Specification and Automatic Checking of Architecture Constraints on Object Oriented Programs

Sahar Kallel[a,b], Chouki Tibermacine[a], Slim Kallel[b,c], Ahmed Hadj Kacem[b], Christophe Dony[a]

[a] *LIRMM, CNRS and University of Montpellier, France*
[b] *ReDCAD, University of Sfax, Tunisia*
[c] *SAMOVAR, Telecom SudParis, CNRS & University of Paris Saclay, France*

Abstract

Context: Architecture constraints are specifications of conditions to which an architecture model must adhere in order to satisfy an architecture decision imposed by a given design principle. These constraints can be specified with predicate languages like OCL at design time and checked on design artifacts.

Objective: Many works in the literature studied the importance of checking these constraints at source code level to guarantee quality at that level of the software life-cycle, and to prevent technical debt and maintenance difficulties. That's why we propose a process to check these constraints at that level.

Method: The proposed process takes as input a textual specification of an architecture constraint and enables its static and dynamic checking. It translates architecture constraints into meta-programs and then it uses them with aspect-oriented programming to check the constraints at the implementation stage and at run-time on object-oriented programs.

Results: We experimented an implementation of this process on a set of 12 architecture constraints. The results of this experimentation indicate that our

Email addresses: sahar.kallel@lirmm.fr (Sahar Kallel), sahar.kallel@redcad.org (Sahar Kallel), chouki.tibermacine@lirmm.fr (Chouki Tibermacine), slim.kallel@telecom-sudparis.eu (Slim Kallel), ahmed.hadjkacem@fsegs.rnu.tn (Ahmed Hadj Kacem), dony@lirmm.fr (Christophe Dony)

process is able to detect statically and dynamically architecture constraint violations on basic Object-oriented applications and even on real projects.

Conclusion: The automatic checking of architecture constraints is important at source code level and at runtime. It avoids architecture decisions knowledge vaporization and then facilitates later the maintenance of the application on which these constraints are specified.

Keywords: Architecture Constraint, Object Constraint Language, Meta-program, Java Reflect, AOP, AspectJ

1. Introduction: Context and Problem Statement

Documenting architecture decisions is an important activity in software development processes [1]. Indeed, this documentation allows for, among other benefits, limiting the evaporation of architectural knowledge. Several models for defining this type of documentation exist [2]. These models include both textual (informal) and formal specifications. These models include, among other elements, the description of the decision itself, its state and its alternative decisions. One of the most important elements that compose this documentation of an architecture decision, are `architecture constraints`.

This kind of constraints should not be confused with functional constraints, which are checked by analyzing the state of the running elements constituting the modeled system and which navigate in models like UML class models. An architecture constraint represents the specification of a condition to which an architecture description must adhere, in order to satisfy an architecture decision [3]. For example, an architect may make the decision to use the `Layered` pattern [4]. An architecture constraint allowing the verification of the adherence to this pattern in an architecture description consists of checking, among other things, that elements of a layer must depend only on elements on the same layer or of lower layers. Architecture constraints navigate in meta-models and not in models. They are frequently specified with predicate languages, like OCL (Object Constraint Language) [1].

Functional constraints are used in Design by Contract for ensuring the definition of accurate and checkable interfaces for software components [5].

[1] `http://www.omg.org/spec/OCL/2.3.1/PDF/`

Architecture constraints are used during the evolution of a software architecture for guaranteeing that changes do not have bad side effects on the applied architecture patterns or styles, and thus on quality [6].

In contrast to constraints in mathematics or in constraint programming, architecture constraints are not conditions that should be satisfied by a solution in a combinatorial problem, where we search for an optimal solution among a lot of possible ones. They are conditions that are evaluated to see whether a given single "fixed" solution (our architecture description) satisfies the conditions or not. If the conditions are not satisfied, we are not led to find another possible solution. We should change the current solution (architecture description), by undoing previous changes for example, and then re-evaluate the conditions.

In the literature and practice of software engineering there exists a large catalog of formalized architecture constraints [7, 8, 9]. But unfortunately, currently architecture constraints can be checked mainly at design time on design artifacts. Checking the conformance of software artifacts, with regard to these constraints, downstream in the software life-cycle (during the implementation stage or at runtime) is equally important. 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)? We argue in this work that it is important to check architecture constraints not only at design time but also later in the software life-cycle in order to preserve and make persistent the software's quality.

To be able to check architecture constraints in the implementation stage and at run-time, it is interesting first to see how to specify architecture constraints at that levels. In this case, two solutions are possible. The first one consists in writing a new interpreter, used at implementation phase, for the language used for constraint specifications at design time (like OCL). But this solution can be quickly discarded because it is time-consuming, and it obliges programmers to learn another language (the language used to specify constraints in the design phase, like OCL) to specify their new architecture constraints, in the implementation phase. The second solution is to rewrite the architecture constraints (specified at design time) entirely with programming languages. This task of rewriting manually all these constraints is tedious, time consuming and error prone. In addition, constraints on the design and implementation stages of development are syntactically different but they are semantically equivalent (conditions on architecture descriptions that are present, even implicitly, in the two stages). Indeed, constraints deal

with architectural aspects which are orthogonal. So why not generate ones from the others, like skeletons of code can be generated from UML models?

In the practice of software development, 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, they only translate *functional* constraints, and not *architectural* ones.

In this paper, we propose an automated multi-step process for translating OCL architecture constraints into code. We are using Java as a target language, but a similar approach may be safely used with other programming languages. The obtained Java code uses the introspection mechanism provided by the standard library of the programming language (Java Reflect) to analyze the structure of the application. This choice is motivated by our wish to use a standard mechanism without falling back on external libraries. The generated code is a "meta-program" which uses the introspection mechanism of the programming language for implementing an architecture constraint. In addition, we propose in this paper a complementary automated micro process, which combines static and dynamic constraint checking, based on the aforementioned generated meta-programs. This checking is fully automatic and seamless for users. This process notifies developers on the possible violations of constraints.

The remaining of this paper is organized as follows. In Section 2, we present the general approach indicating the steps for checking architecture constraints using the generated meta-programs. Sections 3 and 4 describe these steps in detail. Section 5 presents the experimentation we have conducted to evaluate the process. Before concluding and presenting some perspectives, we discuss the related works in Section 6.

2. General Approach

Fig. 1 depicts the general steps of our process, which can be seen as a two-phase process (*meta-program generation* and *constraint checking*), the first phase being composed of three steps (after excluding "loading" steps). The process tests first if the OCL constraint, specified in the UML metamodel needs a refinement in order to make it more concrete. For example, if a constraint has a navigation to Dependency meta-class (on a UML metamodel) then we need to refine this constraint by specifying the different kinds of concrete dependencies (for instance, types of fields or parameters). Then,

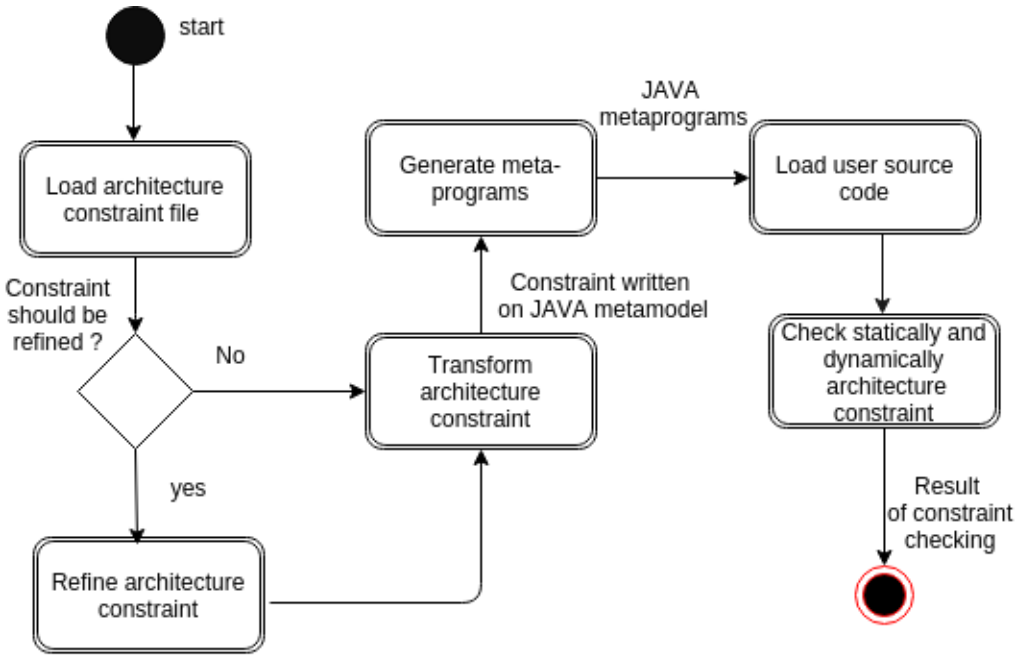

Figure 1: Approach Description

the OCL constraint is transformed to a constraint specified on a JAVA meta-model. Finally, JAVA meta-programs are generated from it.

The constraint checking phase needs the generated meta-programs to check the corresponding architecture constraints at the source code level. This step is based on aspect-oriented software development (AOSD) [10] since the constraints are specified separately from the source code. It combines static and dynamic checking of the constraints.

In the first phase, we did not perform a direct translation from OCL/UML to Java code because this translation includes at the same time several transformations: shifting to a new meta-model, changing the syntax of constraints, etc. Indeed, our approach requires first a mapping from abstractions of design level to abstractions of implementation level (mapping abstractions from UML meta-model to the Java meta-model) and subsequently a translation

of the syntax.

OCL (version 2.3.1)/UML(version 2.4.1) was chosen among many languages enabling the specification of architecture constraints (see [3] for a survey). This choice 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 [11] pointed that UML is not widely used by developers in industry, we all agree that it is a general-purpose modeling language, easy to learn and known by a lot of developers.

We have intuitively chosen to transform constraints in the implementation level into Java programs because it is a main-stream language in object-oriented programming. In addition it implements a small reflective meta-level and provides in its standard library introspection capabilities.

3. Generation of Meta-programs from Constraint Specifications

Before detailing how meta-program generation is performed, we present an example of an architecture constraint specification. This will serve as a running example to illustrate the steps of this first phase of the process.

3.1. Illustrative Example

We introduce an example of an architecture constraint that characterizes the `MVC` (Model-View-Controller) architecture pattern [12]. This constraint navigates in the UML meta-model shown in Fig 2. This meta-model was obtained from the UML language superstructure specification, version 2.4.1 [2]. By "navigating in the meta-model", we refer to OCL navigation expressions specified in the constraints, in which we move from a given meta-class to another meta-class and/or to meta-attributes in order to analyze the architecture elements corresponding to these meta-level elements.

This meta-model focuses on describing classes, packages, attributes, dependencies and profiles. A package is composed of a number of types (left of Fig. 2). A Class inherits from `PackageableElement` and `NamedElement` meta-classes. This means that classes are able to participate in dependencies. The bottom-right part of the figure shows that classes can declare attributes which are instances of `Property`. The left-most part of the figure illustrates the fact that we can apply a profile to a package, and that a profile is composed of a number of stereotypes.

[2] `http://www.omg.org/spec/UML/2.4.1/`

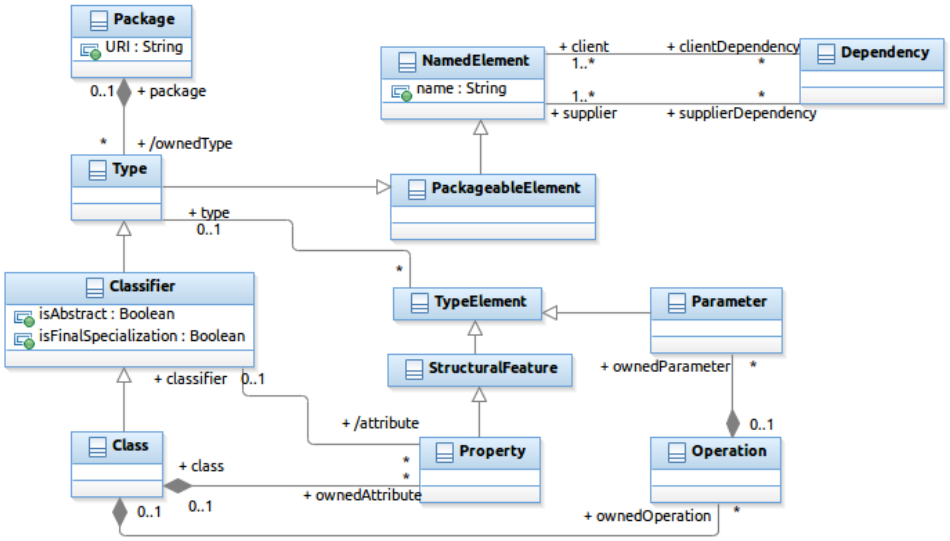

Figure 2: An excerpt of the UML metamodel (Class modeling)

The `MVC` constraint specification is presented in Listing 1. We assume that we have three stereotypes, enabling us to annotate the classes in an application which represent the different roles of the pattern: View, Model and Controller. This constraint states that the classes stereotyped `Model` must not declare dependencies with the classes stereotyped `View` nor `Controller`. 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.

```
1  import uml : 'http://www.eclipse.org/uml2/4.0.0/UML#/'
2  package uml
3  context Package inv:
4  let Model:
5   Set(Class)= self.ownedType->oclAsType(Class)
6    ->select(c:Class|c.getAppliedStereotypes()
7    ->exists(s:Stereotype | s.name='model'))
8  in
9  let View:
10  Set(Class)= self.ownedType->oclAsType(Class)
11   ->select(c:Class|c.getAppliedStereotypes()
12   ->exists(s:Stereotype | s.name='view'))
13 in
14 let Controller:
15  Set(Class)= self.ownedType->oclAsType(Class)
16   ->select(c:Class|c.getAppliedStereotypes()
17   ->exists(s:Stereotype | s.name='controller'))
18 in
```

```
19 -- No dependencies between Model and View or Controller
20 Model->forAll(c:Class|
21   c.supplierDependency.client->forAll(c1:Class|
22     View->excludes(c1) and Controller->excludes(c1)))
23 endpackage
```

Listing 1: MVC constraint in OCL/UML

Line 3 in Listing 1 declares the context of the constraint. The meta-class `Package` is the starting point for all navigations in the rest of the constraint. To refer to the context, we use the keyword `self`. Note that `oclAsType(Class)` operation is used in this constraint to allow navigation between `Type` and `Class` through the specialization indirect relation. Lines 4 to 18 serve to collect together the sets of classes representing the `Model`, the `View` and the `Controller`. For example, we move from the package to look for the types defined in it by `ownedType`. Then, we select only those which have `Model` as an applied stereotype, using the operation `getAppliedStereotypes()`. The remaining of the constraint checks that the classes stereotyped `Model` should not have any dependencies with `View` or `Controller` classes by using `clientDependency.supplier` navigation.

In the following subsections, we explain each step of meta-program generation process illustrated using this example.

3.2. Constraint Refinement

The refinement mechanism is used whenever some abstractions in the UML meta-model do not have a direct equivalence in the JAVA language (like dependencies). There are some navigations in the UML meta-model that do not enable to generate JAVA code. For example, in the previous specification of the `MVC` constraint on the UML meta-model, we have collected all types (classes) which have dependencies with a specific type by using `clientDependency.supplier` (Listing 1, Line 20). This expression has not a direct equivalence in JAVA. As a result, we refine the constraint on the UML meta-model to express the different kinds of dependencies.

Often, a dependency between two classes A and B is translated as: i) the declaration in A of at least one attribute having as a type B, ii) at least one parameter in an operation of A has as type B, or iii) at least one operation of A, has B as a return type or a thrown exception type.

Our constraint is automatically refined to what follows:

```
1 package uml
2 ....
3 -- No dependencies between Model and View or Controller
```

```
4  Model->forAll(c:Class|
5    c.ownedAttribute->forAll(p:Property|
6     View->excludes(p.type) and Controller->excludes(p.type))
7  and
8    c.ownedOperation->forAll(o:Operation|
9     View->excludes(o.type) and Controller->excludes(o.type))
10 and
11   c.ownedOperation->forAll(o:Operation|o.ownedParameter
12      ->forAll(p:Parameter| View->excludes(p.type) and
13            Controller->excludes(p.type)))
14 )
15 ...
16 endpackage
```

Listing 2: MVC refined constraint in OCL/UML

After refinement, our constraint (Listing 2) is composed of three sub-constraints. Each sub-constraint matches one kind of dependency. In Lines 4 to 6, the dependency is primarily verified on all attributes defined in classes. In Lines 8 to 13, the dependency is related to the types of operation parameters and its returned value.

In Listing 2, we have shown that the Model's classes do not have to declare dependencies with the View classes, which makes it a constraint of a static nature, *i.e.* it is checkable on static types. However, according to the existing implementations of the `MVC`, we may find ourselves with a reference to a View object in a Model object at run-time, while statically the classes comply with the constraint. In this case, the "dependency" between the Model and the View can be implemented by the *Observer* pattern: a model object stores a collection of objects listening to changes on the model (the collection can be statically typed by an interface). At runtime, however, this collection will include view objects, whose classes implement the aforementioned interface. In this case, the constraint should take into consideration relations between objects and not only classes. The constraint needs therefore to be further refined by specifying it on the UML meta-model related to instances (Fig. 3) in order to rely on objects rather than classes. This new specification allows to check the values of object slots (slots can be seen as instances of attributes. A class has an attribute and an object has a slot).

In Fig. 3, an instance specification has a `Classifier` which defines it. It includes a number of slots, which have a `StructuralFeature` (e.g. a Property) that declares them. They have values of type `ValueSpecification`. These can be of different types. We are interested in `InstanceValue` (a reference to an instance). This is a "pointer" to an `InstanceSpecification`.

This kind of refinement is applied automatically by our process, whenever

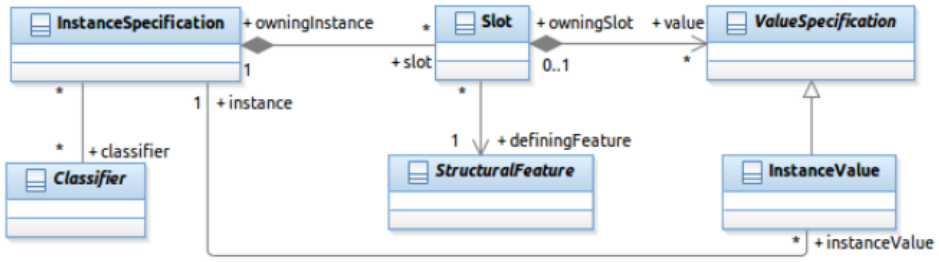

Figure 3: An excerpt of UML metamodel (Instances)

class attributes are introspected by a constraint. The result on the `MVC` constraint is shown in Listing 3.

```
1  context InstanceSpecification inv:
2  let Model:
3   Set(Classifier)= self.classifier
4    ->select(c:Classifier|c.getAppliedStereotypes()
5    ->exists(s:Stereotype | s.name='model'))
6  in
7  let View:
8   Set(Classifier)= self.classifier
9    ->select(c:Classifier|c.getAppliedStereotypes()
10   ->exists(s:Stereotype | s.name='view'))
11 in
12 let Controller:
13  Set(Classifier)= self.classifier
14   ->select(c:Classifier |c.getAppliedStereotypes()
15   ->exists(s:Stereotype | s.name='controller'))
16 in
17 -- No dependencies between Model and View or Controller
18 Model->forAll(c:Classifier |c.instanceSpecification.slot.value.
19  oclAsType(InstanceValue).instance.classifier
20   ->forAll(c:Classifier |View->excludes(c)
21       and Controller->excludes(c))
22 )
23 ...
24 endpackage
```

Listing 3: MVC refined constraint in OCL/UML (Instances)

In Listing 3, `InstanceSpecification` is the constraint context. We therefore assume that the constraint must be verified on all instance specifications making up the application. In the constraint, we navigate to the classifier of the instance specification. We access then to the `Classifier` of the value stored in the slot of the `Model` and we check if `View` and `Controller` are not stereotypes applied on it.

We have implemented the constraint refinement step using an XML mapping between UML meta-model elements. We analyze the AST (Abstract Syntax Tree) generated by a compiler from the text of the constraint. According to the AST node the appropriate refinement is applied. For doing so, we have defined a list of possible refinements. For example, if a node content is "supplierDependency" or "isComposite", the process refines the constraint.

The refinement of a constraint implies a translation of an architecture constraint from a relatively 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, the constraints which are the source and the target of the transformation navigate both in the (UML) meta-model.

3.3. Constraint Transformation

We transform the OCL constraint specified on the UML meta-model into an OCL constraint specified on the JAVA meta-model. We searched in the literature for a JAVA meta-model for our process but unfortunately none of the existing ones satisfied our needs (producing a constraint in an intermediate step towards code generation). We relied on JAVA Reflect library to create a new simplified JAVA meta-model. In fact, we can define our meta-model relying on JAVA specification but we deliberately chose JAVA Reflect because it gives us access to the meta-level of the language which was implemented in the JDK and also because it reflects exactly what we can do in the generated JAVA code. In this meta-model, we limited ourselves to the elements necessary for architecture constraint specification. Fig. 4 depicts the simplified JAVA meta-model that we have defined.

In Fig. 4, classes have fields, methods and constructors. A `Class` belongs to a package. In JAVA, from one package, we cannot know which types are defined there. All these elements can be annotated and have modifiers (except packages), which can have different values listed in the enumeration named `Modifier`. An attribute can have a reference towards another object as its value for a particular object. In constructing this meta-model, we relied on classes defined in the JAVA Reflect API whose methods enable the introspection of JAVA objects. The `get(...)` method of the `Field` meta-class returns the value stored in the field of an object which is passed as an argument. In JAVA there is no equivalent of UML's `Slot` meta-class. Actually, this is a more general problem. It is due to the fact that in JAVA, there is no true coupling between the objects and their meta-objects (instances of

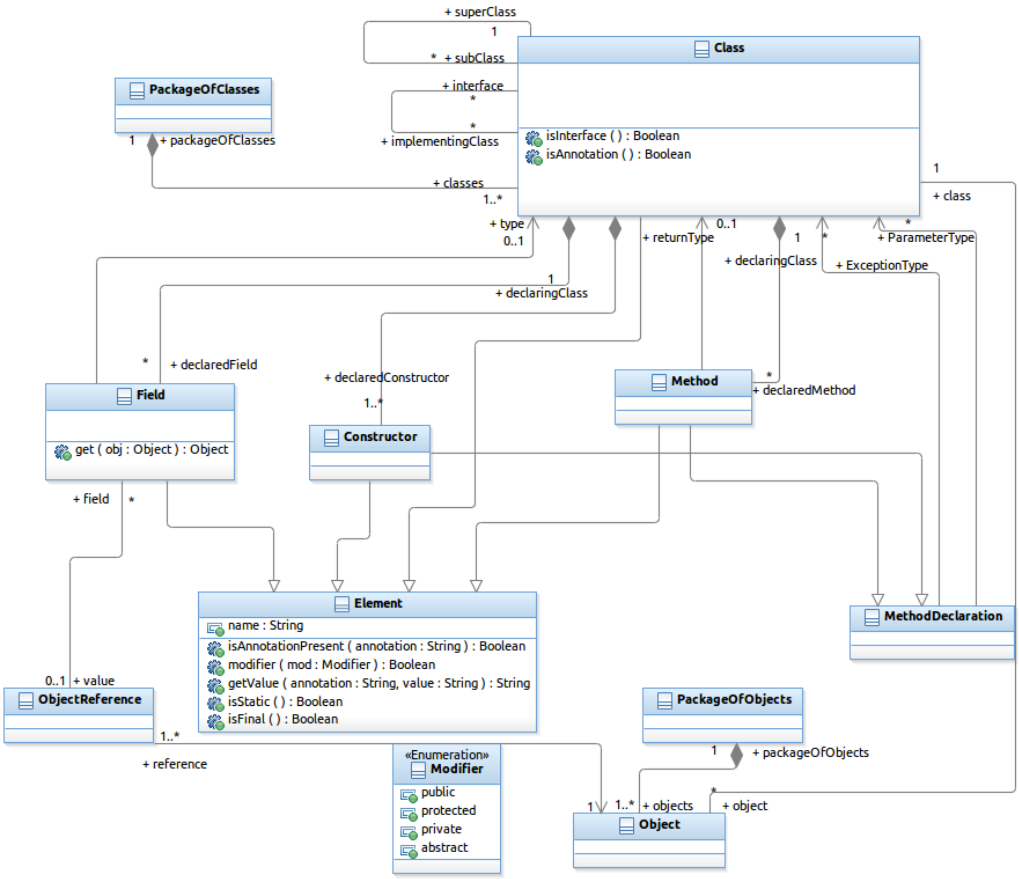

Figure 4: Simplified JAVA meta-model

Class). Once we invoke `getClass()` on an object, we obtain the meta-object, but in this meta-object, there is no reference back to the object (we therefore loose access to the values of its "fields").

Constraint transformation consists in establishing a mapping between UML modeling elements and JAVA programming entities. Mappings are classified in three categories depending on meta-model-level OCL expressions: meta-classes, roles and navigation patterns. Table 1 presents an excerpt of these mappings.

An abstract syntax tree (AST) is generated from the initial constraint (refined OCL constraint, if any). This AST includes the names and the types of the nodes for each expression in the OCL constraint. The process

Table 1: An excerpt of UML-JAVA mappings

		UML	JAVA
Metaclass		**Package**	**ApplicationClasses/ ApplicationObjects**
	Role	ownedAttribute	field
		ownedOperation	method
		superClass	superClass
		getImplementedInterfaces	interface
		ownedType	class / object
		isAbstract	isInterface()
	Nav.	getAppliedStereotypes() ->(s:Stereotype|s.name='N')	isAnnotationPresent('N')
		c.ownedOperation ->(o:Operation|o.name=c.name)	c.declaredConstructor
		visibility=VisiblityKind::public	modifier(Modifier::public)
Metaclass		**Class**	**Class/Object**
Metaclass		**Operation**	**Method**
	Role	type	returnType
		ownedParameter	parameterType
		raisedException	exceptionType
Metaclass		**Property**	**Field**
	Role	type	type
		value	type
		slot	field

automatically parses this AST in depth and according to each matched node, the corresponding part of the constraint is translated into the appropriate part based on the predefined mapping between the two meta-models (Table 1). This translation starts by identifying the navigation patterns, then the roles and finally the meta-classes in the same way as in [13]. After each modification of the AST, a new constraint is generated from it and evaluated with an OCL compiler that validates it on the JAVA meta-model.

In the case of navigation pattern transformation, we need to store some parameters and variables such as the name of the class annotation (Line 9 in Table 1) to put them in the equivalent of this navigation pattern in JAVA. These variables or parameters can be easily obtained from the AST.

We opted for the specification of the mappings in XML, and we have

written an ad-hoc program for implementing the transformation instead of using an existing model transformation language like Kermeta [3] or ATL [14], because we do not consider architecture constraints as models. We might have generated models from constraints and then transform them. But this process is tedious to implement. It requires to transform the text of the constraint to models, then to use a transformation language for transforming these models, and after that to generate again the text of the new constraint from the new model. We decided to follow a simple solution that consists in exploiting simply an OCL compiler.

At the end of this step, two kinds of constraints, which navigate in the JAVA meta-model, are provided: i) a constraint which deals with *classes* and which has as a context `ApplicationsClasses` (for instance, see Listing 4 as the transformation output of Listing 2), and ii) a constraint which deals with *objects* and which has as a context `ApplicationsObjects` (for instance, see Listing 5 as the transformation output of Listing 3).

```
1  package JAVA
2  context ApplicationClasses inv:
3  let Model:
4   Set(Class)= self.classes->oclAsType(Class)
5    ->select(c:Class |c.isAnnotationPresent('model'))
6  in
7  let View:
8   Set(Class)= self.classes->oclAsType(Class)
9   ->select(c:Class |c.isAnnotationPresent('view'))
10 in
11 let Controller:
12  Set(Class)= self.classes->oclAsType(Class)
13  ->select(c:Class |c.isAnnotationPresent('controller'))
14 in
15 -- No dependencies between Model and View or Controller
16 Model->forAll(c:Class|
17 c.declaredField->forAll(f:Field|
18   View->excludes(f.type) and Controller->excludes(f.type))
19 and
20 c.declaredMethod->forAll(m:Method|View->excludes(m.returnType)
21   and Controller->excludes(m.returnType))
22 and
23 c.declaredMethod->forAll(m:Method|m.parameterType
24  ->forAll(p:Class |View->excludes(p) and Controller->excludes(p)))
25 )
26 endpackage
```

Listing 4: MVC constraint in OCL/JAVA

In Listing 4, the context of the constraint is `ApplicationClasses`. It

[3] `http://www.kermeta.org`

is a set of business classes that compose the user application. It excludes classes related to libraries. As indicated in Listing 4, we replace, among others, `ownedAttribute` by `field` and `ownedOperation` by `method`. By parsing the AST, we perform transformations for some complex navigations like `getAppliedStereotypes() ->exists (s:Stereotype | s.name='model')`. This navigation is transformed to `isAnnotationPresent('model')`.

```
1  package JAVA
2  context ApplicationObjects inv:
3  let Model:
4    Set(Object) = self.object
5      ->select(c:Object | c.class.isAnnotationPresent('model'))
6  in
7  let View:
8    Set(Object) = self.object
9      ->select(c:Object | c.class.isAnnotationPresent('view'))
10 in
11 let Controller:
12   Set(Object) = self.object
13     ->select(c:Object | c.class.isAnnotationPresent('controller'))
14 in
15 Model->forAll(o:Object|o.class.field
16    ->forAll(f:Field | Controller->excludes(f.get(o))
17          and View->excludes(f.get(o))))
18 endpackage
```

Listing 5: MVC constraint in OCL/JAVA (Objects)

In Listing 5, the starting point is `ApplicationObjects`. It is a set of objects that compose the user application. The constraint analyzes object relations. To access to a field reference, it uses the `Field`'s `get()` method.

The use of declarative mappings gives us the possibility when the meta-models evolve to modify easily the changed elements. In addition, it allows us to offer a generic method which does not depend on particular meta-models (languages).

After this transformation step, an architecture constraint is ready to be translated into a meta-program.

3.4. Meta-program Generation

The meta-program generation step relies on String Templates [4]. We use String Templates because of their flexibility (easy evolution), simplicity and the existence of a good tool support.

This mechanism is based on the *DepthFirstAdapter* pattern proposed in the DresdenOCL parser used in the implementation. The OCL parser 2.0

[4]String Template : `http://www.stringtemplate.org/`

we have used is from Dresden OCL. The templates have been created using StringTemplate 4.0.8.

Three elements are responsible for generating the JAVA code from the ASTs that are generated from the OCL constraints which navigate in the JAVA meta-model:

- `CodeGenerator`: the role of this element is to traverse in depth the AST and generate code, which depends on the type of the node and its content.
- `Environment`: this element saves all the variables generated and some other information needed during code generation.
- `CodeStacker`: this element manages the code which is generated by the CodeGenerator. The code generated is saved under the StringTemplate format.

The `CodeGenerator` reacts only to the node that it must process. For every type of node (e,g. ArrowrightIteratorPostfixExp, FormalParameter, or DotPropertyCallPostfixExp) we have defined a common default processing. There is a small set of nodes (comparatively with the large set of OCL node types) for which we have defined a different processing. These are the leaves in the AST.

The `CodeGenerator` reads the type of the node from the AST. According to its type (e,g. ArrowrightIteratorPostfixExp), it obtains the template associated to this node. It saves it in a list in the `CodeStacker` and receives its position. Then, it launches the same procedure for its descendant nodes. This procedure is stopped when leaves are found. After the generation of its descendants, it can use every template positioned after the position received above from the `CodeStacker`. The obtained templates are used to fill its own template. In the fulfillment of the template, it uses the introspection methods according to the AST node. After that it removes all the templates that it has used. The `CodeGenerator` has also a map that contains for each used template the associated result. This serves for the complex or the repetitive expressions. When it fills each template, it checks if it has an existing result (a variable) for the template which it uses. If yes, it uses the existing variable, if not, it creates one and uses it. For example, the constraint which contains navigations like the following one: `a.method.returnType`, the `CodeGenerator` creates a variable

named m1 for example which corresponds to the template used for `a.method`. In the code, we have `m1=a.getDeclaredMethod()`. After that, we obtain `m1.getReturnType()`. Every variable created to fill the template must be registered in the `Environment`, as an `InitializedVariable`.

Listing 6 shows an excerpt of the implementation of `CodeGenerator` when it processes the node whose type is `InitializedVariable`.

```
1  public void caseAInitializedVariableCs(AInitializedVariableCs node){
2      STGroup group = new STGroupDir("templates");
3      // Template selection
4      ST st = group.getInstanceOf("InitializedVariable");
5      // Register the chosen Template in Code Stacker
6      int index = codeStacker.put(st);
7      // Descendant step
8      super.caseAInitializedVariableCs(node);
9      // Recovery of Descendant templates
10     ArrayList<ST> generated = codeStacker.getAfter(index);
11     // FormalParameter /* Fulfillment of the template */
12     ST tmp = generated.get(0);
13     st.add("var", tmp.render());
14     // Equals ::> treatment not needed
15     tmp = generated.get(1);
16     // LogicalOclExpression
17     tmp = generated.get(2);
18     st.add("exprs", tmp.render());
19     InitializedVariable var = variableByST.get(tmp);
20     if (var != null) {
21       st.add("value", var.getName()); }/* Fulfillment End */
22     // Suppression of all used templates
23     codeStacker.removeAll(generated);
24   }
```

Listing 6: An example of the generator for the InitializedVariable node type (CodeGenerator.java)

At the end, our process provides two kinds of meta-programs. Each meta-program is a JAVA class that has a public method called `invariant(..)` which returns a Boolean value. The first meta-program is a JAVA class generated from a constraint that has as context "`ApplicationClasses`", such as Listing 4, while the second one is a JAVA class generated from a constraint whose context is "`ApplicationObjects`", such as Listing 5. Listing 7 and Listing 8 present respectively two excerpts of these two meta-programs:

```
1  public class MVCConstraint {
2      //...
3     public boolean invariant(Class<?>[] self) {
4     ArrayList<Class<?>> klass = new ArrayList<Class<?>>();
5      for(Class c : self) {
6        boolean bool = c.isAnnotationPresent(Model.class);
7        if(bool) {
```

```
8           klass.add(c);
9         }
10      }
11      Class<?>[] klass1 = new Class<?>[klass.size()];
12      int selectiterator = 0;
13      for(Class c : klass) {
14        klass1[selectiterator] = c;
15        selectiterator++;
16      }
17      Class[] model = klass1;
18      // same way for controller and view
19
20      boolean bool18 = true;
21      for(Class c : model) {
22        Field[] field = c.getDeclaredFields();
23        for(Field iterator : field) {
24          iterator.setAccessible(true);
25        }
26        boolean bool6 = true;
27        for(Field p : field) {
28          Class<?> klass6 = null;
29    if (p.getGenericType() instanceof ParameterizedType) {
30          klass6 = (Class<?>)((ParameterizedType)(p.getGenericType()))
31              .getActualTypeArguments()[0];
32          }
33          else {
34            klass6 = p.getType();
35          }
36          boolean bool3 = true;
37          for(Class iterator : controller) {
38            if(iterator.equals(klass6)) {
39              bool3 = false;
40            }
41          }
42          //.....
43          boolean bool4 = true;
44          for(Class iterator : view) {
45            if(iterator.equals(klass7)) {
46              bool4 = false;
47            }
48          }
49          boolean bool5 = bool3 && bool4;
50          if(!bool5) {
51            bool6 = false;
52          }
53        }
54        // second constraint
55        Method[] method = c.getDeclaredMethods();
56        boolean bool10 = true;
57        for(Method o : method) {
58          Class<?> klass8 = o.getReturnType();
59          boolean bool7 = true;
60          for(Class iterator : controller) {
61            if(iterator.equals(klass8)) {
62              bool7 = false;
63            }
64          }
```

```
570 65          //....
    66          boolean bool9 = bool7 && bool8;
    67          if(!bool9) {
    68            bool10 = false;
    69          }
575 70        }
    71        boolean bool11 = bool6 && bool10;
    72        //Remaining of the constraint
    73        //....
    74        boolean bool17 = bool11 && bool16;
580 75        if(!bool17) {
    76          bool18 = false;}
    77         }
    78    return bool18;
    79    }
585 80  }
```

Listing 7: An excerpt of the MVC meta-program in JAVA

```
    1  public class MVCConstraintObj {
    2    public boolean invariant(Object[] self)
590 3        throws IllegalArgumentException, IllegalAccessException {
    4       ArrayList<Object> object = new ArrayList<Object>();
    5      for(Object c : self) {
    6        Class<?> klass = c.getClass();
    7          boolean bool = klass.isAnnotationPresent(Model.class);
595 8        if(bool) {
    9          object.add(c);
   10        }
   11      }
   12      Object[] object1 = new Object[object.size()];
600 13     int selectiterator = 0;
   14      for(Object c : object) {
   15        object1[selectiterator] = c;
   16        selectiterator++;
   17      }
605 18        Object[] model = object1;
   19      // for View and Controller
   20
   21      boolean bool7 = true;
   22      for(Object o : model) {
610 23        Class<?> klass3 = o.getClass();
   24        Field[] field = klass3.getDeclaredFields();
   25        for(Field iterator : field) {
   26          iterator.setAccessible(true);
   27        }
615 28        boolean bool6 = true;
   29        for(Field f : field) {
   30        Object obj = f.get(o);
   31          boolean bool3 = true;
   32          for(Object iterator : controller) {
620 33            if(iterator.equals(obj)) {
   34              bool3 = false;
   35            }
   36          }
   37    //....
625 38  }
```

Listing 8: An excerpt of the MVC meta-program in JAVA (Objects)

In Listing 7, Line 3 presents the *invariant* method signature. Lines 4 to 17 present the source code generated to select the classes annotated `Model` (a code generated from the first let expression of Listing 4). Indeed, the generator calls the string template associated to OCL `select` operation. In this template, an array list is created presenting the returned value of this operation. A loop is generated to browse all the classes of the application (it is a parameter of the *invariant* method). It tests for each class if it has an annotation equal to `Model`. The same mechanism is followed to generate the code for obtaining the classes annotated `Controller` and `View`.

The constraint imposes conditions on fields of a class (Lines 17 to 18 in Listing 4). The generator in this case tests if that field has a simple or a generic type in order to get the appropriate type of this field. This is shown in Lines 29 to 34 in the generated meta-program in Listing 7.

From the Line 55, the code generation process generates the source code of the second sub-constraint. This sub-constraint (Lines 20-21 in Listing 4) contains the `forAll` quantifier. So, as we noted above, the process calls the string template associated to this quantifier. This template requires as parameters: i) a collection to iterate, *i.e.* an array is created to collect all the declared methods of the `Model` class, ii) an iterator i,e a loop browses this array, iii) an expression *i.e.* the Java code that corresponds to the OCL expression included in this quantifier: the generator tests if the types of all the method return values are different from the `Controller` ones, and iv) a Boolean variable, *i.e.* it stores the result of the test. The parameter "expression" is the body of the iterator. It uses other filled string templates (the templates which correspond to the descendant nodes of the node that contains this quantifier in the AST) of other quantifiers like `excludes` and OCL variable initialization (see Listing 6). In this case, the generator stores variables and parameters of the first quantifier in the map created by the `CodeStacker` and then puts them in the parameters of the string template associated to the second one when it is called.

Since each sub-constraint is an OCL invariant, for each one, a Boolean variable is initialized to store its result. At the end, all the created Boolean variables are concatenated by the JAVA operator "&&" to give the result of the whole constraint.

The code generation process followed to generate Listing 7 is similar to

the one used for Listing 8. It deals with objects instead of classes. The major difference is related to how to get object slot values. This occurs in Line 30 in Listing 8. This is preceded by several checks to ensure that the object class attribute is not of a primitive type and is accessible (it has a public accessibility). It is the slot value type in question which is checked, to ensure that it does not relate to an annotated Controller class.

Henceforth, architecture constraints are specified in the implementation phase with JAVA language as meta-programs. These meta-programs are executable to check statically and dynamically the initial constraints.

4. Constraint Checking

The goal of this phase is to complete the object-oriented application engineering process by providing a micro process to check architecture constraints on programs. This process exploits the generated meta-programs and Aspect-Oriented Programming (AOP) in order to not be intrusive, since the constraints are specified separately from source code.

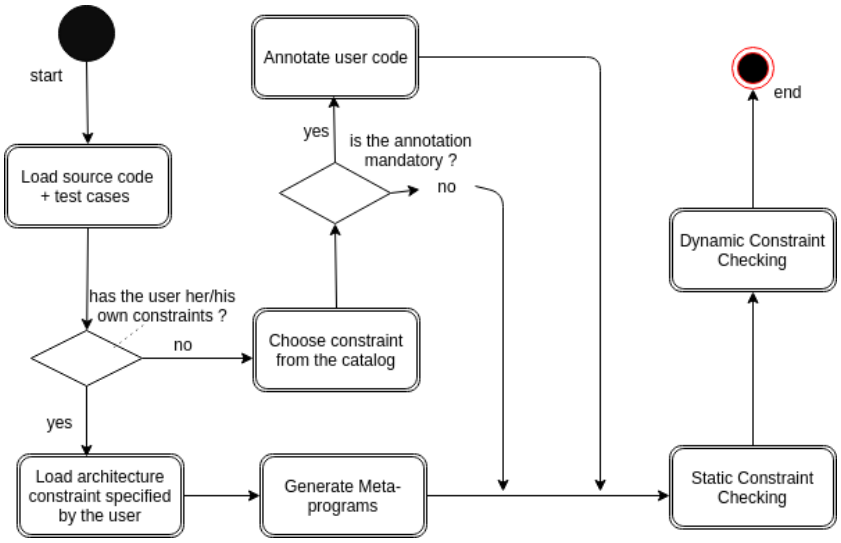

Figure 5: Constraint Checking Description

Fig. 5 presents an activity diagram that explains the micro-process of automatic checking of architecture constraints.

We assume the availability of a catalog composed of a set of architecture constraints written in OCL/UML with their Java meta-programs. First,

the user is asked to load her/his classes accompanied with a set of test cases. Second, she/he is asked to load her/his architecture constraint as an OCL file if this constraint did not belong to the catalog. Meta-programs are generated from the loaded new architecture constraints. If the user chooses a constraint from the catalog, then, the process uses the corresponding pre-generated meta-programs. Sometimes, to be checkable a constraint requires specific annotations in the source code. In this case, the user is asked to indicate the necessary class names in her/his source code, to automatically integrate the annotations and to recompile the code. Then, a static checking is performed using the appropriate meta-programs that analyze the user classes. The next step in our process consists in checking the constraint dynamically using the meta-programs and some pre-defined aspects in our process. Finally, a diagnosis report is provided.

4.1. Static Checking

The static checking consists in invoking the `invariant` method after loading the byte code of user programs in order to collect the classes. It provides a result for each sub-constraint. If at least one sub-constraint is violated, the full constraint is considered not respected.

It should be noted here that unlike approaches for static code analysis, constraint checking and thus execution of the `invariant` method, necessitates loading of the entire application by the class loader, in order to obtain the class objects reifying the different application classes, before passing them as an argument (array) in the invariant method invocation.

Many constraints, beyond their static checking, require to be dynamically checked. This concerns constraints that introspect the application's objects (and not only classes). These constraints need to be valid at the application's runtime.

4.2. Dynamic Checking

In this step of the checking process, the first question being asked is how we can collect all the objects that compose the user's application without modifying the user application by inserting or deleting statements or using an external tool. We found that aspect-oriented programming (AOP) responds to our needs and is an optimal solution to collect the class objects of the user application and then check dynamically the constraints.

AOP provides architectural abstractions and composition mechanisms in order to specify crosscutting concerns into separate functional units, called

aspects. This separation of concerns improves modularity and reusability, and allows having a clean code which is easy to understand.

```
1  aspect Constraint{
2    pointcut ConstraintChecking():
3      call (* *.new(..))
4    //Advice
5    before(): ConstraintChecking(){
6        // execute invariant method
7        ...     }
8  }
```

Listing 9: Example of AspectJ code

One possible language for writing aspect-oriented programs is AspectJ. Listing 9 presents an example of an AspectJ code. A `join point` is a well-defined point in the program flow. In our example, the join point is "When a constructor is called" (Line 3). An `Advice` defines a crosscutting behavior. It is defined in terms of `pointcuts`. The code of a piece of advice runs at every `join point` picked out by its `pointcut` (here a "pointcut" is represented in Lines 2 and 3). Exactly how the code runs depends on the kind of the advice. In Listing 9, we wanted to execute the invariant method before each constructor in the user application is called.

Listing 10 presents an excerpt of the aspect code to collect the class objects making up the user application.

```
1  aspect Constraint{
2    // code
3    pointcut collect():
4      execution(*.new(..));
5    after(): collect() {
6    // collection is the name of the array that
7    // brings together the objects
8    collection.add(thisJoinPoint.getThis()); }
9  }
```

Listing 10: AspectJ code used for collecting objects

When the *new* keyword is interpreted throughout the user source code, an object is created. `thisJoinPoint` (Line 8) is a keyword used to obtain information about the current `join point`. AOP offers a simple and a quick way to collect an object of any object-oriented application with a few number of statements and without requiring any information.

After getting a collection of the objects of an OO application, the `invariant` method of the second kind of the generated meta-programs can be invoked after passing this collection as parameter in order the check the corresponding architecture constraints. We have tried to find the appropriate "pointcuts"

offered by AOP; *i.e.* where in the business source code, the aforementioned `invariant` method should be invoked.

By combining aspects and meta-programs concepts, we profit with the benefits of AOP (simple and dynamic way, easy to extend, maintain and reuse aspects) and we obtain a checking result that details error localization in the source code.

In this process we offer the possibility to check architecture constraints independently from the business source code. In other words, we do not require any analysis of the source code (extracting elements, adding statements, etc). Thus, the developed AspectJ code does not include "pointcuts" that use for example a name of a method, or a field. In this case, because the meta-programs used in this step of the process deal with "objects", we have involved "pointcuts" that manage the places in the source code to identify object states, object relations, object modifications and run-time attribute assignment.

- Object pre-initialization : preinitialization(*.)
- Object initialization : initialization(*.)
- Object creation : execution(*.new(..))
- Object suppression : set(*)
- Constructor call : call(*.new(..))

In some architecture constraint specifications, we need to seek the values of object slots (defined by the attributes declared in the classes of these objects), because we assume that the application has been loaded and launched. Obtaining these object slot values is performed using the reflect API of the programming language. For example, in the `MVC` pattern, we need to check the slot value type in a class which is annotated `Model`, to ensure that it does not relate to a class annotated `Controller`. For doing so, we add the following "pointcut":

- Field set : set(* *)

We can reduce the execution time of the aspect code when we specify exactly for which class of the user application we need to modify the value of the object (last "pointcut") by using the predefined annotations for each

constraint. For example, we defined the following AspectJ code in order to execute the invariant method when a field value in the class annotated `Model` is modified:

```
1 aspect Constraint{
2   pointcut ConstraintChecking():
3     set(* *) && within(@Model *);
4   before(): ConstraintChecking(){
5       verifyConstraint(); }
6 }
```

Listing 11: Constraint checking when model attribute assignment

With this optimization, we produce for each constraint a specific aspect code in order to dynamically verify the source code which is supposed to respect this constraint. If the user has her/his own architecture constraints, a meta-program will be generated and a generic aspect program will be used to automatically reach the checking goal. At the end of our process, a diagnostic report of the checking is provided. Our approach reveals where the constraint is violated.

We take an example of an architecture constraint that formalizes the composition relation at run-time between two classes *ComplexShape* and *Shape*. An example of the checking of this constraint is shown in Fig. 6.

```
déc. 08, 2016 11:17:37 AM Composition.AspectVerification result
INFOS:  preinitialization(Composition.Rectangle()) Call from Composition.Rectangle line 11 to Composition.Rectangle.<init>
Constraint is currently not respected
déc. 08, 2016 11:17:37 AM Composition.AspectVerification result
INFOS:  preinitialization(Composition.Line(int, int, int, int)) Call from Composition.Line line 19 to Composition.Line.<init>
Constraint is currently not respected
déc. 08, 2016 11:17:37 AM Composition.AspectVerification result
INFOS:  execution(Composition.Line(int, int, int, int)) Call from Composition.Line line 19 to Composition.Line.<init>
Constraint is currently respected
déc. 08, 2016 11:17:38 AM Composition.AspectVerification result
INFOS:  preinitialization(Composition.Line(int, int, int, int)) Call from Composition.Line line 19 to Composition.Line.<init>
Constraint is currently respected
déc. 08, 2016 11:17:38 AM Composition.AspectVerification result
INFOS:  preinitialization(Composition.ComplexShape()) Call from Composition.ComplexShape line 11 to Composition.ComplexShape.<init>
Constraint is currently respected
déc. 08, 2016 11:17:38 AM Composition.AspectVerification result
INFOS:  execution(Composition.ComplexShape()) Call from Composition.ComplexShape line 17 to Composition.ComplexShape.<init>
Constraint is currently not respected
déc. 08, 2016 11:17:38 AM Composition.AspectVerification result
INFOS:  preinitialization(Composition.Line(int, int, int, int)) Call from Composition.Line line 19 to Composition.Line.<init>
Constraint is currently not respected
```

Figure 6: An example of dynamic checking output

Fig. 6 shows in each `pointcut` (for instance, Lines 2 and 4), the temporary output of the dynamic checking of the constraint. At first, the constraint is not respected for two times. This is explained by the fact that the annotated objects are not definitively created and the constraint requires that they should not be null and they should realize the composition relation. Then, the process continues to execute. At this level, the objects are created and they

are not null. But we remark the presence of an error when *ComplexShape()* method invocation is done because it makes a null reference of the *Shape* class object. The user is asked to examine her/his source code starting from the `pointcut` corresponding to the first error output.

It is true that the dynamic checking is a crucial step in the process. However, it uses one instance of the user application (one execution scenario). We can obtain different results for other scenarios. In our solution, we apply the checking process on the set of user test cases and we provide for each test case the corresponding report. Our checking process makes alerts on constraint violation by printing log messages about anomalies. It does not abort the execution of the application, but it gives the user all the checking results. She/He can read the log messages and then change her/his code to respect the constraints.

5. Experimental Evaluation

This section reports on some experiments we have conducted to evaluate our entire constraint specification and checking process.

5.1. Research Questions

Our experiments have been conducted in order to answer the following research questions:

- **RQ1:** Does the process allow to generate valid and efficient meta-programs?

  Our automated process generates JAVA meta-programs allowing the checking of architecture constraints. The aim of this research question is to measure: i) the validity of the meta-programs on several object-oriented case examples. These examples have been developed by students. Patterns have been instantiated in these examples and in variants of them (other case examples), in which these patterns have been voluntarily "broken"; ii) the performance of our approach by measuring the time required for generating and executing the meta-programs.

- **RQ2:** Does the constraint checking process provide precise results in JAVA real projects?

  The aim of this research question is to show that the process of constraint checking provides results that are conform to the modifications

made in large-sized JAVA projects (with several patterns in the same project)

5.2. Experiment for RQ1

5.2.1. Data Collection

We invited 6 students to manually accomplish the steps of meta-program generation process. These students know JAVA, its reflection API, and have followed an OCL lecture. We split this group of developers into two groups.

We asked the first one to identify textual constraints of some design patterns and then to formalize their structural conditions with OCL. We chose architecture patterns as data, because they are widely used as a means to characterize an architecture, and are considered as a suitable way to document a part of design decisions. The students have chosen the most popular design patterns which concern only the structural aspect of the architecture. For the second group, we asked them to write a set of OCL constraints and their corresponding JAVA programs using JAVA reflect.

We have collected some descriptive measures (time and size) during the textual identification of the constraints, their formalization with OCL, their transformation in OCL/JAVA, the code generation and the execution of the generated meta-programs. We compared the time spent in each step made manually and automatically. Finally, we have obtained 12 design patterns characterized by their architectural constraints.

Each pattern is represented by its architecture constraint. Each constraint is usually composed of a set of sub-constraints. Each sub-constraint is a formalization of a structural condition that the class diagram of an application in which the pattern is instantiated should respect. The same group of developers have prepared for each pattern (included in our experiment) a toy class model and its corresponding JAVA application. Moreover, they have prepared for each pattern, a set of models each of which invalidates a sub-constraint in the constraint of the pattern.

We take for example a design pattern P characterized by its architecture constraint C. This constraint is composed of two sub-constraints C1 and C2. The developers prepared 4 models. The first one complies with P, the second complies with C1 but not with C2. The third one complies with C2 but not with C1, and the fourth model do not comply neither with C1 nor with C2. Besides, the developers implemented for each model a simple JAVA application. 4 JAVA applications were developed, each of which is the implementation of one of the models previously mentioned.

The experiment data is available online here: `https://seafile.lirmm.fr/f/5221db540b6348d9b9be/`.

We have used as tools in this evaluation, *Eclipse Luna* and the plugin *OCLinEcore* to check the OCL architecture constraints on the pre-defined models.

All the measures taken during this experimentation are presented in Tables 2 and 3.

Table 2: Size of constraints and their meta-programs

Design Pattern	OCL Constraint #tokens	Meta-program LOC	
		Manual	Automatic
Adapter	348	160	381
Bridge	248	142	234
ChainOfResponsibility	225	155	250
Composite	306	120	306
Decorator 1	387	100	500
Decorator 2	387	100	500
Facade	186	135	292
Factory-method	234	167	210
Mediator	190	120	150
MVC	189	40	100
Observer	579	120	300
Proxy	238	117	200

Table 3: Time spent on each step of the process (in seconds)

Design Pattern	Spec UML	UML–JAVA		to Meta-program	
		Manual	Automatic	Manual	Automatic
Adapter	16500	480	0.05	5400	6.21
Bridge	15240	480	0.05	3300	4.38
ChainOfResponsibility	6000	600	0.22	600	4.41
Composite	14100	720	0.11	9000	4.67
Decorator 1	10320	540	0.09	4380	5.91
Decorator 2	2400	120	0.01	4440	3.57
Facade	10620	420	0.18	6060	4.25
Factory-method	7440	540	0.18	4920	4.71
Mediator	10740	660	0.14	4260	3.70
MVC	5400	300	0.19	3900	4.98
Observer	8502	840	0.36	6180	6.88
Proxy	5820	720	0.27	4920	4.24

In Table 2, the first column presents the name of each architecture pattern. The second column shows the size (in terms of number of tokens in

the AST) of the architecture constraints that formalize the pattern. We have chosen constraints with different sizes, ranging from 186 tokens for the smallest to 579 for the largest one. The Third column presents the size (in terms of number of lines of code) of the manually written and the automatically generated meta-programs. As we can observe, the automatically generated meta-programs are larger than the manually created one. Indeed, as mentioned previously the automatically generated code is not optimal in terms of complexity. It was built incrementally without any optimization.

In Table 3, the OCL specification time includes the identification time of the constraints. The developers have manually performed all the steps of our process (constraint specification, constraint transformation and meta-program generation). This work followed a precise order starting with the `Adapter` pattern and finishing with the `Proxy` pattern. We have chosen a precise order for all the developers to examine the correlation, if exists, between the size of the constraint, the time spent in process steps and the acquisition degree of the OCL language.

We observe that in some cases, there is no correlation between the size of the constraint and the time spent specifying it. For instance, the `Observer` pattern is larger than the `Composite` pattern but it took less time for its specification. Indeed, the developers have naturally acquired experience when specifying each time a new constraint. The first constraint took more time to be specified than the others. The average time decreases when specifying more constraints despite of their size variance. Constraints were specified with OCL which is a language easy to learn and to use (as empirically demonstrated in [15]). The students need only to know for each constraint the appropriate navigation in the UML meta-model and used frequently the same "patterns" of OCL expressions.

We can see in Tables 2 and 3 two variants of the `Decorator` pattern. The constraints of these variants have the same size in terms of number of tokens. They share some sub-constraints. This decreases the time spent specifying the second variant.

It takes for a developer an average of 1.47 hours without considering the time of constraint identification, to manually develop a JAVA source code that allows to check an architecture constraint. It is true that this time may be decreased even more when the developer manually develop meta-programs. But it is still significantly higher than the time spent by the automatic generation process.

5.2.2. Protocol and Results:

The protocol followed in this experimentation consists in, on the one hand, checking the OCL constraints on the corresponding pre-defined models. On the other hand, checking the meta-programs generated from these constraints on the different implementations of the aforementioned models.

If we take the same example (introduced in the previous subsection) we get the results presented in Table 4, where:

C: Architecture Constraint	**Mc**: Meta-program generated from C
Mi: Model number i	**Ii**: Implementation of Mi
C/Mi: Result of checking C on Mi	**Mc/Ii**: Checking Mc on Ii
False: Constraint is violated	**True**: Constraint is respected

Table 4: Expected results

C/M1 ->True	——————->	Mc/I1 ->True
C/M2 ->False	——————->	Mc/I2 ->False
C/M3 ->False	——————->	Mc/I3 ->False
C/M4 ->False	——————->	Mc/I4 ->False

The checking of the constraint on the first model must return "True" and on the other models it must return "False". Besides, we must also obtain the same results when checking the meta-program, generated from this constraint, on the implementations of these models. Following this first protocol, we test if each generated meta-program corresponds to the initial constraint or not. These constraints and their meta-programs are checked on several variants of models and programs to avoid any error during the generation process.

We have 12 architecture patterns. We have created more than 250 test cases to check the OCL constraints on models and on source code. All the expected results are obtained for the constraints which require only the static checking. For the other constraints, we were not able to dynamically check the OCL constraints on the static (class-based) models. We have then considered a second evaluation. It consists in applying our constraint checking process (Section 4) using, in one hand, manually written meta-programs and in the other hand the automatically generated ones in the same pattern instance implementation variants (Ii).

Considering the checking results, we noticed that the tool successfully generated valid meta-programs. The two meta-programs notify the same errors (error and code localization) in the source code.

Our approach worked well with the experiment data created by the developers. For improving the process validation, we have evaluated our generated meta-programs and our constraint checking technique on real JAVA projects in which many design patterns are instantiated. This evaluation is presented in Subsection 5.3.

Performance:. Our analysis was about performance. We measured the time our technique required to implement and execute the manually written and the automatically generated meta-programs of our patterns. The results are summarized in Tables 3 and 5 (Table 5 is an extension of Table 3). As expected, the time is proportional to the size of the constraint formalizing the pattern (see Table 2). We must mention that execution time was absolutely within our expectations. The interesting part is when we compare the values in the two columns of Table 5. We can notice that the execution time of the generated source code is lightly higher than the execution time of the manually written one, in all cases. This is explained by the fact that the generated code has a greater complexity than the manually written one. The average overhead of the generated code is +16% (in milliseconds). But this is negligible and does not affect much a process of architecture verification. Indeed, for the moment, the software systems that we target in our work are not real-time ones and this performance overhead does not affect them too much.

Table 5: Execution time of meta-programs (in seconds)

Design Pattern	Execution Time	
	Manual	Automatic
Adapter	0.65	0.86
Bridge	0.54	0.66
ChainOfResponsability	0.57	0. 97
Composite	0.30	0.57
Decorator 1	0.52	0.66
Decorator 2	0.54	0.69
Facade	0.77	0.93
Factory-method	0.69	0.87
Mediator	0.68	0.87
MVC	1.00	1.23
Observer	1.40	1.73
Proxy	1.12	1.69

If we consider the time needed to execute the entire JAVA application, an example of an application reaction time increases to 2,43s instead of 1,99s.

This difference is negligible even if we consider the dynamic aspect of the checking. After considering all the JAVA applications used in our evaluation, we can say that we are quite confident that the overhead of constraint checking at execution time is marginal.

5.3. Experiments of RQ2

We would like here to evaluate our approach on large-sized JAVA projects. The source code of these projects should contain at least two different design pattern instances.

5.3.1. Data Collection

We have conducted our checking process on several JAVA projects: Applied JAVA Patterns (AJP) [16], the Eclipse Pattern Box (EPB) [17], Findbugs [18] and MapperXML [19]. Based on their documentation, we have identified the design patterns that are instantiated in the projects.

5.3.2. Protocol

As a first experimentation, we have applied our checking method on all the source codes (the results are shown in Table 6). Meta-programs were generated form the architecture constraints which formalize the patterns instantiated in each source code. Aspect codes are prepared to check these architecture constraints using their generated meta-programs. The aspect codes include all the pointcuts defined in Section 4.

In the second experimentation, we have invited 3 other persons (1 Phd and 2 Master students) who have enough experience with design patterns. We asked them to introduce some modifications on the source codes of these applications. They have written scripts that describe each pattern architecture as textual items. A master student who was not involved in the last task try to modify the sources by altering at least one item in the script. Therefore, the patterns' source code became not conform to their architecture model, and their constraints became violated. These modifications are made by using the reflective API of Java. The students use the reflective methods especially those responsible for modifying the behavior of the objects, like `invoke()` and `newInstance()`. Then, we have reapplied our process on the altered sources to see whether the patterns are respected or not. Finally, we analyzed the output of the checking to verify its correctness compared to the modifications.

5.3.3. Results and Discussion

To present our results, we use the following notations:

- $\surd$+ : the pattern is well implemented and the result is "pattern respected"
- $\surd$- : the pattern is well implemented but the result is "pattern not respected"
- x+ : the pattern is not well implemented and the result is "pattern not respected"
- x- : the pattern is not well implemented and the result is "pattern respected"

Table 6: Checking Results (before altering the sources)

Patterns	**AJP**	**EPB**	**Findbugs**	**MapperXML**
Abstract-factory	$\surd$+	$\surd$+		
Factory-method	$\surd$+	$\surd$+		$\surd$+
Adapter	$\surd$-	$\surd$-		
Proxy	$\surd$+		$\surd$+	
Bridge	$\surd$+			
Composite	$\surd$-	$\surd$+		
Decorator	$\surd$-			
MVC				$\surd$+
Facade			$\surd$-	

$\surd$+ shows the cases where our method succeeded. Most of the design patterns are correctly verified. We found 10 from 15 pattern occurrences that are well verified, with a success rate of 66.66% (in the first experimentation).

Our tool does not detect the conformance of the sources to the `Adapter` and `Decorator` pattern architectures. Indeed, the `Adapter` and `Decorator` intercept method invocations between the caller and the delegation class. However this relation is neither well defined, nor definable [8]. This point may influence the specification of the constraint and then produces an error in our process validation.

The `Decorator` pattern implementation in the AJP project is not well verified by our tool. This is explained by the fact that there are many variants of this pattern and unfortunately the one implemented in the AJP source is not taken into consideration in our data collection.

The `Composite` pattern is usually tightly related to other patterns. This relation can affect some lightweight modifications on its implementation to be composed with other patterns to answer the user needs. These modifications probably concern composite object states.

In Findbugs project, in the `Facade` pattern occurrence, our tool correctly pointed out that class edu.umd.cs.findbugs.Lookup directly uses class edu.umd.cs.findbugs.ba.XClass without accessing it through the Facade which is called edu.umd.cs.findbugs.ba.Hierarchy as documented in the Findbugs API. With this relation, the `Facade` implementation actually does not strictly satisfy the requirements for the `Facade` design pattern. If the strict interpretation of the `Facade` pattern is to be used, then the fact pointed by the tool is a design flaw.

In the second experimentation, the source code of all the projects is already altered. There were no x- found in the results, shown in this table. All checking results produce "pattern not respected". But, among the x+ are consequences of the first experimentation. Indeed, the $\surd$- showed in Table 6 is automatically changed to x+, considering the errors produced during the first experimentation.

It is true that the `Abstract-factory` pattern implementation in EPB undergoes some modifications and the experimentation result produces "pattern is not well implemented", but the experimentation output and the modifications made are not suitable. The obtained output displays "pattern is currently not respected" throughout the execution but in some of the cases we found that the pattern is respected.

Concerning the `Proxy` pattern, the modifications made in its implementation in AJP source code are performed using reflective methods that affect the pattern architecture. Our constraint and its generated meta-program does not take into consideration this way of modification.

5.4. Threats to validity

We discuss two kinds of threats: to the internal validity and to the external one.

5.4.1. Internal validity

In our experimentation, we have used architecture patterns that are specified from several sources. The constraints that formalize our patterns are specified by participants who have a short experience with OCL language. Besides, in our selected architecture patterns, we can find variants for a

given pattern like the `Decorator`. This increases reuse of the decomposed constraints as well as their relevance. But, we have mitigated this threat by choosing patterns of different sizes and by involving different persons in their specification and transformation.

Besides, we have developed for the evaluation code in AspectJ language that contains all the possible pointcuts to check the constraint. The execution of some large and complex applications under our constraint checking process may produce errors like endless loop especially with generic AspectJ code (without optimization). In this case, we decompose the AspectJ code in several codes and we repeat the execution many times.

5.4.2. External validity

The architecture patterns used in our experimentation have been collected from the literature. We can obviously think that the proposed process works only for this kind of object-oriented architecture patterns or that only constraints written in OCL can be evaluated as input and Java as output. The inputs of the process can be any kind of predicates analyzing architecture descriptions (a parser must exist for their specification language). Besides, the produced output can be a source code written in any language that provides a reflective API. Any kind of architecture constraints can be considered, including component based design patterns or SOA patterns. Constraint transformation step is applicable with any meta-models because it uses external mappings in XML and the AST as output of the parser. The code generation step takes into consideration each node of the AST and uses the corresponding String Template. The Templates can be written in any language that provides a reflective API. The reflective methods provided by this language are mandatory during the use of String Templates.

We have involved many groups of students in the different experiments. In Experiment 1 (Section 5.2), they have manually performed the identification of architecture constraints and prepared examples of models to statically check the OCL constraints. These students may have designed models that are nearly conform to the constraints. To mitigate this threat, we have proposed the second evaluation (Section 5.3) that consists in checking the constraints on external projects. The students have also manually developed the meta-programs. To reduce the implementation error, each student implemented the 12 meta-programs (we have 12 architecture constraints) and the most appropriate meta-program for each pattern was selected.

Two other groups have participated in Experiment 2. A group has ana-

lyzed the Java projects to extract the design pattern architectures and the second has developed scripts allowing the modification of the pattern implementation which has been done by the third group. All the experiments were conducted by involving these groups under the same conditions. Each work is done by one group and validated by the two others in order to reduce the error rate.

6. Related Works

In this section we present works related to i) languages and tools for the specification of architecture constraints, ii) techniques for predicate/constraint transformations, iii) techniques/tools for code generation from OCL, and iv) architecture constraint checking approaches.

A state of the art on languages used for the specification of architecture constraints at design and implementation stages is presented in [3]. Some languages are considered as notations in existing ADLs, like Armani for Acme, FScript for Fractal ADL or REAL for AADL. Others are embedded notations with a logic programming style, like Alloy, LogEn or Spine, or notations with an object-oriented programming (OOP) style or DSLs for OOP languages, like CDL or SCL. There exist in practice some static code quality analysis tools like Sonar, Lattix and Architexa that authorize the specification of architecture constraints. These languages and tools, cited above, do not enable transformation or code generation of specifications in OCL or any other predicate language.

Hassam *et al.* [20] use a model transformation method to transform OCL constraints during UML model refactoring. The others use a mapping table, created under the UML model transformation for transforming OCL constraints of the initial model into OCL constraints of the target one. Their solution of constraint transformation cannot be used straightforwardly because it needs some knowledge about model transformation languages and tools. In our work, constraint transformation is performed in a simple and ad-hoc way without using additional modeling and transformation languages.

Works in [21, 22] focus on UML/OCL transformation into CSP (Constraint Satisfaction Problem). The authors in [21] proposed an approach for instantiating models from meta-models taking into account OCL constraints. Based on CSP, they defined some formal rules to transform models and constraints associated to them. These approaches are similar to our transformation process since the transformed/handled artifacts are the same

(OCL specifications and meta-models). They use the same OCL compiler as us (DresdenOCL [23]) to analyze constraints. In contrast to CSP, our process does not require an external tool for the interpretation of constraints. Besides, in their approaches, everything should be transformed into a CSP to be solved (the constraints + the models/meta-models) while in our approach, we transform only constraints

All these works considered only functional and not architectural constraints. They allow constraint checking only on design phase and they do not provide a way to generate code from them to be specified at the implementation phase. However, in [24] the authors propose to transform constraints into HOL representations before generating Test Data. Thus, the OCL constraint undergoes major modifications. Some features of OCL will be evaporated as confirmed in [25]. Our approach transforms the architecture constraints from OCL/UML to OCL/JAVA before generating code. This transformation considers only the change of the meta-model. The constraints are still written in OCL. In addition, in our case, architecture constraints are specified after transformation in the meta-model of the programming language used later for implementation. This has the benefit that architecture constraints can be documented in a language that all designers and developers understand.

Eclipse OCL [5] and DresdenOCL [23] which provide OCL constraint translation to JAVA, transform constraints which are functional and not architectural. The generated code by Dresden OCL is difficult to understand. In fact, 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.

In [26],the authors integrate constraints translated on JML assertions at compilation time. Jass [27] integrate constraints translated into JAVA comments through source code instrumentation. These works generate skeletons of code in user source code and then use external tools to validate the constraints. In our work, our constraints need to be verified at run-time because they impose conditions on object dependencies which can be obtained only at that level. Our approach allows to generate source code and check the

[5] `http://www.eclipse.org/modeling/mdt/?project=ocl`

constraints without altering the user source code and it uses a standard mechanism, the introspection mechanism offered by the language used for programming the source code.

In [28], the authors translate functional constraints into AspectJ specifications which are checked the at runtime. One disadvantage of this approach is the strong coupling of the aspects to the base code. Pointcut definitions specify the interception points for constraint validation. The definitions are exactly specified with method signatures, class of field names. Any change in the underlying base code undergoes modifications to these definitions. However, in our work, with architecture constraints, this strong coupling between the aspects and the source code does not exist since we need only a collection of the classes or of the objects of the user application which is obtained by using a code separately developed from the source code. Sometimes, this code requires only the class annotations.

Despite the existence of several approaches, as noted above, to check design constraints on code, the gap between the state of the art and the state of the practice has become apparent. In deed, these approaches generally require learning a language different from the programming language to specify design constraints. This increases the learning curve and put at risk the adoption of these approaches. We believe that an approach that allows specification of design constraints in the same language as that of the software can increase the adoption of conformance checking by both designers and programmers.

In this context, an approach that admits this affirmation is presented in [29]. The authors of this approach generate design rules into design tests which are specified in the programming language (Java). These design tests allow to automatically check the conformance of the design rules in the implementation. These design tests are written as JUnit tests. Two frameworks are implemented : a code structure analysis API and a testing framework. The first one is responsible for analyzing the source code and for specifying the design rules through methods offered by this API. The second framework provides assertion routines and an automatic way to execute the generated tests. It is true that the authors in this work rely on the utilization of the language used for implementation to specify design constraints but these constraints are manually specified and the authors use an external framework for the checking instead of using an API provided by the programming language (like Java.reflect) and an extended language (like AspectJ). In this way, the whole process will be more adopted by the designers and the Java

programmers, knowing that in our approach, the developers do not develop entirely AspectJ code and at the most case, they use pre-defined aspects. Besides, the authors in this work, noticed that they do not consider dynamic constraints.

A work presented in [30] presents a comparative study between code analysis tools about their capabilities for architectural conformance checking. The authors proved that: i) The tools the authors investigated do not allow dependency constraint conformance checking at run-time, especially on object-oriented source code and ii) Most of the tools do not succeed to localize where the constraint violation takes place in the source code. In this case, the user should manually inspect the source code in order to determine error location. In contrast, our developed tool allows the static and the dynamic checking of architecture constraints. It is capable to check automatically and dynamically constraints that formalize objects dependencies in object-oriented application. Besides, our checking process, basing on the AOP technique, notifies the user with the traces of constraint violation with a log result.

The authors in [31] introduced a new form of architecture model called Design Rule Space. This model represent the software architecture as an ensemble (a DRSpace) of design rules (e,g, dependency, inheritance, aggregation) and independent modules. This work identifies structural and evolutionary problems between these modules by clustering the code source and visualizes them is structure matrices. The algorithm of clustering provides a hierarchy of the files which are embedded in these modules. The authors, by applying Baldwin and Clark's design rule theory features [32], identify the error prone DRSpace which lead error files. This work considers that software architectures are as multi-layered modules. These later may or may not be equivalent to the abstractions used to express the system's architecture. In deed, this new proposed representation can restrict custom architecture entities representation and thus their architecture constraint specifications. In our work, the source code is considered as one layer and with the introspection mechanism provided by the programming language, we can examine all problems inside this layer.

7. Conclusion

Architecture constraints bring a valuable help for preserving architecture styles, patterns or general design principles in a given application after hav-

ing evolved its architecture model [6]. These architecture constraints are checked at design time. They need also to be checked if the architecture evolves in the implementation artifacts or at runtime. For that purpose, we proposed an automatic process which allows to check these constraints in that development stage and at run-time. This verification process uses the meta-programs that are generated from these constraints and uses the reflection mechanism provided by the programming language.

The proposed automatic process is composed of two phases. The first phase consists in transforming architecture constraints into meta-programs, towards the implementation phase. The second phase is to check these constraints statically by loading the application and executing the appropriate meta-programs, and then to check them dynamically (if necessary) using aspect-oriented programming.

Expressing architecture constraints with the same language as the one used in the implementation phase provides an executable documentation. With this documentation, architecture constraints are more likely to keep in synchronization with the actual implementation. We believe this is especially useful in development teams in which developers change often and can easily miss or misunderstand the previously made design decisions. In our implementation (JAVA code generation), our approach uses Java.reflect API and an other programming language (AspectJ) which are likely known by JAVA developers. The standard introspection mechanism is enough to make this kind of architecture constraints executable at the implementation phase. Besides, for checking them, we require only the user annotated source code. This later is not altered during the process. The checking is fully automatic, seamless for users, flexible and provides a diagnostic result that identifies where the constraints in the code are violated.

One of the limitations of our approach is the fact that it does not cover all OCL language. Some operations, like *OCLIsNew*, *OclAny*, *OclVoid* and *OclInvalid*, are not considered. But these are mainly used in OCL post conditions and not in OCL invariants adopted by our approach. Besides, our tool is flexible, in order to integrate new OCL expressions. We just need to write specific string templates and to implement a method that initializes them.

After working in checking dynamically architecture constraints on object oriented programs and component-based and service-oriented applications [33], we plan as future work to generalize the proposed approach, by specifying architecture constraints in a paradigm-independent way: using

predicates on graphs and operations on them and then making automatic transformations towards a particular object-oriented, component-based or service-oriented programming language.

References

[1] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, R. Little, Documenting software architectures: views and beyond, Pearson Education, 2002.

[2] D. Falessi, G. Cantone, R. Kazman, P. Kruchten, Decision-making techniques for software architecture design: A comparative survey, ACM Computing Surveys (CSUR) 43 (4) (2011) 33.

[3] C. Tibermacine, Software Architecture 2, John Wiley and Sons, New York, USA, 2014, Ch. Architecture Constraints.

[4] A. H. Eden, R. Kazman, Architecture, design, implementation, in: proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, 2003, pp. 149–159.

[5] B. Meyer, Touch of Class, Springer, 2013.

[6] C. Tibermacine, R. Fleurquin, S. Sadou, 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), Springer LNCS, Vasteras, Sweden, 2006, pp. 294–309.

[7] U. Zdun, P. Avgeriou, A catalog of architectural primitives for modeling architectural patterns, Information and Software Technology 50 (9) (2008) 1003–1034.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.

[9] T. Erl, SOA design patterns, Pearson Education, 2008.

[10] R. Filman, T. Elrad, S. Clarke, M. Akşit, Aspect-oriented software development, Addison-Wesley Professional, 2004.

[11] M. Petre, Uml in practice, in: Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), IEEE Press, 2013, pp. 722–731.

[12] J. Peters, J. M. E. van der Werf, A genetic approach to architectural pattern discovery, in: Proccedings of the 10th European Conference on Software Architecture Workshops, ACM, 2016, p. 17.

[13] C. Tibermacine, R. Fleurquin, S. Sadou, Simplifying transformations of architectural constraints, in: Proceedings of the ACM Symposium on Applied Computing (SAC'06), Track on Model Transformation, ACM Press, Dijon, France, 2006, pp. 1270–1244.

[14] F. Jouault, I. Kurtev, Transforming models with atl, in: Satellite Events at the MoDELS 2005 Conference, Springer, 2006, pp. 128–138.

[15] L. C. Briand, Y. Labiche, M. Di Penta, H. D. Yan-Bondoc, An experimental investigation of formality in uml-based development, IEEE Transactions on Software Engineering 31 (2005) 833–849.

[16] S. Stelting, O. Maassen, Applied Java Patterns, Prentice Hall Professional, 2002.

[17] D. Ehms, Patternbox eclipse tool, Retrieved November 14 (2000) 2006.

[18] Findbugs, `http://findbugs.sourceforge.net/`.

[19] MapperXML, `http://essere.disco.unimib.it/svn/DPB/MapperXML%20v1.9.7/`.

[20] K. Hassam, S. Sadou, R. Fleurquin, et al., Adapting ocl constraints after a refactoring of their model using an mde process, in: BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010), 2010, pp. 16–27.

[21] A. Ferdjoukh, A.-E. Baert, A. Chateau, R. Coletta, C. Nebut, A csp approach for metamodel instantiation, in: ICTAI 2013, IEEE International Conference on Tools with Artificial Intelligence, 2013, pp. 1044,1051.

[22] J. Cabot, R. Clarisó, D. Riera, On the verification of uml/ocl class diagrams using constraint programming, Journal of Systems and Software 93 (2014) 1–23.

[23] B. Demuth, The dresden ocl toolkit and its role in information systems development, in: Proc. of the 13th International Conference on Information Systems Development (ISD2004), 2004.

[24] A. D. Brucker, M. P. Krieger, D. Longuet, B. Wolff, A specification-based test case generation method for uml/ocl, in: Models in Software Engineering, Springer, 2011, pp. 334–348.

[25] S. Ali, M. Z. Iqbal, A. Arcuri, L. C. Briand, Generating test data from ocl constraints with search techniques, IEEE Transactions on Software Engineering 39 (10) (2013) 1376–1402.

[26] A. Hamie, Pattern-based mapping of ocl specifications to jml contracts, in: Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on, IEEE, 2014, pp. 193–200.

[27] D. Bartetzko, C. Fischer, M. Möller, H. Wehrheim, Jassjava with assertions, Electronic Notes in Theoretical Computer Science 55 (2) (2001) 103–117.

[28] Y. Cheon, C. Avila, Automating java program testing using ocl and aspectj, in: Information Technology: New Generations (ITNG), 2010 Seventh International Conference on, IEEE, 2010, pp. 1020–1025.

[29] J. Brunet, D. Guerrero, J. Figueiredo, Design tests: An approach to programmatically check your code against design rules, in: Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on, IEEE, 2009, pp. 255–258.

[30] J. Van Eyck, N. Boucké, A. Helleboogh, T. Holvoet, Using code analysis tools for architectural conformance checking, in: Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge, ACM, 2011, pp. 53–54.

[31] L. Xiao, Y. Cai, R. Kazman, Design rule spaces: A new form of architecture insight, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 967–977.

[32] C. Y. Baldwin, K. B. Clark, Design rules: The power of modularity, Vol. 1, MIT press, 2000.

[33] S. Kallel, B. Tramoni, C. Tibermacine, C. Dony, A. Hadj Kacem, Generating reusable, searchable and executable "architecture constraints as services", Journal of Systems and Software 127 (2017) 91–108.