Softw Syst Model (2010) 9:375–402
DOI 10.1007/s10270-009-0136-1

SPECIAL SECTION PAPER

Code generation by model transformation: a case study in transformation modularity

Zef Hemel · Lennart C. L. Kats ·
Danny M. Groenewegen · Eelco Visser

Received: 11 November 2008 / Revised: 5 September 2009 / Accepted: 24 September 2009 / Published online: 6 November 2009

Abstract The realization of model-driven software development requires effective techniques for implementing code generators for domain-specific languages. This paper identifies techniques for improving separation of concerns in the implementation of generators. The core technique is *code generation by model transformation*, that is, the generation of a structured representation (model) of the target program instead of plain text. This approach enables the transformation of code after generation, which in turn enables the extension of the target language with features that allow better modularity in code generation rules. The technique can also be applied to 'internal code generation' for the translation of high-level extensions of a DSL to lower-level constructs within the same DSL using model-to-model transformations. This paper refines our earlier description of code generation by model transformation with an improved architecture for the composition of model-to-model normalization rules, solving the problem of combining type analysis and transformation. Instead of coarse-grained stages that alternate between normalization and type analysis, we have developed a new style of type analysis that can be integrated with normalizing transformations in a fine-grained manner. The normalization strategy has a simple extension interface and integrates non-local, context-sensitive transformation rules. We have applied the techniques in a realistic case study of domain-specific language engineering, i.e. the code generator for WebDSL, using Stratego, a high-level transformation language that integrates model-to-model, model-to-code, and code-to-code transformations.

Keywords Transformation · Transformation engineering · Term rewriting · Webapplication DSL · Combination of analysis and transformation

Communicated by Jeff Gray, Alfonso Pierantonio, and Antonio Vallecillo.

Z. Hemel · L. C. L. Kats · D. M. Groenewegen · E. Visser (✉)
Software Engineering Research Group,
Delft University of Technology, Delft, The Netherlands
e-mail: visser@acm.org, E.Visser@tudelft.nl

Z. Hemel
e-mail: Z.Hemel@tudelft.nl

L. C. L. Kats
e-mail: L.C.L.Kats@tudelft.nl

D. M. Groenewegen
e-mail: D.M.Groenewegen@tudelft.nl

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 [36,53,54]. The essence of the approach is to shift the knowledge about these implementation details from the minds of programmers to the templates of the *code generators* that automatically translate models into implementations.

While model-driven software development should make developing end-user applications easier, the effort of understanding and appropriately using the implementation platforms is shifted to *domain-specific language engineering*, that is, the design and implementation of domain-specific languages (DSLs). DSLs must incorporate the implementation patterns of a complete application domain rather than those sufficient for a single application, possibly amounting to a significantly larger effort. This effort can be amortized by using a language in many projects, and/or by reducing the cost of creating and maintaining domain-specific languages and their generators. Thus, effective methods and techniques for domain-specific language engineering are crucial for realizing model-driven software development. A DSL engineering approach should not only reduce the effort of the initial creation of a DSL, but also the effort needed for maintenance tasks such as the adaptation to a new implementation platform, or the extension with new abstractions.

As a realistic case study in the application of transformation techniques to the systematic development of domain-specific languages for model-driven software development, we are developing WebDSL, a domain-specific language for modeling web applications with a rich data model. In earlier work we described the *process* of designing WebDSL, to contribute to a method for systematic development of new DSLs [62].

In this paper, we describe the *techniques* for *improving separation of concerns* in the implementation of code generators for domain-specific languages, based on our experience with the implementation of WebDSL. The core technique is *code generation by model transformation*, that is, the generation of a structured representation (model) of the target program instead of plain text. This approach enables the transformation of code after generation, which in turn enables the extension of the target language with features that allow better modularity in code generation rules. The technique can also be applied to 'internal code generation' for the translation of high-level extensions of a DSL to lower-level constructs within the same DSL using model-to-model transformations.

This paper refines our earlier description of the *code generation by model transformation* approach [26] with an improved architecture for the composition of model-to-model normalization rules. In particular, in our previous paper [26] we discussed the difficulty of combining type analysis and rewriting. Instead of coarse-grained stages that alternate between normalization and re-application of type analysis, we have developed a new style of type analysis that can be integrated with normalizing transformations in a fine-grained manner. The new normalization strategy has a simpler interface, which makes it easier to extend with rules for new language extensions. Furthermore, the strategy integrates non-local, context-sensitive transformation rules for aspect weaving to support separation of concerns *in WebDSL.*

1.1 Outline

In the next section, we give a brief introduction to WebDSL. In Sect. 3, we discuss the architecture of the implementation of WebDSL. The core of the approach is based on *code generation by term rewriting* (Sect. 4), which employs term rewrite rules with concrete object syntax to transform DSL models to code models. The implementation of the approach is based on Stratego/XT, a language and toolset for program transformation [10,60]. 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* [59] 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. In Sect. 5 we show how this can be used to *extend the target language* to make it better suited for code generation. For example, we have created an extension of Java with identifier composition, interface extraction, and partial classes and methods to simplify code generation rules.

The WebDSL language described in Sect. 2 provides basic abstractions for data models, user interfaces, and actions. While this provides a significant abstraction from the implementation platform, the language still requires repetive code and does not provide optimal separation of concerns. In Sect. 6 we discuss high-level abstractions for web applications that can be expressed by transformation to the base language. We illustrate this using examples of modules, modular data model definitions, user interface templates, high-level polytypic user-interface elements, declarative access control rules [25], and workflow procedures [27], which can all be implemented using model-to-model transformations.

The extended language is implemented by means of *compilation by normalization*, a transformation process in which high-level constructs are gradually transformed to constructs of the base language. The implementation of these transformations often requires more than simple local-to-local rewrite rules. Sect. 7 discusses the implementation of model-to-model transformations using rewrite rules, and discusses techniques for realizing local-to-global and global-to-local transformations such as needed for aspect weaving.

The implementation of the range of language extensions requires the composition of the model-to-model transformations that implement them. A natural approach is to stage the transformations according to level of abstraction, that is, starting with the highest level constructs, gradually transform down to the base language level. However, it turns out that interaction between transformations makes staging of normalization rules cumbersome. In particular, the interaction between type analysis and normalization requires many re-applications of type analysis in a staged setting [26]. In Sect. 8 we present an approach for combining model-to-model transformations into a single normalization strategy that is modularly extensible with rules for new language constructs. In Sect. 9 we present an approach for fine grained

Fig. 1 Blog data model

```
entity Blog {
  title   :: String[1] (name)
  entries <> List[2]<BlogEntry>
  authors -> Set[3]<User>
}
entity User {
  username :: String
  password :: Secret
}
```

```
entity BlogEntry {
  blog    -> Blog (inverse[4]=Blog.entries)
  title   :: String (name)
  author  -> User
  created :: Date[5]
  content :: WikiText[6]
}
```

Fig. 2 Blog page definitions

```
define page blog(b : Blog) {
  title { outputString[7](b.title) }
  section[8] {
    header[9] { outputString(b.title) }
    for[10](entry : BlogEntry in b.entries order by entry.created desc) {
      section {
        header { outputString(entry.title) }
        par { "by " outputString(entry.author)
              "at " outputDate(entry.created) }
        par { outputWikiText(entry.content) }
        navigate(editBlogEntry(entry))[11] { "Edit" } } }
    navigate(createBlogEntry(b)) { "New Blog" } } }
```

Fig. 3 Blog data manipulation

```
define page createBlogEntry(b : Blog) {
  title { "Create Blog Entry for " outputString(b.title) }
  var BlogEntry be[12] := BlogEntry {
    author := securityContext.principal
    be.blog := b }
  section {
    header { "Create Blog Entry for " outputString(b.title) }
    form[13] {
      table {
        row { "Title:" inputString(be.title)[14] }
        row { "Content:" inputWikiText(be.content) } }
      action[15]("Save", save())  } }
  action save[16]() {
    be.created := now();
    be.save();
    return blog[17](b); } }
```

combination of type analysis and transformation, based on a refactoring of the type analysis strategy.

2 WebDSL

WebDSL is a textual domain-specific language for the implementation of web applications with a rich data model. The language provides sublanguages for the specification of data models, for the definition of custom pages for viewing and editing objects in the data model, and for the manipulation of data. Manipulation of data is defined by actions which are contained in pages. This section describes each of these sublanguages using the implementation of a small blogging application as illustration (Figs. 1, 2, 3).

2.1 Data model

A data model specification introduces *entity* definitions (e.g., `Blog`, `BlogEntry`, and `User` in Fig. 1[1]), consisting of *properties* with a name and a type. Types of properties are either value types (indicated by `::`) or associations to other

[1] The code examples in this paper, contain callouts to help the reader. These underlined superscripts connect code fragments in the text to code fragments in the example.

entities defined in the data model. Value types are basic data types such as `String`[1] and `Date`[5], but also domain-specific types that carry additional functionality. For example, the `WikiText`[6] type implies the use of a wiki rendering engine on display of a value of the type. Associations are composite (the referer owns the object, indicated by <>) or referential (the object may be shared, indicated by ->). One-to-many and many-to-many relationships between entities are defined through the use of the `Set`[3] and `List`[2]types. The `inverse`[4]annotation on a property declares a relation with automatic synchronization of two properties.

2.2 User interface

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 definition `blog(b : Blog)` in Fig. 2 defines a page showing all blog entries for blog `b`. WebDSL provides basic markup operators such as `section`[8] and `header`[9] 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 elements such as `outputString`[7]. Collections of data can be presented using the iterator construct `for`[10], which can filter and sort the elements of a collection. Navigation among pages is realized using the `navigate`[11] element, which takes a page with parameters and a link text as arguments.

2.3 Data operations

Data can be manipulated by declaring page actions. The sublanguage used in these page actions is a Java-like imperative language with a simple API. The `createBlogEntry` page in Fig. 3 shows a page that allows the user to create blog entries for a specific blog. Data input elements inside a `form`[13], such as `inputString`[14], are used to let the user enter information. The `save`[16] action stores the blog entry. Execution of an action can result in page navigation, initiated by the `return`[17] statement. An `action`[15] element in a page is used to connect a button with the caption in its first argument to the action call in its second argument. This page also shows that pages can have variables to be used for initialization of data, in this case the var `be`[12]of type `Blog-Entry`.

3 Implementing WebDSL by code generation

The architecture of the WebDSL generator follows the four-level model organization presented by Bézivin [7] as illustrated in Fig. 4. At the M_3 level we define the SDF meta-metamodel, which is the grammar of the Syntax Definition

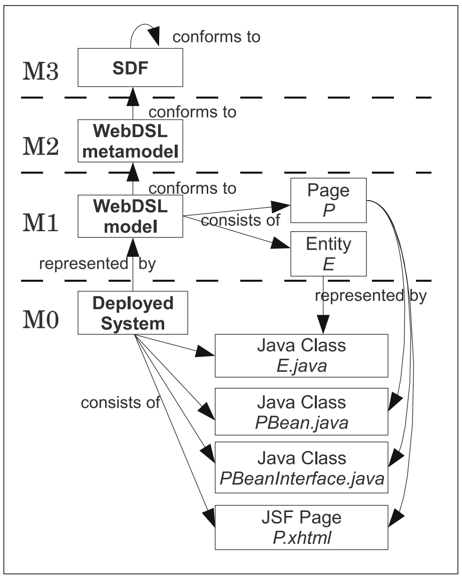


Fig. 4 Organization of models and artifacts of the WebDSL generator

Formalism SDF, which is defined in (and thus conforms to) itself [58]. At the M_2 level we define the WebDSL metamodel, i.e., the grammar of WebDSL defined in SDF. At the M_1 level we define WebDSL models of web applications, consisting of entity and page definitions. At the M_0 level we define the 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 [62], the M_0 systems were initially based on the Java/Seam architecture, consisting of high-level application frameworks, such as the Java Persistence API (JPA) [19], JavaServer Faces (JSF) [13], and the Seam web framework [37]. Currently, alternative back-ends generating plain Java servlets [16] and Python (for the Google AppEngine platform [1]), are under development. In this paper, we will only consider the original Java/Seam back-end. The other back-ends use similar techniques even if the details of the transformations are different.

For each entity definition, a corresponding entity class is generated with fields, accessors, and mutators for the properties of the entity, annotated for object-relational mapping (ORM) according to the JPA. Figure 5 shows an example transformation of a WebDSL entity `Blog`[18]to a JPA entity Java class `Blog`[21]. The `title`[19] property is transformed to a class property `_title`[22], the accessor `getTitle`[23], and the mutator `setTitle`[24]. The second property, `entries`[20], also generates annotations to define static mapping to tables (`@OneToMany`[25]) and dynamic behaviour such as save cascading [`@Cascade(SAVE_UPDATE)`[26]].

For each page definition, a JSF XHTML page, an Enterprise JavaBeans (EJB) [18] session bean, and the required

Fig. 5 Transformation of a WebDSL entity definition to a JPA class

```
entity Blog[18] {
  title[19]:: String
  entries[20]<> List<BlogEntry>
}
```

⇒

```
@Entity public class Blog[21] {
  protected String _title[22] = "";
  public String getTitle()[23] {
    return _title;
  }
  public void setTitle(String value)[24] {
    _title = value;
  }
  @OneToMany[25] @Cascade(SAVE_UPDATE)[26]
  protected List<BlogEntry> _entries;
  public List<BlogEntry> getEntries() {
    return _entries;
  }
  public setEntries(List<BlogEntry> b) {
    _entries = b;
  }
}
```

Fig. 6 Transformation of a WebDSL page to a JSF page with a backing bean

```
define page editBlogEntry(e[27] : BlogEntry)[28] {
  form { table { row { "Title:" inputText(e.title) }
                 row { "Content:" inputWikiText(e.content) }
                 action("Save", save())[29] } }
  action save()[30] { e.save(); } }
```

⇓

```
<html><body>
<h:form><table>
  <tr><td><h:outputText value="Title: "/></td>
      <td><h:inputText value="#{editBlogEntry.e.title}"[31]/></td></tr>
      <tr><td><h:outputText value="Content: "/></td>
      <td><h:inputTextarea value="#{editBlogEntry.e.content}"/></td></tr>
</table><h:actionLink[32] action="#{editBlogEntry.save()}"/></h:form>
</body></html>
```

```
@Stateful @Name("editBlogEntry")[33]
public class EditBlogEntryBean implements EditBlogEntryBeanI {
   @In @Out private BlogEntry e;[34]
   public void setE(e) { this.e = e; }
   public BlogEntry getE() { return this.e; }
   public String save()[35] { entityManager.persist(e); } }
```

```
@Local[36] public interface EditBlogEntryBeanI {
   public void setE(e);
   public BlogEntry getE();
   public String save(); }
```

interface are generated. Figure 6 shows the WebDSL page definition `editBlogEntry`[28] with its transformation to JSF and Java. The page's arguments become properties of the session bean which make them available to the JSF Expression Language (EL). In this example the WebDSL argument `e`[27] is transformed to the java property `e`[34] which can be used in JSF EL expressions, e.g. to show its title with `editBlogEntry.e.title`[31]. The connection to the correct session bean is made by referring to the name specified in the `@Name`[33] Java annotation, in this case `editBlogEntry`. The `save`[30] action defined in the page becomes the `save`[35] method of the session bean. The action can be executed using the `actionLink`[32] JSF element which is generated from the `action`[29] call in the WebDSL page. Finally, an interface is generated for the session bean which makes the methods available and specifies that the bean can be accessed locally[36].

WebDSL is a textual, domain-specific language and its M_2 metamodel is a grammar describing the valid sentences of that language. From the grammar, we automatically generate a parser that transforms the textual representation of a model to an abstract syntax tree (AST). The AST conforms to a regular tree grammar, another M_2 metamodel 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 low-level 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/XT [10], which can apply transformations to any models with an abstract syntax definition.

In the following sections we discuss the organization of the generator as a pipeline of model-to-model transformations, and the idioms used to realize these transformations.

4 Code generation by term rewriting

Most model-driven engineering approaches accomplish code generation by simply writing strings to text files. Sometimes template engines are used to make this process easier. In contrast, in the *code generation by model transformation* approach, code generation is just another model transformation step. Rather than printing strings to a file, source and target models are represented by means of first-order terms, and code generation is expressed by means of term rewriting. For the implementation of WebDSL we use Stratego [10,60], a transformation language with a combination of features that makes it suitable for code generation by model transformation. This section explains code generation by term rewriting and compares it with code generation using template engines as used in other MDE approaches.

4.1 Representing models and code with terms

String template engines such as Velocity [56], StringTemplate [48], and xPand [64] use templates to generate fragments of plain text, which cannot be checked statically for syntactic correctness. Only after a complete output file has been generated for a particular input is it possible to determine if the generated code is syntactically correct.

Stratego rewrite rules operate on a structured representation, using first-order terms to represent models as a tree structure. Any patterns and fragments using this representation can be statically checked for syntactic correctness. For example, consider the following WebDSL entity declaration and its term representation:

```
entity Blog {
  ident :: String
}
```

$\Rightarrow$

```
Entity("Blog", [
  Property("ident", SimpleSort("String"))
])
```

This term corresponds to a tree structure, with as root an `Entity` node, with as children the term representations of the name and properties of the entity declaration. Using the WebDSL meta-model, the structure of this tree can be statically checked for well-formedness. Similarly, Java code can be represented using a term representation. For example, consider the following (tiny) Java class and its term representation:

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

$\Rightarrow$

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

The essence of code generation by model transformation is to represent both the input model and the generated code as structured terms. This requires meta-models for the source language (WebDSL) as well as for the target languages (Java and XML). Using structured representations for input and output makes it possible to divide a large transformation into several, smaller transformation steps. We elaborate on this technique in Sect. 5.

4.2 Rewrite rules

The elementary operations of a transformation 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 of `p1` and the condition `s` succeed, 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 (represented as a term pattern) to a fragment of the target language on the right-hand side (represented as a term pattern). For example, the `entity-to-class` rewrite rule in Fig. 7, rewrites a WebDSL entity declaration to a Java class, using terms to specify both the left-hand side and the right-hand side of the rule. The rewrite rule rewrites the input entity to a class where

```
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(),[]))]))
```

Fig. 7 Rewrite rule generating Java code with abstract syntax

the name *x* of the entity is used as the name of the class and its constructor. For brevity, this rule does not consider the inclusion of generated properties.

4.3 Concrete object syntax

While we will argue that it is useful to have a structured representation of generated code, the right-hand side pattern of the rule in Fig. 7 is not very easy to produce or understand, due to the complexity of the abstract syntax of Java. A language's concrete syntax is usually more concise and more familiar than its abstract syntax. Therefore, Stratego supports *concrete object syntax patterns* [59] in the definition of transformation rules. That is, textual patterns in the syntax of the language concerned that are compiled to the corresponding structured representation pattern. This provides the same level of readability as template engines, while guaranteeing syntactic correctness of code patterns. For example, using concrete object syntax, the `entity-to-class` rule in Fig. 7 can be written as follows:

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

Note that *x* and *prop** are recognized as *meta-variables* for identifiers and lists of properties, respectively. In this paper, we will indicate meta-variables using italics, to distinguish them from identifiers in the subject language. The implementation details are explained elsewhere [59] and are not in the scope of this paper.

A more realistic translation of entity declarations is defined in Fig. 8. Rule `entity-to-class` generates a Java class for an entity declaration. The rule uses the auxiliary rule `property-to-java` to generate the class body declarations (*cbds2**) defining the field and accessors for a property of an entity. Only the case of properties with value type is shown here.

4.4 Rewriting versus template engines

The table in Fig. 9 summarizes the differences between the use of rewriting with concrete syntax and template engines such as Velocity [56] and xPand [64] for code generation. The approaches have in common that the concrete syntax of the target language is used to define (parameterized) code fragments, which makes it easier to define such fragments than with the use of abstract syntax. The approaches use different methods to instantiate the holes in a code fragment. Rewriting uses pattern matching and meta-variables, whereas template engines typically use object accessors inside anti-quotations. This is mainly a difference in programming style, which is not easy to compare. In order to parse code patterns in concrete object syntax, a grammar of the target language is necessary. Since template engines just produce text, they can be applied flexibly for different languages. On the other hand, the lack of a grammar entails no syntactic checks of the code patterns. Finally, the rewriting approach produces a structured representation, while template engines produce flat text. A structured representation means that the target code can be pretty-printed automatically. However, pretty-printers are available for mainstream languages, so this is not necessarily a concern for the use of template engines. The main difference that we are interested in for the purpose of this paper is the fact that the rewriting approach produces a structured representation of the target code, which entails that further transformations can be applied, as we will see in the following sections.

4.5 Composing generation rules

In Stratego, rewrite rules can be composed using *programmable strategies* that control the application of individual rules [10,63]. Stratego provides a few basic combinators for composing transformations from rules. For example, the combinator `s1 ; s2` produces the *sequential composition* of the transformation strategies `s1` and `s2`, and the combinator `s1 <+ s2` produces the *deterministic choice* of `s1` and `s2`. More complex strategies can be constructed from these basic combinators. For example, the strategy definitions

```
try(s)    = s <+ id
repeat(s) = try(s; repeat(s))
```

are part of the Stratego Standard Library. The strategy `try(s)`, tries to apply transformation `s`, but succeeds by producing the original term when `s` fails. The strategy

Fig. 8 Rewrite rules for generating Java entity class from WebDSL entity declaration

```
entity-to-class :
  |[ entity x_ent { prop* } ]| ->
  |[ @Entity class x_ent {
       public x_ent() { }
       cbds*
       ...
     } ]|
  where cbds* := <map(property-to-java)> prop*
      ...
property-to-java :
  |[ x :: srt ]| ->
  |[ private t x_field = e;
     public t x_get() { return x_field; }
     public void x_set(t value) { x_field = value; }
  ]|
  where t       := <java-type> srt
      ; e       := <initialization-expression> srt
      ; x_field := <concat-strings> ["_", x]
      ; X       := <capitalize> x
      ; x_get   := <concat-strings> ["get", X]
      ; x_set   := <concat-strings> ["set", X]
```

Fig. 9 Comparison of rewriting with concrete syntax and textual template engines

Rewriting		**Template Engine**	
+	use syntax of target language	+	use syntax of target language
+	pattern matching	+	object accessors
-	requires grammar	+	flexible: no grammar needed
+	static syntactic checks	-	no static syntactic checks
+	structured representation	-	generation of text
+	automatic pretty-printing	+/-	use existing pretty-printers
+	transformation after generation	-	no transformation after generation

`repeat(s)`, repeatedly applies a transformation `s` until it fails. Traversal strategies are defined using generic traversal operators, and can specify a visit sequence that works on any tree rather than for a particular tree grammar. For example,

```
topdown(s) = s; all(topdown(s))
```

defines a strategy that applies a transformation `s` to all nodes in a tree during a top-down traversal. An application-specific library can collect custom strategy definitions.

Figure 10 shows a basic definition of the `webdsl-to-seam` transformation strategy that transforms a WebDSL application consisting of a list of WebDSL entity and page definitions to Java classes and JSF code. The strategy is a pipeline of transformations that are applied in sequential order to the input model. The `typecheck` transformation checks the consistency of declarations and their uses and annotates identifiers with (a reference to) their declaration or type. The `generate-code` transformation uses `GenerateCode` to map individual WebDSL definitions to XML- or Java class models. The `GenerateCode` strategy is defined using multiple definitions, which is a convenient way to define an extensible composition of alternatives. Thus, the definition of `GenerateCode` in Fig. 10 is equivalent to

```
GenerateCode = entity-to-class + page-to-class
```

that is, the non-deterministic composition of the bodies of the definitions. The non-deterministic composition `s1 + s2` of two strategies entails that either the composition `s1 <+ s2` or the composition `s2 <+ s1` is used. In other words, the order in which the alternaves are tried is undefined.

Finally, the `write-to-file` strategy pretty-prints the Java class and XML models, and writes them to a file.

5 Transforming generated code

Most programming languages are not designed as targets for code generation, which is manifest especially in the lack of composition operators for program fragments of all kinds. For example, identifiers are typically just strings of alphanumeric characters without operators for composing identifiers from smaller strings. Similarly, classes and methods (in Java at least) cannot be assembled from smaller class and method fragments. This lack of compositionality in target languages is reflected in a corresponding lack of compositionality in code generation rules, as is illustrated by the rules in Fig. 8; a rule generating a Java class must produce all ingredients of that class. Since rewrite rules produce a structured representation of the target program, it is possible to apply further

Fig. 10 A basic code generation pipeline

```
strategies
  webdsl-to-seam = typecheck; generate-code; write-to-file
  generate-code  = map(GenerateCode)
  GenerateCode   = entity-to-class
  GenerateCode   = page-to-class
```

transformations to generated code. In this section, we show three example extensions of the target language Java with composition operators that enable better modularization of code generation rules.

5.1 Identifier composition

Code generation often requires the creation of many new identifiers based on identifiers in the source model. Since identifiers in Java (and most programming languages) are simple alphanumeric strings, creation of identifiers requires string manipulation and concatenation. For example, the generation of a field, and a setter and getter method in the `property-to-java` rule in Fig. 8 requires three string concatenations and one string manipulation. Even while this is straightforward code, it takes up quite a bit of real estate in the generation rules, and it is repetitive boilerplate code. To avoid this problem we extended the Java language with the # operator, which composes its two operand identifiers into a single identifier following Java's naming conventions. Thus, `get#name` becomes `getName` and `_#name` becomes `_name`. The new version of the `property-to-java` rule from Fig. 8 using this feature is presented in Fig. 11. The # operator is implemented by a transformation that replaces composite identifiers by regular Java identifiers. The Java extension and the transformation can be reused in all code generators that produce Java code.

5.2 Partial classes and methods

Since its conception, the WebDSL generator has grown considerably. 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). The `entity-to-class` rule in Fig. 8 is a typical starting point for growing such a God rule. As new language extensions are added, these rules grow to a size that no longer fits on a single screen. Thus, this pattern is a code smell that hinders the extensibility and maintainability of the generator.

The employment of God rules is the result of the structure of the target meta-model: Java does not support composition of classes. Other platforms, such as C#, provide partial classes (but not partial methods), which allow subdividing 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 with partial classes and methods. Our extension uses Java's annotation syntax to identify partial classes and methods with the annotation `@Partial`. For example, Fig. 12 shows a refactoring of the generation rules of Fig. 8, in which the `entity-to-java` and `property-to-java` rules independently generate partial classes. Methods can be declared as partial using the same annotation:

```
@Partial void initialize() { stm* }
```

Since code patterns are no longer assembled by a God rule, partial code fragments are emitted to a code repository using rules such as `emit-java-class`. The code fragments are collected and assembled by the `merge-partial-classes` strategy. Partial classes with the same name (within the same Java package) are merged into a single Java class. Similarly, the bodies of partial methods with the same name (in the same class) are merged into a single method definition. The ordering of statements originating from different partial methods is non-deterministic. Hence, the generator should make no assumptions on such an ordering. However, there is a simple technique to enforce some order. If there are several classes of statements such that all statements in one class should be executed before all statements in another class, the partial method can be refactored into a regular method that calls for each stage a partial method to which the statements of the various classes can be added.

5.3 Interface extraction

Seam and EJB require that each session bean implement an interface containing all public methods of that bean. Generating code for these types of interfaces is tedious, since it requires generation rules that shadow the generation rules for the regular class. Therefore, we extended Java to generate these interfaces automatically. The `@RequiresInterface` annotation for a class indicates that a separate interface should be derived from the class including signatures of all its public methods. Again, the extension is implemented

Fig. 11 Code generation rule using identifier composition operator #

```
property-to-java :
  |[ x :: srt ]| ->
  |[ private t _#x = e;
     public t get#x() { return _#x; }
     public void set#x(t value) { _#x = value; }
  ]|
  where t := <java-type> srt
      ; e := <initialization-expression> srt
```

Fig. 12 Generation rules emitting partial classes and methods

```
entity-to-java :
  |[ entity x_ent { prop* } ]| ->
  <emit-java-class> |[
    @Entity @Partial class x_ent {
      public x_ent() { }
    } ]|
property-to-java :
  |[ x :: srt ]| ->
  <emit-java-class> |[
    @Partial class x_ent {
       private t _#x = e;
       public t get#x() { return _#x; }
       public void set#x(t value) { _#x = value; }
     } ]|
  where x_ent := <entity>
      ; t := <java-type> srt
      ; e := <initialization-expression> srt
```

by means of a (library) transformation that carries out the derivation of the interface. Thus, for a page bean generated with the rule in Fig. 13, a corresponding interface `x#PageBeanInterface` is generated as well.

5.4 A revised pipeline

Figure 14 shows a revision of the generator pipeline presented in Fig. 10, incorporating the new code generation technique. Since the `GenerateCode` rules apply not only to top-level definitions, the `map` strategy is no longer sufficient. The new `generate-code` strategy uses the generic `topdown` strategy to traverse the WebDSL model, applying the `GenerateCode` rules to each node of the input model. The succesful applications of `GenerateCode` contribute to a repository of code fragments, which are assembled by `merge-partial-classes`.

6 Model-to-model transformations

The WebDSL language as described in the preceding sections provides basic abstractions for web applications. As the language evolved, we have added new features to achieve higher levels of abstractions, providing better support for particular kinds of web applications and application domains. For some of these abstractions, special support from the platform is needed. However, many can be implemented by incremental model transformation steps, transforming rich WebDSL models to more primitive WebDSL models that only support the feature set of the WebDSL *core* language as described in Sect. 2. The advantage of transforming higher level abstractions to low level core models rather than generating implementation code from them directly is that the domain-specific core language that has to be mapped to implementation target code remains small, increasing portability of the generator. The development of the Servlet and Python back-ends for WebDSL took little time because the WebDSL core language is relatively small.

This section discusses several examples of abstractions we have added to WebDSL and implemented by means of model-to-model transformations. In the next sections we discuss the issues that arise in the implementation of such transformations and the techniques developed to address them. The rationale for the introduction of a core language for WebDSL is discussed more extensively by Visser [62].

6.1 Modules

In the WebDSL core language all definitions of an application have to be defined in a single file. This is not desirable,

Fig. 13 Annotating a generated class for interface extraction

```
page-to-java-bean :
  |[ define page x(param*) {elem*} ]| ->
  <emit-java-class> |[
     @RequiresInterface @Partial @Stateful
     class x#PageBean {
       @In EntityManager em;
       ...
     }
  ]|
```

Fig. 14 Code generation pipeline with partial class generation and assembly

```
strategies
  webdsl-to-seam =
    typecheck; generate-code; merge-partial-classes; write-to-file
  generate-code = topdown(try(GenerateCode))
  GenerateCode  = entity-to-java
  GenerateCode  = property-to-java
  GenerateCode  = page-to-java
  GenerateCode  = parameter-to-java
```

as it impairs maintainability and reusability of applications. A simple module system defined as an extension of the core language addresses this problem. A module is a collection of definitions and may import other modules. WebDSL modules are implemented by means of a model-to-model transformation that replaces module imports by the content of the imported module. This mechanism is similar to the `#include` pragma of the C language. The crucial difference with the C mechanism is that a WebDSL module is parsed *before* its abstract syntax tree is included in the abstract syntax tree of the importing module. Thus, it is not possible to rely on textual composition to compose WebDSL definitions, e.g. using an import in the middle of a definition.

6.2 Modular data models

To support separation of concerns it can be beneficial to spread entity declarations over different modules that deal with various aspects of an application. For example, for our blogging application we might want to reuse a separate, generic user management module with a standard `User` data model. For the purpose of the blogging application the `User` entity needs to have a set of blog entries. Rather than directly implementing this aspect in the `usermanagement` module, we define it in a separate module, using the `extend entity` construct to extend the existing `User` entity. The transformation merges the entity definition and its extensions, mapping it to a regular entity declaration in the core WebDSL language, as illustrated in Fig. 15.

6.3 Template definitions

User-defined *templates* allow the developer to define reusable chunks of user interface model. For example, the `main()` template in Fig. 16 defines a general set-up for pages (with menubar and page body) that is shared among many pages of the application. Pages can call the `main()` template and locally override parts of it. For instance, the `blog` page overrides the default `body` template. Templates can be implemented by recursively inlining the contents of their definitions into the page they are called from. Thus, in the `blog` page, the `main` template call is replaced by a `top` and `body` template call, which are subsequently replaced by a menubar and a section.

6.4 Deriving user interface elements from types

The user interface elements to be used for input and output of entity properties depend on their type. For example, input of a `String` requires a simple text input field, input of a `WikiText` requires a larger text area, while input of a `Date` requires a date picker. Thus, the core WebDSL language provides different `input` elements for different types. For example, for input of the properties of a `BlogEntry e`, we would use `inputString(e.title)`, `inputDate(e.created)`, and `inputWikiText (e.content)`. Since the input element to be used depends on the type of the property, we can simplify the specification of inputs to just `input(e)`, and derive the particular kind

Fig. 15 Joining modular data model definitions

```
module usermanagement
  entity User {
    username :: String
    password :: Secret
  }
module blogentries
  imports usermanagement
  extend entity User {
    entries -> Set<BlogEntry>
  }
```

⇒

```
entity User {
  username :: String
  password :: Secret
  entries -> Set<BlogEntry>
}
```

Fig. 16 Inlining template calls

```
define main() {
  top()
  body()
}
define top() {
  menubar { ... }
}
define page blog(b : Blog){
  main()
  define body() {
    section{
      header{output(b.name)}
      ... } }
}
```

⇒

```
define page blog(b : Blog){
  menubar { ... }
  section{
    header{output(b.name)}
    ...
  }
}
```

from the type of the expression using a model-to-model transformation. Thus, `input(e.title)` is transformed to `inputString(e.title)`, `input(e.created)` to `inputDate(e.created)`, etc.

A recurring pattern in user interfaces are tabular forms for input of the properties of an entity. Such forms can be derived from an entity declaration by considering its properties. WebDSL provides a collection of `derive` statements for deriving different page configurations from data model declarations. For example, the `derive editPage` statement produces a complete edit page for a particular entity, as illustrated in Fig. 17.

6.5 Access control

Most web applications need to adhere to a certain access control policy that defines the permissions of the various users. Such a policy can be encoded in the application using conditional checks defined in the protected resource, such as a page. If access is not permitted the user is notified by a redirect to an error page. This solution requires that access control checks are entangled with template definitions, which makes the policy encoded in the application hard to verify or modify. The WebDSL access control sublanguage [25] supports definition of access control policies as a separate concern. Access control rules are defined in a separate section (or module) in the application, and are woven into the corresponding WebDSL definitions during compilation.

The example in Fig. 18 illustrates the weaving of access control rules. The page `editBlogEntry` is protected by a rule that matches the signature of the page (i.e. the name and type of arguments) and specifies the condition for allowing access. The condition verifies that the currently logged in user, the '`principal`', is the author of the blog entry. The condition is woven into the page definition by a transformation, creating an `init` action which performs the check and redirects if necessary. Such an action is executed before each request to the page it is defined in.

6.6 Workflow

Another recurring aspect of web applications is workflow. Workflow is concerned with the coordination of activities performed by participants involving artifacts. Workflows can be encoded using low-level constructs in WebDSL applications, however these encodings give little insight into the structure of such workflows. Therefore, we have developed WebWorkFlow [27], an extension of WebDSL supporting the definition of tasks on entities and the order in which these have to be performed using process expressions. Although the specifics of this extension are beyond the scope of this paper, it has also been implemented as a model-to-model transformation. Workflow process descriptions are translated to procedures, which are then translated to a combination of entity extensions, access control rules, actions, and page and template definitions [27].

Fig. 17 Derivation of edit page based on data model

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

⇓

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

Fig. 18 Access control rule transformation

```
define page editBlogEntry(e : BlogEntry) { section { ... } }
```

```
rule page editBlogEntry(be : BlogEntry) { principal == be.author }
```

⇓

```
define page editBlogEntry(e : BlogEntry) {
  init{ if(!(principal == e.author)) { redirect accessDenied(); } }
  section { ... }
}
```

7 Implementing model-to-model transformations

In the previous section, we have illustrated how high-level abstractions can be implemented by means of model-to-model transformations to the core language. These transformations can be implemented by means of the same techniques we employed for code generation in Sect. 3, i.e. rewrite rules with concrete syntax and transformation of generated 'code'. Using concrete syntax, it is feasible to handle large code templates, while its underlying structured representation enables cascading transformations after generation. Thus, Stratego unifies model-to-code and model-to-model transformation, avoiding the need for different languages for different types of transformations.

Van Wijngaarden and Visser [66] give a classification of mechanisms for program transformation, distinguishing the *scope* of a transformation (which part of a program it affects or is affected by), its *direction* (whether it is triggered by the source or the target), and the number of *stages* it requires. In the WebDSL generator three classes of transformations are used that differ in *scope*.

- *local-to-local transformations*, which locally transform one term to another
- *global-to-local transformations*, which retrieve information from the surrounding (global) context to perform a local transformation
- *local-to-global transformations*, which retrieve information from a local term for use in the surrounding context

In this section, we discuss the definition of transformations in these classes. In the next section, we consider the composition of such transformations.

7.1 Local-to-local

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 transformations.

A local-to-local rewrite replaces a model fragment with another fragment without using or producing other parts of the model, as illustrated by the examples in Fig. 19. The `normalize-text` 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, `text(blog.title, ": ", blog.author)` is reduced to `text(blog.title) text(": ") text(blog.author)`. Similarly, the `normalize-for` rule rewrites an occurrence of

Fig. 19 Local-to-local syntactic normalization rules

```
normalize-text :
  |[ text(e1,e2,e*) ]| -> |[ text(e1) text(e2,e*) ]|

normalize-for :
  |[ for(x : srt in e1 order by e2) {elem*} ]| ->
  |[ for(x : srt in e1 where true order by e2) { elem* } ]|
```

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.

7.2 Global-to-local

In a global-to-local transformation, model elements are locally transformed using (global) context information. For example, a transformation may depend on the types of identifiers declared elsewhere in the model. Similarly, template calls can be implemented by inlining a template definition defined in the global context.

As an example of a global-to-local transformation consider the derivation of edit pages and input elements in Sect. 6.4, which is implemented by the rewrite rules in Fig. 20. The `derive-edit-page` rule transforms a `derive editPage from` *e* element to a complete edit page, as illustrated in Fig. 17. The transformation is driven by the type of the expression *e*. The `type-of` strategy computes the type of an expression based on declarations of types and identifiers. Given a type expression, the `entity-properties` strategy retrieves the list of properties of an entity type. For each property a row is generated by the `derive-edit-row` rule. The return address of the `save()` action is the view page declared for the entity type, which is obtained with the `view-page` strategy.

The edit rows generated by this rule make use of a generic `input` element, which is specialized with regard to its type by an appropriate `derive-input` rule. For example, an `input` for type `String`, is specialized to an `input String` template, and an `input` for a defined entity type is specialized to a drop-down menu (`select`) that allows selecting an object of that type.

7.3 Local-to-global

A local-to-global transformation can locally rewrite an element of the model, while producing elements or information for use elsewhere in the global context. This pattern can be used to implement the `extend entity` construct of Fig. 15, or similar aspect-oriented programming (AOP) constructs.

Normal rewrite rules in Stratego are *context-free*, that is, a rule locally transforms a term to another term without access to the context in which the term occurs. To express *context-sensitive* transformations, Stratego provides *dynamic rewrite rules* [11]. Dynamic rules are defined at run-time, and inherit information in the context of their definition. The global-to-local strategy `derive-entity` makes use of strategies `type-of` and `entity-properties`, which are implemented using dynamic rules to propagate context information from declarations to uses, as we will see in Sect. 9.3.

Dynamic rules can also be used to implement local-to-global transformations, as illustrated in Fig. 21, which defines a rewrite rule to implement the `extend entity` construct. The `extend-entity` rule is an example of a *non-preserving* local-to-global transformation. That is, the term is rewritten to a placeholder term `RemoveMe()`, which is removed from the model in a later transformation step. In addition, the rule defines a dynamic rule `DynamicExtendEntity`, which rewrites the base entity definition to include the additional properties provided by the `extend entity` definition (as illustrated in Sect. 6.2). The context-sensitivity of the dynamic rule stems from the fact that it inherits the bindings to identifiers from its context. In this case, the identifiers *x* and *prop1** are bound by the left-hand side match of `extend-entity`. Thus, the uses of *x* and *prop1** in the definition of `DynamicExtendEntity` refer to these terms. That is, the properties *prop1** are propagated from the extend entity declaration to the entity declaration for *x*. The `:+` in the definition of the dynamic rule indicates that there may be multiple dynamic rule definitions with the same left-hand side, for the same entity in this case. This allows the rule to support multiple extensions of the same entity declaration.

8 Transformation modularity and extensibility

Separation of concerns is a key strategy in model-driven software development. High-level, declarative models support separation of essential application properties from the code patterns used to implement them, such that the effort of development and maintenance of applications is drastically reduced. Separation of concerns is also important for the maintainability and extensibility of the *implementation* of domain-specific languages. Rapid extension and adapta-

Fig. 20 Rewrite rules to derive edit page elements

```
derive-edit-page :
  |[ derive editPage from e ]| ->
  |[ section{
       header{"Edit " srt " " text(e.name)}
       form {
         table { row* }
         action("Save", save()) }
       action save() { x.save(); return x_view(x); } }
  ]|
  where prop*   := <type-of; entity-properties> e
      ; row*    := <map(derive-edit-row(|x))> prop*
      ; x_view  := <type-of; view-page> e

derive-edit-row(|x) :
  |[ y relation srt ]| -> |[ row{x_text ": " input(x.y)} ]|

derive-input :
  |[ input(e) ]| -> |[ inputString(e) ]|
  where SimpleSort("String") := <type-of> e

derive-input :
  |[ input(e) ]| -> |[ select(s : t, e) ]|
  where t := <type-of> e; <defined-entity> t
```

Fig. 21 Merging extended entities using a local-to-global transformation

```
extend-entity :
  |[ extend entity x { prop1* } ]| -> RemoveMe()
  where rules(
          DynamicExtendEntity :+
            |[ entity x {prop2*} ]| -> |[ entity x {prop2* prop1*} ]|
        )
```

tion of a DSL is important to keep up with evolving insights and requirements.

The use of model-to-model desugaring transformations separates the implementation of high-level abstractions from the implementation of lower-level constructs. Rather than directly bridging the semantic gap between the model and its implementation in one transformation step, the model is gradually transformed. This approach is called compilation by normalization [33]. The code generator does not have to be aware of the presence of high-level constructs. The front-end can use the base language, instead of target language implementation patterns, to implement high-level constructs. Thus, this *vertical separation of concerns* realizes information hiding between compiler stages, and reduces the impact of extending the language.

Horizontal separation of concerns is information hiding within a compiler stage, designed to reduce the dependencies between transformations for different constructs. Again, it should be easy to add transformations for new abstractions without disrupting or otherwise affecting existing rules. For code generation, horizontal separation of concerns is achieved by the generation of partial artifacts, factoring out the assembly of these artifacts into a separate transformation. The `GenerateCode` rules can be extended without affecting existing rules. It is not necessary to locate an existing 'God rule' and plug into its assembly of a Java class.

In this section, we consider two architectures for the application of normalization rules from the perspective of extensibility. In the staged approach rules are divided in clusters, which are applied sequentially. In the normalization approach all rules are applied simultaneously.

8.1 Staged normalization

A natural approach to organize a DSL front-end is as a pipeline of transformation stages that are associated with language extensions, as illustrated in Fig. 22. The `webdsl-to-seam` compiler is divided into two parts. The front-end, `webdsl-to-core`, transforms applications in the extended WebDSL language to WebDSL core. The back-end, `webdsl-core-to-seam`, uses the partial class generation approach of Sect. 5. This pipeline model of the

```
webdsl-to-seam =
  webdsl-to-core
  ; webdsl-core-to-seam

webdsl-to-core =
  import-modules
  ; typecheck
  ; translate-workflow
  ; weave-access-control
  ; extend-entities
  ; derive-pages
  ; typecheck
  ; inline-templates
  ; derive-elements
  ; normalize-syntax

webdsl-core-to-seam =
  generate-code
  ; merge-partial-classes
  ; write-to-file
```

Fig. 22 Pipeline of analysis and transformation stages

transformation is similar to workflow descriptions in other model transformation solutions, such as openArchitectureWare [20].

The staging approach achieves a form of separation of concerns by combining transformations associated with a language extension in a single transformation. However, the information hiding between stages is limited. Each stage should eliminate the constructs of the language extension that it is associated with. After a stage for a particular extension has been applied, the constructs from that extension can no longer be used. Thus, the position in the pipeline determines which language constructs can be used in the generation templates of that stage. Designing the transformations for a new language extension requires an analysis of the order of the stages to determine which language constructs can be used. Furthermore, adding an extension requires modification of the central pipeline, hampering independent extensibility of the language by third party developers. For WebDSL, which composes multiple sublanguages that focus on different aspects of web applications, this phase ordering problem formed a continuous, although slight nuisance for the authors of the various language extensions.

In addition to the presence of certain language constructs, transformation stages may also depend on the results of analysis stages. For example, the application of the `derive-input` rules requires type analysis to be applied to the argument expressions of `input` elements. If such `input` elements are generated by another stage, it may be necessary to perform an additional `typecheck` stage. We consider the problem of the combination of analysis and transformation in the next section. In this section we ignore the issue and focus on an extensible architecture for normalization rules.

8.2 Innermost normalization

Desugaring rules translate high-level constructs into a combination of other constructs. If the result of a desugaring rule would always be a term in the core language, there would be no phase ordering problem. However, desugaring rules for high-level abstractions often rely on transformations for lower-level abstractions that are not core language constructs. For example, WebWorkFlow definitions are translated to a combination of entity extensions, page definitions, and access control rules, which need further desugaring before the back-end can translate them. Thus, we need to reduce the model to *normal form* with respect to the set of desugaring rules.

In the staging approach we try to find an ordering of desugaring rules that ensures that we reach this normal form. However, this requires a dependency analysis on the rules, which is not guaranteed to produce a linear ordering, as illustrated by need to reapply type analysis. The phase ordering problem can be avoided by applying all desugaring rules exhaustively using a fixpoint rewriting strategy. That is, any term that is produced as result of a transformation rule, is inspected to see if other transformation rules can be applied to it. (In Sect. 8.3 we discuss termination of this process for non-confluent rules.)

Figure 23 shows a reimplementation of the `webdsl-to-core` strategy. Instead of a sequential composition of desugaring stages, the `innermost` strategy is used to simultaneously normalize a model with respect to a set of desugaring rules. The `innermost` strategy takes as argument a transformation, typically a list of rules composed with deterministic choice (<+), which it exhaustively applies to the subject term starting with innermost nodes. That is, a term is only considered for transformation, after all its subterms have been normalized. When a rule is applied to a term, the strategy subsequently normalizes the newly produced term. Thus, right-hand sides of rules do not have to be restricted to terms in normal form. As a result it is not necessary to consider the order in which rules are applied.

To extend the new implementation of `webdsl-to-core` with desugaring rules, is simply a matter of adding rules to the argument of `innermost`. However, this still requires changing the definition of the pipeline. Extensibility is further simplified by Stratego's rule extension facility. Instead of directly passing a composition of rules to `innermost`, the strategy `desugar` is passed, which is defined by an extensible set of clauses, in the same manner as the extension of the `GenerateCode` strategy in Sect. 4.5. Defining rules in this fashion allows different desugaring

```
webdsl-to-core =
  import-modules
  ; typecheck
  ; desugar-top

desugar-top =
  innermost(
    normalize-text
    <+ normalize-for
    <+ derive-edit-page
    <+ derive-input
    <+ extend-entity
    <+ ...
  )
```

Fig. 23 Pipeline with innermost normalization

```
desugar-top =
  innermost(desugar)
```
```
desugar = normalize-text
desugar = normalize-for
```
```
desugar = derive-edit-page
desugar = derive-input
```
```
desugar = extend-entity
```

Fig. 24 Extensible definition of `desugar`

steps to be defined across different modules, each extending the `desugar` definition, as illustrated in Fig. 24. This means that different desugaring rules can be modularly defined and implicitly composed and evaluated, without the need for explicit staging. This helps in separation of concerns, and is essential for the scalability of a generator specification.

8.3 Normalization with local-to-global rules

Local-to-global transformations may define new dynamic rules that need to be applied elsewhere in the tree. These could be elements of the model that have already undergone normalization. To ensure that these elements are again revisited by the `desugar-top` traversal, we extend the evaluation strategy with additional logic to ensure that all elements of the tree are revisited. Figure 25 shows an extended version of the `desugar-top` strategy that takes this into account. Using an additional dynamic rule `InnermostApplied`, it keeps track of whether there have been any rule applications in the current traversal. Note that this rule does not have a left-hand side, and is simply restricted to the values `True()` and `False()`, similar to a (scopeable) global variable. If `InnermostApplied` is `True()`, `desugar-top` repeats the traversal after the current pass is completed. This process repeats until no rules are applicable anymore, ensuring exhaustive application of both static and dynamic rules.

A number of transformations in the WebDSL compiler directly relied on the global staging approach used previously in the generator [26]. These features depend on explicit ordering of rewrite rules to handle non-confluent rewrite rules, or may be applied once or a limited number of times to ensure termination. Most rewrite rules *reduce* the input term to a form closer to a normal form (e.g., core WebDSL). One such rule is the local-to-global transformation rule `extend-entity` in Fig. 21. It rewrites a construct to a term that is removed in a different rule, thus ensuring termination of the transformation. However, it also defines a new dynamic rule `DynamicExtendEntity`, which is a preserving global-to-local transformation. Application of that rule results in a term that it can be applied to again, resulting in an infinite sequence of applications.

In general, non-reducing rules can be controlled by explicitly specifying termination criteria. For instance, an additional dynamic rule could be used to indicate that a particular transformation has successfully completed for a particular element. For dynamically defined global-to-local transformations, however, Stratego offers a convenient feature to avoid this. For each dynamic rule, an additional strategy is derived that applies the rule and then removes its definition. We can invoke this strategy using the prefix '`once-`', and integrate it into the system as follows:

```
desugar = once-DynamicExtendEntity
```

For *non-confluent rewrite rules*, the order of their application affects their result. Similar to non-reducing rules, dynamic rules can be used to register which rules may be applied at what time. Using a dynamic rule to explicitly set and check a rule that maintains a stage number, a given set of rewrite rules can be explicitly and internally staged without requiring a global staging mechanism.

9 Combining type analysis and transformation

While the `innermost` strategy solves the extensibility problem of the staging approach, it does not solve the interaction between type analysis and desugaring transformations. In this section, we analyze the interaction problem and present a solution for the fine grained combination of analysis and transformation.

To understand the interaction between type analysis and transformation consider the transformation of `derive editPage` in Sect. 7.2. For example, given the entity definition

```
entity NewsItem { name :: String text :: Text }
```

Fig. 25 Fixpoint iteration of `innermost`

```
desugar-top =
  do-while(
    rules(InnermostApplied := False())
    ; innermost(desugar; rules(InnermostApplied := True()))
    , InnermostApplied => True()
  )
```

type analysis annotates the identifier `i` in the page definition

```
define page editNewsItem(i : NewsItem) {
  derive editPage from i {NewsItem}
}
```

with its type `NewsItem` (indicated in italics). Given this type, the `derive-edit-page` rule transforms the page definition to a page definition with a table and rows (slightly simplified):

```
define page editNewsItem(i : NewsItem) {
  table(){ row{ "Name:" input(i.name) }
           row{ "Text:" input(i.text) } } }
```

After application of the `derive-edit-page` rule, the types of the expression generated as arguments of the `input` statements are unknown. Another round of type analysis is needed to determine the types of the expressions:

```
define page editNewsItem(i : NewsItem) {
  table(){ row{ "Name:" input(i.name {String}) }
           row{ "Text:" input(i.text {Text}) } } }
```

Given these types, the `derive-input` rules can normalize the `input` statements:

```
define page editNewsItem(i : NewsItem) {
  table(){ row{ "Name:" inputString(i.name {String}) }
           row{ "Text:" inputText(i.text {Text}) } } }
```

Thus, after application of the `derive-edit-page` transformation, type analysis is needed before the `derive-input` rules can be applied.

9.1 Integrating type analysis and transformation

In the staged approach of Fig. 22, desugaring transformations are interleaved with invocations of `typecheck` in order to add type information to newly generated terms. While the staging approach solves the problem of the interaction between analysis and transformation, it does so at a cost to the extensibility of the compiler. Since performing type analysis to the entire model after the application of each rewrite rule does not scale, the transformations are divided into coarse grained stages. The transformation stages have to be carefully composed such that type analysis is applied after the application of transformations introducing new, untyped expressions. Fitting in new transformations requires a careful analysis of all existing transformations.

If staging is not the answer to the interaction problem, maybe maintaining a fully typed representation is a solution. That is, require all transformation rules to produce a representation that includes all type and other analysis information that is needed to perform further transformations. For example, the `derive-edit-row` rule that is responsible for the generation of the `input` statements above, can be rewritten to ensure it includes a type annotation in its result:

```
derive-edit-row(|x) :
  |[ y relation srt ]| -> |[ row{x_text ``:'' input(e)} ]|
  where e-untyped := |[ x.y ]|
      ; e := <add-type-information(|srt)> e-untyped
```

With this modification, the rewrite rule now explicitly adds a type annotation to the generated expression *e*. While this approach is effective, it requires additional effort in the development of desugaring rules, and leads to rules that are harder to read and maintain. Furthermore, this violates the principle of separation of concerns by introducing logic related to type analysis in transformation rules. For more complex rewrite rules, the amount of code for type analysis can be significantly larger in size, and may require passing around type information of the surrounding context.

We considered two solutions to the combination of analysis and transformation. Staging analysis and transformation is not extensible. Integrating analysis in transformation rules breaks separation of concerns. Is there a solution that maintains separation of concerns, yet is extensible?

The `typecheck` strategy employed in Fig. 22 is a monolithic traversal that combines three functions, as illustrated by the rules in Fig. 26: *name resolution*, identifying the declaration to which an identifier is associated; *type analysis*, assigning types to identifiers and expressions; and, *error checking*, checking type correctness of expressions and other constructs and generating error messages in case of violations. Factoring these three operations into separate, independently applicable sets of rules, is the key to fine grained combination of type analysis and transformation. Figure 27 shows the composition of `typecheck` as a name resolution phase (`rename-top`) and an error checking phase (`check-constraints`). Type analysis is applied during error checking, but can now also be applied *on demand* during desugaring. We discuss these components and their application during normalization in the rest of this section.

Fig. 26 Rules of monolithic typechecker

```
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
```

```
typecheck =
  rename-top
  ; check-constraints
```

Fig. 27 Typechecker composed from name resolution and constraint checks

9.2 Name resolution

In name resolution, renaming rules annotate identifiers with unique keys. These are associated with the types of the referenced declaration, using a dynamic rule `TypeOf` that rewrites the annotated expression to its declared type. The annotations added are shared for identical identifiers, taking scoping rules into account to ensure that identifiers with the same name in different scopes get different keys.

Figure 29 shows two renaming rules, assigning unique keys to identifier declarations[37] and arguments of page definitions[38]. Each declared identifier is annotated with a unique name by the `add-naming-key` rule[41]. A dynamic rule `Rename`[42] is then generated, which annotates all uses of the declared identifier with the new key. To ensure that the rule is only applied in the lexical scope of its declaration, the dynamic rule is scoped[40]. The `TypeOf` rule, on the other hand, is not scoped. It binds the globally unique key to its associated type, for use in the phases that follow after this analysis. The `rename` rule for page definitions[39] renames all formal arguments *farg1** in a similar fashion as is done for identifier declarations; it declares them as local identifiers in the page's scope.

The naming rules are applied in a top-down fashion by the `rename-top` strategy, which is defined to apply `alltd(rename)`. The `alltd` strategy traverses the tree top-down and attempts to apply its strategy argument to each node it traverses, when the application succeeds the traversal is stopped.

The example in Fig. 28 illustrates the application of renaming rules. The partial page definition on the left is being renamed, the traversal just passed the last line shown. For the renamed identifiers, annotations are displayed in superscript. On the right the active dynamic rewrite rules at this point in the traversal are listed. The identifier declaration u^{u1} has shadowed the page argument u^{u0}, thus there is only one active `Rename` rule.

9.3 Type analysis

After name resolution, all identifiers in the model have a unique key, and an associated `TypeOf` rule that can be used to acquire its type, without the need for contextual (e.g., scoping) information about the identifier.

Context-sensitive transformations, such as those required for deriving page elements from entity types, make use of the type information made available by the name resolution phase. The `type-of` rules can be used to acquire the type of complete expressions. Figure 30 shows the definition of several of such rules. The first rule resolves the type of an identifier, which utilizes the `TypeOf` dynamic rule. The second rule calculates the type of an object field access, for instance `user.name`. It does so by first calculating the type of the object expression (`user`) and then retrieving the type of the property (`name`) of the object's type. A third rule calculates the type of the addition of two expressions, where it also checks that they are type compatible.

9.4 Type constraints

One application of name resolution and type analysis is the static checking of the type correctness of a model. The `typecheck` strategy checks the model for any violations of type constraint rules. Type constraints can be simple checks for types (e.g., conditions must be booleans), or more sophisticated checks such as constraints on the nesting of user interface elements (e.g., all `input` elements must be nested within a `form`). As an example, consider the simple type constraint in Fig. 31 stating that every identifier used needs to be declared. Constraint rules typically specify a negative condition in their `where` clause: this rule only produces an error if the constraint is violated. If there are any constraint

Fig. 28 Rename example

```
define page editing(u$^{u0}$ : User) {
  form {
    input(u$^{u0}$.name)
    var u$^{u1}$ : Blog
    input(u$^{u1}$.title) } }
```

```
TypeOf: u$^{u0}$ -> SimpleSort("User")
TypeOf: u$^{u1}$ -> SimpleSort("Blog")
Rename: u -> u$^{u1}$
```

Fig. 29 Name resolution rules

```
rename-top = alltd(rename)

rename :[37]
  |[ var x : srt ]| -> |[ var y : srt ]|
  where y := <add-naming-key(|srt)> x

rename-page-arg :[38]
  |[ x : srt ]| -> |[ y : srt ]|
  where y := <add-naming-key(|srt)> x

rename :[39]
  |[ define page x(farg1*) { elem1* } ]| ->
  |[ define page x(farg2*) { elem2* } ]|
  where {| Rename[40]
         ; farg2* := <map(rename-page-arg)> farg1*
         ; elem2* := <rename-top> elem1*
         |}

rename = Rename

add-naming-key(|srt) :[41]
  x -> y
  where y := x {<newname> x}
      ; rules (
          Rename : Var(x) -> Var(y)[42]
          TypeOf : y -> srt
        )
```

Fig. 30 Type analysis rules

```
type-of :
  |[ x ]| -> srt
  where srt := <TypeOf> x

type-of :
  |[ e.f ]| -> srt2
  where srt1 := <type-of> e
      ; srt2 := <type-of-property> (srt1, f)

type-of :
  |[ e1 + e2 ]| -> srt1
  where srt1 := <type-of> e1
      ; srt2 := <type-of> e2
      ; <type-compatible> (srt1, srt2)
```

violations, the complete set of errors is reported to the user and the compiler terminates, as shown in Fig. 31. This style of constraint checking rules was inspired by oAW *Check* language [22].

9.5 Type analysis during transformation

Since type analysis has been separated from error checking and name resolution, it can be used on the fly during

Fig. 31 Constraint error rule and check constraints strategy

```
constraint-error :
  |[ x ]| -> ConstraintError(["Variable ", x, " not declared"])
  where not(<type-of> |[ x ]|)

check-constraints = where(
  collect-all(constraint-error)
  ; if not(?[]) then report-errors; <exit> 1 end
)
```

Fig. 32 Fixpoint iteration desugaring with incremental name resolution

```
desugar-top =
  do-while(
    rules(InnermostApplied := False())
    ; innermost(
        desugar
        ; {| Rename: rename-top |};
        ; rules(InnermostApplied := True())
      )
    , InnermostApplied => True()
  )
```

Fig. 33 Desugaring rule for case statements

```
desugar :
  |[ case(e) { alt* } ]| -> |[ { var x : srt := e; stat } ]|
  where srt := <type-of> e
      ; x := <newname> "c"
      ; stat := <case-to-if(|x)> alt*
```

transformation. Thus transformation rules that depend on type information, such as the derive rules in Fig. 20, can use the `type-of` rules to compute the type of an expression without the need for a re-application of type analysis to the entire model.

Moreover, the specification of the transformation rules need not be concerned with name resolution. Figure 32 shows an extension of the `desugar`-top strategy. Each successful application of `desugar` is followed by an application of the `rename-top` rule, which adds unique keys to any identifiers that have not yet been annotated. By adding a scope for the `Rename` dynamic rule, the strategy ensures that any dynamic renaming rules derived from local declarations do not "leak" out of the context of the generated code.

While the initial name resolution phase performs a global analysis, applying `rename-top` during the transformation phase is not a context-sensitive operation. It only adds annotations to the generated fragment of code, with regard to scopes defined in that fragment. For example, consider the desugaring rule of Fig. 33. It introduces pattern matching by means of a case statement into the language. To do so, among other things, it defines a fresh variable *x* with a new name "c". Processed by the `rename-top` strategy, this name is given a new, unique key associated with the local type *srt*. Any non-local definitions, such as identifier uses in the expression *e*, are left alone by the transformation, as they must be defined in the context of the generated fragment. However, as they are defined by the context, and not local to this transformation, it is safe to assume that these identifiers are already annotated with a unique key, copied from the left-hand side of the transformation rule.

Likewise, for the `derive-edit-row` global-to-local transformation of Sect. 7.2, the generated fragment is already annotated and requires no additional effort from the `rename-top` operation. As an optimization, `rename-top` can be cached to avoid inspection of (sub)terms that are already sufficiently annotated.

Figure 34 illustrates the case statement desugaring with a concrete example. The `show` page definition takes two arguments, a `Blog b` and a `String s`. The content of the page depends on the type of view requested in `s`, realized through a case statement. The case statement has been desugared on the right and introduces a new identifier $\texttt{c0}^{c00}$ which has been automatically renamed after the application of the `desugar` rule which is called by `desugar-top` (see Fig. 32). Through this mechanism analysis information is kept intact during transformations.

Fig. 34 Case statement desugaring applied

```
define page show(b^{b0}:Blog,
                 s^{s0}:String) {
  case(s^{s0}.toLowerCase()) {
    "all"{ showAll(b^{b0}) }
    ...
```

```
TypeOf:b^{b0}->SimpleSort("Blog")
TypeOf:s^{s1}->SimpleSort("String")
```

⇒

```
define page show(b^{b0}:Blog,
                 s^{s0}:String) {
  {var c0^{c00} : String :=
               s^{s0}.toLowerCase();
  if(c0^{c00}=="all"){ showAll(b^{b0}) }
  ...
```

```
TypeOf:b^{b0}->SimpleSort("Blog")
TypeOf:s^{s1}->SimpleSort("String")
TypeOf:c0^{c00}->SimpleSort("String")
```

10 Discussion

In this section, we discuss related work, evaluate the code generation by model transformation approach, and discuss future work.

10.1 Compilation by normalization

Normalization of a rich language to a small core language is a well-known design pattern in programming language design and implementation, made popular in particular by the functional programming language Haskell [49]. Haskell is a large and complex language with many ‘syntactic abstractions’. These abstractions are translated by the compiler front-end to a core language, close to the lambda calculus. Furthermore, the Glasgow Haskell Compiler (GHC) uses a transformation-based approach in its optimizer that relies on the application of cascading (small) transformation rules [50].

A difference with the approach in this paper is that the core language in GHC is not a subset of the input language. Using an Intermediate Representation (IR) that is a subset of the source language is useful since it allows the result of compilation to be fed into the front-end of the compiler. This approach is for example taken by Kats et al. [33] in the extension of Java with inline bytecode, which can be used by code generators for DSLs to flexibly combine (pre-compiled) bytecode and source code. The core language design pattern has also been used in the design of Stratego [10,11] and SDF [58], the DSLs used for the transformation and syntax definition of WebDSL.

An approach related to compilation by normalization is the *nanopass compiler infrastructure* of Sarkar et al. [52], which advocates the design of compilers as a long pipeline of very small stages in order to enhance the understandability of the compiler in an educational context. Each stage transforms the program to an intermediate, more low-level form. In contrast, our approach does not employ a strict separation between the application of the different stages, as doing so hinders compositionality of language features. For normalization rules that are non-reducing or non-confluent, individual rules may specify dependencies or restrict their application (discussed in Sect. 8.3), without introducing a form of globally staged application.

10.2 Rewriting tools

The main ingredients for code generation by model transformation are (1) generation rules that generate structured representation instead of text, (2) concrete object syntax to make generation rules readable and maintainable, (3) integration of model-to-code, code-to-code, and model-to-model transformations, (4) incremental normalization of high-level models to low-level ones, (5) extensible definition of transformations, and optionally (6) integration of type analysis and transformation. In the WebDSL implementation and this paper we have used the rewriting-based Stratego/XT toolset [10], which supports all of these ingredients.

Examples of other rewriting-based languages are ASF+SDF [38] and TXL [15]. Both languages support the definition of transformations using concrete syntax and can be used to realize transformations on models and code. However, these languages provide limited programmability for strategies controlling the application of transformation rules. In contrast, Tom [5] supports rewrite rules and strategies. Tom is implemented as an extension of Java, using a preprocessor approach to map the Tom language features to standard Java. The language only supports the use of abstract syntax to specify patterns. Visser’s survey of strategies in rule-based transformation strategies gives an overview of approaches to user defined strategies [61].

10.3 Model transformation

In this section, we reflect on the different aspects of typical MDE systems, and discuss how these compare to the approach presented in this paper. We first focus our discussion on the transformation of models; in Sect. 10.9 we discuss code generation techniques.

10.4 An overview of MDE toolkits

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 by Czarnecki and Helsen in [17]. Various MDE toolkits provide model transformation and code generation facilities, many of which are based on OMG's MDA (openArchitectureWare [20,21], AMMA [40], AndroMDA [3]). Each approach is bound to a particular metamodeling language. A number of them share a standardized foundation, such as MOF [45], Ecore [12], or KM3 [30]). Other MDE tools are based on proprietary formats (DSL Tools [14], MetaEdit+ [35]).

The different MDE toolkits prescribe varying model transformation languages, such as ATL [32], openArchitectureWare's xTend [20], and QVT [6]. The current crop of MDE toolkits are characterized by using a separate language for code generation, such as TCS [31], xPand [64], Velocity [56]). In general, they also use a separate language to define a sequence of transformations, or to combine model transformations and code generation. Examples include openArchitectureWare's workflow language [20] and the Groovy scripting language [3], employed by AndroMDA.

10.5 Representation of models

Using a common metamodeling language can improve interoperability between tools. A number of standardized metametamodels have been developed, such as MOF, Ecore, and KM3. In Stratego/XT, SDF grammars are used as a corresponding notion. These are serialized using the ATerm format, and may be used to interoperate with other tools based on the same technology, such as ASF+SDF [38].

Using a common metamodeling language should not be considered a "silver bullet" for interoperability, however. In practice, metamodels designed using different tools are often incompatible. Similarly, metamodels designed using the same tool, for the same concrete syntax representation of a known language, may lack compatibility. Only when using identical metamodels can models be exchanged across tools. Alternatively, a well-defined (textual) concrete syntax representation may be used to exchange models between tools.

Model management can be based on any algebraic datastructure such as trees, graphs, hypergraphs, or categories [8]. Most current MDE toolkits are based on graphs, while Stratego/XT uses a tree-based representation.

Trees are acyclic, directed graphs. This nature allows them to be efficiently stored using maximal sharing, ensuring a significant decrease in memory usage [9]. Based on maximal sharing, all identical subtrees occupy the same space in memory, allowing constant-time equality tests between branches using pointers. Moreover, terms that are copied can simply be copied as pointers, while modified terms can be efficiently reconstructed (maintaining maximal sharing) rather than destructively updated. In contrast, using destructive updates parts of the tree that may be shared or used in different contexts will be modified in-line. Any local rewrite is performed using a destructive update, as is typical in graph-based rewriting.

Using a tree-based structure allows for simple, intuitive specifications of traversals with clear termination criteria. For this reason, many graph-based systems employ a spanning tree, imposing a tree structure on a graph. In contrast, in Stratego the tree structure is the principal representation. By use of dynamic rules, Stratego can conversely impose graph structures on trees. This makes it possible to model context-sensitive information that cannot easily be expressed using mere trees. For example, a dynamic rule can be used to resolve an identifier reference, essentially connecting the identifier node to the declaration node. We described how dynamic rules can provide context information for global-to-local and local-to-global transformations in Sect. 7.

An alternative approach to using dynamic rules to represent graphs as trees is Balland and Brauner's approach of using de Bruijn indices in terms implemented in Tom [4] In this approach term paths are used to point to terms. For instance in the term `f(s(a, 1.1))`, the path `1.1` will refer to `a` by first taking the first child of `f` and then the first child of `s`. Relative paths can also be represented, such as `f(s(a, -1.1))` in which `-1` indicates going one term level up and navigating from there. Again, `-1.1` refers to `a`. This approach works well in simple cases where little transformations are performed. In the case of a large number of much more complex transformations, such as the transformations demonstrated in Sect. 6, it becomes tedious to keep such paths up to date during transformations.

10.6 Transformation workflow

Stratego/XT does not employ a separate workflow language, but allows the Stratego language itself to control the application of transformations. In contrast, other approaches use a separate language such as Groovy, or a dedicated workflow language as is used in openArchitectureWare. For both examples, a lack of linguistic integration results in a lack of static checking for the validity and definedness of transformations that are specified.

10.7 Consistency management

Consistency management is an important issue in MDE [42]. In principle, models can be kept consistent as part of a transformation. In practice however, doing so 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.

Fig. 35 The Stratego/XT platform and the QVT RFP requirements

Mandatory requirement	Supported
1. Language for querying models	+
2. Language for transforming models	+
3. Meta-models for languages in MOF 2.0	-
4. Expressive transformations	+
5. Creation of views of meta-models	+
6. Declarative transformations enabling incremental changes	+/-
7. Meta-models specified in MOF 2.0	-
Optional requirement	**Supported**
1. Bidirectional transformations	-
2. Transformation traceability	+/-
3. Generic transformation definitions	+
4. Transactional transformations	+/-
5. Other sources of data	+
6. Model updates	+

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 [57] to achieve code-to-model traceability.

Transformation languages such as ATL and xTend allow transformations to be separated in modules, similar 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.

10.8 QVT request for proposals

The Query/View/Transformation (QVT) request for proposals [44] of 2002 sparked much interest in the development and comparison of different tools for model transformations. Figure 35 summarizes the compliance of Stratego/XT as such a tool according to the mandatory and optional requirements as specified by the OMG.

Outlining the mandatory requirements of Fig. 35, Stratego/XT provides excellent support for specifying querying and transformations of models (1,2). In particular, it supports querying using a combination of traversal and pattern matching specifications, and provides the APath library [29] for XPath-like queries. Stratego/XT does not define the abstract syntax of the query, view, and transformation languages in MOF 2.0 (3,7), but uses ATerms instead, where SDF grammars define the meta-model. As Stratego is implemented in itself, the abstract syntax of Stratego is itself also defined in SDF. The present paper demonstrates how the Stratego language is capable of expressing all information required to generate a target model from a source model automatically, as required by (4). Stratego can be used to implement views through transformations (5). For example, a view of all names of pages defined in a WebDSL model can be generated through a transformation that collects all page names in the model. As meta-models are defined in SDF, which can also be transformed using Stratego, views can also be constructed at the meta-model level. Technically, it is possible to incrementally apply changes to source code model into changes in a target model using Stratego (6). However, this incrementality does not come for free, and requires additional work in the implementation of the transformations.

Outlining the optional requirements of Fig. 35, transformations cannot be applied bi-directionally (1); reverse transformations have to be specified separately. It does not provide traceability by default (2). However, with additional runtime or library support limited traceability capabilities can be added [34]. Support for reusing and extending generic transformations is where Stratego really shines (3). Using strategies it is much easier to reuse transformations and traversals than in approaches that do not have such a notion. Although there is no notion of term inheritance; generic transformations based on inheritance structures as suggested in the request for proposals can be supported by emulating inheritance using strategies that implement `is-a` behavior and accessor strategies that set and get properties of terms generically. Stratego uses copy-on-write semantics for transformations, and allows rollback using the `<+` operator (4). However, applying the system in an asynchronous context is considered future work. Access is provided to other sources of data beyond the input model, such as file access and execution of external programs (5). Stratego allows (non-destructive) updates of models (6).

10.9 Code generation

Some other approaches have generated partial artifacts through the use of partial classes, which are then combined by the regular compiler for the target language. Warmer and Kleppe [65] describe experiences with such an approach. These approaches rely on the target language to support this 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.

Generation of partial artifacts has also been applied by Huang and Smaragdakis [28] by use of Meta-AspectJ [68]. Rather than using a full-fledged AOP (meta-)programming language, our approach makes use of the standard Java syntax, with only a small semantic extension. By integration of this functionality in the generator, our approach is independent of the capabilities of the target platform.

There have been other approaches that aspect weaving at the model level rather than using this feature in the generated code [39,55]. In contrast, in our approach we overlay the feature of partial classes and methods directly on the output language. This overlay definition can be used across different applications, i.e. other code generators that produce Java code. In contrast, using the more typical approach of strictly separating model transformation and code generation (using templates), as applied in [39,55], a very low-level, general-purpose model representation would have to be used to achieve the same result.

10.10 Web application generators

Many (visual) languages for modeling web applications have been developed, including WebML [41], MIDAS [46], OOWS [43], Netsilon [51], and UWE [2]. 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. WebML interprets its models rather than generating code from them.

Most approaches apply model transformations with the purpose of retargetability, or with the purpose of expressing "as many artifacts as possible using models as this allows for processing these artifacts using model transformations" (Voelter et al. [64]). Only Netsilon actually models the target source code (but then only XML).

10.11 Evaluation

Throughout this paper we have demonstrated how generator concerns can be better separated. We showed how transformation rules can be made more concise and modularized by extending the target language. We discussed several ways of combining type analysis with rewriting and introduced the approach of three-phased type analysis and transformation, in which name resolution, constraint checking, and rewriting can all be specified as strictly separate concerns.

When additional language abstractions are introduced, they can take advantage of the open extension points provided by the generator. These extension points, as described in Sect. 8 allow the extension to easily plug into the type analysis, model transformation and code generation subsystems. We built a number of language extensions into the generator, most notably the access control and workflow extensions, which are entirely built by plugging into the extension points mentioned.

10.12 Future work

A focal point of the present paper has been to provide an extensible mechanism for language specifications, using the Stratego language and specialized strategic programming idioms and library support. In the future, we would like to further investigate this area, in particular by providing specialized tool and language support for such specifications. Stratego is, as a strategic programming and term rewriting language, a very flexible platform for these endeavors, but specialized tooling could simplify the implementation of these idioms and provide additional static checks.

A number of other tools use attribute grammar to specify analyses, which provide a high-level, declarative, and effective way to specify analysis on trees [47]. Modern attribute grammar systems such as Eli [24], JastAdd [23], and Silver [67] offer many specialized features with respect to the analysis of software languages. More recently, Aster [34] used strategic programming to abstract over common patterns in attribute equations. Specifications made with these systems are highly modular and extensible. Unfortunately, they are lacking in their support for transformations, particularly for context-sensitive transformations that depend directly on the type analysis, and vice versa. Ongoing work focuses on integrating the two paradigms: using the expressiveness of a strategic term rewriting system for transformations and the high-level specification capabilities of attribute grammar systems for analysis. Challenges in this area include the integration of analysis and transformation: as a tree changes through transformation, this should have a proportionate effect on the attribute evaluation system. Normal attribute equations are declarative definitions of immutable properties of nodes in the tree, and may be memoized and computed on demand. In the context of a rewriting system, where the tree is a mutable model that undergoes multiple transformations, these computation principles must be reconsidered. This also raises the question of how control should be determined in a system that combines attribute equations and rewrite rules. Different designs of explicit control, eager or on-demand evaluation, and other approaches can be taken, and have a large impact on the effectiveness of such a system.

11 Conclusion

In this paper, we described several techniques to improve separation of concerns in DSL generators. The core technique is *code generation by model transformation*. The key idea behind code generation by model transformation is to represent both the source model and target code as terms. Current practice is often to directly generate plain text code, using template engines. We demonstrated that generating code by term rewriting has a number of advantages, for instance the ability to ensure syntactical correctness of generated code and the ability to perform further transformations on generated code. This enables extension of the target language with features such as partial classes and methods which greatly improve the modularity and size of rewrite rules.

We have shown how high level abstractions can be built on top of a relatively small core DSL language. Abstractions are gradually transformed to core DSL elements in a process of *compilation by normalization*. We have argued that the advantage of implementing such abstractions as model transformations is that by keeping the core DSL small, the generator becomes more portable, making it feasible to develop multiple generator back-ends.

Many transformations rely on the availability of contextual information, such as type information. In previous work we discussed the difficulty keeping type annotations up-to-date as a model is transformed. In this paper, we introduced a novel approach in which type analysis and rewriting are combined while still keeping the analysis and rewriting generator concerns separate. Repeatedly reanalyzing the entire model, the approach we previously took, is therefore no longer necessary.

Acknowledgments We would like to thank the anonymous reviewers of ICMT 2008 and SOSYM for their comments on earlier versions of this paper. This research was supported by NWO/JACQUARD project 638.001.610, MoDSE: Model-Driven Software Evolution, and 612.063.512, TFA: Transformations for Abstractions.

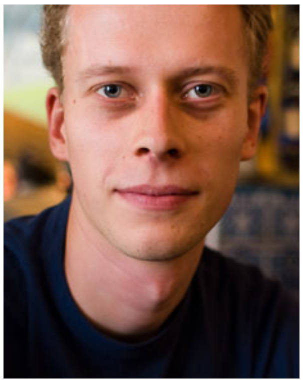
References

1. Google App Engine—Google Code.: http://code.google.com/appengine/ (2008)
2. Kraus, A.K.A., Koch, N.: Model-driven generation of web applications in UWE. In: Model-Driven Web Engineering (MDWE'07), Como, Italy, July (2007)
3. AndroMDA.org.: AndroMDA documentation. http://galaxy.andromda.org (2007)
4. Balland, E., Brauner, P.: Term-graph rewriting in Tom using relative positions. Electronic Notes in Theoretical Computer Science, vol. 203(1), pp. 3–17. Proceedings of the Fourth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2007) (2008)
5. Balland, E., Brauner, P., Kopetz, R., Moreau, P., Reilles, A.: Tom: Piggybacking Rewriting on Java. Lecture Notes in Computer Science **4533**, 36–47 (2007)
6. Bast, W., Belaunde, M., Blanc, X., Duddy, K., Griffin, C., Helsen, S., Lawley, M., Murphree, M., Reddy, S., Sendall, S., Steel, J., Tratt, L., Venkatesh, R., Vojtisek, D.: MOF QVT Final Adopted Specification, Nov 2005. OMG document `ptc/05-11-01`
7. Bézivin, J.: On the unification power of models. Softw. Syst. Model. **4**(2), 171–188 (2005)
8. Bézivin, J.: Model driven engineering: an emerging technical space. In: Lämmel, R., Saraiva, J., Visser, J. (eds) GTTSE, volume 4143 of Lecture Notes in Computer Science, pp. 36–64. Springer, Heidelberg (2006)
9. van den Brand, M.G.J., de Jong, H., Klint, P., Olivier, P.: Efficient annotated terms. Softw. Prac. Exp. **30**(3), 259–291 (2000)
10. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comp. Programm. **72**(1–2), 52–70 (2008)
11. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fund. Inform. **69** (1–2), 123–178 (2006)
12. Budinsky, F., Steinberg, D., Merks, E., Ellersick R., Grose, T.J.: Eclipse Modeling Framework (The Eclipse Series). Addison-Wesley, Reading (2003)
13. Burns, E., Kitain, R. (eds).: JavaServer Faces Specification. Version 1.2. Sun Microsystems (2006)
14. Cook, S., Jones, G., Kent, S., Wills, A.C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, Reading (2007)
15. Cordy, J.: The TXL source transformation language. Sci. Comput. Programm. **61**(3), 190–210 (2006)
16. Coward, D., Yoshida, Y.: Java Servlet Specification. Version 2.4. Sun Microsystems (2003)
17. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
18. DeMichiel, L., Keith, M. (eds.).: JSR 220: Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements. Sun Microsystems (2006)
19. DeMichiel, L., Keith, M. (eds.).: JSR 220: Enterprise JavaBeans, Version 3.0. Java Persistence API. Sun Microsystems (2006)
20. Efftinge, S., Friese, P.: openArchitectureWare. http://www.eclipse.org/gmt/oaw (2007)
21. Efftinge, S., Friese, P., Haase, A., Kadura, C., Kolb, B., Moroff, D., Thoms, K., Völter, M.: openArchitectureWare User Guide. Version 4.2. http://www.openarchitectureware.org (2007)
22. Efftinge, S., Völter M.: oAW xText—a framework for textual DSLs. In: Modeling Symposium, Eclipse Summit (2006)
23. Ekman T., Hedin G.: Rewritable reference attributed grammars. In: Odersky, M., editor, 18th European Conference Object-Oriented Programming (ECOOP 2004), volume 3086 of Lecture Notes in Computer Science, pp. 144–169, Oslo, Norway, July, Springer (2004)

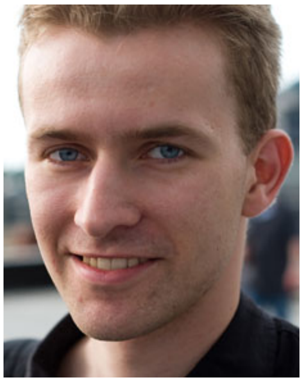
24. Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M.: Eli: a complete, flexible compiler construction system. Commun. ACM **35**(2), 121–130 (1992)
25. Groenewegen, D.M., Visser, E.: Declarative access control for WebDSL: Combining language integration and separation of concerns. In: Schwabe, D., Curbera, F. (eds.) Eighth International Conference on Web Engineering (ICWE 2008), pp. 175–188. IEEE CS Press, New York (2008)
26. Hemel, Z., Kats, L.C.L., Visser, E.: Code generation by model transformation. A case study in transformation modularity. In: Gray, J., Pierantonio, A., Vallecillo, A. (eds.) Theory and Practice of Model Transformations. First International Conference on Model Transformation (ICMT 2008), volume 5063 of Lecture Notes in Computer Science, pp. 183–198, Springer, Heidelberg (2008)
27. Hemel, Z., Verhaaf, R., Visser, E.: WebWorkFlow: an object-oriented workflow modeling language for web applications. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008), volume 5301 of Lecture Notes in Computer Science, pp. 113–127, Springer, Heidelberg, 2008
28. Huang, S.S. Smaragdakis, Y.: Easy language extension with Meta-AspectJ. In: ICSE '06: Proceeding of the 28th International Conference on Software Engineering, pp. 865–868. ACM, New York (2006)
29. Janssenn, N.: Transformation tool composition. Master's thesis, Institute of Information and Computing Sciences Utrecht University, Utrecht, The Netherlands (2005)
30. Jouault F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Formal Methods for Open Object-Based Distributed Systems, volume 4037 of LNCS, pp. 171–185, Bologna, Italy, Springer, Heidelberg (2006)
31. 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'06), pp. 249–254. ACM, New York (2006)
32. Jouault, F., Kurtev, I. Transforming models with ATL. In: Satellite Events at the MoDELS 2005 Conference, volume 3844 of LNCS, pp. 128–138. Springer, Heidelberg (2006)
33. Kats, L.C.L., Bravenboer, M., Visser, E.: Mixing source and bytecode. A case for compilation by normalization. In: Kiczales, G., (ed.) Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008), pp. 91–108, October. ACM, New York (2008)
34. Kats, L.C.L., Kalleberg, K.T., Visser, E.: Domain-specific languages for composable editor plugins. In: Ekman, T., Vinju, J., (eds.) Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009), Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, 2009 (to appear)
35. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: a fully configurable multi-user and multi-tool CASE and CAME environment. In: CAiSE, pp. 1–21 (1996)
36. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling. Enabling Full Code Generation. Wiley, New York (2008)
37. Kittoli, S. (ed). Seam—Contextual Components. A Framework for Enterprise Java. Red Hat Middleware, LLC (2008)
38. Klint, P.: A meta-environment for generating programming environments. ACM Trans. Softw. Eng. Methodol. **2**(2), 176–201 (1993)
39. Kulkarni, V., Reddy, S.: An abstraction for reusable mdd components: model-based generation of model-based code generators. In: GPCE '08: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, pp. 181–184. ACM, New York (2008)
40. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: Companion to OOPSLA'06, pp. 602–616. ACM, New York (2006)
41. 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, October (2007)
42. Mens, T., van Gorp, P.: A taxonomy of model transformation. In: Graph and Model Transformation (GraMoT 2005), vol. 152, pp. 125–142 (2006)
43. Pastor, V.P.O., Fons, J.: OOWS: a method to develop web applications from web-oriented conceptual models. In: Web Oriented Software Technology (IWWOST'03), pp. 65–70 (2003)
44. Object Management Group (OMG).: OMG/RFP/QVT MOF 2.0 query/views/transformations RFP, (2003)
45. Object Management Group (OMG).: Meta object facility (MOF) core specification. OMG available specification. Version 2.0. http://www.omg.org (2006)
46. Cáceres, B.V.P., Marcos, E.: A MDA-based approach for web information system development. In: Proceedings of Workshop in Software Model Engineering (2003)
47. Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. ACM Comput. Surv. (CSUR) **27**(2), 196–255 (1995)
48. Parr, T.J.: Enforcing strict model-view separation in template engines. In: WWW '04: Proceedings of the 13th International Conference on World Wide Web, pp. 224–233, New York (2004)
49. Peyton Jones, S., (ed).: Haskell98 Language and Libraries. The Revised Report. Cambridge University Press, Cambridge (2003)
50. Peyton Jones, S.L., Santos, A.L.M.: A transformation-based optimiser for Haskell. Sci. Comp. Programm. **32**(1–3), 3–47 (1998)
51. Pierre-Alain Muller, F.F., Studer, P., Bézivin, J.: Platform independent web application modeling and development with Netsilon. Softw. Syst. Model. **4**(4), 424–442 (2005)
52. Sarkar, D., Waddell, O., Dybvig, R.K.: A nanopass infrastructure for compiler education. In: ICFP '04: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, pp. 201–212. ACM, New York (2004)
53. Schmidt, D.C.: Model-driven engineering. IEEE Comp. **39**(2), 25–31 (2006)
54. Stahl, T., Völter, M.: Model-Driven Software Development. Wiley, New York (2005)
55. Suzuki, J., Yamamoto, Y.: Extending UML with aspects: aspect support in the design phase. Lecture Notes in Computer Science, pp. 299–299 (1999)
56. The Apache Foundation. Velocity User Guide. http://velocity.apache.org/engine/devel/user-guide.html (2007)
57. Deursen, A.van , Klint, P., Tip, F.: Origin tracking. J. Symbol. Comput. **15**(5/6), 523–545 (1993)
58. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. Thesis, University of Amsterdam (1997)
59. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) Generative Programming and Component Engineering (GPCE 2002), volume 2487 of Lecture Notes in Computer Science, pp. 299–315, Pittsburgh. Springer, Heidelberg (2002)
60. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In: Lengauer, C., et al. (eds.) Domain-Specific Program Generation, volume 3016 of Lecture Notes in Computer Science, pp. 216–238. Spinger, Heidelberg (2004)
61. Visser, E.: A survey of strategies in rule-based program transformation systems. J. Symbol. Comput. 40(1):831–873 (2005). Special issue on Reduction Strategies in Rewriting and Programming
62. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Interna-

tional Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007), volume 5235 of Lecture Notes in Computer Science, pp. 291–373. Springer, Heidelberg (2008)
63. Visser, E., Benaissa, Z.-E.-A., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP 1998), pp. 13–26. ACM, New York (1998)
64. Voelter, M., Groher, I.: Handling Variability in Model Transformations and Generators. In: Domain-Specific Modeling (DSM'07) (2007)
65. Warmer, J.B., Kleppe A.G.: Building a flexible software factory using partial domain specific models. In: Domain-Specific Modeling (DSM'06), Portland, Oregon, USA, pp. 15–22 (2006)
66. van Wijngaarden, J., Visser, E.: Program Transformation Mechanics. A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University (2003)
67. Wyk, E. V., Krishnan, L., Bodin D., Schwerdfeger, A.: Attribute Grammar-Based Language Extensions for Java. In: Ernst E. (ed.) 21st European Conference on Object-Oriented Programming (ECOOP 2007), volume 4609 of Lecture Notes in Computer Science, pp. 575–599, Berlin, Springer, Germany (2007)
68. Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ Programs with Meta-AspectJ. In: Karsai, G. Visser, E. (eds.) Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings, volume 3286 of Lecture Notes in Computer Science, pp. 1–18. Springer, Heidelberg (2004)

Author Biographies



Zef Hemel is a Ph.D. student at Delft University of Technology focussing on the design and implementation techniques of domain-specific languages. He is one of the core developers of WebDSL and in particular worked on WebWorkFlow, a WebDSL sub-language to describe workflows declaratively. He also develops PIL, the Platform Independent Language, which simplified the implementation and maintenance of DSL back-ends generating code for multiple software platforms.



Lennart C. L. Kats is a Ph.D. student at Delft University of Technology, where he works on techniques and tool support for developing domain-specific languages. He is the lead developer of the Spoofax/IMP project, which aims to provide state-of-the-art, Eclipse-based IDE support for domain-specific languages developed with Stratego/XT. He also developed Aster, a new language that combines Stratego's strategic programming with attribute grammars, and the Dryad Compiler, an experimental open compiler for Java.

Danny M. Groenewegen is a Ph.D. student at Delft University of Technology, where he works on abstractions for the web domain and their implementation as domain-specific languages and code generator. He is the main developer of the Java back-end of WebDSL and he designed and implemented the access control and data validation sublanguages of WebDSL.

Eelco Visser is associate professor at Delft University of Technology where he conducts research in the areas of model-driven engineering, domain-specific languages, program transformation, and software deployment. Together with his students he has designed and implemented domain-specific languages for syntax definition (SDF), program transformation (Stratego), software deployment (Nix), and web application development (WebDSL). In the research project 'Model-Driven Software Evolution' he is investigating the introduction of domain-specific languages as a standard software development tool, including the effective and efficient construction of DSLs, but also the maintenance of DSLs and systems built with them.