Projective Template-Based Code Generation

Zeev Chared[1] and Shmuel Tyszberowicz[2]

[1] CodiScent, Ltd.,
zeev@codiscent.com
[2] The Academic College of Tel-Aviv Yaffo,
tyshbe@tau.ac.il

Abstract. Template-based code generation (TBCG) is a generative technology that transforms data into structured text, e.g. code, through the use of templates. The data, referred to as the model, represents an abstract design while the templates provide the mold of the target code used by the generator in the process of code production.
A classical template is a mixture of text blocks and imperative control code segments. The text blocks' concern is to describe textual patterns while the control code's is to retrieve, manipulate, and substitute the data into place holders within the text. The mixture of these concerns reduces the readability, expressiveness, reusability, and analyzability of the textual patterns and the control code alike.
We introduce the projector templates, a TBCG technology that adheres to separation of concerns of the template, model, and the template control code. Inspired by the relational theory, projective template blocks are treated as annotated relations that can derive their input from the source model through the use of relational operators. The projector templates are thus code-free, coherent, their transformation process reusable, and capable of elegantly coping with complex code-generation scenarios.

1 Introduction

The difficulty of writing large software systems has led to an increased interest in generative programming where one program generates another program [9, 1]. Generator programs transform abstract data (model) into code making it possible to manage the software at a favorable abstract level of the model [4].

Template-based code generation(TBCG) uses templates to describe the structure of the target text. It offers a high level of control over the generated output providing the level of flexibility required by custom generation scenarios.

TBCG is used extensively throughout the industry [3], especially in web related technologies. Past and recent research attempts to find ways to further employ code generation to cope with more complex generation scenarios and heterogeneous development environment [11, 12, 7]. There are many available products for TBCG such as Microsoft T4 [14, 15], Acceleo [13], Xpand [17, 2] and XVCL [8]. The approaches commonly employed by most engines are either wrapping text block within control blocks (T4/Acceleo), or fragmenting the template among named blocks (Xpand/XVCL). These approaches limit the templates readability, textual flexibility and reusability, ultimately curbing their effectiveness as the generation scenarios complexity grows.

Ideally, a template should be an abstract representation of the code it describes. Consequently, our research is focused on inferring the control code from the templates textual structure without requiring any explicit control-logic code. This paper presents the projector templates-based code generation technology that focuses on transforming relational models into text by control code-free templates that can express complex textual patterns coherently and elegantly. Relations are analyzable [6] and easily created using available technologies (Excel/XML). Also, a lot of meta-data is already available in relational format from various modeling frameworks.

The rest of the paper is organized as follows: in Section 2 we present the relational view of templates. Section 3 presents the projective templates approach to common generation scenarios. Section 4 demonstrates the use of the technology in the context of a use case. Section 5 discusses the technology's benefits and limitations, and Section 6 concludes with a short summary.

2 The Projective Templates Inference Framework

A projective template is a hierarchy of textual blocks (blocks) where each block contains text and block variables, called the block attributes that reference corresponding metadata attributes to be substituted in their place. The block variable set is used to deduce a relational projection to be performed on a source metadata relation (model) associated with the template. Using the inferred projections, the code generation algorithm can extract data from the model, order it, and substitute it into the template in order to produce the target code.

2.1 Relational View of Templates

A relation is a set of tuples, or rows, that have the same set of column names called the relation header [5]. For example, the relation Name-Age = {Name, Age} contains 2-tuple set where the first tuple is name and the second is age.

A relational projection operator Π {A1,...,An} returns a relation that is derived from a source relation by selecting a distinct set of rows with the header {A1,...,An}. For instance, the projection Π {Name} applied to the relation Name-Age returns the distinct set of all the names in it.

We define a new operator called the annotated projection operator (APO) that takes a projection Π {A1,...,An} and a block that references the projection header's attributes. The operator transforms each projected row into an annotated row by substituting the projected row's column values in place of the block variables. Columns that are not referenced by the block do not affect the operator's output; hence the projection header can be inferred directly from the template blocks providing the basis of the projective TBGC.

We define the projection inferred from the block as the induced projection; the dataset returned from applying the induced projection to a metadata model as the block's dataset. The image resulting from substituting one row values in place of the block's variables as an annotated row, the concatenation of the annotated rows as the block image, and the concatenation of the block images as the template image.

3 Approach to Common Generation Scenarios

This section describes the projective template approach to common generation scenarios. The output is produced with the Projective Template Generator IDE by CodiScent, LTD available at www.codiscent.com.

Child blocks. A template-block can contain other template-blocks in a parent-child recursive structure. A child block is evaluated for each processed instance of the parent for the subset of rows associated with that instance. The child block's image is substituted in place of the child block in the parent processed instance.

The following example shows a template for table creation code and the output of its application to the model M = {Table, Column, DataType, Length}.

```
Model M:
 {{Customer, AddressLine1, varchar, 100} ,
 {Customer, AddressLine2, varchar, 100 } ,
 ...
 {Product, Description, varchar, 50} ...}

Template:
[ create table $Table$
  ([ $Column$ $DataType$ [($Length$)] ^,\n]) ^\n]

Generated code:

create table Customer (
   AddressLine1 varchar (100) ,
   AddressLine2 varchar (100) ,
   ...
 create table Product (
   Description varchar (100),
   ... )
```

Optional Attributes and **Required Blocks.** A block attribute is required by default. The block generates an image only for rows with non-empty values in their required variables. An attribute can be defined as optional by prefixing its name with '*'. A child block is optional by default, but can be designated as required by prefixing the block with "!" . A required child block prevents the production of its container block's image in case it has no instances. In the following example the block that contains the keyword 'Where' is generated only when there is a constraints clause.

[select * from tableX [where ![$Condition$]]

Ordering. A block dataset is sorted in an ascending order of the concatenated values of its block attributes where the attributes are ordered according to their order of appearance in the block.

Silent Attributes. There are generation scenarios that require a sort order other than the default order. The projective templates resolve that with *silent attributes,* designated by prefixing the attribute with a tilde, whose output is deactivated.

Instance separators. A block instance separator is designated by a '^' prefixed string attached to the ending block marker. A block instance separator is introduced between every pair of adjacent block images, for example, a comma between arguments in a function call.

Conditional generation: existence check. There is no need to test for attributes' null values. A zero instance block is simply not generated.

Conditional generation: derived attributes. The projective templates can *extend* the model by introducing derived attributes defined at the template or template group level. Derived attributes are evaluated only if their instantiation conditions are satisfied. In the following example the block containing PKAtt (derived from AttName) is generated only for rows that satisfy the PkAtt instantiation constraint, that is, rows with *AttName='ID'*:

```
[@EXTEND Derive PKAtt isa AttName
         where AttName='ID';@
delete $EntityName$ where [$PKAtt$ = 3]]
```

Conditional generation: text polymorphism. There are situations where the generated code varies slightly from instance to instance. For example, the syntax for loading string values into type-safe objects in C# requires parsing for non-string data types. The following template demonstrates the generation of a type-safe conversion code.

```
@EXTEND
derive NonStringDataType isa Column
       where C#DataType not in ('String')@
public bool LoadValues([string inp$Attribute$])
{[ $Attribute$ =
[$NonStringDataType$.
Parse(]inp$Attribute$[$~NonStringDataType$)] ; ^\n]}}
```

The template generates the necessary parse command for non-string data types. The silent NonStringDataType variable ensures that the parse command is matched with a closing bracket.

To summarize, the projective template technology provides an inference mechanism for all of the template control code tasks, specifically:

Data Retrieval. The dataset for each block is defined by its header and the instance of the parent block being processed. It is derived by joining the parent instance values with the source model, projecting the child block header onto the resulting relation and further filtered the result set according to the block required/optional attributes' designation and conditional constraints.

Data Substitution. The projective template code generator processes a block by iterating its input data. Hence, the imperative iteration control code construct (for-each) that is common in many template-based technology is unnecessary.

Conditional Generation. The model extension mechanism makes it possible to apply conditional generation without the need of explicit imperative control code such as an if-then-else construct.

4 Use Case

The following use case demonstrates some of the projective template features. Following is a template that generates SQL code for comparing tables contents in different database instances highlighting with '***' changed columns in each record. The metadata model is derived from querying the database system tables and generating M = {EntityName,ColumnName,DataType,Length,PK), the relation model. The model is further extended with the derived attributes PKColumnName and NonStringColumnName that contain columns with PK='PK' designation and columns that have a non-string data type, correspondingly.

```
[@EXTEND
Derive PKColumnName isa ColumnName where PK='PK';
Derive NonStringColumnName isa ColumnName
     where DataType not in ('varchar,nvarchar');@
[    select ' $EntityName$ ' as EntityName,'CHANGED ROWS' ,
    [ case a.$ColumnName$ when b.$ColumnName$ then ''
    else '***' end +
[convert(varchar(80),$~NonStringColumnName$]a.
 $ColumnName$[$~NonStringColumnName$)]
        as $ColumnName$ ^,\n#3]
    from  DL1..$EntityName$ a join DL2..$EntityName$ b
    on [a.$PKColumnName$=b.$PKColumnName$ ^ and ]
    where [(a.$ColumnName$<>b.$ColumnName$ or
         a.$ColumnName$ is null and b.$ColumnName$ is not null or
         b.$ColumnName$ is null and a.$ColumnName$ is not null )
         ^ or \n]]]
```

The outer most block is generated for each EntityName. It includes the first selected column with the entity name constant and the from-clause with the compared tables. The first child block selects the tables' columns designating changed data values with '***'. The second child block contains the join clause and is generated only for PKColumnName attributes. The last child block contains the where clause that selects record with any changed value. Using a traditional, imperative approach would require adding data access code to retrieve, test, filter, iterate and substitute the model data in the template.

5 Benefits and Limitations

5.1 Benefits

- *Separation of model and template.* A projector template is strictly separated from the template control logic that is inferred by the code generator in a consistent and predictable manner. Reusing the templates as well as the metadata models is as easy as associating the relevant templates and metadata sources.
- *Clarity and Conciseness.* The projective templates are clear and readable as they do not contain any template-control code or data access logic. They merely represent an abstract description of their target code and thus do not require the template developer to know any additional language.
- *Standard Control Code Inference*. Since the control code is inferred from the template block structure it is innately reusable among templates and template blocks as it is always guaranteed to be the same for similar blocks.
- *Format flexibility*. The template is flexible with respect to the text it can generate as there are no formatting limitations imposed by a control language. Generating a function call with parameters at the same line is as easy as generating each parameters on a different line simply by changing the separator associated with the block.
- *Change flexibility*. The template is very easy to change since there is no need to change the control code logic or rearrange the template blocks among different control blocks.

5.2 Limitations

- *Lack of imperative control code.* The lack of imperative control code prevents the transformation from changing the model; however, this limitation is introduced by design as in our view a transformation should not change its source model [10].
- *Change Impact.* Changing the model or the template may have a great impact on the generated code-base. Unless managed correctly, model changes can become an impediment to development process. That is, however, a potential issue with any generative technology and the projective templates approach makes it possible to assess the impact of model changes by comparing before/after projections.
- *Over generation.* Having a powerful generation technology may encourage the use of code-generation in place of standard coding techniques. On the other hand, it sometimes makes sense to generate explicit code rather than use a generic framework as code can be straightforward to understand and analyze.

6 Conclusion

In this paper we presented the projective templates, a new approach to template-based code generation featuring relational projective templates that provide a coherent and readable description of the target code. The projective template generator infers the control code by employing relational algebra, treating the template blocks as annotated relations that operate on the associated source generating the instance set for each block. We have presented the benefits of separating the concerns of textual patterns and control code specifications, specifically, the template readability, consistency, and clarity offered by the relational approach as well as the increased opportunity for control code reuse. Future work may include expanding the projective data selection capabilities to handle information that resides in several relational sets and non-relational formats.

References

1. B.J. Arnold, *An Illumination of the Template Enigma.* IPA dissertation series 2011-02, Eindhoven University of Technology, 2011
2. B. Klatt, *Xpand: A Closer Look at the model2textTransformation Language.* Institute for Program Structures and Data Organization, 2007
3. C. Henthorne and E. Tilevich, *Code Generation on Steroids: Enhancing COTS Code Generators via Generative Aspects.* Virginia Tech, 2007
4. D. Jackson, *Software Abstractions: logic, language, and analysis.* MIT Press Cambridge, MA, 2006
5. E. F. Codd, *A Relational Model of Data for Large Shared Data Banks.* IBM Research Laboratory, San Jose, CACM 13(6), June 1970
6. E. Torlak and D. Jackson, *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications.* Doctor of Philosophy, Massachusetts Institute of Technology, 2009
7. G. Vos, *Issues of iterative MDA-based software development processes, Master thesis.* University of Twente, the Netherlands, 2008
8. S. Jarzabek, *Software Reuse Beyond Components with XVCL.* National University of Singapore, 2007
9. T. Parr, *A Functional Language For Generating Structured Text.* University of San Francisco, 2006
10. T. Parr, *Enforcing Strict Model-View Separation in Template Engines.* University of San Francisco, 2004
11. W. R. Otte, A. Gokhale, D. C. Schmidt, and J. Willemsen, *Infrastructure for Component-Based DDS Application Development.* GPCE, 2011
12. Y. Zheng and R. N. Taylor, *A Rationalization of Confusion, Challenges and Techniques in Model Based Software Development.* University of California Irvine, ISR Technical Report UCI-ISR-11-5, 2011
13. Acceleo, *http://www.acceleo.org.* Accessed on 2012
14. Code Generation Using T4 Templates, *http://learneveryday.net/dot-net/code-generation-using-t4-templates.* 2010
15. Microsoft Code Generation and Text Templates (T4), *http://msdn.microsoft.com/en-us/library/bb126445.aspx.* Accessed on 2013
16. The Free Dictionary, *http://www.thefreedictionary.com/template.* Accessed on 2013
17. Xpand Reference, *http://www.openarchitectureware.org/pub/ newline documentation/4.3.1/html/contents/core_reference.html.* Accessed on 2013