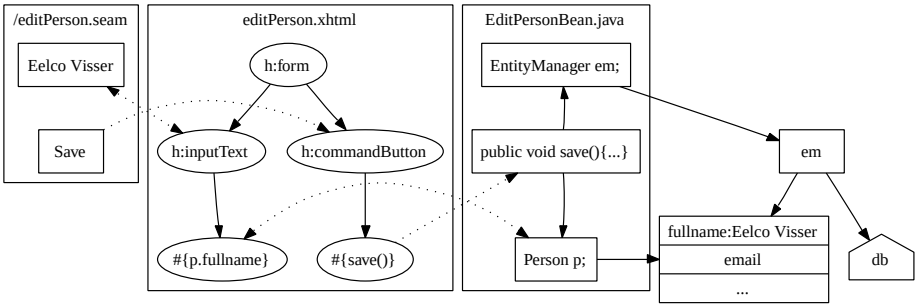

Fig. 3. Sketch of JSF/Seam architecture.

database, with which it synchronizes objects such as `Person p`. When the input field at the client side gets a new value and the form is submitted by a push of the `Save` button, the value of the input field is assigned to the field pointed at by the expression of the `h:inputText` component (by calling the corresponding setter method). Subsequently, the `save()` action method of the session bean, which is specified in the `action` attribute of the `h:commandButton` corresponding to the `Save` button, is called. This method, then invokes the entity manager to update the database.

Thus, to implement a CRUD interface for domain objects we must generate for each page a JSF XHTML document that defines the layout of the userinterface and the data used in its elements, and a Seam session bean that manages the objects referred to in the JSF document.

3.1 Generating JSF Pages

Figure 4 illustrates the structure of the JSF XHTML document for the edit page in Figure 2. Besides common HTML tags, the document uses JSF components such as `h:form`, `h:outputText`, `h:inputText`, and `h:commandButton`. Such a document can again be generated using rewrite rules transforming entity declarations to XHTML documents.

```
entity-to-edit-page :
  Entity(x_Class, prop*) ->
  %><html ...> ... <body><h:form><table>
      <%= rows ::* %>
      <tr><td>
        <h:commandButton value="Save" action="#{<%=editX%>.save()}"/>
      </td><td></td></tr>
    </table></h:form><%
  where editX := <concat-strings>["edit", x_Class]
      ; rows := <map(row-in-edit-form(|editX))> props
```

This rule generates the overall setup of an edit page from an entity declaration. Just as was the case with generation of Java code, this rule uses the concrete

```
<html ...> ... <body>
<h:form>
  <table>
  <tr><td> <h:outputText value="Fullname"/> </td>
      <td> <h:inputText value="#{editPerson.person.fullname}"/>
      </td> </tr>
  <tr><td><h:commandButton value="Save" action="#{editPerson.save()}"/>
      </td> <td></td></tr>
  </table>
</h:form>
</body> </html>
```

Fig. 4. editPage.xhtml with JSF components.

syntax of XML in the right-hand side of the rule [4]. (The quotation marks `%>` and `<%` were inspired by template engines such as JSP [12]). The XML fragment is syntactically checked at compile-time of the generator and the rule then uses the underlying abstract representation of the fragment. (Note that the ellipses `...` are not part of the formal syntax, but just indicate that some elements were left out to save space.)

The `entity-to-edit-page` rule calls `row-in-edit-form` to generate for each property a row in the table.

```
row-in-edit-form(|editX)  :
  prop@Property(x_prop, _, _, _) ->
  %>
    <tr><td><h:outputText value="<%=x_prop%>"/></td>
        <td><%= input %></td></tr>
  <%
  where input := <property-to-edit-component(|editX)> prop
```

The left column in the table contains the name of the property, and right column an appropriate input component , which is generated by the `property-to-edit-component` rule. In the case of the `String` type a simple `inputText` component is generated.

```
property-to-edit-component(|editX) :
  Property(x_prop, SimpleSort("String")) ->
  %>
    <h:inputText value="#{<%=editX%>.<%=x_prop%>}"/>
  <%
```

Other types may require more complex JSF configurations. For instance, an entity association (such as the `user` property of `Person`) requires a way to enter references to existing entities. The page in Figure 2 uses a drop-down selection menu for this purpose.

The generation of a view page is largely similar to the generation of an edit page, but instead of generating an `inputText` component, an `outputText` component is generated:

```
property-to-view-component(|editX) :
  Property(x_prop, SimpleSort("String")) ->
  %>
    <h:outputText value="#{<%=editX%>.<%=x_prop%>}"/>
  <%
```

3.2 Seam Session Beans

As explained above, the JSF components get the data to display from an EJB session bean. The Seam framework provides an infrastructure for implementing session beans such that the connections to the environment, such as the application logger and the entity manager, are made automatically via *dependency injection* [7]. To get an idea, here is the session bean class for the `editPerson` page:

```
@Stateful
@Name("editPerson")
public class EditPersonBean implements EditPersonBeanInterface{
  @Logger private Log log;

  @PersistenceContext(type = EXTENDED) private EntityManager em;

  @In private FacesMessages facesMessages;

  @Destroy @Remove public void destroy() { }

  // specific fields and methods
}
```

EJB3 and Seam use Java 5 annotations to provide application configuration information within Java classes, instead of the more traditional XML configuration files. The use of annotations is also an alternative to implementing interfaces; instead of having to implement a number of methods with a fixed name, fields and methods can be named as is appropriate for the application, and declared to play a certain role using annotations.

The `@Stateful` annotation indicates that this is a stateful session bean, which means that it can keep state between requests. The `@Name` annotation specifies the Seam *component* name. This is the prefix to object and method references from JSF documents that we saw in Figure 4. Seam scans class files at deployment time to link component names to implementing classes, such that it can create the appropriate objects when these components are referenced from a JSF instance. The `destroy` method is indicated as the method to be invoked when the session bean is `@Remove`d or `@Destroy`ed.

The fields `log`, `em`, and `facesMessages` are annotated for *dependency injection* [7]. That is, instead of creating the references for these objects using a factory, the application context finds these fields based on their annotations and injects an object implementing the expected interface. In particular, `log` and

`facesMessages` are services for sending messages, for system logging, and user messages respectively. The `em` field expects a reference to an `EntityManager`, which is the JPA database connection service.

All the above was mostly boilerplate that can be found in any session bean class. The real meat of a session bean is in the fields in methods specific for the JSF page (or pages) it supports. In the CRUD scenarios we are currently considering, a view or edit page has a property for the object under consideration. That is, in the case of the `editPerson` page, it has a property of type `Person`:

```
private Person person;
public void setPerson(Person person) { this.person = person;}
public Person getPerson() { return person; }
```

Next, a page is called with URL `/editPerson.seam?person=`x, where x is the identity of the object being edited. The problem of looking up the value of the `person` parameter in the request object, is also solved by dependency injection in Seam. That is, the following field definition

```
@RequestParameter("person") private Long personId;
```

declares that the value of the the `@RequestParameter` with the name `person` should be bound to the field `personId`, where the string value of the parameter is automatically converted to a `Long`.

To access the object corresponding to the identity passed in as parameter, the following `initialize` method is defined:

```
@Create
public void initialize() {
  if (personId == null) {
    person = new Person();
  } else {
    person = em.find(Person.class, personId);
  }
}
```

The method is annoted with `@Create` to indicate that it should be called upon creation of the bean (and thus the page). The method uses the entity manager `em` to find the object with the given identity. The case that the request parameter is `null` occurs when no idenity is passed to the requrest. Handling this case enables, supports the creation of new objects.

Finally, a push of the `Save` button leads to a call to the following `save()` method, which invokes the entity manager to save the changes to the object to the database:

```
public String save()
{
  em.persist(this.getPerson());
  return "/viewPerson.seam?person=" + person.getId();
}
```

The return value of the method is used to determine the page flow after saving, which is in this case to go to the view page for the object just saved.

3.3 Generating Session Beans

Generating the session beans for view and edit pages comes down to taking the boilerplate code we saw above and generalizing it by taking out the names related to entity under consideration and replacing it with holes. Thus, following rule sketches the structure of such a generator rule:

```
entity-to-session-bean :
  Entity(x_Class, prop*) -> |[
    @Stateful @Name("~viewX")
    public class x_ViewBean implements x_ViewBeanInterface {
      ...
      @Destroy @Remove public void destroy() { }
    }
  ]|
  where viewX := ...; x_ViewBean := ...; x_ViewBeanInterface := ...
```

Such rules are very similar to the generation rules we saw in Section 2.

3.4 Deriving Interfaces

A stateful session bean should implement an interface declaring all the methods that should be callable from JSF pages. Instead having a separate (set of) rule(s) that generates the interface from an entity, such an interface can be generated automatically from the bean class. This is one of the advantages of generating structured code instead of text. The following rules and strategy define a transformation that turns a Java class into an interface with all the public methods of the class.

```
create-local-interface(|x_Interface) :
  class -> |[ @Local public interface x_Interface { ~*methodsdecs } ]|
  where methodsdecs := <extract-method-signatures> class

extract-method-signatures =
  collect(method-dec-to-abstract-method-dec)

method-dec-to-abstract-method-dec :
  MethodDecHead(mods, x , t, x_method, args, y) ->
  AbstractMethodDec(mods, x, t, x_method, args, y)
  where <fetch(?Public())> mods
```

4 Refining Programming Patterns

The domain DSL with CRUD generator that we set up in the previous two sections allows us to generate the data layer of a web application and a simple user interface for viewing and editing objects. Even for functionality as straightforward as CRUD operations, the flavour that we introduced so far is very simplistic. Before we consider how to be able to define the user interface more flexibly, we consider first a couple of refinements to the domain modeling language and the view/edit page generator.

4.1 Strings in Many Flavours

The association types that we saw in the previous sections were either `String`s or references to other defined entities. While strings are useful for storing many (if not most) values in typical applications, the type name does not provide us with much information about the nature of those data. By introducing application-domain specific value types we can generate a lot of functionality 'for free'. For example, the following domain models for `Person` and `User` still use mostly string valued data, but using alias types:

```
Person {                  User {
  fullname  : String        username : String
  email     : Email         password : Secret
  homepage  : URL           person   : Person
  photo     : Image       }
  address   : Address
  user      : User
}
```

Thus, the type `Email` represents email addresses, `URL` internet addresses, `Image` image locations, `Text` long pieces of text, and `Secret` passwords. Based on these types a better tuned user interface can be generated. For example, the following rules generate different input fields based on the type alias:

```
property-to-edit-component(|x_component) :
  Property(x_prop, SimpleSort("Text")) ->
  %><h:inputTextarea value="#{<%=x_component%>.<%=x_prop%>}"/><%

property-to-edit-component(|x_component) :
  Property(x_prop, SimpleSort("Secret")) ->
  %><h:inputSecret value="#{<%=x_component%>.<%=x_prop%>}"/><%
```

A textarea is generated for a property of type `Text`, and a password input field is generated for a property of type `Secret`.

4.2 Collections

Another ommission so far was that associations had only singular associations. Often it is useful to have associations with collections of values or entities. Of course, such collections can be modeled using the basic modeling language. For example, define

```
PersonList { hd : Person tl : PersonList }
```

However, in the first place this is annoying to define for every collection, and furthermore, misses the opportunity for attaching standard functionality to collections. Thus, we introduce a general notion of generic sorts, borrowing from Java 5 generics the notation `X<Y,Z>` for a generic sort `X` with sort parameters `Y` and `Z`. For the time being this notation is only used to introduce collection

associations using the generic sorts `List` and `Set`. For example, a `Publication` with a list of authors and associated to several projects can then be modeled as follows:

```
Publication {
  title    : String
  authors  : List<Person>
  year     : Int
  abstract : Text
  projects : Set<Project>
  pdf      : URL
}
```

Many-to-many associations Introduction of collections requires extending the generation of entity classes. The following rule maps a property with a list type to a Java property with list type and persistence annotation `@ManyToMany`, assuming that objects in the association can be referred to by many objects from the parent entity:

```
property-to-property-code(|x_Class) :
  Property(x_prop, GenericSort("List", [y])) -> |[
    @ManyToMany
    @Cascade({
      CascadeType.PERSIST,CascadeType.SAVE_UPDATE,CascadeType.MERGE
    })
    private List<t> x_prop = new LinkedList<t>();
  ]|
```

The `@Cascade` annotation declares how the entity manager should behave when persistence operations are applied to the parent object, that is whether these operations should be propagated (cascaded) to the objects at the other end of the association. The annotation above states that only operations that save the parent object should be propagated, but not, for example, deletions.

Collections also requires an extension of the userinterface. This will be discussed later in the paper.

4.3 Refining Association

Yet another omission in the domain modeling language is with regard to the nature of associations. In UML terminology, are associations to other entities *compositions* or *aggregations*? In other words, does the referring entity *own* the objects at the other end of the association or not? To make this distinction we refine properties to be either value type (e.g. `title :: String`), composite (e.g. `address <> Address`), or reference (e.g. `authors -> List<Person>`) associations. Figure 5 illustrates the use of special value types, collections, and composite and reference associations

Based on the association type different code can be generated. For example, a composite collection, i.e. one in which the referrer owns the objects in the collection, gets a different cascading strategy than the one for (reference) collections

```
Publication {                Person {                 Address {
  title     :: String          fullname  :: String      street :: String
  authors   -> List<Person>    email     :: Email       city   :: String
  year      :: Int             homepage  :: URL         phone  :: String
  abstract  :: Text            photo     :: Image     }
  projects  -> Set<Project>    address   <> Address
  pdf       :: URL             user      -> User
}                            }
```

Fig. 5. Domain model with composite and reference associations.

above, namely one in which the associated objects are deleted with their owner. Furthermore, collections of value types are treated differently than collections of entities.

Unfolding Associations One particular decision that can be made based on association type is to unfold composite associations in view and edit pages. This is what is already done in Figures 1 and 2. In Figure 5 entity `Person` has a composite association with `Address`. Thus, an address is owned by a person. Therefore, when viewing or editing a person object we can just as well view/edit the address. The following rule achieves this by *unfolding* an entity reference, i.e. instead of including an input field for the entity, the edit rows for that entity are inserted:

```
row-in-edit-form(|editY) :
  |[ x_prop <> s ]| ->
  %>
    <tr><td><h:outputText value="<%=x_prop%>"/></td><td></td></tr>
    <%= row* ::*%>
  <%
  where <defined-java-type> s
      ; prop*  := <properties> s
      ; editYX := <concat-strings>[editY,".",x_prop]
      ; row*   := <map(row-in-edit-form(|editYX))> prop*
```

As an aside, note how the EL expression passed to the recursive call of `row-in-edit-form` is built up using string concatenation (variable `editYX`). This rather bad style is an artifact of the XML representation for JSF; the attributes in which EL expressions are represented are just strings without structure. This can be solved by defining a proper syntax of JSF XML by embedding a syntax of EL expressions.

5 Scrap your Boilertemplate™

Time to take stock. What have seen in the previous sections? We analyzed the programming patterns for JPA entity classes and for view/edit pages implemented using JSF and Seam. Then we abstracted out the commonality in these

programming patterns and turned them into code generation rules with as input a domain modeling language providing some variability in the form of built-in types and kinds of associations. The boilerplate in the generated code is considerable. For example, for the entity `Publication` in Figure 5 here is a breakdown of the source files generated and their size:

file	LOC
Publication.java	121
EditPublicationBeanInterface.java	56
EditPublicationBean.java	214
ViewPublicationBeanInterface.java	28
ViewPublicationBean.java	117
editPublication.xhtml	181
viewPublication.xhtml	153
total	870

It should be noted that this includes code for editing entity associations that has not been discussed yet. The one but last row in Figure 2 gives an example of the user interface; for a reference to an entity, the user interface provides a drop-down menu with (names of) all objects of that type. This is responsible for quite a bit of the code in `EditPublicationBean.java`.

With 8 lines of model input, the ratio of generated lines of code to source lines of code is over 100! Now the question is what that buys us. If there was a market for boring CRUD applications this would be great, but in practice we want a much richer application with fine tuned view and edit pages. If we would continue on the path taken here, we could add new sets of generator rules to generate new types of pages. For example, we might want to have pages for searching objects, pages list all objects of some type, pages providing selections and summaries, etc. But then we would hit an interesting barrier: code duplication in the code generator. The very phenomenon that we were trying to overcome in the first place, code duplication in application code, shows up again, but now in the form of target code fragments that appear in more than one rule (in slightly differen forms), sets of generator rule that are very similar, but generate code for a different type of page, etc. In other words, this smells like boilerplate templates, or boilertemplates, for short.

The boilertemplate smell is characterized by similar target coding patterns used in different templates, only large chunks of target code (a complete page type) considered as a reusable programming pattern, and limited expressivity, since adding a slightly different pattern (type of page) already requires extending the generator.

High time for some generator refactoring. The refactoring we're going to use here is called **find an intermediate language** also known as *scrap your boilertemplate*[2]. In order to gain expressivity we need to better cover the variability in the application domain. While implementing the domain model DSL, we have

[2] Google for 'scrap your boilerplate'.

explored the capabilities of the target platform, so by now we have a better idea how to implement variations on the CRUD theme by combining the basics of JSF and Seam in different ways. What we now need is a language that sits in between the high-level domain modeling language and the low-level details of JSF/Seam and allows us to provide more variability to application developers while still maintaining an advantage over direct programming.

Consider the following domain model for an entity `Group`:

```
Group {
  acronym   :: String (name)
  fullname  :: String
  mission   :: Text
  logo      :: Image
  members   -> Set<Person>
  projects  -> Set<ResearchProject>
  colloquia -> Set<Colloquium>
  news      -> List<News>
}
```

While a standard edit page is sufficient for this model, we want to create custom presentation pages that highlight different elements. We will use this example to design a basic language for page flow and presentation. Then we develop a generator that translates page definitions to JSF pages and supporting Seam session beans.

5.1 Page Flow

The view/edit pages in Section 3 had URLs of the form

```
/viewGroup.seam?group=x
```

where `x` is the identity of the object to be presented. Thus, a page has a name and arguments, so an obvious syntax for page definitions is

```
define page viewGroup(group : Group) {
  <presentation>
}
```

The parameter is a variable local to the page definition. While in the implementation, the URL to call a page uses object identities, within a page definition the parameter variable can be treated as referring to the corresponding object. Of course, a page definition can have any number of parameters, including zero. *Navigation* to a page has the following form in HTML:

```
<a href="/viewGroup.seam?group=1">SERG</a>
```

The identity of the argument object is passed and a string is used for the anchor of the link. In WebDSL we'll use the following form for specifying page navigation:

Fig. 6. View of `Group` object.

```
define page viewGroup(group:Group){
section {
  header{text(group.fullname)}
  section {
    header{"Mission"}
    outputText(group.mission)
  }
  section {
    header{"Recent Publications"}
    list { ... }
  }
  section {
    header{"People"}
    list { ... }
  }
} }
```

Fig. 7. Markup for `Group` view.

```
navigate(pers.group.acronym, viewGroup(pers.group))
```

The first argument is a specification of the text for the anchor, which can be a literal string, or a string value obtained from some data object. The second argument is a 'call' to the appropriate page definition.

5.2 Content Markup and Layout

Next we are concerned with presenting the data of objects on a page. For instance, a starting page for a research group might be presented as in Figure 6. The layout of such a page is defined using a presentation markup language that can access the data objects passed as arguments to a page. The elements for composition of a presentation are well known from document definition languages such as LaTeX, HTML, and DocBook and don't require much imagination. We need things such as sections with headers, paragraphs, lists, tables, text blocks, etc. Figure 7 shows the top-level markup for the view in Figure 6. There we see sections with headers, nested sections, lists, and a text block obtained by taking the `Text` from `group.mission`. The intention of these markup constructs is that they don't allow any configuration for visual formatting. That is, `section` does not have parameters or attribute for declaring the font-size, text color, are text alignment mode. The markup is purely intended to indicate the *structure* of the document. Visual formatting can be done using cascading stylesheets [16].

While the presentation elements above are appropriate for text documents, web pages often have a more two-dimensional layout. That is, in addition to the

Fig. 8. Logos, sidebars, drop-down menus.

body, which is layed out as a text document, a web page often contains elements such as a toolbar with drop-down menus, a sidebar with (contextual) links, a logo, etc. Figure 8 illustrates this by an extension of the `Group` view page of Figure 6 with a sidebar, menubar with drop-down menus and a main logo.

WebDSL takes a simple view at the problem of two-dimensional layout. A page can be composed of `block`s, which can be nested, and have a name. For instance, here is the (top-level) markup for the page in Figure 8:

```
define page viewGroup(group : Group) {
  block("outersidebar"){
    block("logo"){ ... }
    block("sidebar"){ ... }
  }
  block("outerbody"){
    block("menubar"){
      block("menu"){ ... }
    }
    block("body"){
      section { header{text(group.fullname)} ... }
    }
  }
}
```

This definition states that a page is composed of two main blocks, `outersidebar` and `outerbody`, which form the left and right column in Figure 8. These blocks are further subdivided into, logo and sidebar, and menubar and body, respectively. By mapping `block`s to `div`s in HTML with the block name as CSS class,

```
.outersidebar {                      .outerbody {
  position   : absolute;               position : absolute;
  overflow   : hidden;                 top      : 10px;
  top        : 0px;                    left     : 12.5em;
  left       : 10px;                   right    : 40px;
  margin-top : 10px;                 }
  width      : 10em;                 .menubar {
}                                      height        : 62px;
.logo {                                border-bottom : 1px dotted;
  text-align : left;                   color         : darkblue;
}                                    }
.sidebar {                           .body {
  top          : 0px;                  position      : relative;
  margin-top   : 20px;                 top           : 20px;
  color        : darkblue;             margin-bottom : 2.5em;
  border-right : 1px dotted;         }
}
```

Fig. 9. Cascading stylesheet for block layout.

the layout can be determined again using CSS. For instance, the layout of Figure 8 is obtained using the stylesheet in Figure 9.

Other layout problems can be solved in a similar way using CSS. For example, the sidebar in Figure 9 is simply structured as a list:

```
block("sidebar"){
  list {
    listitem { navigate(group.acronym, viewGroup(group)) }
    listitem { navigate("People", groupMembers(group)) }
    listitem { navigate("Publications", groupPublications(group)) }
    listitem { navigate("Projects", groupProjects(group)) list{ ... } }
  }
}
```

Using CSS the default indented and bulleted listitem layout can be redefined to the form of Figure 9 (no indentation, block icon for sublists, etc.).

Even drop-down menus can be defined using CSS. The menus in Figure 9 are again structured as lists, that is, each menu is a list with a single list item, which has a sublist declaring the menu entries:

```
block("menu") {
  list { listitem { "People"   list { ... } } }
  list { listitem { "Projects" list { ... } } }
  ...
}
```

Using the `:hover` element, such lists can be specified to be visible only when the mouse is over the menu.

Thus, using simple structural markup elements without any visual configuration, we can achieve a good separation of the definition of the structure of a page and its visual layout using cascading stylesheets. This approach can be easily extended to more fancy userinterface elements by targetting AJAX in addition to pure HTML. There again the aim should be to keep WebDSL specifications free of visual layout.

5.3 Language Constructs

We have now developed a basic idea for a page presentation language with concepts such as sections, lists, and blocks. The next question is how to define a language in which we can write these structures. The approach that novice language designers tend to take is to define a syntactic production for each markup element. Experience shows that such language definitions become rather unwieldy and make the language difficult to extend. To add a new markup construct, the syntax needs to be extended, and thus all operations that operate on the abstract syntax tree. Lets be clear that a rich syntax is a good idea, but only where it concerns constructs that are really different. Thus, rather than introducing a syntactic language construct for each markup element we saw above, WebDSL has a single generic syntactic structure, the *template call* (why it is called *template* call will become clear later).

Template Call In general, a template call has the following form:

```
f(e1,...,em) {elem1 ... elemn}
```

That is, a template call has a name `f`, a list of *expressions* `e` and a list of *template elements* `elem`. The abstract syntax for the construct is

```
TemplateCall(f, [e1,...,em], [elem1, ..., elemn])
```

Both the expression and element argument lists are optional. The name of the call determines the type of markup and is mapped by the back-end to some appropriate implementation in a target markup language.

The element arguments of a call are nested presentation elements. For example, a `section` has as arguments, among others, headers and paragraphs

```
section{ header{ ... } par{ ... } par{ ... } }
```

a `list` has as elements `listitem`s

```
list { listitem { ... } ... }
```

and a `table` has `row`s

```
table { row{ ... } row{ ... } }
```

The expression arguments of a call can be simple strings, such as the name of a block:

```
block("menu") { list { ... } }
```

However, mostly expressions provide the mechanism to access data from entity objects. For example, the `text` element takes a reference to a string value and displays it:

```
text(group.name)
```

Similarly, the `navigate` element takes a string value to be used as the anchor, and a method call expression that is interpreted as a call to a page.

```
navigate(pub.name, viewPublication(pub))
```

Iteration While the template call element is fairly versatile, it is not sufficient for everything we need to express. In particular, we need a mechanism for iterating over collections of objects or values. This is the role of the `for` iterator element, which has the following concrete syntax:

```
for( x : sort in e ) { elem* }
```

The reason that this construct cannot be expressed using the syntax of a template call is the variable which is bound locally in the body of the iterator. The abstract syntax of the `for` element is

```
For(x, sort, e, elem*)
```

The iterator is typically used to list objects in a collection. For example, the following fragment of a page involving a `group` of type `Group`, which has a collection of `projects`,

```
list {
  for(p : ResearchProject in group.projects) {
    listitem {
      navigate(p.acronym, viewResearchProject(p))
    }
  }
}
```

5.4 Mapping Pages to JSF+Seam

In Section 3 we saw how to generate a web application for viewing and editing objects in a domain model using a row-based interface targetting the JSF and Seam frameworks. We can now use the knowledge of that implementation approach to define a mapping from the new fine grained presentation elements to JSF+Seam. Figure 10 illustrates the mapping for tiny page definition. The mapping from a page definition to JSF involves creating an XML JSF document with

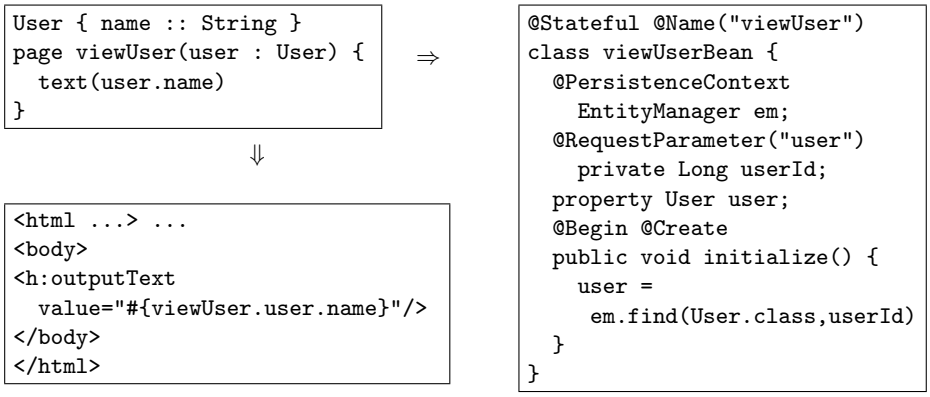

Fig. 10. Mapping from page definition (upper left) to session bean (right) and JSF (lower left).

as body the body of the page definition, mapping presentation elements to JSF components and HTML, and object access expressions to JSF EL expressions. The mapping from a page definition to a Seam session bean involves creating the usual boilerplate, `@RequestParameter`s with corresponding properties, and appropriate statements in the initialization method. In the rest of section we consider some of the translation rules.

Generating JSF The mapping from page elements to JSF is a fairly straightforward set of recursive rules that translate individual elements to corresponding JSF components. Note that while the syntax of template calls is generic, the mapping is *not* generic. First, while the syntax allows to use arbitrary identifiers as template names, only a (small) subset is actually supported. Second, there are separate generation rules to define the semantics of different template calls. The essence of the domain-specific language is in these code generation rules. They store the knowledge about the target domain that we reuse by writing DSL models. We consider some representative examples of the mapping to JSF.

Text The rule for `text` is a base case of the mapping. A `text(e)` element displays the string value of the `e` expression using the `outputText` JSF component.

```
elem-to-xhtml :
  |[ text(e) ]| -> %> <h:outputText value="<%=el%>"/> <%
  where el := <arg-to-value-string> e
```

The `arg-to-value-string` rules translate an expression to a JSF EL expression.

Block The rule for `block` is an example of a recursive rule definition. Note the application of the rule `elems-to-xhtml` in the antiquotation.

```
elem-to-xhtml :
  |[ block(str){elem*} ]| ->
  %>
    <div class="<%= str %>">
      <%= <elems-to-xhtml> elem* ::*%>
    </div>
 <%
```

The auxiliary `elems-to-xhtml` strategy is a map over the elements in a list:

```
elems-to-xhtml = map(elem-to-xhtml)
```

Iteration While iteration might seem one of the complicated constructs of WebDSL, its implementation turns out the be very simple. An iteration such as the following

```
list{ for ( project : ResearchProject ) {
  listitem { text(viewGroup.project.acronym) }
}}
```

is translated to the JSF `ui:repeat` component, which iterates over the elements of the collection that is produced by the expression in the `value` attribute, using the variable named in the `var` attribute as index in the collection.

```
<ul> <ui:repeat var="project"
              value="#{viewGroup.group.projectsList}">
   <li> <h:outputText value=\"#{viewGroup.project.acronym}\" </li>
</ui:repeat> </ul>
```

This mapping is defined in the following rule:

```
elem-to-xhtml :
  |[ for(x : s in e) { elem1* } ]| ->
  %>
    <ui:repeat var="<%= x %>" value="<%= el %>">
      <%= elem2* ::*%>
    </ui:repeat>
  <%
  where el := <arg-to-value-string> e
      ; elem2* := <elems-to-xhtml> elem1*
```

Navigation The translation of a navigation element is slightly more complicated, since it involves context-sensitive information. As example, consider the following `navigate` element:

```
navigate(viewPerson(p)){text(p.name)}
```

It is a variation over the form before. Here the text of the anchor is defined in the list of argument elements. Such a navigation should be translated to the following JSF code:

```
<s:link view="/viewPerson.xhtml">
  <f:param name="person" value="#{p.id}" />
  <h:outputText value="#{p.name}" />
</s:link>
```

While most of this is straightforward, the complication comes from the parameter. The `f:param` component defines for a URL parameter the name and value. However, the name of the parameter (`person` in the example) is not provided in the call (`viewPerson`). The following rule solves this by means of the dynamic rule `TemplateArguments`:

```
elem-to-xhtml :
  |[ navigate(p(e*)){elem1*} ]|
  %> <s:link view = "/<%= p %>.xhtml"><%=
       <conc>(param*,elem2*) ::*
  %></s:link> <%
  where <IsPage> p
      ; farg*  := <TemplateArguments> p
      ; param* := <zip(bind-param)> (farg*, args)
      ; elem2* := <elems-to-xhtml> elem1*
```

In a similar way as `declare-entity` in Section 2 declares the mapping of declared entities to Java types, for each page definition dynamic rules are defined that (1) record the fact that a page with name `p` is defined (`IsPage`), and (2) map the page name to the list of formal parameters of the page (`TemplateArguments`). Then, creating the list of `f:param`s is just a matter of zipping together the list of formal parameters and actual parameters using the following `bind-param` rule:

```
bind-param :
  (|[ x : $X ]|, e) ->
  %><f:param name="<%= x %>" value="<%= el %>" /><%
  where <defined-java-type> $X
      ; el := <arg-to-value-string> |[ e.id ]|
```

The rule combines a formal parameter `x` and an actual parameter expression `e` into a `f:param` element with as name the name of the formal parameter, and as value the EL expression corresponding to `e`.

Sections A final example is that of nested sections. Contrary to the custom of using fixed section header levels, WebDSL assigns header levels according to the section nesting level. Thus, a fragment such as

```
section { header{"Foo"} ... section { header{"Bar"} ... } }
```

should be mapped to HTML as follows:

```
<h1>Foo</h1> ... <h2>Bar</h2> ...
```

This is again an example of context-sensitive information, which is solved using a dynamic rule. The rules for sections just maps its argument elements. But before making the recursive call, the `SectionDepth` counter is increased.

```
elem-to-xhtml :
  |[ section() { elem1* } ]| -> |[ elem2* ]|
  where {| SectionDepth
         : rules( SectionDepth := <(SectionDepth <+ !0); inc> )
         ; elem2* := <elems-to-xhtml> elem1*
         |}
```

The dynamic rule scope {| SectionDepth : ... |} ensures that the variable is restored to its original value after translating all elements of the section.

The rule for the header element uses the SectionDepth variable to generate a HTML header with the right level.

```
elem-to-xhtml :
  |[ header(){ elem* } ]| ->
  %>
    <~n:tag><%= <elems-to-xhtml> elems ::*%></~n:tag>
  <%
  where n := <SectionDepth <+ !1>
      ; tag := <concat-strings>["h", <int-to-string> n]
```

Interesting about this example is that the dynamic rules mechanism makes it possible to propagate values during translation without the need to store these values in parameters of the translation rules and strategies.

5.5 Generating Seam Session Beans

The mapping from page definitions to Seam is less exciting than the mapping to JSF. At this point there are only to aspects to the mapping. First a page definition gives rise to a compilation unit defining a stateful session bean with as Seam component name the name of the page, and the usual boilerplate for session beans.

```
page-to-java :
  def@|[ define page x_page(farg*) { elem1* } ]| ->
  compilation-unit|[
    @Stateful @Name("~x_page")
    public class x_PageBean implements x_PageBeanInterface {
      @PersistenceContext private EntityManager em;
      @Create @Begin public void initialize() { bstm* }
      @Destroy @Remove public void destroy() {}
      ~*cbd*
    }
  ]|)
  where x_Page     := <capitalize-string> x_page
      ; x_PageBean := <concat-strings> [x_Page, "Bean"]
      ; cbd*       := <collect(page-elem-to-method)> def
      ; bstm*      := <collect(page-elem-to-init)> def
```

Secondly, for each argument of the page, a @RequestParameter with corresponding property is generated as discussed in Section 3.

```
argument-to-bean-property :
  |[ x : x_Class ]| ->
  |[
    @RequestParameter("~x") private Long x_Id;
    private x_Class x;
    public void x_set(x_Class x) { this.x = x; }
    public x_Class x_get() { return x; }
  ]|
  where x_Id  := <concat-strings>[x, "Id"]
      ; x_get := <property-getter> x
      ; x_set := <property-setter> x
```

Furthermore, code is generated for initializing the property by loading the object corresponding to the identity when the session bean is created.

```
argument-to-initialization :
  |[ x : x_Class ]| ->
  |[
    if (x_Id == null) { x = new x_Class(); }
    else { x = em.find(x_Class.class, x_Id); }
  ]|
  where x_Id := <concat-strings>[x, "Id"]
```

5.6 Boilertemplate Scrapped

This concludes the generator refactoring 'scrap your boilertemplate'. We have introduced a language that provides a much better coverage of the user interface domain; we can now create presentations by arranging the set of templates in different ways. The resulting mapping now looks much more like a compiler; the language constructs do one thing and the translation rules are fairly small. Next we consider several extensions of the language.

6 Extensions

In the core language developed in the previous section some aspects were, conciously, ignored to keep the presentation simple. In this section we consider several necessary extensions.

6.1 Typechecking

Java is a statically typed language, which ensures that many common programming errors are caught at compile-time. Surprisingly, however, this does not mean that web applications developed with frameworks such as JSF and Seam are free of 'type' errors after compilation.

JSF pages are 'compiled' at run-time, which means that many causes of errors are unchecked. Typical examples are missing or non-supported tags, references

to non-existing properties, and references to non-existing components. Some of these errors cause run-time exceptions, but others are silently ignored.

While this is typical of template-like data, it is interesting to observe that a framework such as Seam, which relies on annotations in Java programs for configuration, has similar problems. The main cause is that Seam component annotations are scanned and linked at deployment-time, and not checked at compile-time for consistency. Thus, uses of components (e.g. in JSF pages) are not checked. Injection and outjection of data enables loose coupling between components/classes, which is good, but as a result, the compiler can no longer check data flow properties, such as guaranteeing that a variable is always initialized before it is used. Another symptom of interacting frameworks is the fact that a method that is not declared in the `@Local` interface of a session bean, is silently ignored when invoked in JSF.

Finally, JPA and Hibernate queries are composed using string concatenation. Therefore, syntactic and type errors (e.g. non-existing column) become manifest only at run-time. Most of these types of errors will show up during testing, but vulnerabilities to injection attacks in queries only manifest themselves when the system is attacked, unless they are tested for.

Typechecking WebDSL To avoid the kind of problems mentioned above, WebDSL programs are statically typechecked to find such errors early. The types of expressions in template calls are checked against the types of definition parameters and properties of entity definitions to avoid use of non-existing properties or ill-typed expressions. The existence of pages that are navigated to is checked. For example, for the following WebDSL program

```
User { name :: String }
define page viewUser(user : User) {
  text(user.fullname)
  text(us.name)
}
```

the typechecker finds the following errors:

```
$ dsl-to-seam -i test.app
[error] definition viewUser/text/:
        expression 'user.fullname' has type error
[error] definition viewUser/text/:
        variable 'us' has no declared type
```

Yes, the error messages could be more informative.

Typechecking Rules The typechecker is a transformation on WebDSL programs, which checks the type correctness of expressions and annotates expressions with their type. These annotations will turn out very useful later on when considering higher-level abstractions. The following typecking rule for the iterator construct, illustrates some aspects of the implementation of the typechecker.

```
typecheck-iterator :
  |[ for(x : srt in e1){elem1*} ]| -> |[ for(x : s in e2){elem2*} ]|
  where in-tc-context(id
        ; e2 := <typecheck-expression> e1
        ; <should-have-list-type> e2
        ; {| TypeOf
           : if not(<java-type> s) then
               typecheck-error(|["index ", x, " has invalid type ", srt])
             else
               rules( TypeOf : x -> srt )
             end
           ; elems2 := <typecheck-page-elements> elems1
           |}
        | ["iterator ", x, "/"] )
```

First, the typechecker is a *transformation*, that is, rather than just checking, constructs are transformed by adding annotations. Thus, in this rule, the iterator expression and elements in the body are replaced by the result of typechecking them. Next, constraints on the construct are checked and errors reported with `typecheck-error`. The `in-tc-context` wrapper strategy is responsible for building up a context string for use in error messages. Finally, the local iterator variable `x` is bound to its type in the `TypeOf` dynamic rule. The dynamic rule scope `{| TypeOf : ... |}` ensures that the binding is only visible while typechecking the body of the iterator. The binding is used to annotate variables with their type, as expressed in the `typecheck-variable` rule:

```
typecheck-variable :
  Var(x) -> Var(x){Type(t)}
  where if not(t := <TypeOf> x) then
          typecheck-error(|["variable ", x, " has no declared type"])
          ; t := "Error"
        end
```

6.2 Data Input and Actions

The core language of the previous section only dealt with *presentation* of data. Data input is of course an essential requirement for web applications. To make edit pages we need constructs to create input components that bind data to object fields, forms, and buttons and actions to save the data. Figure 11 shows a WebDSL page definition for a simple edit page with a single input field and a `Save` button, as well as the mapping to JSF and Java/Seam. The language constructs are straightforward. The `form` element builds a form, the `inputString)e)` element creates an input field bound to the contents of the field point at by `e`, and the `action` element creates a button, which executes a call to a defined action when pushed. The mapping to JSF is straightforward as well. The action definition is mapped to a method of the session bean.

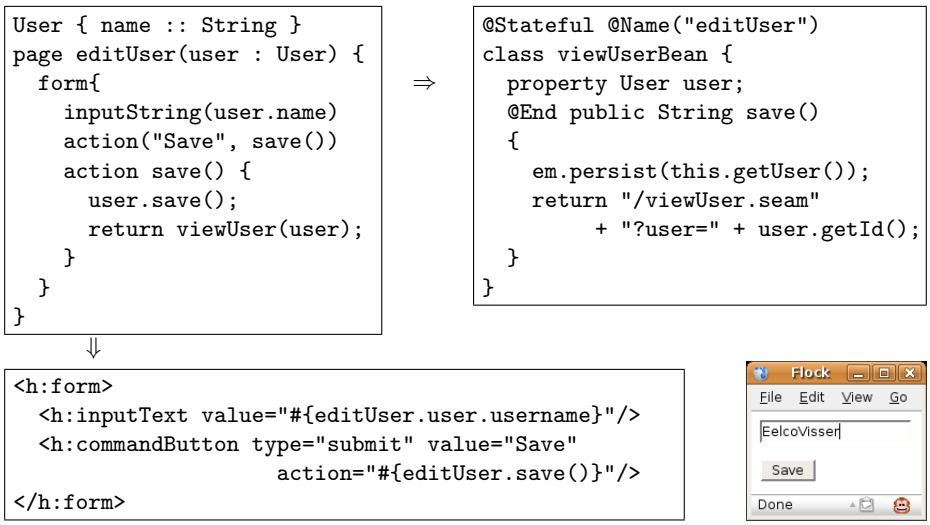

Fig. 11. Mapping form, input field, and action to JSF and Java/Seam.

Action Language The statement language that can be used in action definitions is a simple imperative language with the usual constructs. Assignment such as `person.blog := Blog{ title := name};` bind a value to a variable or field. Method calls `publication.authors.remove(author);` invoke an operation on an object. Currently the language only supports a fixed set of methods, such as some standard operations on collections, and persistence operations such as `save`. The latter can be applied directly to entity objects, hiding the interaction with an entity manager from the WebDSL developer. The return statement is somewhat unusual, as it is interpreted as a page-flow directive, that is, a statement `return viewUser(u);` is interpreted as a page redirect with appropriate parameters. Conditional execution is achieved using the usual control-flow constructs.

Expressions consist of variables, constant values (e.g. strings, integers), field access, and object creation. Rather than having to assign values to fields after creating an object, this can be done with the creation expression. Thus, object creation has the form `Person{ name := e ... }`, where fields can be directly given a value. There is also special syntax for creating sets (`{ e1, e2, ... }`) and lists `[ e1, e2, ... ]`.

Java Embedding The current design of the action language is a little ad hoc and should be generalized. A conventional crititique of domain-specific languages is that they require the redesign of such things as statements and expressions, which is hard to get right and complete. An alternative would be to directly embed the syntax of Java (statements and expressions), importing the full expressivity of that language. Indeed this is what is done with the Hibernate Query Language later in this section. However, I'm somewhat hesitant to do this for Java. While

```
User { name :: String }
page createUser() {
  var user : User := User{};
  form{
    inputString(user.name)
    action("Save", save())
    action save() {
      user.save();
      return viewUser(user);
    }
  }
}
```

⇒

```
@Stateful @Name("editUser")
class createUserBean {
  property User user;
  @Create @Begin
  public void initialize() {
    user = new User();
  }
  @End public String save() {
    em.persist(this.getUser());
    return "/viewUser.seam"
         + "?user=" + user.getId();
  }
}
```

⇓

```
<h:form>
  <h:inputText value="#{createUser.user.username}"/>
  <h:commandButton type="submit" value="Save"
                   action="#{createUser.save()}"/>
</h:form>
```

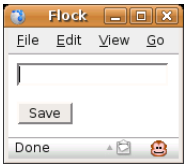

Fig. 12. Page local variables.

the embedding would provide a language with solid syntax and semantics, it would also mean a loss off control over the expressivity of the language, and over its portability to other platforms. A more viable direction seems to be to keep the action language simple, but provide foreign method interface, which gives access to functionality implemented in external libraries to be linked with the application.

6.3 Page Local Variables

So far we have considered pages that operated on objects passed as parameters. Sometimes it is necessary for a page to have local variables. For example, a page for creating a new object cannot operate on an existing object and needs to create a fresh object. Page local variables support this scenario. Figure 12 illustrates the use of a local variable in the definition of a page for creating new `User` objects, which is mostly similar as the edit page, except for the local variable.

6.4 Queries

The presentation language supports the access of data via (chained) field accesses. Thus, if we have an object, we can access all objects to which it has (indirect) associations. Sometimes, we may want to access objects that are not available through associations. For example, in the domain model in Figure 5, a `Publication` has a list of `authors` of type `Person`, but a `Person` has no (inverse) association to the publications he is author of. In these situations we need

```
User{ name :: String } Publication{ authors -> List<User> }
page viewUser(user : User) {
  var pubs : List<Publication> :=
      select pub from Publication as pub, User as u
       where (u.id = ~user.id) and (u member of pub.authors)
       order by pub.year descending;
  for(p : Publication in pubs) { ... }
}
```

⇓

```
class viewUserBean {
  property List<Publication> pubs;
  @Factory("pubs") public void initPubs() {
    pubs = em.createQuery(
      "select pub from Publication as pub, User as u" +
      " where (u.id = :param1) and (u member of pub.authors)" +
      " order by pub.year descending"
    ).setParameter("param1", this.getUser().getId())
     .getResultList();
  }
}
```

Fig. 13. Mapping embedded HQL queries to string-based query construction in Java.

a query mechanism to reconstruct the implicit association. In general, queries allow filtering of data.

There is no need to invent a DSL for querying. The Hibernate Query Language (HQL), an adaptation of the relational query language SQL to ORM, provides an excellent query language[2]. To make HQL available in WebDSL we follow the language embedding pattern described in [14]. Figure 13 illustrates the embedding and its implementation. The query retrieves the publications for which the `user` is an author. A HQL query is added to the WebDSL syntax as an expression. For now we assume the result of a query is assigned to a local page variable, which can then be accessed anywhere on the page. Queries can refer to values of page objects by means of the antiquotation . In Figure 13, this is used to find the user with same identity as the `user` object of the page. The query is translated to a `@Factory` method, which uses the entity manager to create the query using string composition. Antiquoted expressions become parameters of the query.

While the use of HQL in WebDSL does not provide a dramatic decrease in code size, there are some other advantages over the use of HQL in Java. In Java programs, Hibernate queries are composed as strings and parsed at run-time. This means that syntax errors in queries are only caught at run-time, which is hopefully during testing, but maybe during production is testing is thorough. The `getParameter` mechanism of HQL appears to do a job when it comes to

escaping special characters. However, when developers ignore this mechanism and splice values directly into the query string, the risks of injection attacks are high. In WebDSL are not composed as strings, but integrated in the syntax of the language. Thus, syntactic errors are caught at compile-time and it is not possible to splice in strings originating from user input without escaping. Another advantage is that the WebDSL typechecker can check the consistency of queries against the domain model and local variable declarations. The consistency of HQL queries in Java programs is only checked at run-time.

7 Not all Abstraction can be Generative

In the previous two sections we have extended the domain modelling language with a language for page presentation, data input, and page flow. The resulting language provides the required flexibility such that we can easily create different types of pages without having to extend or change the generator. The generator now encapsulates a lot of knowledge about basic implementation patterns.

The thesis of this section is that *not all abstraction can be generative*. That is, beyond the abstractions provided by the DSL, the application programmer needs mechanisms to create new abstractions. In order to avoid code duplication, it should be possible to name reusable fragments of code. Such a mechanism is vital for building a library. In this section we will introduce two simple abstraction mechanisms. Templates are named pieces of code with parameters and hooks. Modules are named collections of definitions defined in a separate file, which can be imported into other modules. Modules are essential for organizing a code base and to form a library of reusable code.

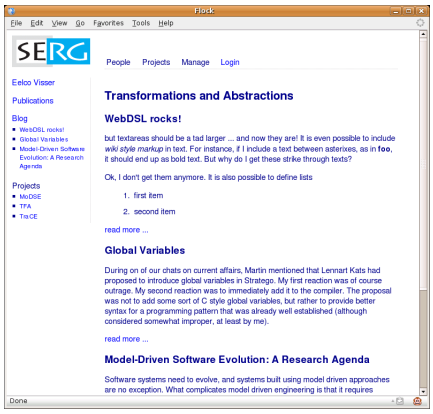

Fig. 14. Instance of viewBlog page.

To motivate the need for templates and modules consider the `viewBlog` page in Figure 14. The page displays entries of a blog, which are instances of the domain model defined in Figure 15, a menu with references to people and projects, and a sidebar with references relating to the author of the blog. Figure 16 gives metrics about the sizes of the WebDSL code involved in defining this page and the code generated from it. The entity classes generated from the domain model have a ratio of 12.5 lines of generated Java code to one line of source code. However, the mapping from `viewBlog.app` definition gives a ratio of only 3.6 lines of generated code to one line of source code. This is much less of a gain than in the case of entity classes.

```
Blog {                                  BlogEntry {
  title      :: String (name)             blog     -> Blog
  author     -> Person                    title    :: String (name)
  entries    <> List<BlogEntry>           created  :: Date
  categories -> List<Category>            category -> Category
}                                         intro    :: Text
                                          body     :: Text
                                          comments <> List<BlogComment>
                                        }
```

Fig. 15. Domain model for blogs and blog entries.

Now compare this to the metrics in Figure 17, which are for (a snapshot of) the SERG application (used to test WebDSL), which consists of some 30 entities, including the one for the blog. Here we see that the ratio of generated code to source code is 39! The difference in the ratios between Figure 16 and Figure 17 of almost a factor 10, is explained by template expansion. The size of `viewBlog` measured in Figure 16 is the all the WebDSL code needed to build that page. However, many elements of the page are used in other pages of the SERG application as well. The overall composition of the page is the same on all pages, as well as the menu. The sidebar is used on all pages related to `Person`s. By reusing such common elements between pages, the ratio of source code to generated code can be raised from 3.6 to 39. This depends on an intermediate step of template expansion and model-to-model transformations (next section).

7.1 Reusing Page Fragments with Templates

Template definitions provide a mechanism for giving a name to frequently used page fragments. A *template definition* has the form `define f(farg*){elem*}`, with `farg*` a list of formal parameters and `elem*` a list of template elements. The use of a defined template in a template call, which were introduced before, leads to the replacement of the call by the body of the definition. For example,

file name	LOC
Blog + BlogEntry	16
BlogEntry.java	116
Blog.java	85
generated : source	201 : 16 = 12.5
viewBlog.app	91
viewBlog.xhtml	164
ViewBlogBeanInterface.java	32
ViewBlogBean.java	131
generated : source	327 : 91 = 3.6

Fig. 16. Size metrics for `viewBlog`.

file name	LOC
serg.app	983
serg-full.app	9165
generated : source	9.3
generated total	38834
generated : source	4.2

Fig. 17. Metrics for serg.app

the following parameterless template definitions define literal fragments `logo`, `footer`, and `menu`:

```
define logo() { navigate(home()){image("/img/serg-logo.png")} }
define footer() {
  "generated with "
  navigate("Stratego/XT", url("http://www.strategoxt.org"))
}
define menu() {
  list{ listitem { "People" ... } } ...
}
```

Such fragments can be reused in many pages, as in the following page definition:

```
define page home() {
  block("menubar"){ logo() menu() }
  section{ ... }
  footer()
}
```

Literal template definitions are of limited use. To support reuse of partial fragments, which have holes that should be filled in by the reuse context, templates can have hooks in the form of template calls that can be locally (re)defined. For example, the following `main` template calls templates `logo`, `sidebar`, `menu`, `body`, and `footer`.

```
define main() {
  block("outersidebar") { logo() sidebar() }
  block("outerbody") {
    block("menubar") { menu() }
    body()
    footer()
  }
}
```

Some of these templates may have a global definition, such as the ones above, but others may (only) be defined locally in the context where `main` is called. For example, the following page definition calls the `main` template and defines `sidebar` and `body` (overriding any top-level definitions), thus instantiating the calls to these templates in the definition of `main`:

```
define page viewBlog(blog : Blog) {
  main()
  define sidebar(){ blogSidebar(blog) }
  define body() {
    section{ header{ text(blog.title) }
      for(entry : BlogEntry in blog.entries) { ... }
    }
  }
}
```

Finally, templates may need to access objects. Therefore, templates can have parameters. For example, the following definitions for a sidebar, defines links specific to a particular `Person` object `p`.

```
define personSidebar(p : Person) {
  list {
    listitem { navigate(p.name, viewPerson(p)) }
    listitem { navigate("Publications", personPublications(p)) }
    listitem { navigate("Blog", viewBlog(p.blog)) blogEntries() }
    listitem { "Projects" listProjectAcronyms(p) }
  }
}
```

This allows templates to be reused in different contexts. For example, the template above, can be used to create the sidebar for the view page for a `Person`, as well as for the publications page of that person.

```
define page viewPerson(person : Person) {
  main()
  define sidebar() { personSidebar(person) } ...
}
define page personPublications(person : Person) {
  main()
  define sidebar() { personSidebar(person) } ...
}
```

Template Expansion Template expansion is a context-sensitive transformation, whic again relies on dynamic rules for its implementation. For each template definition a dynamic rule `TemplateDef` is defined that maps the name of the template to its complete definition.

```
declare-template-definition =
  ?def@|[ define mod* x(farg*){elem*} ]|
  ; rules( TemplateDef : x -> def )
```

The dynamic rule is used to retrieve the definition when encountering a temlate call. Subsequently, all bound variables in the definition are renamed to avoid capture of free variables.

```
expand-template-call :
  |[ x(e*){elem1*} ]| -> |[ block(str){elem2*} ]|
  where <TemplateDef; rename> x => |[define mod* x(farg*){elem3*}]|
      ; {| Subst
         : <zip(bind-variable)> (farg*, <alltd(Subst)> e*)
         ; elem2* := <map(expand-element)> elem3*
         ; str := x
         |}
```

The formal parameters of the template are bound to the actual parameters of the call in the dynamic rule `Subst`:

```
bind-variable = ?(Arg(x, s), e); rules( Subst : Var(x) -> e )
```

A Trail of Blocks Note that the right-hand side of the rule for expansion of a template call creates a `block` around the expanded body. For example, the page definition

```
define page viewBlog(blog : Blog) {
  main()
  define sidebar(){ ... }
  define body() { ... }
}
```

expands to

```
define page viewBlog(blog : Blog) {
  block("main"){
    block("outersidebar") {
      block("logo"){ ... } block("sidebar"){ ... }
    }
    block("outerbody") {
      block("menubar") { block("menu") { ... } }
      block("body") { ... } block("footer") {  }
    }
  }
}
```

The trail of blocks can be used in stylesheets.

7.2 Modules

A module system allows an application definition to be divided into separate files. This is useful to keep overview of the parts of an application, and is necessary for building up a library. Module systems come in different measures of complexity. Module systems supporting separate compilation can become quite complex, especially if the units of compilation in the DSL don't match the units of compilation of the target platform. For this version of WebDSL a very simple module system has been chosen that supports distributing functionality over files, without separate compilation. A module is a collection of domain model and template definitions and can be imported into other modules as illustrated in Figures 18 and 19. The generator first reads in all imported modules before applying other transformations. The implementation of import chasing is extremely easy:

```
import-modules = topdown(try(already-imported <+ import-module))

already-imported : Imports(name) -> Section(name, [])
  where <Imported> name

import-module : Imports(name) -> mod
  where mod := <xtc-parse-webdsl-module>FILE(<concat-strings>[name, ".app"])
      ; rules( Imported : name )
```

The dynamic rule `Imported` is used to prevent re-importing the same module.

```
module publications
section domain definition.

  Publication {
    title    :: String (name)
    subtitle :: String
    year     :: Int
    pdf      :: URL
    authors  -> List<Person>
    abstract :: Text
    projects -> Set<ResearchProject>
  }
section presenting publications.

 define showPublication(pub : Publication) {
    for(author : Person in pub.authors){
      navigate(author.name, viewPerson(author)) ", " }
    navigate(pub.name, viewPublication(pub)) ", "
    text(pub.year) "."
  }
```

Fig. 18. Module definition.

```
application org.webdsl.serg

description
  This application organizes information relevant for a
  research group, including people, publications, students,
  projects, colloquia, etc.
end

imports app/templates
imports app/people
imports app/access
imports app/blog
imports app/colloquium
imports app/publications
imports app/projects
imports app/groups
imports app/news
imports app/issues
```

Fig. 19. Application importing modules.

8 More Sugar, Please!

With the core language introduced in Sections 5 and 6 we have obtained expressivity to define a wide range of presentations. With the templates and modules

from the previous section we have obtained a mechanism for avoiding code duplication. However, there are more generic patterns that are tedious to encode for which templates are not sufficient. Even if a language provides basic expressivity, it may not provide the right-level of abstraction. So if we encounter reocurring programming patterns in our DSL, the next step is to design higher-level abstractions that capture these patterns. Since the basic expressivity is present we can express these abstractions by means of transformations from the extended DSL to the core DSL. Such transformations are known as model-to-model transformations or *desugarings*, since the high-level abstractions are known as *syntactic sugar*. In the section we discuss three abstractions and their corresponding desugarings.

8.1 Output Entity Links

Creating a link to the view page for an object is done similar to the following WebDSL fragment:

```
navigate(viewPublication(pub)){text(pub.name)}
```

While not a lot of code to write, it becomes tedious, especially if we consider that the code can be derived from the *type* of the variable. Thus, we can replace this pattern by the simple element

```
output(pub)
```

This abstraction is implemented by the following desugaring rule, which uses the *type* of the expression to determine that the expression points to an entity object:

```
DeriveOutputSimpleRefAssociation :
  |[ output(e){} ]| -> |[ navigate($viewY(e)){text(e.name)} ]|
  where |[ $Y ]| := <type-of> e
      ; <defined-java-type> |[ $Y ]|
      ; $viewY := <concat-strings>["view", $Y]
```

This desugaring is enabled by the type annotations on expressions, which are put there by the typechecker. Similar desugaring rules can be defined for other types. For example,

```
DeriveOutputText :
  |[ output(e){} ]| -> |[ navigate(url(e)){text(e)} ]|
  where |[ URL ]| := <type-of> e

DeriveOutputText :
  |[ output(e){} ]| -> |[ image(e){} ]|
  where |[ Image ]| := <type-of> e
```

As a consequence of this abstraction, it is sufficient to write `output(e)` to produce the presentation of the object indicated by the expression `e`.

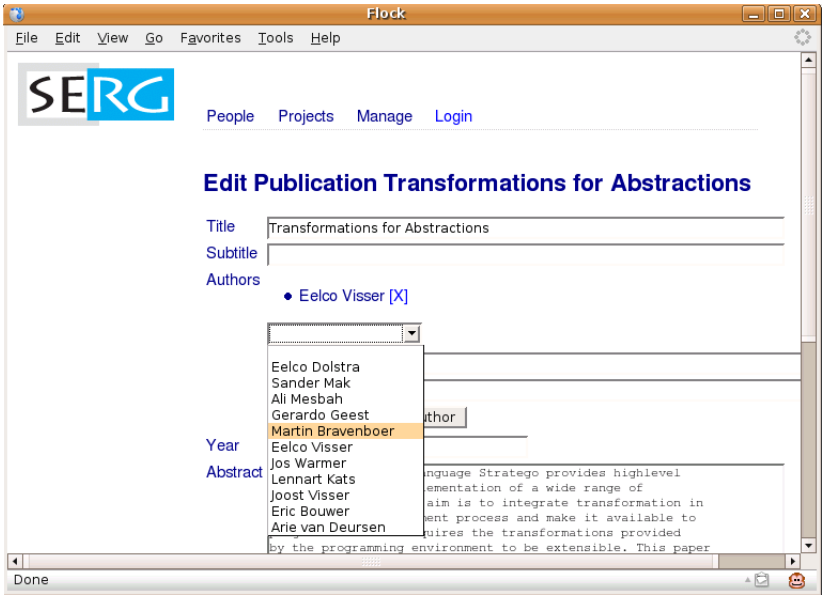

Fig. 20. Editing collection association.

8.2 Editing Entity Collection Associations

Editing a collection of entities Figure 20 is not as simple as editing a string or text property. Instead of typing in the value we'd like to select an existing object from some kind of menu. Consider the edit page for a publication in Figure 20. Editing the `authors` association requires the following ingredients: a list of names of entities already in collection; a link [X] to remove the entity from the collection; a select menu to add a new (existing) entity to the collection.' This is implemented by the following WebDSL pattern:

```
list { for(person : Person in publication.authors) {
  listitem{ text(person.name) " "
            actionLink("[X]", removePerson(person)) }
} }
select(person : Person, addPerson(person))
action removePerson(person : Person) {
  publication.authors.remove(person);
}
action addPerson(person : Person) {
  publication.authors.add(person);
}
```

Creating this pattern can be done automatically by considering the type and name of the association, which is done by the following desugaring rule:

```
DeriveInputAssociationList :
  elem|[ input(e){} ]| ->
  elem|[
    block("inputAssociationList"){
      list { for(x : $X in e){ listitem {
        text(x.name) " "
        actionLink("[X]", $removeX(x))
        action $removeX(x : $X) { e.remove(x); }
      }} }
      select(x1 : $X, $addX(x1))
      action $addX(x : $X) { e.add(x); }
    }
  ]|
  where |[ List<$X> ]| := <type-of> e
      ; x        := <decapitalize-string; newname> $X
      ; x1       := <decapitalize-string; newname> $X
      ; $viewX   := <concat-strings>["view", $X]
      ; $removeX := <concat-strings; newname>["remove", $X]
      ; $addX    := <concat-strings; newname>["add", $X]
```

Thus, an `input(pub.authors)` is now sufficient for producing the implementation of an association editor. Similar rules can be defined for other types:

```
DeriveInputText :
  |[ input(e){} ]| -> |[ inputText(e){} ]|
  where SimpleSort("Text") := <type-of> e

DeriveInputSecret :
  |[ input(e){} ]| -> |[ inputSecret(e){} ]|
  where SimpleSort("Secret") := <type-of> e
```

As a consequence, the `input(e)` call is now sufficient for producing the appropriate input interface.

8.3 Edit Page

The presentation layer language allows us to define our own very flexible user interface. It is now also possible to reformulate the generation of standard CRUD interface as a model to model transformation (Figure 21). That is, rather than directly generating Java and JSF code, we can generate a presentation model from an entity declaration. Then the general presentation generator will generate the implementation. The ingredients are an input box for each property of an entity organized in a table, and save and cancel buttons. The pattern for the (body of) an edit page is:

```
form {
  table {
    row{ "Blog"     input(entry.blog) }
    row{ "Title"    input(entry.title) }
```

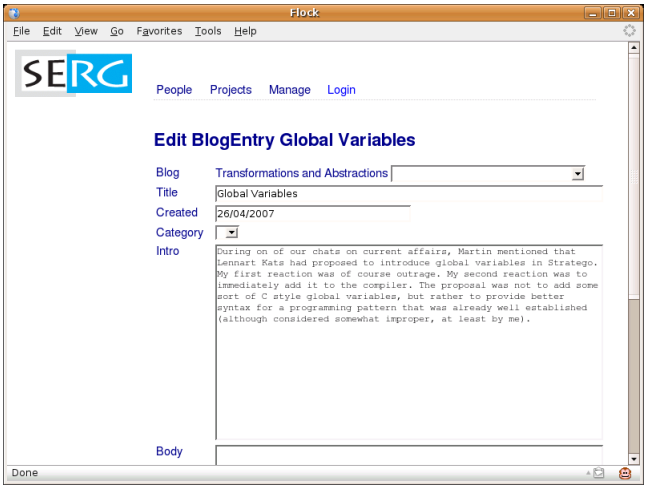

Fig. 21. Edit BlogEntry

```
    row{ "Created"  input(entry.created) }
    row{ "Category" input(entry.category) }
    row{ "Intro"    input(entry.intro) }
    row{ "Body"     input(entry.body) }
  }
  action("Save", save()) action("Cancel", cancel())
  action cancel() { return viewBlogEntry(entry); }
  action save() { entry.save(); return viewBlogEntry(entry); }
}
```

This pattern is captured in the following desugaring rule:

```
entity-to-edit-form :
  |[ $X : $Y { prop* } ]| ->
  |[
     form {
       table { elem* }
       action("Save",   save())
       action("Cancel", cancel())
     }
     action cancel() { return $viewX(x); }
     action save() { x.save(); return $viewX(x); }
  ]|
  where $viewX := <concat-strings>["view", $X]
      ; x      := <decapitalize-string> $X
      ; str    := $X
      ; elem*  := <map(property-to-edit-row(|x))> prop*

property-to-edit-row(|x) :
```

```
|[ y k s (anno*) ]| -> |[ row { str input(x.y) } ]|
where str := <capitalize-string> y
```

9 Unfinished Business

The WebDSL developed in this paper is just a first prototype. There are many loose ends and larger research and engineering questions. This section contains a list of these issues. Even the list itself requires more work!

9.1 Modeling Web Applications

With the current WebDSL, implementation of (simple) webapplications is no longer an obstacle. Just write a domain model and start experimenting with presentations. Yet there is much room for experimenting with domain modeling, and evaluating the impact on the persistence mapping and the interaction patterns.

9.2 Completeness of WebDSL

- Loose ends
  - Pagination of query results
  - Collections of value types
  - Punctuation in generated output (commas, delimiters, ...)
  - Better URLs
- More default interaction patterns
  - Identify styles of interaction and generate good defaults
  - In particular associations
- Rich(er) userinterface
  - Integration of iteration with UI components
  - Using AJAX JSF components
  - Single page user interface (e.g. using Echo2) (Jonathan Joubert)
- Input validation and conversion
- Security
  - authentication and access control (Danny Groenewegen)
  - Preventing injection attacks (seems to be covered well by base frameworks?)
- Workflow: business process modeling
- and of course: business logic
  - what is needed? (what is business logic, by the way?)

9.3 DSL Design and Implementation

Implementation of WebDSL

- Pretty-printed error messages (instead of dumping terms)
- Templates that abstract over template element (not only via hooks)
- Fully typechecking HQL expressions
- Easier name mangling with guaranteed consistency (?)
- Optimization of database queries

General Concerns

- DSL interaction and separate compilation (Sander Mak)
  - modular typechecking, template expansion, ...
  - generate modular code (depends on target platform)
- Reusable framework for DSL implementation
  - parameterized with syntax definition
  - organizes main generator pipeline
  - generation of multiple files
  - import chasing

9.4 Programming Environment

IDEs for DSLs

- New DSL not supported by IDE (Eclipse)
- Generate Eclipse plugin from language definition
  - syntax highlighting
  - syntax checking
  - typechecking
  - refactoring
  - ...
- Integrate Stratego/XT with Eclipse (Safari, EMF)

Visualization

- Visual views
  - class diagrams
  - page flow diagrams
- Editing via visual views?

9.5 Deployment

Current Status

- Generation of JSF and Java source files
- Skeleton of application source tree generated by seam-gen
- Manual build steps
  - .app to code (make)
  - code to .war/.ear (ant)
  - activation of database & webserver

Future

- Generate complete source tree
- Integrate building of the source tree (build .war file)
- Automatic deployment and activation of the webserver
- WebDSL virtual machine
  - drop `foo.app` and activate
  - server takes care of code generation, deployment, activation
  - using Nix deployment system

9.6 Evolution

Data conversion

- Adapting entity declarations leads to new database scheme
- Convert data in old database to new one
- Define relation mapping old entities to new ones
- Generate scripts for existing tools?

Model migration

- Changing DSL definition requires adapting existing models

Abstraction evolution

- Model sweetening: apply new sugar to old models

Harvesting from legacy code

- Transform legacy EJB applications to WebDSL?
- JSF to page definitions
- Entity classes to entity declarations
- Session beans to actions

10 Related Work

This work was partly inspired by Jos Warmer's Software Factory for web applications developed at Ordina [17]. Otherwise I have conciously *not* studied other domain-specific / web programming languages, nor approaches for developing domain-specific languages. The idea of the experiment was to consider the technology available for the domain and based on that develop a DSL. For the final version of this paper a comparison with web programming languages and other DSL building approaches will be included.

11 Discussion

Summary: Properties of a good DSL

- Core language that covers needed domain expressivity
- Syntactic extensions that allow concise expression
- Facilities to build a library
  - Modules for organization of code base
  - Parametric abstraction over DSL fragments

Summary: How to develop a DSL?

- Choose high-level technology
  - DSL should not readdress problems already solved by technology
- Start with large chunks of programs
  - Understand the technology
  - Recognize common patterns
- Setup a basic generator early on
  - makes it easy to experiment with alternative implementation strategies
- Don't try to find core language from the start
  - result may be too close to target
  - e.g., modeling language that covers all EJB concepts
- Don't over specialize syntax
  - template call vs header, section, ... as constructs
- Don't over generalize syntax (XML)

Future

- Extend WebDSL (see ideas before)
- Apply to industrial case studies
- Abstractions for application (business) domains?
  - finance, insurance, ...
- Repeat exercise for other domains
- Develop systematic method for building new modeling languages

Acknowledgements

First of all I would like to thank Ralf Lämmel and Joost Visser for inviting me to give a tutorial at GTTSE'07 in August 2006. This invitation provided a perfect target and outlet for the rather uncertain sabbatical project that I had conceived to build a domain-specific language for web applications. Along the way I had many inspiring discussions about various aspects of this enterprise. I would like to thank the following people for their input: Jos Warmer, Sander Mak, William Cook, Anneke Kleppe, Jonathan Joubert, Martin Bravenboer, Rob Schellhoorn, Danny Groenewegen.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
2. C. Bauer and G. King. *Java Persistence with Hibernate*. Manning, Greenwhich, NY, USA, 2007.
3. M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
4. M. Bravenboer. Connecting XML processing and term rewriting with tree grammars. Master's thesis, Utrecht University, Utrecht, The Netherlands, November 2003.
5. M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.
6. M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.
7. M. Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html, January 2004.
8. JBoss Seam. *Seam - Contextual Components. A Framework for Java EE 5*, 1.2.1.ga edition, 2007. http://www.jboss.com/products/seam.
9. K. D. Mann. *JavaServer Faces in Action*. Manning, Greenwhich, NY, USA, 2005.
10. J. F. Nusairat. *Beginning JBoss Seam*. Apress, New York, USA, 2007.
11. Sun Microsystems. *JSR 220: Enterprise JavaBeans^{TM}, Version 3.0. Java Persistence API*, May 2 2006.
12. J. van Wijngaarden. Code generation from a domain specific language. designing and implementing complex program transformations. Master's thesis, Utrecht University, Utrecht, The Netherlands, July 2003. INF/SCR-03-29.
13. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
14. E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
15. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

16. W3C. *Cascading Style Sheets, level 2. CSS2 Specification*, May 1998. http://www.w3.org/TR/REC-CSS2/.
17. J. Warmer. A model driven software factory using domain specific languages. In *Model Driven Architecture - Foundations and Applications proceedings of the Third European Conference (ECMDA-FA 2007)*, Haifa, Israel, June 2007.

TUD-SERG-2007-017
ISSN 1872-5392