Java code generation based on OCL rules

Marijana Rackov [*], Sebastijan Kaplar[**], Milorad Filipović[**], Gordana Milosavljević[**]
[*] Prozone, Novi Sad, Serbia
[**] Faculty of Technical Sciences, Novi Sad, Serbia
*marijanarackov@gmail.com, **{kaplar, mfili, grist}@uns.ac.rs

***Abstract*— The paper presents an overview of the underlying framework that provides Java code generation based on OCL constraints used in Kroki tool. Kroki is a mockup-driven and model-driven tool that enables development of enterprise applications based on participatory design. Unlike other similar tools, Kroki can generate a custom set of business rules along with standard CRUD operations. These rules are specified as OCL constraints and associated with classes and attributes of the developed application.**

I. INTRODUCTION

Software development on internet time dictates very fast and flexible approach to programming where traditional coding "from scratch" is often substituted with model-driven development. The main factors in this shift are market requirements imposed on the companies, forcing them to adapt to constant changes. A lack of communication between client and developers is another source of constant changes which affect delivery time even more [1].

Kroki [4] is mockup-driven and model-driven development tool developed at the Chair of Informatics of the Faculty of Technical Sciences from Novi Sad, Serbia that uses business application mockups and models in order to generate a fully functional prototype of the developed system. Resulting prototype is an operational three-tier business application which can be executed within seconds and used for hands-on evaluation by the customers immediately. Developed prototype offers not only basic CRUD operations that can be easily generated from the specification, but also, an implementation of the custom business rules specific for the currently developed system. These rules are represented as OCL [5] constraints.

In order to transform these OCL constraints into working programming code, a Dresden OCL parser is incorporated into Kroki tool and custom code generator for OCL is developed. This paper presents mechanisms and techniques used to implement the code generator and provide examples how it is used and what the resulting programming code looks like.

The paper is structured as follows: Section 2 reviews the related work. Section 3 covers the basic concepts of the solution. It covers introduction to Kroki elements that require OCL rules specification and an overview of the implemented components that enable OCL constraints specification and processing. Section 4 presents some techniques used to generate Java code from OCL expressions, while Section 5 gives an example of how it is incorporated into Kroki tool.

II. RELATED WORK

Related work presented here focuses primarily on the OCL implementations and accompanying tools used to generate programming code from it. Most of the analyzed tools do not implement the complete OCL specification.

A. Dresden OCL

Dresden OCL [6] is an open-source tool for writing, parsing and implementation of the OCL code and the generation of Java, AspectJ and SQL code from it. It can be used as stand-alone Java library or as Eclipse plugin. If used as Eclipse plugin, it provides its own Eclipse perspective which simplifies constraint definition. Generated AspectJ code can be highly customized by specifying various verification and error handling options. SQL code can be generated as one of the following versions: Standard SQL, Postgre SQL, Oracle, and MySQL. It supports Ecore [7] models import and models specified from reverse-engineered Java programs.

Dresden OCL parser is used in our solution for parsing OCL expressions and constraints, but we have implemented our own code generator for Java.

B. Octopus

Octopus (OCL Tool for Precise UML Specifications) [8] is an open-source Eclipse plugin that provides an editor for writing and verification of OCL constraints and generation of Java programming code. It also supports model creation via textual UML specification or by importing XMl representation exported from external modeling tools. Octopus generates a Java method for each OCL constraint and provides numerous customization options for generated code. There is also an option for generating additional Java classes to store hand-written code which is integrated with the generated code.

C. OCLE

OCLE (OCL Environment) [9] is a stand-alone application that enables the creation of OCL models and constraints and code generation. It provides its own graphical modeling editor in which new models can be created, but it also supports XMI model import. OCL constraint editor enables code formatting and highlighting which is also very helpful. OCLE supports all of the OCL constraint and data types, but not all of the operators.

OCLE generates Java code which depends on some libraries contained in the OCLE tool which need to be imported separately. Also, a Java method is generated for every constraint, which makes some of the generated methods somewhat vast and cumbersome.

D. OCL2J

OCL2J [9] is another tool that generates AspectJ code from OCL constraints. It supports most of the OCL operations, but not all of the constraint types. Constraint code and the modeled application code can be generated

separately. If the constraint is broken, generated code throws an exception.

III. IMPLEMENTING THE OCL CONSTRAINTS

This section describes Java code generation process based on OCL constraints attached to Kroki derived fields (attributes), classes and class operations.

Kroki has three derived field types: Aggregated, Calculated and Lookup fields. These fields expand the *VisibleProperty* stereotype and add their own tags (Figure 1) [2].

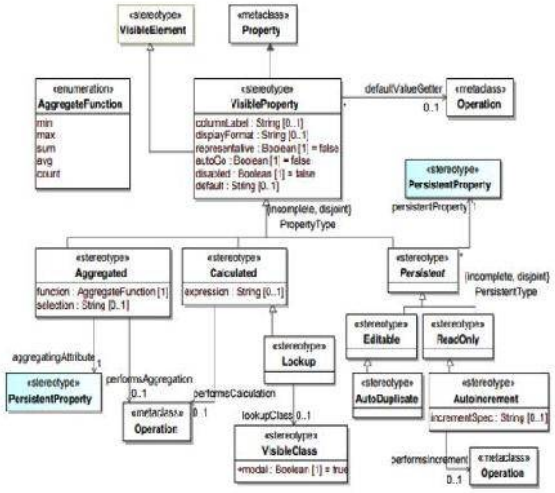

Figure 1. A part of a metamodel that defines form fields within Kroki [2]

Values of aggregated fields are calculated with the use of functions such as: Min, Max, Sum, Average, and Count. Field being aggregated can be of any kind (including derived fields).

Calculated fields are those whose values are calculated based on a formula. Formula is given with the OCL expression by filling the expression attribute in calculated field (please see Figures 5 and 7).

Values of a lookup fields are based on the value of OCL expressions that specify navigation to the class and its attribute we want to display.

A. *Processing OCL constraints*

Process started in `ProjectExporter`, Kroki's class for exporting modeled application to one of its engines (figure 2). `ProjectExporter` parses OCL expressions and prepares them for further analysis with `ConstraintAnalyzer` and `ConstraitBodyAnalyzer` classes. Finally, prepared set of rules is passed to `ConstraintGenerator` class which uses freemarker [11] templates to generate `EntityConstraint` classes. Figure 2 shows main steps of generating Java classes based on the given constraint. Blue color depicts classes that are used for generating Java code, while the yellow color shows helper classes that are used in the particular step.

`EJBClass` which instance is created for every defined class in the modeled application is extended with additional references to classes that model OCL constraint and its programming metadata. Those classes are `Constraint`, `Operation`, and `Parameter`. Figure 3 shows relationships of these classes.

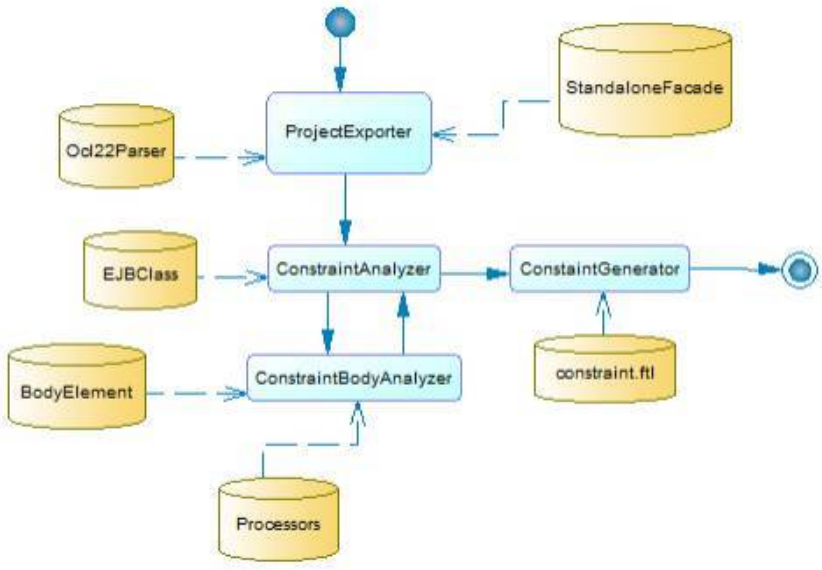

Figure 2. Constraint generation process

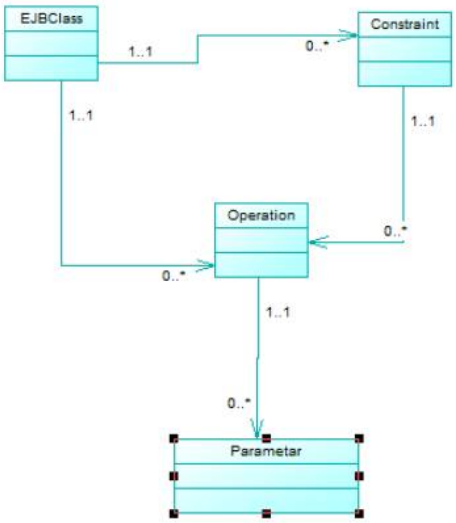

Figure 3. Kroki classes that support definition of OCL constraints

The `Constraint` class represents an OCL constraint while the `Operation` represents Java method that enforces given constraint. Method parameters are modeled with the `Parameter` class.

B. OCL expressions parsing

Parsing of the OCL constraints and generation of the corresponding Java methods is incorporated in the process of Kroki prototype execution. When analyzing each entity in the developed system, OCL expressions are extracted from derived fields and passed to Dresden OCL `OCL22Parser` parser [6], which assembles a list of parsed constraints for the given model (Kroki project). `ConstraintAnalyzer` associates OCL constraints from the parsed list with the corresponding `EJBClass`.

C. Java operation parsing

Another task for `ConstraintAnalyzer` class is building the `Operation` class instances for each `Constraint` instance. Data extraction starts with parameter definition. For ordinary (invariant) OCL constraints, parameters are built from the `Constraint` parameter list, while the definition constraint parameters are parsed from the OCL expression. Once the parameters have been prepared, the operation header can be constructed. The operation header is a string that represents future Java function signature. After that, OCL expressions from the constraint body are parsed by the `ConstraintBodyAnalyzer` class. If the expression contains an operation (data operation, mathematical or logical expression), it is delegated to one of the corresponding `Processor` classes for the further processing. The Figure 4 shows a list of all of the available `Processor` classes and the way they are connected to `ConstraintBodyAnalyzer` via the `Controller` class.

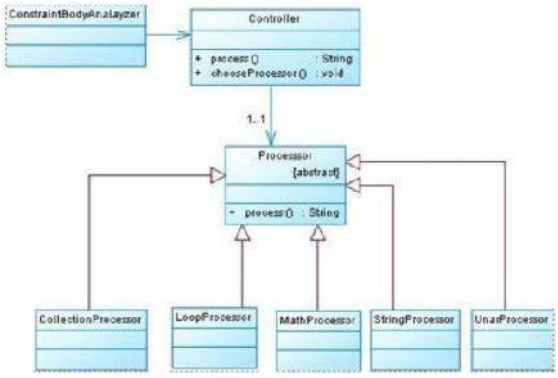

Figure 4. Operation processors

1) Binary operations

Supported binary logical and arithmetic operations are given in Table 1.

Since Java expressions are being built recursively, complexity of the operation is not an obstacle for parsing. The first step when processing the operations is checking whether the operator syntax is the same in Java and OCL, and altering the expression if it is not the case. An example of this are logical operators where OCL "and" “and” needs to be translated to Java "&&" and so on.

The binary operation that is not directly available in Java programming language is implication (implies). In order to support this operation, a custom function that checks whether one boolean parameter implies the other is integrated into each generated class. Another operator worth mentioning is the equality operator. When parsing equality check, if the operands (or return types of the functions used as operands) are integer or floating point numbers, the "==" operator is used, of all other types, "equals" function is generated.

Operation	Operator
Addition	`+`
Subtraction	`-`
Division	`/`
Multiplication	`*`
Modulo	`%`
Integer division	`div`
Equality	`=`
Less than	`<`
Greater than	`>`
Less than or equal	`<=`
Greater than or equal	`>=`
Not equal	`<>`
Logical and	`and`
Logical or	`or`
Exclusive or	`xor`
Implication	`implies`

Table 1. Supported binary operations

2) Other operations

For the sake of simplicity, a detailed overview of processing of the other operations is hereby skipped. This subsection covers some basic information on this topic

and provides an insight into some specific cases. All mathematic operations processed by the `MathProcessor` (`abs`, `min`, `max`, `round`, `floor`) are directly supported in Java. The same holds for the supported string operations (`concat`, `size`, `toLowerCase`, `toUpperCase`, `substring`), with the exception of the size function, which is used for calculating a collection size in Java. An additional check needs to be executed on the operand of the size function and, if its type is string, `length` is generated instead of a `size` function.

IV. JAVA CODE GENERATION

After all OCL expressions have been processed and the constraint lists of `EJBClass` objects have been populated, processed data is passed to `ConstraintGenerator` which generates corresponding Java classes. `ConstraintGenerator` is a generator class added to Kroki tool in order to support code generation based on OCL expressions. A full list of Kroki code generators is presented in [4].

`ConstraintGenerator` uses freemarker [11] template engine to generate files. For each entity of the modeled application, a Java class is generated in the same package with the constraints specified for it. The constraint class name is generated based on the `EJBClass` name, according to the pattern: `<Class name>Constraints.java`. The constraints class also extends the corresponding `EJBClass` in order to gain access to its methods and attributes. Also, during constraint processing, a list of Java import declaration is assembled for all of the used data types and classes that need explicit imports in Java. According to the list, a custom import section is generated for each constraint class. Basic constraint types (invariant, precondition, postcondition, definition, body) are generated in the constraint template while additional operations are defined in separate template files and imported into constraint template. That way, a dynamic support for OCL operations is enabled where additional functions can be added by specifying a template file and mapping it to a function name in the `ConstraintAnalyzer` class. As an example of one external template, the Listing 1 contains a freemarker template for the `exists` Java function that is generated for OCL function of the same name.

```
public Boolean ${method.name}{
  Iterator<${ method.iterType}>
    iter=${ method.forParam}.iterator();
  while(iter.hasNext()){
    ${ method.iterType} x =
       (${method.iterType})iter.next();

         if(${method.ifCondition})
           return true;
         }
         return false;
}
```

Listing 1. Freemarker template for *exist* OCL function

This function implements the OCL operation `exists`, that checks whether at least one element of the collection conforms to the specified condition. The `method` attribute in the template is `Operation` instance from the class operations list. Its attributes contain the information processed by the `ConstraintAnalyzer`. The `iterType` attribute holds the data type contained in the collection, and `ifCondtion` is Java `if` statement parsed from the OCL expression. The usage of iterators ensures that the generated method can be used on any Java collection type (if `get()` function is used, it could only work on `List` collections). If the method with the same name is already generated in the same class (e.g. an OCL defined method with the same name, but different functionality), a unique numeric suffix is added to the function name. If the operation is called upon another operation that returns a collection, the mentioned operation needs to be processed and generated before processing the current operation. An iterator is then created on the collection returned from the first operation.

V. OCL CONSTRAINTS IN KROKI

As mentioned before, Kroki supports special UI elements called *derived fields*, defined in the underlying DSL (EUIS DSL[12]) Those fields are used to hold the values that are not entered by the user, but instead, they are calculated from other values in the business system. Derived fields are located in the Actions section of the Kroki component palette (Figure 5).

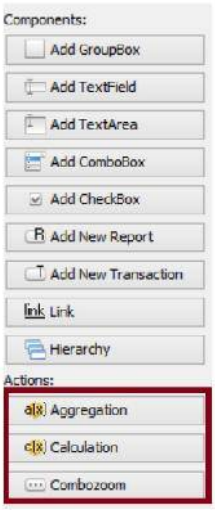

Figure 5. Kroki derived fields

Aggregated fields hold the values that are calculated according to one of the supported functions (`min`, `max`, `sum`, `avg`), as shown in Figure 6.

Calculated fields can specify a custom expression for their values calculation. The settings panel for calculated field is shown in Figure 7 and it contains "Expression" text area which is used to specify OCL expression that defines rules imposed on the specified field value. The following subsections illustrate the usage of the calculated fields with two examples of the OCL rules and the accompanying generated code (Figure 7).

```
context Driver inv:
self.age >= 18
```

Listing 2. Driver age OCL constraint

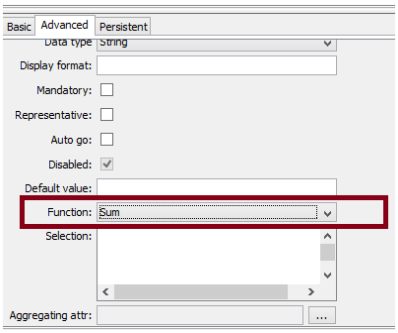

Figure 6. Aggregated field settings

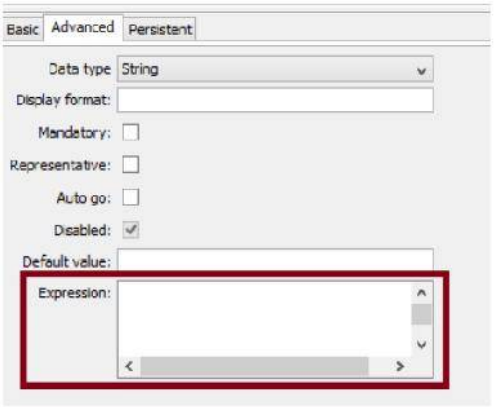

Figure 7. Calculated fields settings

A. *Driver age restriction*

Suppose that enterprise application we are developing contains a record about drivers and the vehicles used in client's company. The rule is that a driver must be at least 18 years old. In order to impose this restriction, a field that specifies driver age has to be a calculated field, and an OCL invariant constraint given in the Listing 2 has to be applied to it.

`ConstraintAnalyzer` output while processing specified expression is given in Listing 3, and the generated Java function is shown in Listing 4.

```
>=     OperationCallExpImpl       Boolean
age    PropertyCallExpImpl        Integer
18     IntegerLiteralExpImpl      Integer
```

Listing 3. OCL expression processing steps

```
public void checkdriverInvariant0()
  throws InvariantException {
  boolean result=false;
  try {
    result = (getA_age() >= 18);
  }catch (Exception e) {
      e.printStackTrace();
  }
  if(!result ) {
    String message = "invariant " +
    + "checkdriverInvariant0" +
    + "is broken in object '" +
    this.getObjectName() +
    + "' of type '" +
    this.getClass().getName() + "'";
    throw new InvariantException(message);
  }
}
```

Listing 4. Generated Java function that checks driver age restriction

As can be noticed from this example, all generated functions based on the boolean constraint are void Java functions, where `InvariantException` is thrown if constraint is broken.

B. *Number of vehicles restriction*

Another restriction for drivers in the observed system is that one driver can have up to three associated vehicles. The OCL constraint for this rule is shown in Listing 5, whereas the generated Java function is in Listing 6.

```
context Driver inv:
self.vehicles -> size <= 3
```

Listing 5. OCL constraint for vehicle number

```
public void checkDriverInvariant1()
throws InvariantException {
  boolean result=false;
  try {
     result = ( getA_vehicles().size() <= 3);
  }catch (Exception e) {
      e.printStackTrace();
  }
  if(!result) {
    String message = "invariant" +
      + "checkDriverInvariant1 " +
      + "is broken in object '" +
      + this.getObjectName() +
      + "' of type '" +
      + this.getClass().getName() + "'";
   throw new InvariantException(message);
  }
}
```

Listing 6. Generated Java function for vehicle number restriction

VI. CONCLUSIONS

The paper presented a process of implementing Java code generation functionality based on the OCL constraints specification in the Kroki tool. Implementing a custom rule specification is a very powerful feature of a development tool since most of the available software of that kind is able only to produce GUI skeletons or working prototypes that support only basic CRUD and navigation features. Implementing OCL constraints

specification combined with the already existing ability to export Kroki models as Eclipse projects and integrate hand-written code with the generated portion [13] makes Kroki prototypes one step closer to the final products.

The solution presented here requested a development of the custom OCL parsing framework based on Dresden OCL parser implementation and custom-made code generator for Kroki tool. The implemented OCL parser supports all of the OCL data types and most of the functions and operations. Currently `oclMessage` and `Tuple` data types are not supported as well as the functions that require AspectJ support in generated Java code. As a further enhancement, a syntax highlighting and a code completion support can be built in the Kroki OCL expression editor areas.

REFERENCES

[1] A. Cockburn, J. Highsmith, "Agile Software Development: The Business Of Innovation", IEEE Computer, pp. 120-122, Sept. 2001.

[2] B. Perišić, G. Milosavljević, I. Dejanović, B. Milosavljević, "UML Profile for Specifying User Interfaces of Business Applications", Computer Science and Information Systems,Vol. 8, No. 2, pp. 405-426., 2011.

[3] A. Kleppe, J. Warmer, W. Bast, MDA Explained – The Model Driven Architecture: Practice and Promise, Addison-Wesley, Boston, 2003.

[4] G. Milosavljevic, M. Filipovic, V. Marsenic, D. Pejakovic, I. Dejanovic, Kroki: A mockup-based tool for participatory development of business applications. SoMeT (p./pp. 235-242), : IEEE. ISBN: 978-1-4799-0419-8, 2013.

[5] OMG. OCL 2.2 specification.

[6] B. Demuth, "The Dresden OCL toolkit and its role in information systems development", Dresden University of Technology, http://dresden-ocl.sourceforge.net/downloads/pdfs/BirgitDemuth_TheDresdenOCLToolkit.pdf

[7] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf

[8] Octopus OCL. http://octopus.sourceforge.net/

[9] OCLE. http://lci.cs.ubbcluj.ro/ocle/

[10] A. Rensink, J. Warmer, "Model Driven Architecture – Foundations and Applications0", Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. Proceedings

[11] Freemarker Java template engine, http://freemarker.incubator.apache.org

[12] B. Perišić, G. Milosavljević, I. Dejanović, B. Milosavljević, "UML Profile for Specifying User Interfaces of Business Applications", Computer Science and Information Systems,Vol. 8, No. 2, pp. 405-426., 2011

[13] M. Filipović, S. Kaplar, R.Vaderna, Ž.Ivković, G.Milosavljević, "Aspect-Oriented Engines for Kroki Models Execution", 5th International Conference on Information Society and Technology, Kopaonik, Serbia, March 8-11, 2015, Proceedings

[14] Kroki tool, http://kroki-mde.net/