Generating Code using Reflection in the Context of Computer-assisted Legacy System Analyses and Reengineering

Peter Krall

Dr. Peter Krall IT Consulting

peter.krall.it@googlemail.com

Abstract

The paper presents a two-step approach for DSM-based generating of substantial parts of tools for legacy-system analysis. In the first step a parser is generated from a formal grammar, representing the syntax of the legacy system's code. This parser is able to recognize the constructs of the legacy code but unaware of any analytic functionality. In the second step the code for the analytic functionality and glue code for integrating the parser-generated code into a framework is generated from a domain-specific model capturing properties of and relations between code constructs to be analyzed. Integration with parser functionality generated in first step and framework is achieved by loading classes and retrieving information using reflection and annotations in the second step.

Categories and Subject Descriptors: *Specialized application languages, Computer-aided software engineering*

General Terms Languages, Theory

Keywords Domain Specific Modeling, Code Generation, Legacy System Analysis, Reflection.

1. Introduction

Legacy system analysis and reengineering is based on associating the existing code with additional semantics by applying additional rules for interpretation of the language. This task, whether done by a computer or a human engineer, requires understanding of the legacy code's syntax. Due to the complexity of such traditional languages as SQL or COBOL, a parser generated from a formal grammar will therefore be part of solutions for legacy system analysis by a computer program. If the tool for legacy code analysis is to be generated from a model, the formal grammar will therefore need to be embedded into the model and transformation of the grammar into a parser, lexer and so will be part of the transformation of the model into a solution.

The generated SQL-, COBOL-, parser will capture language recognition concepts but not the more specific concepts of a narrower domain, following the standard DSM paradigm with respect to model semantics [2,5]. Somehow the functionality required for the concrete task has to be implemented by weaving generated language recognition code and more domain specific generated code into a framework.

YACC-style parser generators accomplished the task by capturing semantic actions as grammar annotations and projecting them into the generated code. This is not optimal with respect to support generation of code for the semantic actions themselves, unless this were to be done by generating annotations in a formal definition from some other source. The new antlr4-architecture supports stricter separation by generating a parser that will build a parse tree from parsed sources, together with a listener [8]. This architecture is the background for the solution presented in the rest of this paper.

2. Reference Project: Data-Flow Analyses

While presenting a concrete project is not the purpose of this paper, reference to a particular project allows drawing illustrations from this project. Therefore this reference project shall be introduced to the extent necessary for understanding the illustrations.

The task of the reference project is analyzing dependencies and data flows in a database by parsing and interpreting SQL sources. Important forces shaping the solution are:

- Since SQL sources must be interpreted, the solution must be able to parse SQL-statements. Therefore the solution will include a parser. Therefore a formal representation of SQL-grammar will be part of the model for generating the solution and a parser generator will be part of the tool-chain.

  Writing a custom parser generator based on a custom DSL for representation of SQL grammar would require a lot of effort. Using an existing parser generator and representing SQL syntax with means of a formal grammar accepted by the parser generator is thus almost enforced.
- While concepts of formal language recognition domain are part of the relevant concepts, the application cannot be reduced to these concepts. Rather, it needs additional concepts like data-flow or dependency which are more specific to the narrower domain. An example may illustrate this. Consider the very simple Create-View Statement:

  ```
  CREATE  VIEW Cute_Animals AS SELECT id, given_name
    FROM Animals WHERE species = 'cat';
  ```

  From the viewpoint of language recognition, this is just a code construct, which is an instance of the construct:

  ```
  create_view_stmt
   : K_CREATE ( K_TEMP | K_TEMPORARY )?
     K_VIEW ( K_IF K_NOT K_EXISTS )?
     qualified_view_name K_AS select_stmt;
  ```

  The semantics of the statement are irrelevant for pure language recognition. From the viewpoint of dependency analysis, the CreateView Statement is interpreted as information concerning the relation between two database objects: View

Cute_Animals depends on Table Animals and, moreover on the value of columns species, id given_name.

- Reusability is a major concern. The solution is conceived as a pioneer project for a family of solutions addressing similar but not identical themes in legacy system analysis and computer-assisted refactoring.

The forces of the concrete project are more or less typical for legacy code analysis. In particular the code which is to be analyzed will usually be written in some feature-rich language and therefore there is no way around the requirement for the application to understand such languages.

3. Generating solutions from composite DSMs including formal grammar

In projects where recognition of language plays a major role, the models used for generating code can be conceptually separated into two parts: First the grammar and second the part capturing the semantic concepts of the narrower domain, which will be called nDSM (for narrow DSM) subsequently. Any DSM in this scenario can thus be conceived as a composite: DSM = grammar+nDSM. In the reference projects the nDSM-part would capture concepts like 'dependency' or 'information-flow'. Some future variants, say: for generating scripts for data-model normalization – will share the grammar part but vary with respect to the nDSM-part. Other variants will require modification of the grammar, e.g. due to vendor-specific extensions of standard SQL or because the customer uses a specific preprocessor.

By virtue of capturing semantic domain concepts from a narrow domain, the role of the nDSM part in this scenario corresponds to the standard DSM-role described in literature [2,5] while the grammar-part may be seen as the model for a more general, or at least different domain. In this view the task of generating solutions from composite (grammar+nDSM)-DSMs is a task of implementing the specific intelligence required for dealing with composite models containing components with different semantics [6,9].

3.1 Resolution of forces in generating solutions from composite DSMs

The most important force in the rationale of the solution presented here results from the existence of language recognition tools and the prohibitive costs for replacing or modifying them. This force yielded the decision to use the grammar→code transformation provided by the antlr4[8] generator as is.

The previous decision led to the question how to integrate code resulting from nDSM→code transformation with the generated parser code. There are two principal alternatives: either the nDSM→code generator must inspect the formal grammar and predict what the parser generator will make of it or it must look up the results of the parser generator's production. The latter alternative can be broken down further into two alternatives: parse the antlr4-generated SQL-parser's java-code or load the class-object and use reflection for information retrieval.

Loading class objects and retrieving information from these objects is often used for runtime code generation but can also be used for generating GPL-code that does not depend on reflection[4]. Using reflection is easier than parsing Java-code. The approach also matches the idea that only information about the imperative semantics of the antlr4-generated programs should be used, not information about the code itself, and that standard OO-mechanisms rather than modification of code from other sources should be used for integrating the custom code generator's productions.

The resolution of forces led to a two-step process:

1. The language recognizer classes are generated from a formal grammar by the standard antlr4 parser generator.
2. A specific codegenerator loads the generated classes and additional nDSM-information, as well as some complementary information retrieved from framework packages. It inspects the class structure using reflection, relates it to the additional information and generates code that will provide the solution. In this second step the generated language recognition functionality is considered part of the target platform.

In this context there is no real model-collaboration. The parser is generated by antlr4 without any concern for the nDSM and the nDSM→code generator does not know anything about the origin of the classes it inspects. This is a particular kind of intelligence implemented in the generator but no mechanism of model collaboration. A main advantage is the elimination of the need for a mapping between the meta-models of grammar and of the nDSM-part: The nDSM→code generator just needs to know about the reflection API, not about specification of languages by formal grammars.

3.2 Technical aspects of the solution

The architecture of the solution is based on the antlr4-generated language recognition functionality implementing the observer pattern by a generated listener: the parse tree can be walked depth-first and a Listener class will receive messages when an enter- or exit-event on a node happens.

By default, all messages when traversing the parse tree will be received by a generated listener. The name of course depends on the name of the grammar. In the reference project it is SQLBaseListener. Semantic actions can be added to language recognition by overriding the event handling of the generated listener class in a specialization class

```
SQListener extends SQLBaseListener.
```

For example, assume we wanted to write the tokens sequence of a statement into database whenever our SQL-parser encounters a SELECT-statement. Then we could override the method in SQListener handling enter-context event:

```
@override
enterSQL_stmt_context(SQL_stmt_context ctx) {
  write2database(ctx.getTokens());
}
```

where 'write2database' is a method defined as part of the framework. Our solution for combining code generation by antlr4 and the custom code generator is based on this mechanism:

- In the preliminary step antlr4 is used to generate parser and listener classes from a formal grammar. No particular modifications are made to the code generator.
- The custom code-generator ('custom' distinguished from antlr4-code generator) loads the antlr4-generated parser class:
  `Class<?> PARSER_CLASS = SQLParser.class;`
- The parser class declares a context class for every rule. The custom code generator iterates over the set of context classes

and, for every such class, generates code of a type for the respective code construct class:

```
for(Class<?>codeConstructContextType:
  PARSER_CLASS.getClasses()){
   if (PARSER_RULE_CONTEXT_CLASS
    .isAssignableFrom(codeConstructContextType)){
      generateType(codeConstructContextType);
   }
}
```

- The constant *PARSER_RULE_CONTEXT_CLASS* refers to the standard generated context-class. The invoked method *generateType()* will emit the code of a Java class with the name of the defining rule (with capitalized first letter), e.g. *Select_stmt or With_clause,* etc.
- There is a non-generated base class *CodeConstructBaseType* which essentially provides basic tree node functionality.

  CodeConstructBaseType contains a collection

  Set<CodeConstructBaseType> children

  and implements a traverse method.
- A generated class

  class CodeConstructGenericType

  extends CodeCostructBaseType

  is created in every generator run in addition to classes corresponding to production rules of the grammar. So, after the generator run, there are classes

  class CodeConstructGenericType

  extends CodeConstructBaseType

  and classes

  class Assigment extends CodeConstructGenericType

  ..

  class From_clause extends CodeConstructGenericType

  .. and so on, for every production rule.

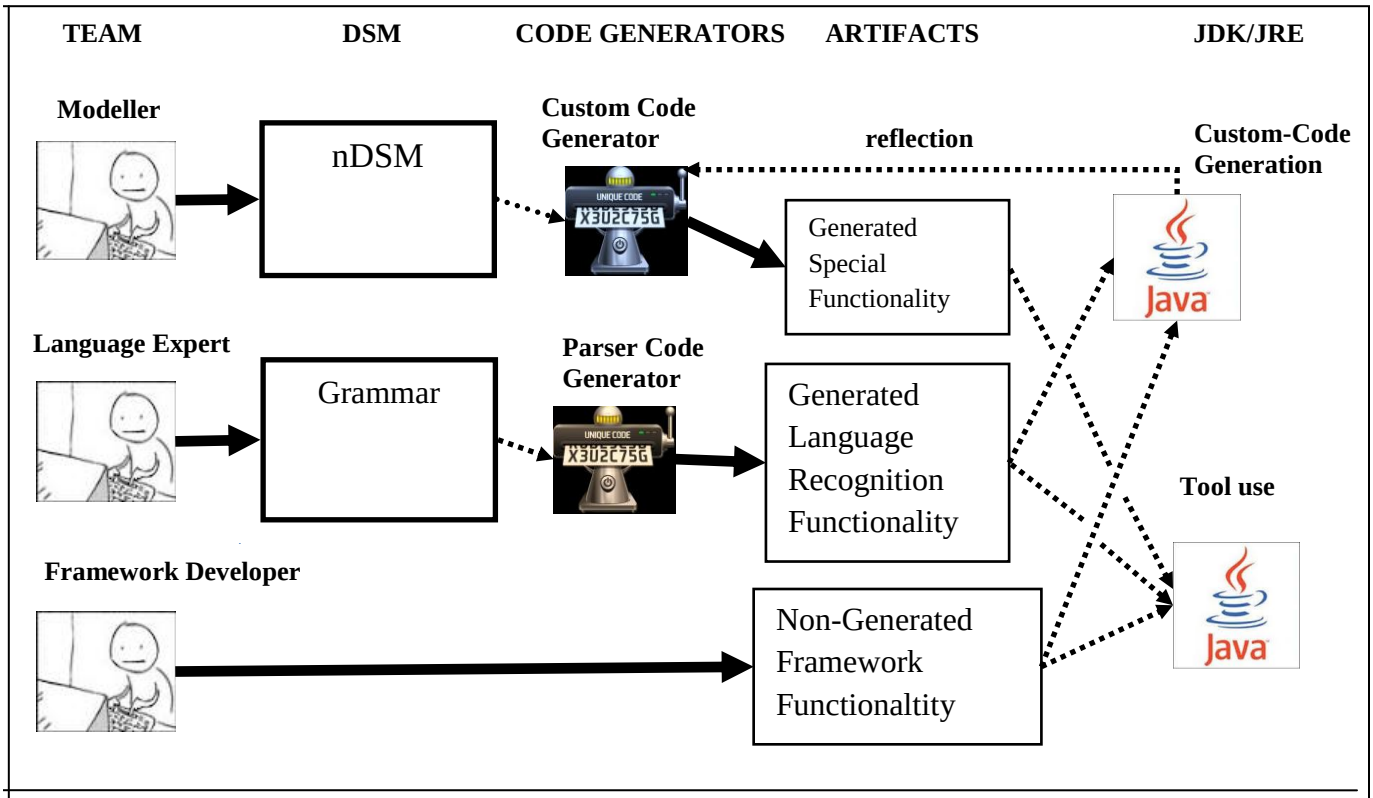

Figure 1: Flows of information and artifact production in the composite-model scenario

Legend:

→ artifact production: The artifact at the arrow's head is produced by the actor at the other end

⇢ supply and use of information: The using actor (at the arrow's head) retrieves information provided by the supplier at the other end of the arrow and uses it for some action, i.e.: code generators retrieve information from models plus, in the case of the custom code generator, the JRE. JRE/JDK retrieve information from source code (compilation and class loading steps are not distinguished).

Remark: The code generated by the Custom Code Generator is part of the application but not used in the context of code generation; therefore there is no arrow from 'Generated Special Functionality' to JDK/JRE in 'Custom Code Generation'.

- The generated CodeConstructGenericType class implements an empty 'accept' method for every special node class.

```
public class CodeConstructGenericType
      extends CodeCon structBaseType {
  public void accept (Alter_table_stmt child) {}
  public void accept (Analyze_stmt child) {}
  public void accept (Any_name child) {}
```

  and so on, for every production rule. This method is invoked (by generated code) when the syntax tree is built up whenever the parent of a node is set. This is one of the extension points used by code generated from nDSM, as will be explained below.

- The generator also creates a listener which is indirectly based on the antlr4-generated class with a non-generated class providing utility functionality injected into the class hierarchy between the antlr4-generated base class and the custom-generator generated class:

```
class SQLExtendedBaseListener extends SQLBaseListener
class SQLListener extends SQLExtendedBaseListener
```

  The classes in this hierarchy have three different origins:

  SQLBaseListener is generated by antlr from the grammar.

  SQLExtendedBaseListener is created manually.

  SQLListener is generated by the custom code generator, using the nDSM as well as information retrieved by inspecting *SQLBaseListener.*

- The generated listener builds a tree of code constructs by invoking the constructors of matching CodeConstruct-classes and invoking push- and pop methods. This is done by overloading the enter- and exit-methods for every production:

```
class SQLListener extends SQLExtendedBaseListener{
  @Override
  public void enterAlter_table_stmt
       (SQLParser.Alter_table_stmtContext ctx) {
    push(new Alter_table_stmt(ctx));
  }
  @Override
  public void exitAlter_table_stmt
     (SQLParser.Alter_table_stmtContext ctx) {
    pop();
  }
```

  … and so on, for every production rule.

  The syntax tree built thereby consists of nodes of different classes, each of which corresponds to a production of the grammar.

- The first possibility for adding special code generated from the nDSM-part of the model is by overloading the 'accept'- and 'traverse'-methods in the antlr4-generated *CodeConstruct*-classes by methods in their specialization generated by the Custom Code Generator. For example, assume that the 'From-clause' has a special role *TableAliasProvider* in the Select statement. By a mechanism described below, From clause will be declared in implementation of interface *TableAliasProvider:*

```
class From_clause extends CodeConstructGenericType
    implements TableAliasProvider
```

- The *Select_stmt* class will declare a member of the type *TableAliasProvider* and set the value when a *From-clause* is added:

```
public class Select_stmt extends StatementType {
  public Select_stmt (Select_stmtContext context) {
    super(context)
  }
  TableAliasProvider tableAliasProvider
  @Override
  public void accept(From_clause child) {
    this.tableAliasProvider = child;
  }
}
```

- The *tableAliasProvider* can then be used by the traverse-method of Select-statement for resolving references. This illustrates the relation between the model semantics of the grammar and the nDSM-part: From the viewpoint of language recognition, the From-clause just is a sequence of symbols appearing as part of the Select-statement. From the viewpoint of dependency analysis, it defines the source tables for the select and their local names.

- There is a second kind of mechanism used by the nDSM→code generator which also works using reflection: The CodeConstruct-classes can be declared to have roles or inherit functionality declared in framework interfaces or implemented in framework classes specializing the generated CodeConstructGenericType class. However, the model should not know about the implementation of the framework.The solution works on basis of custom annotations. The nDSM defines abstract roles, like

  '*STATEMENT*' or '*TABLE_ALIAS_PROVIDER*'.

  A non-generated class or interface can declare responsibility for this role with a custom annotation, e.g.:

```
@SemanticMetaData(getAbstractType = "STATEMENT")
public abstract class StatementType
   extends CodeConstructGenericType
```

  The code generator loads all classes from the framework:

```
Reflections reflections =
 new Reflections(CODE_CONSTRUCT_TYPE.getPackage()
                                    .getName());
codeConstructBaseTypes =
 refletions.getSubTypesOf(CodeConstructBaseType.class);
```

  When generating a type associated with some special base functionality by the nDSM for a grammar production, the Custom Code Generator will look for a class with matching annotation and use this as base class:

```
if (codeConstructBaseType
        .getAnnotation(CodeConstructBaseType
                       .SemanticMetaData.class)
        .getAbstractType().equals(declaredBaseType))..,
```

  where the symbolic type (e.g. STATEMENT) is associated with a production in the nDSM. This mechanism is responsible for appearance of StatementType rather than CodeCon-

structGenericType in the base-class declaration of class *Select_stmt:*

public class Select_stmt extends StatementType

Interfaces are handled analogously. This allows adding non-generated aspects to generated classes without manipulating generated code.

The solution supports clear separation of concerns and avoids any kind of code manipulation in the aftermath: Whatever is generated by either the antlr4-generator or the custom code generator is concerned 'sealed' and, conversely, manually edited code is not modified by either generator.

3.3 Alternatives and related work

Even though this paper is intended to be a technical-level presentation of a solution rather than a meta-level comparison of different approaches, some sketch of alternatives and the rationale for the presented solution may be appropriate. The most important alternatives are:

a. Decoupled model-code transformations with integration based on reflection. This is the solution described in this paper.
b. Extended Grammar: Integration of all information used for code generation in one source, including the grammar and everything else used by a single, or possibly several code generators.
c. Component-architecture: Several models capture different aspects. They are integrated by a framework for component integration.
d. Bytecode generation: Also uses reflection but generates bytecode for VM directly rather than GPL-sources for compilation by a standard compiler.

The dominating criterion for comparison between these approaches has been the support for dealing with two dimensions of variation in requirements:

- Rather special requirements will pop up in dealing with legacy systems: Oracle allows to use the keyword 'DEFAULT' or ':=' for assignment of a default value to a parameter, so that two sequences of characters should be treated as alternative productions of the same rule - unless the customer wants to change the database vendor in which case the use of expressions supported by Oracle but not by SQL Server will become the subject of interest. Or: In some countries, like Germany, insurance liability insurance is issued for single vehicles bur in other countries, like Austria, they are associated with license plates which might be used for up to three cars. What happens if a formerly 1-1 relation becomes m-n when the company wants to use the IT-system developed for Germany for both countries? Even comments can become subject of analysis: Say: a team member is going to retire - in which change histories is he the last one to appear?
- There is some variation with respect to legacy systems' code. This includes vendor-specific extensions of standard languages but does not necessarily reduce to such extensions. An example for another source of variation are customer-specific pre-processors.

Rather than aiming at development of a tool providing complete coverage for possible challenges out of the box, focus has been on responsiveness to unpredicted requirements. This yields preference for decoupling models capturing the two dimensions of variation and for a lightweight, easily adoptable integration mechanism.

Option a) is the solution presented here. The approach separates concerns associated with the two dimensions of variation in requirements. The reflection mechanism used for integration is stable and easy to use. There are some consequences that might be arguments against the approach in other scenarios, as discussed lateron, but in the concrete scenario of the project this has not been an issue.

Option b) corresponds to embedding definitions of semantic actions into definitions of rules in a grammar as known from yacc-like parser generators. More recent developments include improved support for modular extension of reusable grammar core with various extensions [3]. The argument for preference of a) against b) concerns responsiveness for anticipated use cases: If the customer wants to know the answer to some specific question about a legacy system, the most efficient way for building the tool will be writing some code for special functionality, modelling the relation between grammatical constructs and the special functionality in a nDSM and generating the glue code. Using reflection is a simple way for retrieving information already present in the class system, thereby allowing to limit the information to be captured in the nDSM. This is a purely pragmatic argument based on anticipated efforts for anticipated future requirements and thus no more than rational motivation can be claimed.

Option c) has been proposed for capture of different aspects of a domain in separate models[9]. The combination of grammar and nDSM may be seen as an instance of the general class of composite meta-models. The argument for favouring a) over c) has mainly been a consideration for costs: reflection comes for free and is easy to use while integration into a component framework would have required additional work, particularly for integrating the parser generator in the meta-model.

Option d) has been successfully applied for automated embedding of special business logics into frameworks, e.g. in the context of adding persistence or publication as web [1]. From a logical point of view this is a related challenge to adding analytic functionality to a parser. The option has been discarded nevertheless due to the additional technical complexity of bytecode generation. Also, the solution discussed here does not require adaptation of program functionalities based on information becoming available only at runtime, which would be the key argument for generating directly loadable byte- rather than source-code.

As always when forces need to be weighted for finding a balance, there is no conclusive proof for superiority of the selected approach against all alternatives. Moreover, the tie of the nDSM to implementation may deserve attention in some scenarios: The DSM contains information concerning additional function to be associated with constructs in the legacy system's code, either by generating these functionality or by generating glue code for integration into the framework. This implies that nDSM-modelling requires understanding of the grammar and both, the antlr-generated code and the framework code. A model of this nature thus does not shield domain experts from technical aspects. The reflection-based integration concepts thus is incompatible with the goal of integrating experts from non-technical domains, at least in the form presented here. The concept indeed requires profound technical understanding on the side of the modellers. This constraint being fulfilled, efficiency is improved and architecture is stabilized by lifting level of abstraction and leaving the implementation to the code generators.

3.4 Further Work

The current solution is hoped to become the core of a solution family for legacy system tools based on parsing SQL and, possibly, other languages. For this purpose the nDSM-modeling has to be improved.

Currently the nDSM just is a XML edited model without any specific support. The modeler needs to know what classes exist in the antlr-generated class system and how they relate to grammar elements. Also, there is little protection against inconsistency of the nDSM with grammar or framework.

As long as the modeler, framework builder and grammar definer are one person or a small team of experts understanding what the others are doing, effects of this weakness are not too severe, especially since eclipse IDE in combination with antlr4-plugin provides support for search and navigation. Still, there is room for improvement.

Apart from GUI-related aspects, the model editor should also provide lookup and validity checking. Part of this functionality is already implemented in the code generator – e.g. retrieving a list of all productions defined in the grammar is done already, as is retrieving the classes of the framework and their annotations. But support for nDSM-modeling also requires the nDSM's ability to look up information from grammar: if the modeler shall see that a 'Select' statement contains a 'From' clause which can be associated with a certain role, and if he or she shall not need to read the formal grammar for this purpose, then the nDSM-metamodel must implement functionality for looking up and presenting this information. Unlike in the code generation context, this implies the need for model collaboration in the model-building context. Editor-related aspects of composite models based on different base languages is beyond the scope of this paper, which focusses on code generation, but has been described elsewhere[7,9].

4. Summary

In some scenarios DSMs are composed of a formal grammar and, additionally, a 'nDSM' (= narrow-domain specific model) component. The semantics of the grammar in defined in the domain of language recognition whereas the semantics of the nDSM is defined in some narrow, task-specific domain. This particularly holds for legacy system analysis domains since the legacy systems' code will be written in a language defined independently from the DSM-tool.

The solution described here to handle code-generation for composite DSMs is by a two-step process: First, the grammar is transformed into parser source code without concern for the nDSM. In the second step a custom code generator loads the generated classes, inspects them by reflection and combines the information with the nDSM-interpretation. This eliminates the need for mapping the meta-models of the two components making up a composite DSM and replaces it by using reflection for code generation.

Acknowledgment

The author would like to thank Regine Wegner from ASG Software Solutions for reading earlier versions of this manuscript and giving helpful advice.

References

[1] Chiba, S., "Load-Time Structural Reflection in Java", Proceedings of ECOOP 2000 , pp.311-336, Lecture Notes in Computer Science 1850, Springer-Verlag (2000)

[2] Greenfield J., Short K., Cook S., Kent S., Crupi.J. "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools" ISBN: 978-0-471-20284-4, Wiley (2004)

[3] Efftinge, S and M. Voelter,M ".oAW xText: A framework for textual DSLs". In Workshop on Modeling Symposium at Eclipse-Summit (2006).

[4] Hutchins, DL " Partial Evaluation + Reflection = Domain-Specific Aspect Languages". GPCE Workshop on Domain-Specific Aspect Languages (DSAL) (2006).

[5] Kelly, S. & Tolvanen, J-P."Domain-Specific Modeling". ISBN: 978-0-470-03666-2 Wiley-IEEE Computer Society Press (2008)

[6] Krall, P." How does Intelligent Functionality Provided by the Meta-Model in Model Driven Development Relate to Model Semantics?" In Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.) Proceedings of the 7th OOPSLA workshop on Domain-Specific Modeling. Montreal, Canada. University of Jyväskylä, Technical Reports, TR-38, Finland (2007)

[7] Kuhn,T. et al "Multi-Language Development of Embedded Systems", in Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling, M. Rossi et al., Eds., pp. 21-27. (2009)

[8] Parr, T. "The Definitive ANTLR 4 Reference: Building Domain-Specific Languages (1st ed.)", Pragmatic Bookshelf, p. 325, ISBN 1934356999 (2013)

[9] Occello, A., Casile,O. Pinna-Déry,A-M., Riveill, M. (2007) Making Domain-Specific Models Collaborate"in 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Jonathan Sprinkle, Jeff Gray, Matti Rossi, Juha-Pekka Tolvanen (eds.), pages 79-86, Montréal, Canada. (2007)