Code Generation by Model Transformation
A Case Study in Transformation Modularity

Zef Hemel, Lennart C.L. Kats, and Eelco Visser

Software Engineering Research Group, Delft University of Technology,
The Netherlands
Z.Hemel@tudelft.nl, L.C.L.Kats@tudelft.nl,
visser@acm.org

Abstract. The realization of model-driven software development requires effective techniques for implementing code generators. In this paper, we present a case study of *code generation by model transformation* with Stratego, a high-level transformation language based on the paradigm of rewrite rules with programmable strategies that integrates model-to-model, model-to-code, and code-to-code transformations. The use of *concrete object syntax* guarantees syntactic correctness of code patterns, and enables the subsequent transformation of generated code. The composability of strategies supports *two dimensions of transformation modularity*. *Vertical* modularity is achieved by designing a generator as a pipeline of model-to-model transformations that gradually transforms a high-level input model to an implementation. *Horizontal* modularity is achieved by supporting the definition of plugins which implement all aspects of a language feature. We discuss the application of these techniques in the implementation of WebDSL, a domain-specific language for dynamic web applications with a rich data model.

1 Introduction

Model-driven software development aims at improving productivity and maintainability of software by raising the level of abstraction from source code in a general purpose language to high-level, domain-specific models such that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. The essence of the approach is to shift the knowledge about these implementation details from the minds of programmers to the templates of *code generators* that automatically translate models into implementations. Since the code generators themselves need to be developed and maintained as well, effective languages and tools for implementing generators are crucial for realizing model-driven software development. Many paradigms and technologies for transformation and generation are under development. In order to compare the various proposals, large scale case studies are needed. To this end we are developing WebDSL, a domain-specific language (DSL) for modeling dynamic web applications with a rich data model. In earlier work we described the development of WebDSL as a case study in domain-specific language engineering, i.e. a method to find the design of a new DSL [24].

A. Vallecillo, J. Gray, A. Pierantonio (Eds.): ICMT 2008, LNCS 5063, pp. 183–198, 2008.

In this paper, we discuss a case study in *code generation by model transformation*, an approach to the organization of DSL implementations that we use in the implementation of WebDSL. We have implemented the approach with the Stratego/XT program transformation system [23,5]. Stratego is a high-level transformation language that integrates model-to-model, model-to-code, and code-to-code transformations. The language provides *rewrite rules* for the definition of basic transformations, and *programmable strategies* for building complex transformations that control the application of rules. The use of *concrete object syntax* [22] in the definition of transformation rules improves the readability of rules, guarantees syntactic correctness of code patterns, *and* supports the subsequent transformation of generated code, which is not the case for text-based template engines such as Velocity [19] or xPand [25].

The composability of strategies supports *two dimensions of transformation modularity* used to realize separation of concerns in DSL implementations. First, *vertical modularization* is used to reduce the semantic gap between input and output model. Rather than directly generating code from the input model, the generator is constructed as a pipeline of model-to-model transformations that gradually transform a high-level input model to a low-level implementation model. Since even the generated code has a structured model representation to which transformations can be applied, any restrictions in modularity of the target language can be alleviated by extending it with new constructs to support better modularity. For example, we have created an extension of Java with partial classes, interface extraction, and name generation in order to simplify code generation rules.

Secondly, the approach supports *horizontal modularization*, that is, the separate definition of all transformations for a single language construct. This is the basis for *meta-model extensibility* through *generator extensibility*. The basic transformation pipeline provides an implementation for a base language. Extensions to the base language are implemented as plug-ins that extend the basic pipeline. Combining horizontal and vertical extensibility makes it possible to implement new domain-specific abstractions as plug-ins to the base language.

In the next section we give a brief introduction to WebDSL and the architecture of its implementation. In the rest of the paper we discuss the core ideas of the code generation by model transformation approach, i.e., code generation by rewriting (Section 3), model-to-model transformations to reduce input models to implementation models (Section 4), the role of semantic analyses and annotations (Section 5), and modularity and extensibility of the transformations (Section 6). We compare the approach to related work in Section 7.

2 WebDSL

WebDSL is a domain-specific language for the implementation of dynamic web applications with a rich data model. The language provides sub-languages for the specification of data models and for the definition of custom *pages* for viewing

```
entity Blog {
  title   :: String (name)
  entries <> List<BlogEntry>
}

entity BlogEntry {
  blog    -> Blog
             (inverse=Blog.entries)
  title   :: String (name)
  author  -> User
  created :: Date
  content :: WikiText
}
```

```
define view page blog(b : Blog) {
  main()
  title{ text(b.title) }
  define body() {
    section{
      header{ output(b) }
      for(entry : BlogEntry in b.entries
                  order by entry.created desc) {
        section {
          header { output(entry) }
          par{ "by " output(entry.author)
               " at " output(entry.created) }
          par{ output(entry.content) } } } } } }
```

Fig. 1. Example WebDSL data model and page definition

and editing objects in the data model. Fig. 1 illustrates this by means of a data model and view page for a blogging application.

The data model introduces *entity* definitions (e.g., `Blog`, `BlogEntry`), consisting of *properties* with a name and a type. Types of properties are either value types (indicated by `::`) or associations to other entities defined in the data model. Value types are basic data types such as `String` and `Date`, but also domain-specific types such as `WikiText` that carry additional functionality. Associations are composite (the referrer owns the object, indicated by `<>`) or referential (the object may be shared, indicated by `->`). The `inverse` annotation on a property declares a relation with automatic synchronization of two properties.

Page definitions consist of the name of the page, the names and types of the objects used as parameters, and a presentation of the data contained in the parameter objects. For example, the `blog(b : Blog)` definition in Fig. 1 creates a page showing all blog entries for blog `b`. WebDSL provides basic markup operators such as `section`, `header`, and `list` for defining the structure of a page. Data from the object parameters (and the objects they refer to) are injected in the page by data access operations such as `output`. Collections of data can be presented using the iterator construct `for`, which can filter and sort the elements of a collection. It is also possible to present content conditionally on some property of an object, for example, whether the user has the right access control permissions. User-defined *templates* allow the developer to define reusable chunks of WebDSL code. For example, the `main()` template used in Fig. 1 defines a general set-up for the page (navigation sidebars and menus) that is shared among many pages of the application. Finally, WebDSL supports separation of concerns by means of a module mechanism, and a separate sub-language for access control, which is beyond the scope of this paper.

The architecture of the WebDSL generator follows the four-level model organization of Bézivin [3] as illustrated in Fig. 2. At the M_3 level we find the SDF metametamodel, which is the grammar of the Syntax Definition Formalism SDF, which is defined in (and thus conforms to) itself [21]. At the M_2 level we find the WebDSL meta-model, i.e., the grammar of WebDSL defined in SDF. At the M_1 level we find WebDSL models of web applications, consisting of entity and

page definitions. At the M_0 level we find the actual web applications consisting of Java classes and XHTML pages, which represent the models at the M_1 level. In the implementation of WebDSL that we have realized [24], the M_0 systems are based upon the Java/Seam architecture, consisting of high-level application frameworks, such as the Java Persistence API (JPA), JavaServer Faces (JSF), and the Seam web framework. For each entity definition, a corresponding entity class is generated with fields, getters, and setters for the properties of the entity, annotated for object-relational mapping according to the JPA. For each page definition, a JSF XHTML page, a Seam Java bean class, and an accompanying interface are generated. In the following sections we discuss the organization of the generator as a pipeline of model-to-model transformations, and the techniques used to realize these transformations. The transformations are expressed in the Stratego transformation language [23,5], which is based on the paradigm of rewrite rules with programmable rewriting strategies.

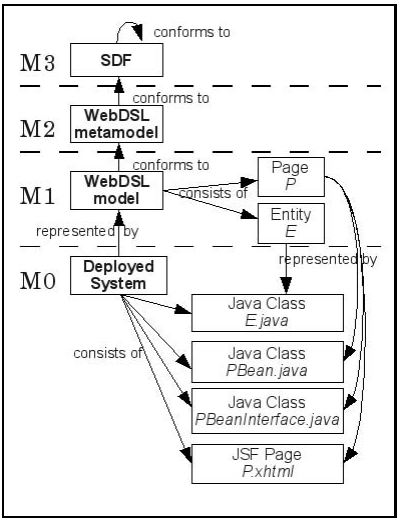


Fig. 2. Organization of models and artifacts of the WebDSL generator

3 Code Generation by Rewriting

WebDSL is a textual, domain-specific language and its M_2 meta-model is a grammar describing the valid sentences of that language. From the grammar, we automatically generate a parser, which transforms the textual representation of a model to an abstract syntax tree (AST). The AST conforms to a regular tree grammar, another M_2 meta-model that defines a set of valid trees, and which is obtained automatically from the grammar. All subsequent transformations are applied to the AST corresponding to the textual representation of the model. The WebDSL generator transforms high-level models into Java code and XML files. These target languages are also described by a grammar and a derived abstract syntax definition. All transformations are expressed in Stratego, which can apply transformations to any models with an abstract syntax definition.

The WebDSL generator can be decomposed into three main steps, which can be expressed and combined in Stratego as a *strategy*. A strategy is essentially a function that controls the order of application of more basic transformations. The basic strategy `webdsl-to-seam` is defined as a sequence of three steps, which are applied to the input model. First, starting with the main module of an application, all imported modules are parsed. Next,

```
webdsl-to-seam =
  import-modules
  ; generate-code
  ; output-generated-files
```

the combined model is transformed to a model of the generated Java and XML files. Finally, these code models are written to files and packaged for deployment to a web server. In later sections we will discuss refinements of this basic strategy.

3.1 Code Generation Rules

The elementary transformations that are combined by strategies are rewrite rules of the form `L : p1->p2 where s`. The name `L` of a rule can be used to invoke it in a strategy. When applied, the left-hand side pattern `p1` is matched against the subject term, binding any variables in the pattern to corresponding sub-terms of the subject term. When the match succeeds, and the condition `s` succeeds as well, the subject term is replaced with the instantiation of the right-hand side pattern `p2`. Rewrite rules are used for code generation by translating a fragment of the source language on the left-hand side to a fragment of the target language on the right-hand side. This is illustrated in Fig. 3 with a rewrite rule that rewrites a WebDSL page parameter, such as `b : Blog` in the page definition of Fig. 1, into a fragment of Java code that includes fields, accessors, and initialization code implementing the processing of a page parameter in a Seam page bean.

```
parameter-to-bean-property :
  |[ x : srt ]| ->
  <emit-java-code> |[
     @Partial class x_PageBean {
       @RequestParameter("~x") private Long x#Id;
       private t _#x;
       public void set#x(t x) { _#x = x; }
       public t    get#x()    { return x; }
       @Partial void initializeParameter() { bstm* }
     }
  ]|
  where bstm*      := <parameter-to-initialization>
      ; t          := <defined-java-type> srt
      ; x_PageBean := <CurrentPageBean>
```

Fig. 3. Rewrite rule transforming WebDSL source to Java target code using concrete syntax

Rewrite rules in Stratego can make use of the *concrete syntax* of the transformed language [22] using the `|[` and `]|` quotation construct. For example, a Java `return` statement can be expressed as `|[ return true; ]|`, rather than the abstract syntax form `Return(Some(Lit(True())))`. A language's concrete syntax is usually more concise and more familiar than its abstract syntax. The Stratego compiler parses concrete syntax quotations at compile-time, checking their syntax and replacing them with equivalent abstract syntax fragments.

Using *meta-variables* in concrete syntax fragments (written in *italics*), the rule in Fig. 3 matches any parameter *x* of type (or "sort") *srt*. In the `where` clause of the rule, a number of meta-variables are set for use in the produced Java fragment. For instance, *t* is set to the Java equivalent of WebDSL type *srt*, and *x_PageBean* is set to the current page bean.

In Stratego, the application of rewrite rules is under the control of *programmable strategies*, such that transformations can be explicitly staged. For example, the WebDSL `generate-code` transformation strategy uses a top-down traversal to visit all model elements for which code needs to be generated. This is expressed using the generic `topdown` traversal strategy as follows:

```
generate-code = topdown(try(argument-to-bean-property <+ ...))
```

Different rewrite rules are combined using the `<+` operator, which tries to apply each rule in the given order. Using the `try` strategy, the `generate-code` strategy will *try* to apply these rules, but will not fail if no rules are applicable, so that the `topdown` traversal will proceed even if none of the rules match.

3.2 Transforming Generated Code

In Stratego, generated code has a structured representation just like the input model of the generator — as opposed to the flat textual representation produced by traditional template engines. Therefore, additional transformations can be applied to generated code. This enables the use of an enriched version of the target language in generation, reducing the semantic gap between model and code, and thus simplifying the generator by capturing common generation patterns, and ensuring separation of concerns in their implementation.

For instance, in Fig. 3, the generated Java code takes the form of a *partial class*. That is, the rule defines only part of the generated class, as indicated by the `@Partial` annotation. In a later stage of the code generation process, all partial class fragments for the same class are merged. This approach eliminates the need for maintaining an aggregated model at this stage of the generator. In particular, the generated fragment is not used locally to replace the model fragment. Rather, in this case using `emit-java-code`, all code fragments are collected centrally for later assembly. Similarly, the generated `initializeParameter` method is a `@Partial` method, so that it can be extended for other page parameters. The order in which the statements of partial methods are merged is unspecified, thus no dependencies between statements in different definitions of a partial method should exist, an invariant that should be maintained by the developer of the generator.

Another extension of Java that is designed to simplify code generation, is the `#` identifier concatenation operator. It is used to generate the names of accessors, field, and classes that are commonly built up from different parts. For example, for accessors, `get#`x is used to generate a 'get' accessor for meta-variable x. In a later stage of the generator, such concatenations are evaluated and capitalized according to the Java conventions (e.g., using *camelCase* for method names).

Generated page bean classes require a matching interface definition. This interface is automatically generated in a separate generation stage: after merging all partial classes, such an interface is extracted from each generated class annotated with a `@RequiresInterface` annotation.

4 Semantic Analysis and Annotation

```
webdsl-to-seam =
  import-modules
  ; typecheck
  ; generate-code
  ; output-generated-files
```

Not all models that conform to the WebDSL syntax are valid. For instance, identifiers may refer to a non-existing entity, property, or function. Such models violate the static semantic constraints of WebDSL. A separate typechecking stage of the generator checks these constraints, and reports any violations found. The semantic information gathered at this stage is

also used to provide context information for other transformations, as we will discuss in the next section.

Typechecking involves a *context-sensitive* global-to-local transformation in which type information is propagated from the declaration site of an identifier to its use sites. Stratego provides support for such transformations through its mechanism of *dynamic rewrite rules* [6], which allows the definition of new rewrite rules at run-time. For example, the `typecheck-variable` rule in Fig. 4 defines the checking of the *use* of a variable with abstract syntax `Var(x)`. The dynamic rule `TypeOf` is used to rewrite the identifier `x` to its type `t`. If this fails an error is reported. Otherwise, the variable is *annotated* with its type. The `TypeOf` rule is *defined* when a variable declaration, which may be a page parameter or a local variable, is encountered. For example, the `declare-page-argument` rule checks that the type used in the declaration of a page parameter is a valid type (using the `TypeExists` dynamic rule). If the type does exist, the `rules` construct is used to define a new instance of the `TypeOf` rule specific to the values of `x` and `srt` encountered in the declaration. Dynamic rule *scopes* are used to limit the scope of rules to the traversal of a fragment of the model. For example, the `TypeOf` rule for a page parameter is valid only during the traversal of that page. Similarly, functions and `for` loops also define a local scope.

```
typecheck-variable :
  Var(x) -> Var(x){Type(t)}
  where if not(t := <TypeOf> x) then
          typecheck-error(|
            ["Undeclared variable ",x," referenced"])
        end
declare-page-argument :
  |[ x : srt ]| -> |[ x : srt ]|
  where if not(<TypeExists> srt) then
          typecheck-error(|
            ["Illegal type ",srt," for parameter ",x])
        else
          rules( TypeOf : x -> srt )
        end
```

Fig. 4. Typechecking with dynamic rules

5 Model-to-Model Transformations

Extending the target language helps in simplifying the translation from models to code. However, directly translating input models to code may still require complex transformations, in particular, when adding higher-level abstractions. Instead of a complex model-to-code translation, the WebDSL generator pipeline consists of several stages of model-to-model transformations that reduce models in the full WebDSL language to *core* WebDSL, which is domain-specific, yet relatively close to the target platform. As a result, only normalized core language constructs have to be transformed to the target platform during code generation, which improves retargetability. All the abstractions built on top of the core language can be ignored by the back-end. Staging the transformations in a pipeline is a case of *vertical modularity*; each stage is a separately defined transformation that is

```
webdsl-to-seam =
  import-modules
  ; typecheck
  ; normalize-syntax
  ; expand-page-templates
  ; derive
  ; merge-emitted-decs
  ; generate-code
  ; merge-partial-classes
  ; output-generated-files
```

only concerned with one aspect of the code generator. In this section we illustrate this with a discussion of typical transformations applied in such a pipeline: syntactic normalization, and the implementation of user-defined and generative abstractions. We also discuss the problem of preserving or reproducing the annotations produced by semantic analyses.

5.1 Syntactic Normalization

Syntactic abstractions, also known as syntactic sugar, provide new language constructs that support expression of functionality that is already provided by the base language in a more compact manner. The implementation of such abstractions can often be realized by means of simple local-to-local transformation rules (Fig. 5), but sometimes, more complex local-to-global rules (Fig. 7) are needed.

```
NormalizeSyntax :
  |[ text(e1,e2,e*){} elem* ]| ->
  |[ text(e1) text(e2,e*){} elem* ]|
NormalizeSyntax :
  |[ for(x : srt in e1
         order by e2){elem*} ]| ->
  |[ for(x : srt in e1
         where true
         order by e2){elem*} ]|
normalize-syntax =
  topdown(repeat(NormalizeSyntax))
```

Fig. 5. Local-to-local syntactic normalization rules

A local-to-local rewrite replaces a model fragment with another without using or producing other parts of the model, as illustrated by the examples in Fig. 5. The first rule normalizes applications of the `text` construct with multiple arguments to a list of applications of `text` with a single argument. More precisely, it splits off the first argument of a multi-argument application. Repeated application of the rule ensures that only singleton applications remain. For example, the application `text(blog.title, ": ", blog.author)` is reduced to `text(blog.title) text(": ") text(blog.author)`. Similarly, the second rule rewrites an occurrence of the `for` statement without a `where` clause to one with the universally valid `where true` clause. These normalizations ensure that later stages of the code generator only need to deal with one syntactic variant, i.e., singleton applications of `text`, and `for` statements with a `where` clause. The application of normalization rules is controlled by the `normalize-syntax` strategy, which performs a top-down traversal, which repeatedly applies rules to each element.

A local-to-global transformation rewrites a local element, but also produces elements that should be placed elsewhere in the model. An example of such a transformation is the lifting of list comprehensions. These provide declarative manipulations and queries on lists and sets, i.e., a combined map, filter and sort operation. As an example, consider the expression in Fig. 6, which retrieves the list of blog entries created after `date`, sorted in reverse chronological order. Such expressions can be computed by means of the `for` *statement* of WebDSL, as shown in the second part of Fig. 6. Statements, however, may not be used as expressions.

```
[e.title
 for(e : BlogEntry in b.entries
     where e.created > date
     order by e.created desc)]
```

```
globals { function lcf_33
(b : Blog, date : Date) {
 var y : List<String> := [];
 for(e : BlogEntry in b.entries
     where e.created > date
     order by e.created desc)
   { y.add(e.title); } } }
```

Fig. 6. List comprehension and implementation

```
Lift :
  |[ [e for(x : srt in e2 where e3 order by e4)] ]| ->
  |[ x_fun(arg*) ]|
  where x_fun     := <newname> "lcf"
      ; free-vars := <collect-free-vars> (e,e2,e3,e4)
      ; param*    := <map(build-param)> free-vars
      ; arg*      := <map(build-arg)> free-vars
      ; <emit-webdsl-dec> |[
          globals {
            function x_fun(param*) : List<srt> {
              var y : List<srt> := [];
              for(x : srt in e2 where e3 order by e4)
                { y.add(e); }
              return y; } } ]|
```

Fig. 7. Local-to-global syntactic normalization

The transformation in Fig. 7 *lifts* a list comprehension to a new global function definition and replaces the expression with a call to the generated function The free variables of the list comprehension expression are extracted and passed as parameters to the generated function. The `emit-webdsl-dec` rule takes the newly defined function and stores it in a dynamic rule. Declarations emitted in this manner are merged into the model during the `merge-emitted-decs` generator stage. (A pattern also applied in the form of partial classes during code generation).

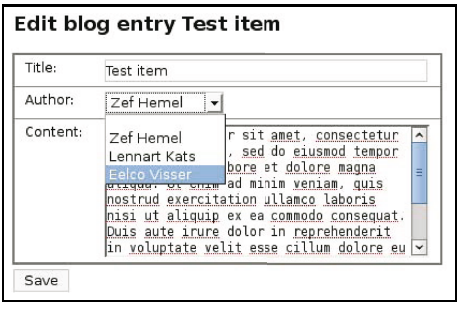
In a global-to-local transformation, constructs are locally transformed using (global) context information. The typechecking rules in the previous section are an example. Another example is the expansion (inlining) of user-defined templates by the `expand-page-templates` strategy. It collects top-level and local template definitions and replaces calls to these template definitions by their bodies, substituting actual parameters for formal parameters. This mechanism allows WebDSL developers to capture reoccurring patterns in page definitions for reuse.

5.2 Generative Abstractions

Generative abstractions are abstractions that explicitly invoke the generator to derive some functionality. Here we discuss an example of type-based derivation. Consider the edit page in Fig. 9, which provides an interface for editing the values of the properties of a `BlogEntry`. Depending on the type of the property, a different interface element is used; a simple string input box for `title`, a select box for `author`, and a text area for `content`. The definition of the edit page in Fig. 9 simply invokes `input(e.`*`prop`*`)` to declare an edit interface for property *prop*. The specific implementation for each input type is derived from the type of the expression. For example, the `DeriveInput` rule in Fig. 8 derives for an `input` of a property with a 'defined entity' type a select box for that type. Similarly, the `DeriveOutput` rule derives a rendering mechanism for an expression based on its type. For example, the use of `output(e.author)`

```
DeriveInput :
  |[ input(e){} ]| ->
  |[ select(s : srt, "Select", e) ]|
  where SimpleSort(srt) := <get-type> e
      ; <defined-entity> SimpleSort(srt)
DeriveOutput :
  |[ output(e){} ]| ->
  |[ navigate(x_view(e)){text(e.name)} ]|
  where SimpleSort(s) := <get-type> e
      ; <defined-entity> SimpleSort(s)
      ; x_view := <view-page-for-entity> s
```

Fig. 8. Type-based derivation




```
define page editBlogEntry(e : BlogEntry) {
  section {
    header{"Edit blog entry "output(e.title)}
    form { table {
        row { "Title:"   input(e.title) }
        row { "Author:"  input(e.author) }
        row { "Content:" input(e.content) } }
      action("Save", save())
      action save() {
        e.save(); return blogEntry(e);} } } }
```

```
define page editBlogEntry(e : BlogEntry) {
  derive editPage for e from BlogEntry }
```

Fig. 9. Screenshot of an edit page with low-level and high-level page definition

in a page definition results in a link (`navigate`) to the view page for the object that is referred to by `e.author`. The `e.author.name` property of that object is used as anchor for the link.

The next step in generative abstraction is the generation of complete page definitions. The structure of an edit page can often be fairly straightforward, say a table with a row for each property with an appropriate input interface. Such a structure can be derived automatically from the declaration of the entity. The implementation of `editBlogEntry` in the lower right of Fig. 9 uses the `derive` construct to automatically generate the implementation of the body of the edit page from the `BlogEntry` entity. The `derive-page` rule in Fig. 10 implements this derivation. The rows of the table are generated by a map of the `derive-edit-row` transformation over the properties of the entity *srt*, which are obtained by applying the dynamic rule `entity-properties`.

```
derive-page :
  |[ derive editPage for x from srt ]| ->
  |[ section{ header{"Edit " srt " " text(x.name)}
       form { table { row* }
         action("Save", save()) } }
     action save() {
       x.save(); return x_view(x); } ]|
  where x_view := <decapitalize-string> x
      ; prop*  := <entity-properties> srt
      ; row*   := <map(derive-edit-row(|x))> prop*
derive-edit-row(|x) :
  |[y k srt (anno*)]| -> |[row{x_text input(x.y)}]|
  where x_text := <concat-strings> [x, ": "]
```

Fig. 10. Rules to derive edit page elements

5.3 Restoring Annotations

As a model undergoes transformation, type annotations may be lost. Rewrite rules may introduce new variables or entire fragments of code that do not include type annotations. For example, the `derive-edit-row` rule in Fig. 10 does not attach a type annotation to the expression of the `input` element it generates. Defining the rules to create correct type annotations would

```
webdsl-to-seam =
 import-modules           ; typecheck
 ; normalize-syntax       ; typecheck
 ; expand-page-templates  ; typecheck
 ; derive                 ; typecheck
 ; merge-emitted-decs
 ; generate-code
 ; merge-partial-classes
 ; output-generated-files
```

be quite tedious and would require duplication of the knowledge encapsulated in the typechecking rules. Following the principle of separation of concerns, the typechecking rules are used to introduce type annotations in freshly generated code. A question then is what the granularity of applying typechecking rules should be. Since the type checker is currently defined as a complete traversal over the model, reapplying the type checker after each application of a transformation rule would be prohibitive. Instead, we take a rather course grained approach, re-applying the type checker after each transformation stage, as illustrated in the strategy above. This strategy requires these stages to be designed such that no transformation opportunities are missed by missing type annotations. Combination of analysis and transformation in an efficient way is a topic for research; it would be desirable to automatically infer an optimal incremental analysis strategy.

6 Transformation Modularity and Extensibility

Since its conception, the WebDSL generator has grown more and more complex. Initially, the generator was constructed in a centralized fashion, with a single *"God rule"* associated with each generated artifact. Much like a "God class", an anti-pattern in object-oriented programming, such a God rule dispatches a large number of smaller transformation rules to generate a monolithic target artifact (e.g., a Java class). As new language extensions were added, these rules grew to a size that they would no longer fit on a single screen. As such, this pattern was quickly identified as a code smell that hindered the extensibility and maintainability of the generator.

```
Derive :
  |[ input(e){} ]| -> |[ inputDate(e){} ]|
  where SimpleSort("Date") := <type-of> e

Derive :
  |[ output(e){} ]| -> |[ outputDate(e){} ]|
  where SimpleSort("Date") := <type-of> e

GenerateXML :
  .. generate xhtml controls for
  inputDate and outputDate ...

GenerateJavaExpr :
  |[ Date(d) ]| ->
  |[ org.webdsl.tools.Utils.parseDate(e1) ]|
  where e1 := <expression-to-java> d

GenerateJavaExpr :
  |[ now() ]| -> |[ new java.util.Date() ]|
```

Fig. 11. Modular definition of the primitive type `Date`

The employment of God rules was the unfortunate result of the structure of the target metamodel: Java provides only limited means of modularization of classes. Other platforms, such as C#, offer partial classes (but not partial methods), that can help further subdivide classes into smaller units. The lack of such a construct makes it difficult to decompose rewrite rules that generate large classes. This platform limitation can be resolved by extension of the target language, in the form of partial classes and methods. In a separate generator stage (`expand-partial-classes`), all partial classes and methods are merged for processing with a regular Java compiler.

To support both modularity and extensibility of transformation definitions, Stratego provides the notion of *strategy and rule definition extension*. Strategies

and rules can be extended by declaring a new instance with the same name. All such definitions are merged together, and evaluated in an unspecified order when invoked, until one of the definitions succeeds or all fail. The different stages of the generator make use of this facility, by defining rules that are extended in separate transformation modules. For example, Fig. 11 shows an implementation of an extension of WebDSL with a `Date` value type that makes use of this facility. It extends the definition of the `Derive` rule used in the derivation stage, and a number of rules in the code generation stage. (Not shown here are the mappings to the Java `Date` type and the corresponding JPA annotations.) Another, more elaborate extension that has been implemented is the addition of access control constraints to the model, which is outside the context of this paper.

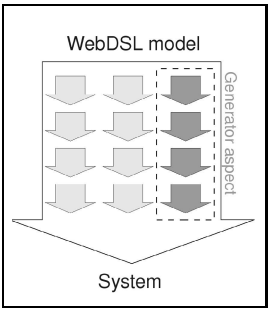


Fig. 12. Dimensions of modularity

As seen in the preceding section, transformations for an aspect of the generator can be divided into different stages. This *vertical modularity* helps in separation of concerns and retargetability. Further modularity can be achieved in a second dimension, by subdividing rules that operate on a single level. This is a form of *horizontal modularity* and is supported by rule definition extension and generation of partial artifacts. Horizontal modularity is essential for the extensibility of the generator. Fig. 12 illustrates the two dimensions of the transformation architecture. Highlighted is a horizontal extension of the generator (such as the `Date` extension), which consists of a number of vertical transformation stages.

The definition extension feature of Stratego elegantly combines rewrite rules that operate on different elements of a source model. However, it will only evaluate a single rule if multiple rules are applicable to the same element (e.g., a page parameter that has different rules to generate Java and JSF code for the same page parameter). While Stratego does not offer a direct facility for this, we build upon the notion of strategy extension to accomplish this, as shown below.

By use of a `fail` statement at the end of each definition of `GenerateCode`, all alternatives will "fail", ensuring each will be tried. The result of the applications can then be collected as a side effect using dynamic rules (e.g., using `emit-java-code` in Fig. 3). This pattern efficiently achieves the desired composition of definitions. Direct language support and accompanying compile-time checks for this within Stratego could prove useful, and could prevent subtle bugs that may occur if an extension programmer now forgets to include `fail` at the end of a definition, or mistypes its name.

```
GenerateCode =
  page-to-java; fail
GenerateCode =
  parameter-to-bean-property; fail
```

7 Discussion

Since the advent of model-driven engineering, several modeling methodologies and model transformation approaches have been introduced. A classification of

a number of such systems is given in [7]. Various MDE toolkits provide model transformation and code generation facilities, many of which are based on OMG's MDA (openArchitectureWare [8], AMMA [12], AndroMDA [2]). These generally consist of a metamodeling language (MOF [16], Ecore, KM3 [9]), model transformation language (ATL [11], xTend [8]), code generation language (TCS [10], xPand [25], Velocity [19]), and a language to define the sequence of transformations (oAW's workflow language, Groovy scripting language).

Model management can be based on any algebraic datastructure such as trees, graphs, hypergraphs, or categories [4]. Most current MDE toolkits are based on graphs, while Stratego/XT uses trees. By combining trees with dynamic rules, graphs *can* be represented in Stratego, which makes it possible to model context-sensitive information that cannot easily be expressed using just trees.

Consistency management is an important issue in MDE [14]. It is possible to keep models consistent as part of transformations, but in practice this also tends to make transformations much more complex. In our approach we chose to separate the concern of typechecking from the model transformation at hand. The drawback of this approach is that models need to be reanalyzed after applying transformations. Incremental analysis and transformation techniques are an important research topic. By analyzing models before any transformations are performed, we detect inconsistencies early and can report them to the developer. However, problems that occur while the system is running turn out to be difficult to trace back to errors in the model. In the future, we intend to investigate the feasibility of origin tracking [20] to achieve code-to-model traceability.

Transformation languages such as ATL and xTend allow transformations to be separated in modules, similarly to Stratego. However, extensibility of transformations is more difficult to realize, especially if transformation extensions have to operate on the same modeling elements, which is forbidden in ATL, for instance. In existing MDE toolkits, vertical modularity in transformations is often realized using a separate workflow language, such as the oAW workflow language and Groovy in AndroMDA. Stratego not only integrates model-to-model and model-to-code transformations, but also the overall generator workflow. Thus, a single transformation composition language is used for micro and macro compositions.

Some approaches [26] generate partial artifacts through the use of partial classes, which are then combined by the regular compiler for the target language. However, these approaches only work if the target language supports these features. In our approach, code is treated as a model, while most MDE approaches generate code through the use of textual template engines, which produce plain text, not amenable to further transformation. By treating generated code as a model, it is possible to extend the target language and add convenient language features such as partial classes and methods, and interface extraction.

Many (visual) languages for modeling web applications have been developed, including WebML [13], MIDAS [17], OOWS [15], Netsilon [18], and UWE [1]. UWE generates JSP code via a model representation conforming to a JSP metamodel. Netsilon uses an intermediate language for code generation in order to

increase retargetability of the generator. The other approaches use textual, usually template-based code generation.

8 Conclusions

In this paper we presented a case study of the *code generation by model transformation* approach applied in the development of WebDSL. WebDSL is a substantial DSL code generator, consisting of a total of 1300 rules and strategies (see Fig. 13). It has been employed for a number of web applications, most significantly the webdsl.org project website (which is currently used in production). The site features a generic project management interface, including a wiki-based documentation system, an issue tracker, blogs, and discussion forums. Fig. 13 gives an indication of the scale of the project: it is defined using 146 page and entity definitions, written in a total of 2366 lines of text. The code generated from these definitions spans nearly 80.000 lines of code. Even if this is not the volume of code one would produce manually for such an application, it seems justified to conclude that an order of magnitude reduction in code can be achieved. As such, we believe that employment of the WebDSL generator enables a significant gain in productivity, resulting from the high level of abstraction it provides.

We have shown how a pipeline of model-to-model transformations helps achieve high-level abstractions in models. By applying two-dimensional modularity—vertically in stages and horizontally in a core language and extensions—we ensure maintainability and extensibility of the generator. We have also demonstrated the benefits of generating models, rather than text, and how this technique aids horizontal modularity. The modular design of WebDSL provides a solid foundation for further research into higher-level domain-specific abstractions for web-based software systems. The approach should also be usable in the implementation of other DSLs. The approach can be further improved by research into incrementality of analysis and transformations, and the application of origin tracking.

134	**Modeling elements**
103	core model elements
1298	**Rules and Strategies**
459	in code generation stage
318	in model-to-model stage
277	in typechecking
126	rules for access control
	webdsl.org application
2366	lines in webdsl.org model
38395	lines of generated Java code
39216	lines of generated JSF code

Fig. 13. WebDSL statistics

Acknowledgments. This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution.*

References

1. Kraus, A.K.A., Koch, N.: Model-driven generation of web applications in UWE. In: Model-Driven Web Engineering (MDWE 2007), Como, Italy (July 2007)
2. AndroMDA.org. AndroMDA documentation (2007), http://galaxy.andromda.org

3. Bézivin, J.: On the unification power of models. Software and System Modeling 4(2), 171–188 (2005)
4. Bézivin, J.: Model Driven Engineering: An Emerging Technical Space. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 36–64. Springer, Heidelberg (2006)
5. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.16. Components for transformation systems. In: Partial Evaluation and Program Manipulation (PEPM 2006), Charleston, South Carolina. ACM, New York (2006)
6. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fund. Informaticae 69(1–2), 123–178 (2006)
7. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
8. Efftinge, S., Friese, P.: openArchitectureWare (2007), http://www.eclipse.org/gmt/oaw
9. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
10. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Generative programming and component engineering (GPCE 2006), pp. 249–254. ACM Press, New York (2006)
11. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
12. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: Companion to OOPSLA 2006, pp. 602–616. ACM Press, New York (2006)
13. Brambilla, P.F.M., Comai, S., Matera, M.: Designing web applications with WebML and WebRatio. In: Rossi, G., et al. (eds.) Web Engineering: Modelling and Implementing Web Applications. Human-Computer Interaction Series. Springer, Heidelberg (2007)
14. Mens, T., van Gorp, P.: A taxonomy of model transformation. In: Graph and Model Transformation (GraMoT 2005), vol. 152, pp. 125–142 (March 2006)
15. Pastor, V.P.O., Fons, J.: OOWS: A method to develop web applications from web-oriented conceptual models. In: Web Oriented Software Technology (IWWOST 2003), pp. 65–70 (2003)
16. Object Management Group (OMG). Meta object facility (MOF) core specification. OMG available specification. Version 2.0 (January 2006), http://www.omg.org
17. Cáceres, B.V.P., Marcos, E.: A MDA-Based approach for web information system development. In: Proceedings of Workshop in Software Model Engineering (2003)
18. Pierre-Alain Muller, F.F., Studer, P., Bézivin, J.: Platform independent web application modeling and development with Netsilon. Software and Systems Modeling 4(4), 424–442 (2005)
19. The Apache Foundation. Velocity User Guide (2007), http://velocity.apache.org/engine/devel/user-guide.html
20. van Deursen, A., Klint, P., Tip, F.: Origin tracking. Journal of Symbolic Computation 15(5/6), 523–545 (1993)
21. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
22. Visser, E.: Meta-programming with Concrete Object Syntax. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, Springer, Heidelberg (2002)
23. Visser, E.: Program Transformation with Stratego/XT. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 216–238. Springer, Heidelberg (2004)

24. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Generative and Transformational Techniques in Software Engineering (GTTSE 2007). LNCS. Springer, Heidelberg (2008)
25. Voelter, M., Groher, I.: Handling variability in model transformations and generators. In: Domain-Specific Modeling (DSM 2007) (2007)
26. Warmer, J.B., Kleppe, A.G.: Building a flexible software factory using partial domain specific models. In: Domain-Specific Modeling (DSM 2006), Portland, Oregon, USA, pp. 15–22 (October 2006)