Automatic Translation of OCL Meta-Level Constraints into Java Meta-programs

Sahar Kallel, Chouki Tibermacine, Bastien Tramoni, Christophe Dony and Ahmed Hadj Kacem

Abstract In order to make explicit and tangible their design choices, software developers integrate, in their applications' models, constraints that their models and their implemetations should satisfy. Various environments enable constraint checking during the modeling stage, but in most cases they do not generate code that would enable the checking of these constraints during the implementation stage. It turns out that this is possible in a number of cases. Environments that provide this functionality only offer it for functional constraints (related to the states of objects in applications) and not for architectural ones (related to the structure of applications). Considering this limitation, we describe in this paper a system that generates metaprograms starting from architecture constraints, written in OCL at the metamodel level, and associated to a specific UML model of an application. These metaprograms enable the checking of these constraints at runtime.

Keywords: Software Architecture, Architecture Constraint, Object Constraint Language, Java Reflect

Sahar Kallel
Lirmm, Montpellier University, France, e-mail: sahar.kallel@lirmm.fr

Chouki Tibermacine
Lirmm, Montpellier University, Farance e-mail: chouki.tibermacine@lirmm.fr

Bastien Tramoni
Lirmm, Montpellier University, France e-mail: bastien.tramoni@lirmm.fr

Christophe Dony
Lirmm, Montpellier University, France e-mail: dony@lirmm.fr

Ahmed Hadj Kacem
ReDCAD, Sfax University, Tunisie e-mail: ahmed.hadjkacem@fsegs.rnu.tn

1 Introduction

Software architecture description is one of the main building blocks of an application's design [4]. It gives us an overview of the application organization that helps us to reason about certain properties, such as quality attributes. In this context, architecture description languages have been created to specify and verify such application architectures without worrying, at first, about the implementation of their functionality. The verification can be especially based on constraints that those languages associate to architecture descriptions. These constraints can be classified into two categories: functional and architectural.

Functional constraints check the state of the architecture's objects. For example, if we consider a UML model (an architecture description) containing a class `Employee` (a component in that architecture) which has an integer attribute `age`, a functional constraint presenting an invariant in this class could impose that the values of this attribute (slot of an object of that class) must be included in the interval [16-70] for all instances of this class. On the other side, architecture constraints analyze the structure of the application, and not objects states. For example, they define invariants (boolean conditions) imposed by the choice of a particular architectural style or pattern, like the layered architecture style [22]. All these constraints can be specified at design stage through a constraint language like the "Object Constraint Language" (OCL) [19], the OMG standard.

In the literature and practice of software engineering there exists a large number of architecture patterns [25, 11, 9] whose architecture constraints have been formalized. But unfortunately, currently architecture constraints can be checked only at design time on design artifacts; they are ignored in the implementation stage. Therefore, a part of the knowledge and the expertise in the implementation of a software project "evaporates". To guarantee that architecture pattern source code will not undergo changes during evolution in the implementation artifacts or at runtime, we need to find a way to check the associated architecture constraints at the implementation stage knowing that with OCL language (for example), we can not check the architecture constraints at this stage. We can opt to rewrite them entirely with languages used by the developers at that development stage. And this task of rewriting all these constraints is tedious, time consuming and error prone. Constraints on the two stages of development (design and implementation) are syntactically different but they are semantically equivalent (conditions on architecture descriptions that are present in the two stages). So why not generate the ones from the others, like code can be generated from UML models? Moreover, most of existing tools for model-to-text (code) generation do not consider the generation of code for constraints associated to models. For those which exist [1, 18, 8], they only translate functional constraints, and not architectural ones.

Considering these limitations, we propose a multi-steps process for translating OCL architecture constraints into Java code. The obtained Java code uses the introspection mechanism provided by the programming language (Java Reflect) to analyze the structure of the application. This choice is motivated by our willingness to use a standard mechanism without resorting to external libraries. Reflection

(introspection) enables language users to analyze architectures and to examine the structure of their classes at runtime. In our work, the generated code is considered as a "metaprogram" since it uses the introspection mechanism of the programming language for implementing an architecture constraint.

The remaining of this paper is organized as follows. In the following section, we illustrate the input and the output of the proposed process to better understand the context of our work. These will serve as running examples throughout the paper. In Section 3, we present our general approach indicating the steps for generating constraints into Java metaprograms. Sections 4, 5 and 6 describe these steps in detail. Before concluding and presenting some perspectives, we discuss the related work in Section 7.

2 Illustrative Example

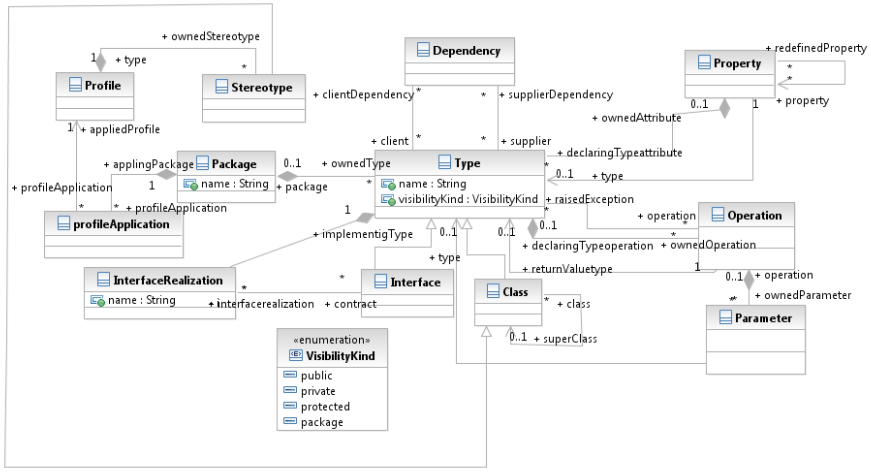

Fig. 1 An excerpt of UML metamodel

To introduce the context of our work, we present an example of an architecture constraint enabling the checking of the "MVC (Model-View-Controller) pattern" [21]. We assume that we have three stereotypes, allowing us to annotate the classes in an application which represent the view (View), the model (Model) and the controller (Controller). This constraint states that the classes stereotyped Model must not declare dependencies with the classes stereotyped View. This makes it possible, among other things, to have several views for the same model, and thus to uncouple these classes that play different roles in the pattern. In addition, the

classes stereotyped Model must not have dependencies with the classes stereotyped Controller. This makes it possible to have several possible controllers for the model.

Using OCL and the UML metamodel (Fig. 1), we obtain the following constraint:

```
1 context Class inv :
2 self.package.profileApplication.appliedProfile
3 .ownedStereotype-> exists(s:Stereotype|s.name='Model')
4 implies
5 self.supplierDependency.client->forAll(t:Type |
6 not(t.oclAsType(Class).package.profileApplication
7 .appliedProfile.ownedStereotype->exists(s:Stereotype |
8 s.name='View' or s.name='Controller')))
```

Listing 1 MVC pattern constraint in OCL/UML

The first line in the Listing 1 declares the context of the constraint. It indicates that the constraint applies to each class of the application ; the meta-class Class is then the starting point for all navigations in the rest of the constraint. Lines 2 to 3 serve to collect the set of classes representing the model (having the stereotype `Model`) by using the navigation package.profileApplication.appliedProfile. ownedStereotype. UML metamodel allows us to get an applied stereotype only starting from the package that contains the modeling element (a class, in our case) and not from the element itself. The problem is resolved in some tools like RSA-IBM where the UML metamodel has been extended with an operation named *getAppliedStereotypes()*, which is inherited by the Class metaclass. In Line 5 we obtain the set of classes which have a direct dependency with the context of the constraint. The remaining of the Listing allows to iterate over the set of class instances and test if it contains classes stereotyped with `View` or `Controller`.

Our goal is to obtain automatically a metaprogram generated from an OCL/UML architecture constraint. The result would be expressed in Java as follows:

```
1  public boolean invariant(Class<?> aClass) {
2    if(aClass.isAnnotationPresent(Model.class)) {
3      Field[] fields = aClass.getDeclaredFields();
4      for(Field aField : fields){
5        Class<?> fieldType = aField.getType();
6        if(fieldType.isAnnotationPresent(View.class)
7        || fieldType.isAnnotationPresent(Controller.class))
8         return false;
9        }
10   }
11   return true;
12 }
```

Listing 2 MVC pattern constraint in Java

The method invariant(...) in Listing 2 accepts as a parameter an object of type Class, representing each of the classes of the application (the classes which compose the application business domain. This excludes classes of the libraries used by the application). Unfortunately, we cannot start navigation from the Package object representing the application package, because in java.reflect, this object does not enable to obtain references to the classes which are declared inside it. The Package object relates to a simple object containing information about the package(e.g. its name). We assume that the dependencies between classes in UML is translated as

the declaration of at least one field in the first class having as a type the second class. In addition, we assume that the equivalent of stereotypes in UML are annotations in Java. The method invariant (..) uses the Java reflect library by invoking, for example, getDeclaredFields() in Line 3 to collect fields, and isAnnotationPresent(..) in Lines 6 and 7 to check if a given type has been marked with a particular annotation.

3 General Approach

We propose a three-step process for generating executable Java code from architecture constraints. We note the presence of two metamodels the first one is the UML metamodel and the second is the Java metamodel that are presented in the following sections. Fig. 2 depicts the process of metaprogram generation. If the OCL constraint needs a refinement, the first step consists in rewriting the OCL constraint in order to make it more accurate and concrete. For example, if the constraint has a navigation to Dependency metaclass (in UML metamodel) then we need to refine this constraint by specifying the different levels of dependencies. Else, the step of transforming OCL constraints from UML metamodel to Java metamodel is established in order to go forward in the process, to the Java code generation. These steps are detailed in the following sections. We did not perform a direct translation from OCL/UML to Java because this translation includes at the same time several transformations: shifting to a new metamodel, changing the syntax of constraints, etc. In fact, our approach requires first a mapping from abstractions of design level to abstractions of implementation level (mapping abstractions from UML metamodel to the Java metamodel) and subsequently a translation of the syntax.

In the literature, there are many languages enabling the specification of architecture constraints (see [23] for a survey). The choice of OCL and UML is motivated by the fact that UML is the *de facto* standard modeling language, and that OCL is its original constraint language. Even if a recent study [20] pointed that UML is not widely used by developers, we all agree that it is a general-purpose modeling language known by a lot of developers. We have intuitively chosen to make constraints programmable in the implementation level in Java because it is a main-stream language in object-oriented programming, which provides introspection capabilities.

4 Constraint Refinement

The refinement mechanism is used whenever some abstractions in the UML metamodel do not have an equivalence in the JAVA language. For example, in the specification of the OCL constraint expressed on the UML metamodel, we have collected all types (Classes) which have dependencies with a specific type by using `supplierDependency.client`. This expression has not a direct equivalence

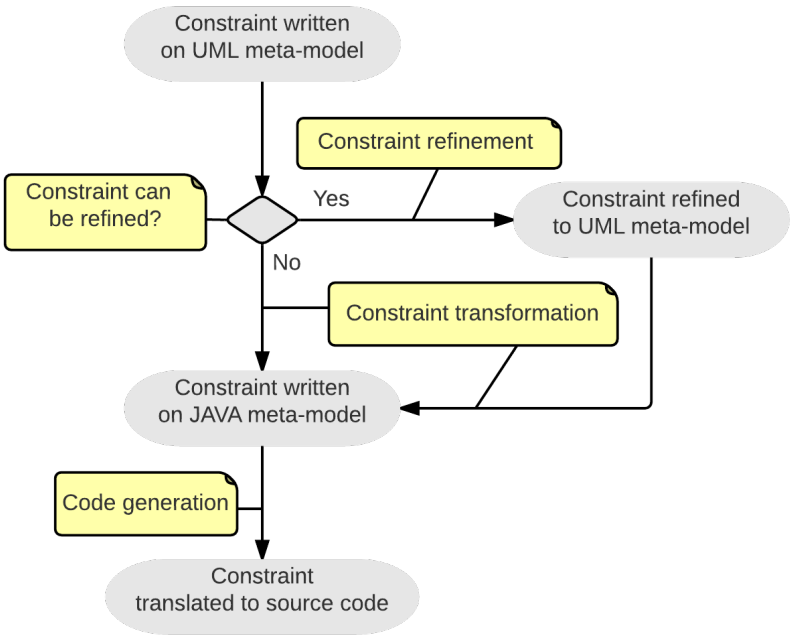

Fig. 2 Approach Description

in Java. As a result, we refine the constraint in the UML metamodel to express the different levels of dependencies.

Often, a dependency between two classes is translated as: i) the declaration in the first class of at least one attribute having as type the second class, ii) some parameters in operations of the first class, have as type the second class, or iii) some operations of the first class, have as a return type the second class.

The previous constraint (Listing 1) is refined as follows:

```
1  context Class inv :
2   self.package.profileApplication.appliedProfile
3   .ownedStereotype-> exists(s:Stereotype|s.name='Model')
4  implies
5   self.ownedAttribute.type->forAll(t:Type |
6    not(t.oclAsType(Class).package.profileApplication
7     .appliedProfile.ownedStereotype->exists(s:Stereotype |
8      s.name='View' or s.name='Controller')))
9   and
10  self.ownedOperation.returnValuetype->forAll(t:Type |
11   not(t.oclAsType(Class).package.profileApplication
12    .appliedProfile.ownedStereotype->exists(s:Stereotype |
13     s.name='View' or s.name='Controller')))
14  and
15  self.ownedOperation.ownedParameter.type->forAll(t:Type |
16   not(t.oclAsType(Class).package.profileApplication
17    .appliedProfile.ownedStereotype->exists(s:Stereotype |
```

```
18|     s.name='View' or s.name='Controller')))
```

Listing 3 Refined MVC pattern constraint

Our constraint in Listing 3 (after refinement) is composed of three sub-constraints (Lines 5- 8, Lines 10- 13 and Lines 15- 18). Each sub-constraint matches one level of the dependencies. In Line 5, the dependency is primarily verified on all attributes defined in classes. Note that oclAsType(Class) operation is used in this constraint to allow navigation between Type and Class through the specialization relation. In Lines 10 and 15, the dependency is related to the types of operation parameters and their returned values.

The refinement of a constraint means a translation of this constraint from an abstract level to a concrete one. In contrast to the translation detailed in the following section, in this step, the translation is an endogenous transformation, since the constraints which are the source and the target of the transformation both navigate in the same (UML) metamodel.

5 Constraint transformation

Before generating code, we transform in this step the OCL constraint specified on the UML metamodel into an OCL constraint specified on the Java metamodel. This simplifies the translation into Java code, since the mapping of abstractions from UML to Java is performed in this step. In order to perform constraint transformation we used a Java metamodel. Unfortunately, none of the metamodels found in the literature and practice satisfied our needs. We relied on Java Reflect library to create a new simplified Java metamodel. In fact, we can define our metamodel relying on Java specification but we deliberately chose Java Reflect because it gives us access to the meta-level of the language and also because it reflects exactly what we can do in the generated Java code. In this metamodel, we limited ourselves to the elements necessary for architecture constraint specification. Fig 3 depicts the Java metamodel that we have defined[1].

The goal of constraint transformation is to replace in an architecture constraint the UML metamodel vocabulary by Java metamodel vocabulary. It had to establish a mapping between UML terms and Java terms that are classified in three categories: metaclasses, roles and navigations.

Table 1 presents for each UML metaclass, role and navigation its equivalent in Java.

We opted for the specification of these mappings in xml, and we have written an ad-hoc program for implementing the transformation instead of using an existing model transformation language like Acceleo [3], Kermeta [2] or ATL [16]. In fact, architecture constraints are not models. We might have generated models from

[1] We assume in this paper that the reader is familiar with UML and Java languages. This is the reason why the two metamodels are not detailed. They are depicted only for accompanying OCL constraints in order to see how navigations in the metamodels are established.

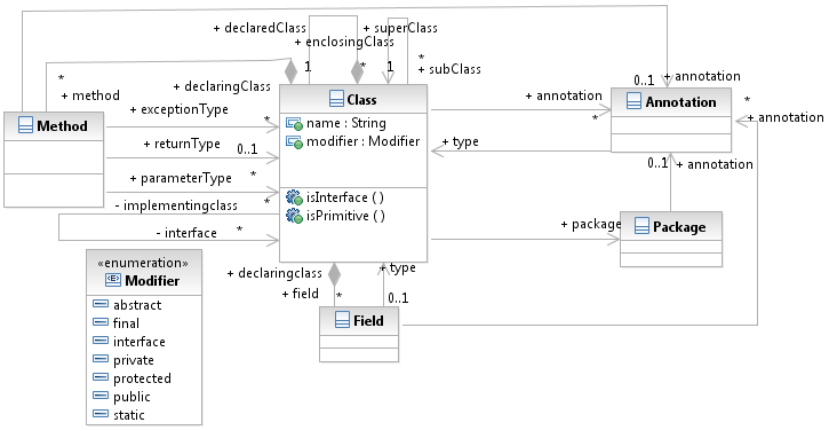

Fig. 3 Java metamodel

	UML	**Java**
Metaclass	**Class**	**Class**
Role	ownedAttribute	field
	ownedOperation	method
	superClass	superClass
	nestedType	declaringClass
	interfaceRealization	interface
	package	package
Navigation	package.profileApplication	
	.appliedProfile.ownedStereotype	annotation
Metaclass	**Property**	**Field**
Role	type	type
	declaringTypeattribute	declaringClass
Metaclass	**Operation**	**Method**
Role	returnValuetype	returnType
	declaringTypeoperation	declaringClass
	ownedParameter	parameterType
	raisedException	exceptionType
Metaclass	**Stereotype**	**Annotation**
Metaclass	**Package**	**Package**

Table 1 Mapping UML-Java (Metaclass, Role, Navigation)

constraints. But this process is complex to implement. It requires to transform the text of the constraint in models, to use a transformation language for transforming these models and then generate again the text of the new constraint from the new model. We opted for a simple solution that consists in exploiting an OCL compiler. It allows to generate an abstract syntax tree (AST) from the text of the constraint. This AST allows us to apply easily different transformations.

We apply the table presented before (Table 1) on the generated AST in order to obtain a constraint expressed in Java metamodel. For applying mappings, we start by navigations, then the roles and finally the metaclasses. The following Listing 4 presents our constraint example after applying the transformation method:

```
1  context Class inv :
2   self.annotation-> exists(s:Annotation|s.name='Model')
3  implies
4   self.field.type->forAll(t:Type |
5    not(t.oclAsType(Class).annotation->exists(s:Annotation|
6     s.name='View' or s.name='Controller')))
7  and
8   self.method.returnType->forAll(t: Type |
9    not(t.oclAsType(Class).annotation->exists(s:Annotation|
10    s.name='View' or s.name='Controller')))
11 and
12  self.method.ParameterType->forAll(t: Type |
13   not(t.oclAsType(Class).annotation->exists(s:Annotation|
14    s.name='View' or s.name='Controller')))
```

Listing 4 MVC pattern constraint in OCL/Java

As indicated in Listing 4, we replace, among others, package.profileApplication. appliedProfile.ownedStereotype by annotation, ownedOperation by method, by respecting the mappings defined before.

The use of declarative mappings gives us the opportunity when the metamodels evolve to modify easily the changed elements. In addition, it allows us to offer a generic method which does not depend on particular metamodels.

6 Constraints generation into Java metaprograms

Code generation consists in translating the constraint expressed in Java metamodel into a Java metaprogram. To generate this code, we relied on the following steps. First, we generate the abstract syntax tree (AST) from the constraint expressed in Java metamodel. Then, when traverse this tree in a Depth-First Pre-Order way in order to generate progressively the java code by relying on rules presented below. It is worth mentioning that the first rule is applied only once in a constraint generation code. The other rules are applied along the analysis of the type of the AST nodes. In fact, if it is a role or navigation then we must apply Rule 2. If is is a quantifier, the rule 3 is then applied and so on.

1. We must consider first that a constraint is represented by a Java method that returns a boolean, which takes as parameter an object of type the metaclass on which the constraint applies (its context). This method is located in a Java class and invokes if necessary other methods that are implemented during the code generation.
2. Each role and navigation in the Java metamodel will be transformed to its accessor method defined in Java. For example, if we navigate to Field, we apply

getDeclaredFields()[2], and if we would like to access to a method return type we call getReturnType().

3. Concerning the OCL quantifiers and the operations, we defined for each one a Java template. Examples are presented in Table 2. select(...) method presented in the last row of the table can be applied on different OCL collection types, like Set or Sequence. During the code generation , each OCL type will be replaced by its Java equivalent.

forAll	ocl	`forAll(ex:OclExpression): Boolean`
	java	`private boolean forAll(Collection c) { for(Iterator i = c.iterator(); c.hasNext();) { if(!exInJava) return false; } return true; }`
exists	ocl	`exists(ex:OclExpression):Boolean`
	java	`private boolean exists(Collection c) { for (Iterator i = c.iterator(); c.hasNext();) { ElementType e = (ElementType) i.next(); if(exInJava) return true; } return false; }`
select	ocl	`select(ex:OclExpression):Sequence`
	java	`List result = new ...(); private list select(Collection c) { for (Iterator i = c.iterator(); c.hasNext();){ ElementType e = (ElementType) i.next(); if (exInJava) { result.add(e); } } return result; }`

Table 2 OCL Quantifiers and operations generation in Java

4. In each quantifier or operation, we traverse recursively the evaluated expression as a sub-constraint and we generate again the corresponding code: if we meet a role or navigation in Java metamodel, we re-apply rule 2. If the quantifier is nested, we re-apply rule 3, and so on.
5. In the case of a nested quantifier (two quantifiers for example are defined one inside the other), the second quantifier frequently needs to use the variables of the first one to define its expression. So, in this case, we store the variables of the

[2] We use getDeclaredField() instead of getFields() to retrieve all attributes (private and public). For those we inherit, we must specify them in the OCL constraint using the role superClass.

first one (parameters of method that correspond to the first quantifier) in order to pass them among the parameters of the method corresponding to the second one.
6. Concerning the logic operators (and, not..), we defined also methods equivalent for each one. These methods are implemented in a class called LogicalOperator. If the constraint contains a logic operator, This class will be declared as a super class of the generated class that contains the invariant method.
7. The arithmetic operations ($>$, $<$, =, ...) and the types (Integer, Real, String, ...) are the same in the generated metaprogram.

In order to better explain the code generation process, Table 3 presents an example of a metaprogram which is generated from our MVC constraint presented in 2. For simplicity reasons, we consider for the dependency between two classes that the first class has at least one method return type having as type the second class.

We have presented in Table 3, for each part of constraint, its equivalent Java code by respecting the rules that was explained previously. The generated code uses the introspection of Java in order to examine the application structure at runtime (`getAnnotations()`, `getMethods()`). This code should be called before and after each method and affectation implemented in the application.

It is worth noting that this code is syntactically different from the optimal code presented at the beginning of the paper (see Listing 2) but they are semantically equivalent. It is evident that the automatic translation does not allow to obtain a code having an optimal complexity. However, it is a valuable tool for developers who will rather focus on implementing the business logic of their application.

7 Related Work

In this section we present works related to OCL constraint transformation and OCL code generation. Hassam *et al.* [13] proposed a method for transforming OCL constraints during UML model refactoring using model transformations. Their approach uses first an annotation method for marking the initial UML model, in order to obtain an annotated target model. Then, a mapping table is created from these two annotations in order to transform OCL constraints of the initial model into OCL constraints of the target one. Their solution of constraint transformations is difficult to establish and it needs some knowledge about model transformation languages and tools. In our work, constraint transformation is simple. It is performed in an ad-hoc way without using additional modeling and transformation languages. In [10], the authors propose an approach to generate (instantiate) models from metamodels taking into account OCL constraints, using CSP (Constraint Satisfaction Problem). They defined some mathematical rules to transform models and constraints associated to them. Cabot *et al.* [7] worked also on UML/OCL transformation into CSP in order to check quality properties of models. These approaches are similar to our transformation process because they use an OCL compiler (DresdenOCL [8]) to transform constraints. But in our approach, we consider source code generation from these constraints, in order to make them executable with application's code.

Constraint	Java Metaprogram

context Class
  inv :
	/*Rule 1*/
	public class Constraint{
	Boolean invariant(Class aClass){
	  //To be completed }
	}

self.annotation	/*Rule 2*/ Annotation [] annotations= aClass.getAnnotations();

->
exists
(a:Annotation|
 a.name='Model')
	/*Rule 3*/
	resultexists1 =
	  exists1(annotations);
	/* Rule 4*/
	private Boolean exists1(Annotation[]
	  annotations)
	{
	 for(Annotation annota: annotations){
	  Class a = annota.annotationType();
	  if(a.equals(Model.class)){
	  return true; }
	  return false;
	}

self.method	/*Rule 2*/ Method[] methods= cl.getDeclaredMethods();

->
forAll(m:method|
not(m.returnType
  .annotation
->
exists
(a:Annotation|
a.name='View'))
)
	/*Rule 3*/
	Boolean resultforAll1=
	  forAll1(methods);
	/*Rule 4 and Rule 6*/
	private Boolean forAll1(Method[]
	  methods)
	{
	 for(Method m : methods){
	  Type type = m.getReturnType();
	  Annotation[] annotations =
	  type.getAnnotations();
	  resultexist2 =
	  not(exists2(annotations));
	  if(!resultexist2) return false; }
	  return true;
	}

Table 3 Example of MVC constraint Code generation

In contrast to CSP, this does not require an external tool for the interpretation of constraints.

In the practice of model-driven engineering, there exist several tools like Eclipse OCL [1], Octopus [18], and DresdenOCL [8, 14, 17] which aim to translate OCL constraints in Java source code. They however transform constraints which are functional and not architectural. These tools translate this kind of constraints into object-oriented programs which do not use the introspection mechanism. The generated code by Dresden OCL is difficult to understand. Indeed, it is true that Dresden OCL is the first tool implemented in this domain, but it extensively uses a vocabulary proposed only by its APIs. This code is normally intended to developers who master, and will continue to use, Dresden OCL, contrary to our work, where code is intended to be used by any Java developer. Besides, with these tools, we need to create beforehand the classes of the model before generating constraints. Other works like Briand *et al.* in [6] and Hamie *et al.* in [12] proposed a tool to transform functional (and not architectural) constraints respectively into Java using aspect-oriented programming and JML contracts.

8 Conclusion

It has been demonstrated that architecture constraints bring a valuable help for preserving architecture styles, patterns or general design principles in a given application after having evolved its architecture description [24]. These architecture constraints are checked at design time. But what if the architecture evolves in the implementation artifacts (the application's programs)? Or, what if the architecture evolves at runtime (through dynamic adaptation, for example)? To be able to check these constraints in that development stage and at runtime, architecture constraints should be translated into an appropriate format: meta-programs.

We have presented in this paper a process for generating Java code starting from OCL architecture constraint specifications expressed in the UML metamodel. This Java code uses the introspection mechanism provided by the programming language. Our process is composed of three steps. The first optional one consists in refining the constraints. The second step allows to transform them into OCL constraints expressed in Java metamodel. The last step generates Java source code relying on specific code generation rules. The reflection (instrospection) mechanism used in our approach is a standard mechanism in Java. Otherwise, we can use static analysis libraries like JDT [15] or ByteCode libraries like BCEL [5] but our goal was to use what is standard in Java and not resort to external libraries. In addition, with reflection, architecture constraints can be checked at runtime (by invoking the invariant method in all the methods of the application where the architecture is changed: new objects are created, references to objects are assigned to fields, etc.).

In our proposal, OCL coverage is not complete. We have implemented a prototype called *MOJaRT: Meta-OCL to JAva Reflect Translator*. It is available for download here: `https://github.com/saharkallel/mojart.git/`. which does

not take into consideration some OCL constructions, like some collection operations (`union`, for example). But this does not have any impact on the work proposed in this paper.

As a future work, we plan to generalize the proposed approach, by specifying architecture constraints in a language-independent way: using predicates on graphs and operations on them and then making automatic transformations towards a particular object-oriented programming language.

References

1. Eclipse ocl. http://www.eclipse.org/modeling/mdt/?project=ocl. URL http://www.eclipse.org/modeling/mdt/?project=ocl
2. Kermeta. http://www.kermeta.org. URL http://www.kermeta.org
3. Acceleo: Implementation of mof to text language. http://www.omg.org/news/meetings/tc/mn/specialevents/ecl/Juliot-Acceleo.pdf. URL http://www.omg.org/news/meetings/tc/mn/specialevents/ecl/Juliot-Acceleo.pdf
4. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley (2012)
5. BCEL: The byte code engineering library. http://commons.apache.org/proper/commons-bcel/. URL http://commons.apache.org/proper/commons-bcel/
6. Briand, L.C., Dzidek, W., Labiche, Y.: Using aspect-oriented programming to instrument ocl contracts in java. Technical Report, Carlton University, Canada (2004)
7. Cabot, J., Clarisó, R., Riera, D.: Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 547–548. ACM (2007)
8. Demuth, B.: The dresden ocl toolkit and its role in information systems development. In: Proc. of the 13th International Conference on Information Systems Development (ISD2004) (2004)
9. Erl, T.: SOA design patterns. Pearson Education (2008)
10. Ferdjoukh, A., Baert, A.E., Chateau, A., Coletta, R., Nebut, C.: A csp approach for metamodel instantiation. In: ICTAI 2013, IEEE Internationnal Conference on Tools with Artificial Intelligence, pp. 1044,1051 (2013)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of Reusable Object-Oriented Software. Addison Wesley (1994)
12. Hamie, A.: Pattern-based mapping of ocl specifications to jml contracts. In: Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on, pp. 193–200. IEEE (2014)
13. Hassam, K., Sadou, S., Fleurquin, R., et al.: Adapting ocl constraints after a refactoring of their model using an mde process. In: BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010), pp. 16–27 (2010)
14. Hussmann, H., Demuth, B., Finger, F.: Modular architecture for a toolset supporting ocl. In: ł UML 2000The Unified Modeling Language, pp. 278–293. Springer (2000)
15. JDT: Java development tools. http://www.eclipse.org/jdt/. URL http://www.eclipse.org/jdt/
16. Jouault, F., Kurtev, I.: Transforming models with atl. In: Satellite Events at the MoDELS 2005 Conference, pp. 128–138. Springer (2006)
17. LCI: Object constraint language environnement. http://lci.cs.ubbcluj.ro/ocle/. URL http://lci.cs.ubbcluj.ro/ocle/
18. Octopus: Ocl tool for precise uml specifications. http://octopus.sourceforge.net. URL http://octopus.sourceforge.net

19. OMG: Object constraint language, version 2.3.1, document formal/2012-01-01. http://www.omg.org/spec/OCL/2.3.1/PDF/. URL http://www.omg.org/spec/OCL/2.3.1/PDF/
20. Petre, M.: Uml in practice. In: Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), pp. 722–731. IEEE Press (2013)
21. Reenskaug, T.: Thing-model-view editor an example from a planning system, xerox parc technical note (may 1979)
22. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
23. Tibermacine, C.: Software Architecture 2, chap. Software Architecture: Architecture Constraints. John Wiley and Sons, New York, USA (2014)
24. Tibermacine, C., Fleurquin, R., Sadou, S.: On-demand quality-oriented assistance in component-based software evolution. In: Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06), pp. 294–309. Springer LNCS, Vasteras, Sweden (2006)
25. Zdun, U., Avgeriou, P.: A catalog of architectural primitives for modeling architectural patterns. Information and Software Technology **50**(9), 1003–1034 (2008)