Generation Technique for Django MVC Web Framework Using the Stratego Transformation Language

D.V. Annenkov*, E.A. Cherkashin**
* National Research Irkutsk State Technical University, Irkutsk, Russia
** Institute of System Dynamics and Control Theory SB RAS, Irkutsk, Russia
annenkov@ib-soft.ru, eugeneai@icc.ru

Abstract – Domain-Specific Languages (DSLs) allow for raising the level of abstraction, improving development productivity, and establishing an equitable communication between domain experts and developers. Language-oriented programming (LOP) is a new paradigm based on DSL construction. LOP facilitates separating domain-specific and technology-specific aspects of a system under development sharing some ideas with model-driven architecture and model-driven development. Spoofax Language Workbench is used as a primary tool for DSL design in the present work. It is based on Stratego (a transformation language with programmable rewriting strategies), and Syntax Definition Formalism (a language for grammar definition). As an example of a DSL, a simple textual language for domain modelling is considered. Rewriting rules and strategies are used as a uniform approach to generate, validate, and make arbitrary abstract syntax tree transformations of the DSL code. Rules for code generation are implemented using so-called "string interpolation" technique. The source DSL code translated into python code that can be deployed within the Django web framework, resulting in a web-application with the create/update/delete functionality on a corresponding database. The developed DSL is an example of the "definition by transformation" approach. Adding more domain-specific features to the DSL would allow for better practical applicability.

I. INTRODUCTION

Language-oriented programming (LOP) can be considered as a style of software development which involves the use of domain-specific languages (DSLs) instead of general-purpose languages (GPLs) [1]. DSLs allow for capturing requirements in the users' terms. A DSL is a programming language designed for a particular domain. Because of the focusing on a specific class of problems, it allows expressing the domain in more precise terms, as compared to GPL and other modelling tools such as UML.

There is an interconnection between model-driven development (MDD) and LOP: both of them tends to reduce the gap between the problem domain and its implementation. It is a step in raising the level of abstraction after GPLs such as Java, C#, Python etc (Fig.1). LOP can be considered as a new programming paradigm that tends to unite such approaches as generative programming, model-driven approaches (Model-Driven Development, MDD, Model Driven Architecture, MDA), intentional programming [2].

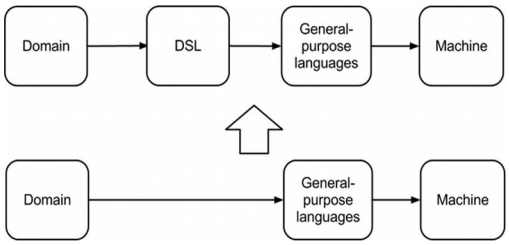

Figure 1. Raising the level of abstraction from GPL to DSL

The main advantages of LOP:

- improved developers productivity;
- communication with domain experts;
- a declarative approach to programming (define *what* one is going to obtain, not *how* it is to be obtained).

In order to support language-oriented programming, one needs a development tool that is known as a *language workbench* [3]. Language workbenches provide tools for defining DSLs (parsing, transformation, code generation), integration between DSLs, rich editing environment (code highlighting, static analysis, code completion and other modern IDE features).

There is a number of different approaches to how to develop and use a DSL. The comparison of some approaches, such as internal DSL, compile-time metaprogramming and strategic term rewriting is presented in [4]. The xText framework is one of the relatively wide-used tools for textual external DSL development. It works on the Eclipse platform and uses EBNF for syntax definition with ANTLR as a parser generator [5]. Another approach is *projectional editors (PE)* implemented in JetBrains Metaprogramming System (MPS) [6]. Projectional editors serve as an alternative to source editors. In MPS developer deals with an abstract syntactic tree (AST) directly using PE to edit the tree.

In this article, we consider an approach for DSL development based on strategic term rewriting. Spoofax language workbench is used as a primary tool for DSL

building. Spoofax is based on Stratego, which is a transformation language with programmable rewriting strategies, and Syntax Definition Formalism (SDF) [7], as a language for grammar definition. In Stratego, DSLs are implemented through term rewriting, where a source DSL program is transformed into a target program (Python, Java etc.) using a set of transformation rules and strategies.

Spoofax language workbench covers all the main aspects of DSL construction [8]:

- grammar definition and parsing;
- semantic analysis (DSL program validation) ;
- DSL code transformations;
- target code generation;
- integration of the DSL and its tools into an IDE.

One of the best practices of DSL development is that the *semantic model* is a part of a DSL. The term "semantic model" is used in this work in the same sense as in [1]: the semantic model is a library or framework that the DSL populates. The semantic model provides a runtime context of the code generated from the DSL.

We use the Django web framework as a semantic model. Source DSL code is translated to Python code that can be deployed within this web framework, resulting in a web-application administration subsystem with the create/ update/delete functionality on a corresponding database.

II. Defining the DSL

We consider a simple textual DSL which describes a system under development as a set of related entities.[1] The DSL may be used as a programming language or as a modelling language. The difference between the modelling and programming properties of DSLs is somewhat blurred and not precisely defined. Using the criteria proposed in [9] we can define our DSL to be closer to a modelling language rather than a programming language.

Customer Relationship Management is considered as a problem domain. We model primary data describing customers and their contacts using the DSL. Let us start with an example.

```
entity Customer
  name            : String
  description     : String
  website         : URL
  repr name
end
```

This fragment of code represents a `Customer` entity with attributes `name`, `description` and `website`. The `repr` keyword is used to define a string representation of the entity. In this example the value of the `name` field used as a string representation of the Customer entity.

Entities can be related using many-to-one associations. For example:

```
entity Contact
  name   : String
  phone  : String(11)
  email  : Email
  customer -> Customer
end
```

The `Contact` entity associated with the `Customer` entity using the `customer` property.

Using Stratego we can define the grammar of the DSL using the SDF notation:

```
context-free start-symbols
        Start
  context-free syntax
    "module" ID Definition* -> Start {cons("Module")}
    "entity" ID Property* Repr? "end" -> Definition
                                        {cons("Entity")}
    "repr" ID           -> Repr {cons("Repr")}
    ID ":" Type         -> Property {cons("Property")}
    ID "->" EntityAssoc -> Property {cons("Property")}
    ID                  -> Type {cons("Type")}
    ID "(" PINT ")"     -> Type {cons("Type")}
    ID                  -> EntityAssoc
                           {cons("EntityAssoc")}
```

Productions are annotated with a constructor name `n` to uniquely identify them in the abstract syntax tree using the `{cons(n)}` annotation. ID represents an identifier consisting of chars, digits and underscore symbols:

```
[a-zA-Z][a-zA-Z0-9\_]* -> ID
```

PINT represents positive integer with no leading zeros:

```
[1-9][0-9]*  -> PINT
```

Stratego generates corresponding algebraic signatures that describe the abstract syntax of the DSL.

```
signature
  constructors
    EntityAssoc : ID -> EntityAssoc
    Type        : ID * INT -> Type
    Type        : ID -> Type
    Property    : ID * EntityAssoc -> Property
    Property    : ID * Type -> Property
    Repr        : ID -> Repr
    Entity      : ID * List(Property) * Option(Repr) →
                    Definition
    Module      : ID * List(Definition) -> Start
                : String -> ID
                : String -> PINT
```

Now we can define rewriting rules for transformation and code generation. A rewrite rule has the form

`R: p1 → p2`, where `R` is the rule's name, `p1` is a left-hand side pattern and `p2` is a right-hand side pattern. Patterns are terms with variables.

Let us write a rewriting rule for a top-level form of the DSL – a module. A module has a name and contains one or more entities.

```
to-django-model:
        Module(x, d*) ->
        $[ # -*- coding: utf-8 -*-

            from django.db import models

            [d'*]
        ]
        with
            d'* := <map(to-django-model)> d*
```

The left side of *`to-django-model`* is a pattern `Module(x, d*)`. The pattern matches against AST nodes and, if successful, `x` binds a module name and `d*` –

1 The source code is available at https://github.com/annenkov/entity-model

a list of entities. Body of the rule represents a code template using a *string interpolation* technique. The text within `$[...]` block remains unchanged, except the block in the square brackets (like `[d'*]`) that is interpreted as a variable. In the `to-django-model` rule, the above variable `d'*` is assigned in a "`with`" clause. The expression `<map(to-django-model)> d*` is similar to the `map` function in *functional languages*: `to-django-model` rule applied to every item in list d*.

Let us define rules for translating `Entity`.

```
to-django-model:
    Entity(x, p*, r) ->
         $[ class [x](models.Model):
         [p'*]
         def __unicode__(self):
                 return "[x]: {0}".format([to-string])
         ]
    with
         to-string := <to-string-repr> r;
         p'* := <map(to-django-model)> p*
```

Here, we use the same name `to-django-model` for the rewriting rule, but this rule has a different pattern to match. In this case, a successful match binds `x` an entity name, `p*` a property list and `r` a `repr` field. The `r` value is optional and can be either `None()` or `Some(Repr(name))`. And we have two rules for these cases:

```
to-string-repr:
         Some(Repr(name)) ->
         $[self.[name]]

to-string-repr:
         None() ->
         $[self.pk]
```

The rules can be read as follows: use a field with a specified name or use the `pk` field, if no `repr` provided. The variable p* is used to obtain transformed code for each property using the <map(`to-django-model`)> strategy with the following rule:

```
to-django-model:
         Property(x, t) ->
       $[    [x] = [field_type]
         ]
         with
             field_type := <to-django-model> t
```

And at the last step is to define rules for every type used in our DSL. Every rule will translate our DSL types to the corresponding model field types of the Django web framework.

```
to-django-model:
    Type("String") ->
        $[models.CharField(max_length=256)]
to-django-model:
    Type("String", len) ->
        $[models.CharField(max_length=[len])]
to-django-model:
    Type("Int") ->
        $[models.IntegerField()]
...
```

Other type-translating rules have the same form and we omit them here for brevity.

Association fields are handled by a separate rule:

```
to-django-model:
       EntityAssoc(e)  -> $[models.ForeignKey([e])]
```

Applying the `to-django-model` rule to the top-level form of the DSL (to a module) gives a complete module for the Django web framework.

Similarly, we can define rules for translation to the Django admin settings. Also, we need an additional rule to write the generated code to Python source code files:

```
generate-django-app:
    (selected, position, ast, path, project-path) ->
        None()
   with
     module_name := <get-module-name> selected;
     models := <to-django-model> selected;
     models_file :=
   $[[<project-path>]
[module_name]/models_generated.py];
     <debug> $[Writing [models_file]];
     mf_handle := <fopen> (models_file, "w");
     <fclose> <fputs> (models, mf_handle);
     amdin_file :=
   $[[<project-path>]
[module_name]/admin_generated.py];
     <debug> $[Writing [amdin_file]];
     af_handle := <fopen> (amdin_file, "w");
     <fclose> <fputs>
       (<to-django-admin> selected, af_handle)
```

The Stratego language allows one to define side-effects in rewriting rules. In the rule above, we use the <`fputs`> strategy to write the result of the code generation and <`debug`> strategy to provide some information to the console.

One can assign the `project-path` variable to a path to the Django project and set this rule on "save" action using Spoofax which is activated on saving changes in the source code. As a result, all changes to the DSL code will be reflected in the files of the Django project.

III. Code checking and completion

One of the great advantages of strategic term rewriting techniques is the ability to express different aspects of a DSL, such as code transformation, code validation and context completion using the same notation. Let us consider the support of code checking and completion using the Stratego language.

Spoofax workbench generates sample rules for code analysis when creating a project. The main rule is as follows:

```
editor-analyze:
    (ast, path, project-path) ->
        (ast, errors, warnings, notes)
    with
        editor-init;
        analyze;
        errors   :=
           <collect-all(constraint-error, conc)> ast;
        warnings :=
           <collect-all(constraint-warning, conc)> ast;
        notes    :=
           <collect-all(constraint-note, conc)> ast
```

One can add rules such as `constraint-error`, `constraint-warning`, `constraint-note` to define custom error checking, warning and notes. The `collect-all(s, un)` strategy collects all subterms where strategy `s` succeed with a user-defined union operator `un`. In this case, the union operator is a list concatenation.

Consider a rule that checks whether the property specified in `repr` clause belongs to the entity.

```
constraint-error:
    Entity(x, p*, Some(Repr(prop))) ->
          (prop, $[[prop] is not a [x] property])
    where
         not(<some(?Property(prop, _))> p*)
```

This is just an ordinary rewriting rule that rewrites an `Entity` node to a tuple, consisting of a property name and an error message. This rewriting succeeds only if the condition in the `where` clause succeeds. The rule `not(<some(?Property(prop, _))> p*)` used as condition reads as follows: succeed only if there is no property named `prop` that belongs to the entity. The expression `?Property(prop, _)` is a pattern to match. The `some(s)` strategy applies the parameter strategy `s` to as many direct subterms as possible and at least one. The application of `some(s)` strategy fails if no successful applications of the parameter strategy `s` occurs.

Code completion or autocomplete allows predicting of code fragments without the user actually typing them completely. It simplifies and speeds up the development in text-oriented environments. Spoofax provides the ability to define custom code completion rules for DSLs.

Consider a simple rule that defines autocompletion for built-in types:

```
editor-complete:
    (Type(COMPLETION(prefix)), position, ast, path,
project-path) ->
       ["String", "Int", "Email", "URL", "Date"]
```

Or another rule for completion of association entities:

```
editor-complete:
    (EntityAssoc(COMPLETION(prefix)), position, ast,
path, project-path) -> proposals
    where
      proposals :=
        <collect-all(?Entity(<id>, _, _), conc)> ast
```

The `?Entity(<id>, _, _)` pattern is a *term projection* that used to extract the entity name. The `collect-all` strategy is used to get all entity names and put them into a list which is assigned to `proposals` variable.

Stratego provides a powerful pattern matching engine. Combined with special strategies, it allows making AST queries. Consider some examples.

```
get-all-string-props =
   collect-all(?Property(_,Type("String")), conc)
get-all-entities-with-2-props =
  collect-all(
    where(?Entity(_,<length => 2>,_)), conc)
get-all-entities-assoc-with-customer =
  collect-all(
     where(?Entity(_,
       <some(?Property(_,EntityAssoc("Customer")))>,_)),
      conc)
```

The `get-all-string-props` collects all properties that have the `String` type. Query `get-all-entities-with-2-props` returns all entities having exactly two properties. We use the *term projection* technique to impose a constraint on properties count. To test a condition (but not to rewrite a node) `where` strategy is used. The `get-all-entities-assoc-with-customer` query is slightly more complicated. It returns all the entities associated with the `Customer` entity. Here we use the `some` strategy with the `?Property(_, EntityAssoc("Customer"))` pattern to find associations with `Customer` if any. One can use similar querying strategies to perform analysis and validation of the source DSL code.

IV. Conlusion

Rewriting language fits well to DSL development activity using “language definition by transformation” approach. With good support of the IDE, one can get a productive environment for DSL development.

Developed DSL is an example of the “definition by transformation” approach. It can be used for rapid prototyping of Django applications. To get real benefits from the DSL one need to add more domain-specific features to it. For example, DSL can be extended with the ability to add constraints imposed by the domain. Constraints can be translated into runtime validation rules for input data using the Django form-handling library. Some constraints can be translated not only to Python code. One can perform client-side validation by JavaScript code and server-side validation by Python code generated from the same constraint. The DSL allows defining business rules in a single place. Since in the general case, the rules implemented or generated as source code are located in various modules within a Django project, it is not easy to see the overall picture of the defined business rules from that code. For that reason, the business rules represented in the DSL give a better way of system modelling.

Other domain-specific features also can be added easily using rewriting rules with the full support of IDE-like features from Spoofax workbench. The approach allows one to use the DSL as a fully-featured language for the domain-driven design approach.

References

[1] Martin Fowler, “Domain Specific Languages”. Addison-Wesley Professional, 2010.

[2] Sergey Dmitriev. “Language Oriented Programming: The Next Programming Paradigm”. URL: [2] http://www.onboard.jetbrains.com/is1/articles/04/10/lop/ (access date: 30.01.2013).

[3] Martin Fowler, “Language Workbenches: The Killer-App for Domain Specific Languages?” 2005. URL:http://www.martinfowler.com/articles/languageWorkbench.html (access date: 30.01.2013).

[4] Naveneetha Vasudevan, Laurence Tratt. “Comparative Study of DSL Tools”, Electronic Notes in Theoretical Computer Science (ENTCS), Volume 264, Issue 5, July, 2011, 103-121p.

[5] “xText page on Eclipse platform” web-site. URL: http://www.eclipse.org/Xtext/ (access date: 30.01.2013).

[6] “MPS page on JetBrains” web-site. URL: [6] http://www.jetbrains.com/mps/ (access date: 30.01.2013).

[7] Mark van den Brand, Paul Klint, Jurgen Vinju. “The Syntax Definition Formalism SDF”. CWI, 2007. URL: http://homepages.cwi.nl/~daybuild/daily-books/learning-about/sdf/sdf.pdf (access date: 30.01.2013)

[8] Lennart C. L. Kats, Eelco Visser. “The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs”. OOPSLA '10 Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2010, pp. 444-463.

[9] Yu Sun, Zekai Demirezen, Marjan Mernik, Jeff Gray, Barrett Bryant. “Is My DSL a Modeling or Programming Language?” University of Alabama at Birmingham, USA, 2008.