Automatic Translation of Architecture Constraint Specifications into Components

Sahar Kallel[1,2](✉), Bastien Tramoni[1], Chouki Tibermacine[1](✉), Christophe Dony[1], and Ahmed Hadj Kacem[2]

[1] Lirmm, Montpellier University, Montpellier, France
{sahar.kallel,bastien.tramoni,chouki.tibermacine,dony}@lirmm.fr
[2] ReDCAD, Sfax University, Sfax, Tunisie
sahar.kallel@redcad.org, ahmed.hadjkacem@fsegs.rnu.tn

Abstract. Architecture constraints are specifications defined by developers at design-time and checked on design artifacts (architecture descriptions, like UML models). They enable to check, after an evolution, whether an architecture description still conforms to the conditions imposed by an architecture pattern, style or any design principle. One possible language for specifying such constraints is the OMG's OCL. Most of these architecture constraints are formalized as "gross" specifications, without any structure or parameterization possibilities. This causes difficulties in their reuse. We propose in this work a process for translating architecture constraints into a special kind of components called constraint-components. This makes these specifications reusable (easily put and checked out in/from repositories), parametrizable (generic and applicable in different contexts) and composable with others. We implemented this process by considering the translation of OCL constraints into constraint-components described with an ADL called CLACS.

Keywords: Architecture constraint · UML metamodel · CLACS · Constraint-component · Reusability · OCL definition · Automatic translation

1 Introduction: Context and Problem Statement

Architecture constraints are specifications of invariants that are checked by analyzing architecture descriptions. This kind of constraints should not be confused with functional constraints, which are checked by analyzing the state of the running components constituting the architecture. For example, if we consider a UML model (an architecture description) containing a class `Employee` (a component in that architecture) which has an integer attribute `age`, a functional constraint presenting an invariant in this class could impose that the values of this attribute (slot of an object) must be included in the interval [16-70] for all instances of this class. This kind of constraints is inherently dynamic. They can be checked only at runtime.

© Springer International Publishing Switzerland 2015
D. Weyns et al. (Eds.): ECSA 2015, LNCS 9278, pp. 322–338, 2015.
DOI: 10.1007/978-3-319-23727-5_27

On the other side, architecture constraints are specifications where architecture descriptions, and not component states, are analyzed [28]. They define invariants imposed by the choice of a particular design principle, architectural style or pattern, like the layered architecture style [25], where "components in non-adjacent layers must not be directly connected together". This is an example of an architecture constraint. OCL(Object Constraint Language) [16] is an OMG standard which specify two types of constraints : functional (constraints navigate in UML models) and architectural (constraints navigate in MOF metamodels).

Functional constraints are used in Design by Contract for ensuring the definition of accurate and checkable interfaces for software components [21]. 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 the quality [29].

Many architecture constraints have been formalized for the existing architecture patterns proposed in the literature and practice of software engineering [3,14,33]. But unfortunately, most of them are "gross" textual specifications. They do not offer any structure. Therefore, it is difficult to reuse them in other/different contexts. This is the reason why we propose in this paper a process to transform them into more structured assets in order to facilitate their reuse. In addition, our experience with architecture constraint specification leads us to say that most of the time, architecture constraints are composed of many "independent" parts that are assembled together via logical operators. Some of these parts are shared between several architecture constraints and have their own semantics. The idea of this paper is to propose a way to build OCL basic constraints as entities embedded in a special kind of software components, that can be reused, assembled, composed into higher-level ones and customized using standard component-based techniques.

In this paper, we propose to translate automatically architecture constraints specified in design stage into "*constraint-components*". We propose a two-step process which takes as input a gross OCL architecture constraint specification expressed in the UML metamodel, and which provides as output constraints-components expressed with CLACS Architecture Description Language. We propose to generate architecture constraints as "constraint-components" [31] so that we can put them on "shelves" and thereafter make them reusable, customizable and composable with others to produce more complex constraints.

The remaining of this paper is organized as follows. In the following section, we give an illustrative example of the input and the output of the proposed process. These will serve as running examples throughout the paper. In Section 3, we describe in detail the steps of our process. In Section 4, we expose an evaluation of the approach. Before concluding and presenting the future work, we discuss the related work in Section 5.

2 Illustrative Example

To better understand the context of this work, we introduce an example of an architecture constraint (Listing 1.1) enabling the checking of the topological

conditions imposed by the "Service Bus Architecture Pattern" [5]. This pattern introduces three kinds of components: the customers, the producers and the bus. The bus is defined as an adapter that establishes the communication between customers and producers as they may have mismatching interfaces. The architecture constraint which specifies the conditions imposed by this pattern is expressed in OCL using the UML metamodel [28] in the following listing.

```
1  context Component inv :
2  let bus:Component
3  = self.realization.realizingClassifier
4  ->select(c:Classifier | c.oclIsKindOf(Component)
5   and c.oclAsType(Component).name='esbImpl')
6  customers : Set(Component)
7  = self.realization.realizingClassifier
8  ->select(c:Classifier | c.oclIsKindOf(Component)
9   and (c.oclAsType(Component).name='cust1'
10    or c.oclAsType(Component).name='cust2'
11    or c.oclAsType(Component).name='cust3'))
12 producers : Set(Component)
13 = self.realization.realizingClassifier
14 ->select(c:Classifier | c.oclIsKindOf(Component)
15  and (c.oclAsType(Component).name= 'prod1'
16 or c.oclAsType(Component).name='prod2'
17 or c.oclAsType(Component).name='prod3'))
18 in
19 -- The bus should have at least one input port
20 -- and one output port
21 bus.ownedPort->exists(p1,p2:Port|
22    p1.provided->notEmpty() and p2.required->notEmpty())
23 and
24 --Customers should have output ports only
25 customers->forAll(c:Component|
26   c.ownedPort->forAll(required->notEmpty()
27     and provided->isEmpty()))
28 and
29 --Customers should be connected to the bus only
30 customers->forAll(com:Component|
31   com.port->forAll(p:Port|p.end->notEmpty()
32     implies
33       self.ownedConnector ->exists(con:Connector |
34          bus.ownedPort->exists( pb:Port|
35            con.end.role->includes(pb)) and
36              con.end->includes(p.end))))
37 and
38 --Producers should have input ports only
39 producers->forAll(c:Component|
40   c.ownedPort->forAll(provided->notEmpty()
41     and required->isEmpty()))
42 and
43 --Producers should be connected to the bus only
44 producers->forAll(com:Component|
45   com.port->forAll(p:Port|p.end->notEmpty()
46     implies
47       self.ownedConnector->exists(con:Connector|
48         bus.ownedPort->exists( pb:Port|
49           con.end.role->includes(pb)) and
50             con.end->includes(p.end))))
```

Listing 1.1. Bus architecture pattern constraint in OCL/UML

When applying our proposed approach, we change the format of the constraint (Listing 1.1) from a textual "gross" specification into an architecture description made of "constraint-components" and "query-components". These components are described with an ADL named CLACS [31] (pronounced Klax). By "gross" specification, we mean a specification that does not offer enough structure, reusability and parameterization.

In the literature, there are many languages enabling the specification of architecture constraints (see [28] for a survey). Each one has its advantages and its particular application context. However, CLACS is the only language that provides a component model for software architecture constraint specification. The architecture constraints modeled with this language are constraint-components in which the checked invariants are still specified using OCL. But these OCL constraints navigate in CLACS metamodel and not in the UML's one. The choice of UML is simply motivated by the fact that it is an industrial standard[1], and that OCL is its original constraint language. We can consider here a repository of architecture constraints that can be fed by the software architecture community, by using these general modeling languages, which are UML and OCL. The result of our translation process is shown in Fig. 1. We notice the presence of two kinds of component descriptors (query and constraint). Query-components embed OCL `definition` constraints that return a value whose type is different from Boolean and constraint-components embed OCL `definition` constraints that return only Boolean values. Indeed, our architecture constraint specification will be decomposed in a set of OCL `definition` constraints and these constraints will be embedded in these two kinds of components to reuse them.

There are three let expressions in the architecture constraint (Listing 1.1). Each one (Lines 2–5, 6–11, 12–17) is supposed to be defined basically in a separate query-component descriptor. But let expressions 2 and 3 are similar according to a similarity measure which is defined in the following section. That is why they are represented by only one query-component (`ParticipantsIdentification`).

There are five constraint-components on the right of the figure. These components represent the OCL `definitions` that are extracted from our initial constraint and then parametrized. These `definitions` are called throughout the constraint and they will potentially serve other constraints.

There are in total five sub-constraints in the architecture constraint (Listing 1.1). Each one (Lines 21–22, 25–27, 30–36, 39–41 and 44–50) is supposed to be defined basically in a separate component descriptor. But in this example, sub-constraints 2 and 4 can be grouped in the same component descriptor (`PortConstraint`) because they check similar "aspects". They check if all the components in a given set of instances (`customers` in the first sub-constraint and `producers` in the second) have specific kinds of ports (input or output).

[1] Even if a recent empirical study [23] found out that UML is not fully (but selectively) used by developers in industry, and that it is used informally, there is a general agreement that UML is the *de facto* standard modeling language known by a large number of developers.

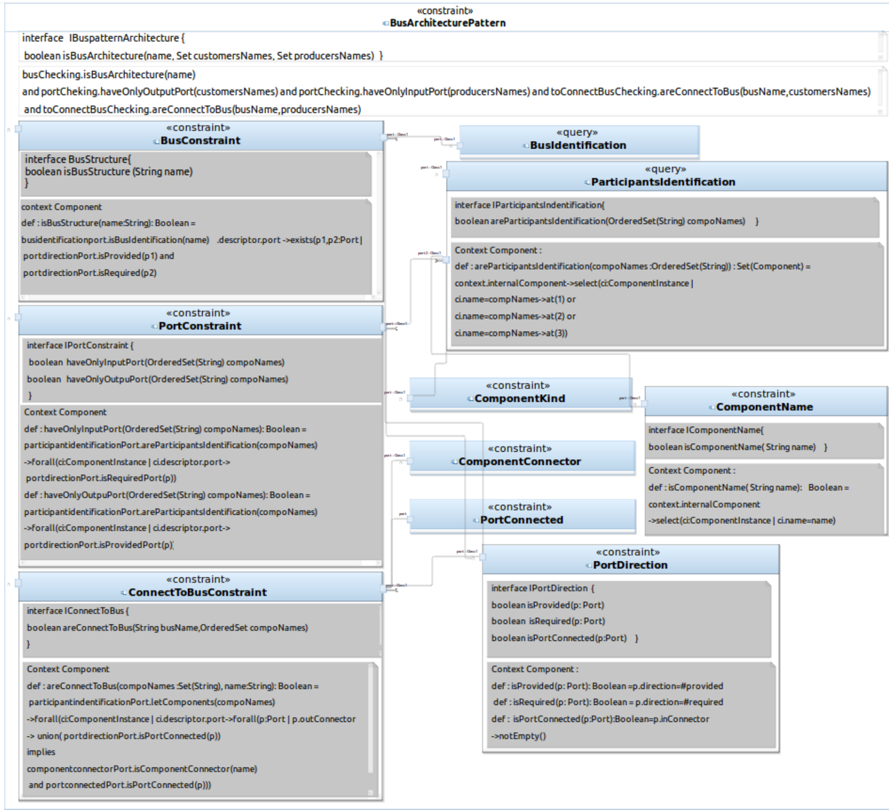

Fig. 1. Sample of approach results

PortConstraint descriptor provides two operations which enable the checking of these two sub-constraints. On the other side, sub-constraints 3 and 5 check exactly the same invariant (in contrast to sub-constraints 2 and 4), except that they apply on different sets of components (customers for sub-constraint 3 and producers for sub-constraint 5). Thus, there is a single component descriptor (ConnectToBusConstraint) which is generated for these two sub-constraints. This constraint-component provides a single operation which is parameterized with the set of components on which the constraint should be checked.

We can see (on the top of the figure) the constraint checked by the composite, in which there are five operation invocations to the three internal components (on the left of the figure). These internal components (that constitute our initial constraint) call the operations that are declared in the others components using the name of the provided port. These later descriptors will be registered in a repository and will be potentially useful for other constraints. In other words, for each new "gross" constraint specification to decompose, we will measure the similarity between the OCL definitions extracted from it i.e. after applying

the decomposition and the paramterization (see Section 3, subsection 3.1), and the registered OCL `definitions` embedded in the components, According to the similarity result, we can reuse an existing OCL `definition` constraint and also modify it, if necessary.

For this example, we will obtain, in addition to the descriptor of the "main" component (`BusPatternArchitecture`), three constraint-component descriptors (instead of five) corresponding to our initial constraint. These three components are connected to the two query-components (`BusIdentification` and `ParticipantsIdentification`) and five other constraint-components. These query-components provide queries that are shared between the constraint- components.

Through this "*componentization*", CLACS constraint-component and query-component descriptors can be reusable (instantiated many times in different contexts), composable (instances of them can be connected together or connected within a composite component to build complex constraint-components) and parameterizable (to check that `customers` or `producers` are connected only to the `bus`, we can pass the right arguments to the operation of `ConnectToBusConstraint` descriptor).

In the following section, we describe in detail the steps of the constraint translation process illustrated with examples.

3 Transformation of Constraints into Components

Our process is composed of two main steps. The first one consists in extracting sub-constraints from the constraint. These sub-constraints will be specified as parametrized OCL `definitions`. The second step consists in embedding these generated OCL `definitions` into components in order to make them reusable. We will detail these two steps in the following subsections.

Note that OCL constraints are predicates in the first order logic. They have a simple and intuitive concrete syntax. Even if the transformations presented in this paper apply on OCL, the proposed work can be generalized to any equivalent predicate logic language. This is not demonstrated experimentally in our work, but as the reader can notice, the syntactic tokens handled in our transformations are general to predicate logic.

3.1 Constraint Refactoring

We propose first to extract sub-constraints as OCL `definitions` and then we identify parameters for them and we will obtain at the end an invariant which uses these definitions. These `definitions` are parametrizable and will be registered in a repository to be used by other constraints. To obtain this new form of our invariant, we propose a multi-step transformation micro-process. All steps use as input the abstract syntax tree of the initial constraint.

Variable Declaration Extraction. Sometimes a sub-expression is used several times in an OCL constraint. The operator `let` allows to report and set the value (i.e initialize) a variable that can be used in the expression which follows the `inv`. `def` is a type of constraints which is used to declare and define the values of attributes or returned values of operations. The first step in our approach is to extract the let expressions from our textual constraint specification and define them as constraints stereotyped with `def`. These OCL `definition` constraints must return a value whose type is different from Boolean. At the same time, we modify our textual constraint i.e, the constraint undergoes changes and call these generated OCL `definitions` in their appropriate places. At this level, our initial constraint will be as follows:

```
1  context Component
2  --let expressions extraction
3  def: letBus(): Component = self.realization.
4  realizingClassifier->select(c : Classifier | c.oclIsKindOf(
       Component)
5   and c.oclAsType(Component).name = 'esbImpl')
6  def: letCustomers(): Set(Component) = self.realization.
7  realizingClassifier->select(c : Classifier | c.oclIsKindOf(
       Component)
8  and (c.oclAsType(Component).name = 'cust1' or
9   c.oclAsType(Component).name = 'cust2' or
10  c.oclAsType(Component).name = 'cust3'))
11 def: letProducers(): ...
12 inv:
13 letBus().ownedPort ->includes(p1, p2 : Port | p1.provided
14     ->notEmpty() and p2.required ->notEmpty())
15 and
16 letCustomers()->forAll(c:Component|c.ownedPort
17 ->forAll(required->notEmpty()and provided->isEmpty()))
18  and ...
19  and ...
20  and ...
```

Listing 1.2. Constraint after extracting let expressions

Constraint Decomposition. Second, we decompose automatically the obtained constraint into a set of sub-constraints. This decomposition is primary based on logical operators used at the top level (Lines 15, 18, 19 and 20 in Listing 1.2). Operands of these operators are considered here as sub-constraints. This set of sub-constraints is refined recursively into a tree of sub-constraints if these sub-constraints can be decomposed again. The stopping condition of the recursion is that no logic operator is found in the sub-constraint. All these sub-constraints will be represented as OCL `definition` constraints. The refactoring of the constraint (i.e modification of the constraint invariant) is performed every time we generate a new `definition`. At this level we obtain a bag of OCL `definition` constraints that return a Boolean value. Listing 1.3 represents an excerpt of our constraint during the decomposition stage.

```
1  context Component
2  def: def1(c:Classifier): Boolean = c.oclIsKindOf(Component)
3  def: def2(c:Classifier): Boolean = c.oclAsType(Component).name
4  = 'esbImpl'
5  def: letBus(): Component = self.realization.realizingClassifier
6  ->select(c : Classifier | def1(c) and def2(c))
7  def: def3(c : Classifier): Boolean = c.oclIsKindOf(Component)
8  def: def4(c : Classifier): Boolean = c.oclAsType(Component).name
9   = 'cust1' or c.oclAsType(Component).name = 'cust2' or
10  c.oclAsType(Component).name = 'cust3'
11 def: letCustomers(): Set(Component) = self.realization.
12 realizingClassifier->select(c : Classifier | def3(c) and def4(c))
13 ...
14 def: part1(): Boolean = letBus().ownedPort
15 ->includes(p1, p2 : Port | def7(p1) and def8(p2))
16 def: part2(): ...
17 def: def11(p: Port): Boolean = p.end->notEmpty()
18 def: def12(p: Port): Boolean = self.ownedConnector
19 ->exists(con : Connector | letBus().ownedPort ->exists(pb:Port|
20 con.end.role ->includes(pb)) and con.end ->includes(p.end))
21 def: part3(): Boolean = letCustomers()
22 ->forAll(com : Component | com.port
23     ->forAll(p : Port |def11(p) implies def12(p)))
24 def: part4(): ...
25 def: part5(): ...
26 inv:
27 part1() and part2() and part3() and part4() and part5()
```

Listing 1.3. Bus architecture pattern Constraint during the decomposition stage

In Listing 1.3, the constraint is composed of five “main” OCL sub-constraints (part1(), part2(), part3(), part4() and part5()). These sub-constraints can be decomposed again into other sub-constraints due to the recursive process[2]. For instance `def4()` contains the operator `or`, so it will be decomposed again. All these sub-constraints are defined as OCL `definitions` (`def:`) presented before the `inv:`. We can observe that there are some OCL `definitions` that have parameters. The reason to make some parameters at this stage (the decomposition) is to have the possibility to define all the generated OCL `definitions` with the same context as that of the constraint (Line 1).

Redundancy Removal. After the constraints decomposition, we obtain a bag of OCL definitions. In this step, we remove all redundant `definitions` and then we update the constraint. For instance, in Listing 1.3 `def1()` and `def3()` are syntactically identical. Now we have a set of OCL `definition` constraints that constitute our textual constraint.

Constraint Parameterization. When creating the signature of the operation that wraps a constraint, we add a parameter in this signature everywhere we find a literal value of a given data type. The type of these parameters is obtained from the abstract syntax tree of the constraint. For instance `def2()` in Listing 1.3 will be defined as follows:

[2] In Listing 1.3, the decomposition is stopped in part4().

```
1 context Component
2 def: def2(c:Classifer, name:String): Boolean = c.oclAsType(Component
      ).name = name
```

Listing 1.4. OCL definition constraint paramatrizable

In this stage, we need to measure the similarity between the OCL `definitions`. This measure allows to optimize our process, i,e. remove some redundant OCL `definitions` (obtained in the parametrization stage). For example `def4()` in Listing 1.3 will be defined at this stage as follows:

```
1 context Component
2 def: def17(c:Classifier, name1:String): Boolean =
3 c.oclAsType(Component).name = name1
4 def: def18(c:Classifier, name2:String): Boolean =
5 c.oclAsType(Component).name = name2
6 def: def19(c:Classifier, name3:String): Boolean =
7 c.oclAsType(Component).name = name3
8 def: def4(c:Classifier, name1:String,name2:String,name3:String):
9 Boolean = def17(c,name1)and def18(c,name2) and def19(c,name3).
```

Listing 1.5. Example of parametrization

We remark that `def2()` (see Listing 1.4), `def17()`, `def18()` and `def19()` are similar. They are different only by the name of the parameter (the same type of the parameter). Then, we remove `def17()`, `def18()` and `def19()` and replace them by `def2()` presented in Listing 1.4. We also optimize the `def4()` `definition` which will take as parameter `c:Classifier` and `consumersNames: Set(String)`. This is performed when comparing the OCL expressions before the "=" (`c.name`) in each literal value. This comparison is done using the AST of the OCL constraint. Concerning how we measured the similarity between OCL `definitions`, we implemented an automated process by analyzing the abstract syntax trees of `definitions` body. Each pair of trees is compared. These should share a common root and a minimal sub-tree (obtained in a breadth-first traversal). This ensures, to some extent, that constraints define predicates on the same kind of architectural elements, which are obtained through navigations in the OCL definition (reflected by these sub-trees). For the remaining sub-tree, an edit distance [27] is measured between each pair of sub-trees. If this measure is less than a threshold[3], we consider that the two `definitions` are similar.

At the end of this step, our invariant is completely decomposed in OCL `definition` constraints. These constraints will be registered in a repository in order to reuse them to create others constraint specifications.

[3] The value of this threshold will be fixed empirically.

3.2 Constraint Transformation into CLACS Components

In this section, we describe the transformation of OCL `definitions` generated in the first step into CLACS components. A CLACS component is an instance of a component descriptor (like an object is an instance of a class). A component has a name, a description and a kind (business or constraint). It declares ports, which are characterized by a direction (required or provided) and a visibility (internal or external). Each port has an interface which specifies a set of operation signatures. Ports are linked via connectors. A connector receives operation invocations through its source port and transmits them through its target port. For generating CLACS components, we proposed a multi-step transformation micro-process:

Operation Grouping. Each CLACS query-component descriptor will embed an OCL `definition` which returns a value whose type is different from Boolean and each CLACS constraint-component will embed an OCL `definition` which returns only boolean values. From the other side, among the generated OCL `definitions`, each one that corresponds to a `let` in the constraint (Subsection 3.1, like `letConsumers()`) will be embedded in a query-component descriptor and each one among the others will be embedded in a constraint-component. In this case, we can obtain a large number of components. Therefore, we propose to put together OCL `definitions` that check similar "aspects" in the same component descriptors. By checking similar aspects, we mean checking the connection, testing the kind, or some other property of a given architectural element (a port or a connector for example). For that we use the same technique of similarity measurement described before (Subsection 3.1, step *Constraints Parametrization*). For example, the OCL `definitions part2()` and `part4()` check the same aspect which is the kind of an architectural element (a `Port`). The two trees of these two sub-constraints have a common root which is a component and a common sub-tree generated from the expression `.ownedPort->includes(p1,p2:Port| )`. For the remaining sub-trees generated from the remaining expressions of the two sub-constraints, we can observe that there is a similarity between them (only two edit operations (node substitutions): `required` and `provided` tokens are inverted). So these are grouped as two operations in the same component descriptor.

Metamodel Migration. In this step, we transform constraint navigations written in OCL/UML into OCL/CLACS. This is performed using a simple set of declarative mappings that we have specified between the two metamodels (UML and CLACS). These have been defined using the same template as in [30]. For reasons of space limitation, we do not show these mappings. But note that, the `self` keyword [4] is replaced by `context`, which is resolved to an implicit required port connected to a meta-descriptor of the business component on which the constraint is checked. This connection resolution is made (lazily) when the checking is launched.

[4] *self* is located in the initial constraint written in UML metamodel.

CLACS Architecture Description Generation. Starting from the tree obtained in the first step, a component-based architecture description in CLACS is generated. This architecture description contains all the necessary constraint-components and query-components (instances) connected together. These components embed the refactored [5] architecture constraints that navigate henceforth in CLACS metamodel. These generated components will be instantiated and then connected to the business components in order to be verified.

4 Process Evaluation

We collected 25 architecture constraints that characterize patterns which concern only structural allure of the architecture. In order to measure the reusability obtained in the result of our transformation process, we choose the metric proposed by Gaffney and Durek in [13]. It allows to calculate the proportion and the number of the reuse constraints. This metric is defined as follows:

$$C = \left(b + \left(\frac{E}{n}\right) - 1\right) R + 1$$

where:

- C: is the cost of software development (specification of an architecture constraint)
- b: is the cost of integrating the reused elements into the new artifact (integration of constraint-components in a composite)
- E: is the cost of developing a reusable element (a constraint-component)
- n: is the number of uses of reused elements
- R: is the proportion of reused elements

C is an important indicator of the effectiveness of the reuse obtained in the final result of our transformation process. If there is no reuse at all, C is equal to 1. The more effective the reuse is, the less C is. b and E relate to the estimated cost of incorporating and developing, respectively, the reused elements. b is supposed to be greater than 0 because it always takes effort to reuse an element. E is supposed to be greater than 1 because the creation of a reusable element requires an extra effort. E is the sum of the costs of developing a new element (without reuse support) and reusing elements. For our experiment, R represents the proportion of the patterns (constraint's) structure which is reused to construct other patterns (constraints). R is the number of the reused constraints divided by the total number of constraints in the same pattern.

Fig 2 shows the values of R for all patterns. As we can observe, the R value is in the range 20-100. We can also observe that there are 13 (out of 25) patterns having 100 % of their structure reused elsewhere. This reinforces our idea to transform architectural constraints into a reusable structure.

[5] A constraint is refactored when the different steps described above have been applied on it.

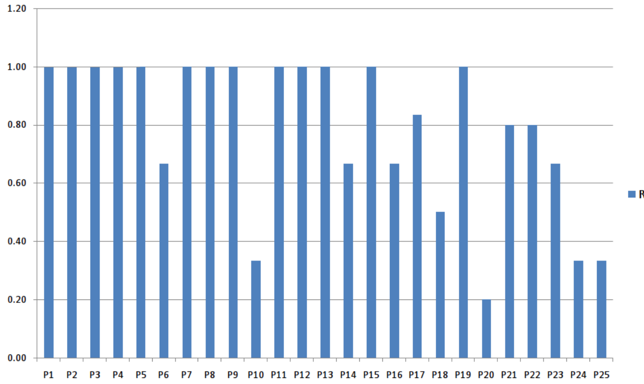

Fig. 2. R values for all patterns

Another value that we have measured is n, which represents how many times a structure is reused in the whole set of evaluated constraints. Fig. 3 depicts the frequency of reusable constraints in each pattern. This demonstrates the potential to promote the reusability of pattern structure in the construction of a pattern library. We can see in Fig. 3 that the pattern P8 is composed of constraint-components that are reused 55 times by other patterns. We have six patterns that have a reusable structure called more than 50 times.

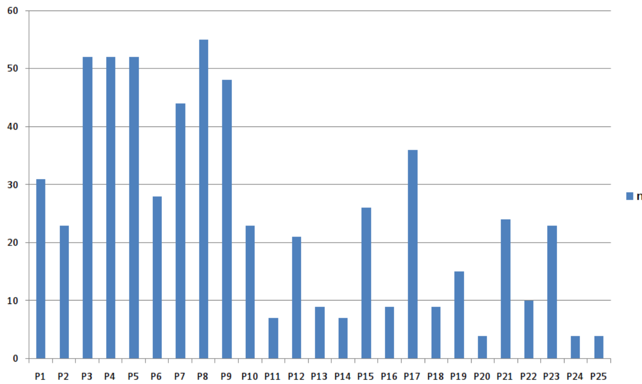

Fig. 3. n values for all patterns

b and E are difficult to measure because of various reasons as explained in [12]. We take the b and E values estimated by [10] since our evaluation falls into the polylithic category[6]. Thus, b and E are equal to 0.15 and 1.2 respectively in our experiment.

[6] This category concerns structures that can be divided into individual parts and each of them can be independently manipulated.

Fig. 4 shows the cost of constructing the 25 patterns. C is in the range of 18 to 89. As we can observe, all of the patterns have a cost less than 1 which means that the obtained reuse really has an effect in reducing pattern construction cost.

5 Related Works

Works related to our approach can be classified in different categories: i) languages and tools for the specification of architecture constraints, ii) techniques for predicate/constraint transformations, iii) techniques for OCL constraints refactoring and iv) methods for constraint reuse.

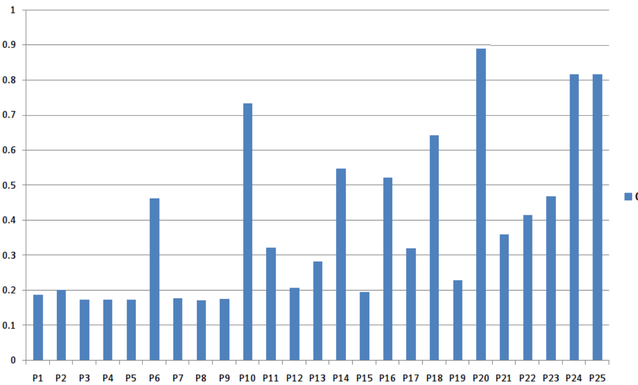

Fig. 4. C values for all patterns

A state of the art on languages used for the specification of architecture constraints at design and at implementation stages is given in [28]. These languages vary from embedded notations in existing ADLs, like Armani [22] for Acme [15], to notations with a logic programming style, like LogEn [9] or Spine [2], or notations with (or for) object-oriented programming style, like CDL [19]or SCL [18]. In practice there are several tools for static code quality analysis that enable the specification of architecture constraints, like Sonar [26], Lattix [20], among others. All these languages and tools do not provide any way for transforming or generating code starting from specifications in OCL or any other predicate language. In addition, they provide either no or a limited parameterization and reusability of architecture constraints.
Hassam *et al.* [17] proposed a method for transforming OCL constraints during UML model refactoring, using model transformations. Their approach uses first an annotation method for marking the initial UML model in order to obtain an annotated target model. Then, a mapping table is created from these two annotations in order to use it 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 an ad-hoc way without using additional modeling and transformation languages. In [11], the authors propose an approach for generating (instantiate) models from metamodels taking into account OCL constraints. Their approach is based on CSP (Constraint Satisfaction Problem). They defined some formal rules to transform models and constraints associated to them. Cabot *et al.* [4] worked also on UML/OCL transformation into CSP in order to check quality properties of models. These approaches are similar to our transformation process since the transformed/handled artifacts are the same (OCL specifications and metamodels). They use the same OCL compiler as us (DresdenOCL [8]) to analyze constraints. In contrast to CSP, this does not require an external tool for the interpretation of constraints. In addition, in our approach, we transform only constraints. In the other approaches, everything should be transformed into a CSP to be solved (the constraints + the models/metamodels). Moreover, Bajwa and Lee presented in [1] a two-step process for transforming SBVR rules (Semantics of Business Vocabulary and Business Rules) into OCL constraints. The first step consists in realizing a mapping between SBVR rules elements and UML model elements. This step ensures that the OCL constraint that will be generated is semantically checkable in a UMl model. The second step consists in transforming an OCL model instance from SBVR model instance using a mapping between the two metamodels (OCL and SBVR). This paper uses model transformations techniques. Their process is troublesome when the constraints have a gross specification (very large models). The generated constraints are complex, not reusable and parametrizable.

OCL refactoring consists in simplifying the constraints and making them more expressive. In [7], Correa *et al.* have as goal to improve the readability and the comprehensibility of the constraint. Therefore, they prepared a catalog of smells. They proposed refactorings for removing a given smell in the constraint. It is true that this refactoring allows a greater comprehensibility of the constraints (validation in the paper) but these do not consider reuse. Besides, the authors consider in their approach only the functional constraints and not architectural ones. In [24], Reimann *et al.* complete the previous work of Correa *et et al.*, they proposed new smells and new refactorings like a decomposition of OCL constraints in atomic sub-constraints. These new refactorings does not address the parameterization of the constraint which enables more reuse.

In [6], Chimak-Opoka proposed a library OCLLib which contains a group of valid OCL constraints. The main objective of this library is to offer a set of OCL constraints that are reliable, tested and can be reusable. But, no method explain how to make the constraints customizable is presented. In [32], Ton That *et al.* proposed a catalog of architecture pattern as constraint-components. They defined for each pattern its architectural constraints, they decomposed the constraints manually and embarked them in components. The component-constraints built are reusable and parametrizable. In our approach, we realized these transformations automatically and we use the result of this paper as an oracle for our experimentation.

6 Conclusion and Future Work

Architecture constraints are predicates that bring a valuable help for preserving architecture styles, patterns or general design principles in a given application after having evolved its architecture description. Such kind of specifications is subject to reuse. They are frequently assembled together to build more complex architecture constraints [31]. We have presented in this paper a process for translating architecture constraints into components. Our process is composed of two main steps. The first one consists in describing OCL constraints, extracted from "gross" textual constraint specifications, as OCL `def`initions. The second step consists in generating automatically constraint-components from these definitions. These components provide operations for checking the constraints. They are specified in an ADL named CLACS.

As a future work, we plan first to demonstrate the generality of our approach studying other predicate logic language than OCL and then we propose to make these generated constraint-components checkable in the implementation stage on component-based programs. We would like to automatically translate these constraint-components into checkable descriptors at runtime.

References

1. Bajwa, I.S., Lee, M.G.: Transformation rules for translating business rules to OCL constraints. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 132–143. Springer, Heidelberg (2011)
2. Blewitt, A., Bundy, A., Stark, I.: Automatic verification of design patterns in java. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), pp. 224–232. ACM (2005)
3. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture. On Patterns and Pattern Languages, vol. 5. Wiley, April 2007
4. Cabot, J., Clarisó, R., Riera, D.: Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 547–548. ACM (2007)
5. Chappell, D.: Enterprise Service Bus: Theory in Practice. O'Reilly Media (2004)
6. Chimiak-Opoka, J.: OCLLib, OCLUnit, OCLDoc: pragmatic extensions for the object constraint language. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 665–669. Springer, Heidelberg (2009)
7. Correa, A., Werner, C., Barros, M.: Refactoring to improve the understandability of specifications written in object constraint language. IET Software **2**, 69–90 (2009)
8. Demuth, B.: The dresden OCL toolkit and its role in information systems development. In: ISD 2004 (2004)
9. Eichberg, M., Kloppenburg, S., Klose, K., Mezini, M.: Defining and continuous checking of structural program dependencies. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008). ACM (2008)
10. Favaro, J.: What price reusability?: a case study. In: ACM SIGAda Ada Letters, vol. 11. ACM (1991)

11. Ferdjoukh, A., Baert, A.-E., Chateau, A., Coletta, R., Nebut, C.: A CSP approach for metamodel instantiation. In: IEEE Internationnal Conference on Tools with Artificial Intelligence, ICTAI 2013, pp. 1044–1051 (2013)
12. Frakes, W., Terry, C.: Software reuse: metrics and models. ACM Computing Surveys (CSUR) **28** (1996)
13. Gaffney, J.E., Durek, T.A.: Software reuse key to enhanced productivity: some quantitative models. Information and Software Technology **31**(5) (1989)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, October 1994
15. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press (2000)
16. OMG: Object Management Group. Object constraint language (ocl), v2.4, specification: Omg document formal/2014-02-03, February 2014. http://www.omg.org/spec/OCL/2.4/
17. Hassam, K., Sadou, S., Fleurquin, R., et al.: Adapting OCL constraints after a refactoring of their model using an mde process. In: BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010), pp. 16–27 (2010)
18. Hou, D., Hoover, H.J.: Using scl to specify and check design intent in source code. IEEE Transactions on Software Engineering **32**(6), 404–423 (2006)
19. Klarlund, N., Koistinen, J., Schwartzbach, M.I.: Formal design constraints. In: Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Jose, CA, USA, pp. 370–383. ACM Press (1996)
20. Lattix. http://lattix.com/
21. Meyer, B.: Touch of Class. Springer, June 2013
22. Monroe, R.T.: Capturing software architecture design expertise with armani. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (2001)
23. Petre, M.: Uml in practice. In: Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), pp. 722–731. IEEE Press, May 2013
24. Reimann, J., Wilke, C., Demuth, B., Muck, M., Aßmann, U.: Tool supported OCL refactoring catalogue. In: Proceedings of the 12th Workshop on OCL and Textual Modelling, pp. 7–12. ACM (2012)
25. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
26. Sonar. http://www.sonarqube.org/
27. Tai, K.-C.: The tree-to-tree correction problem. Journal of the ACM **26**(3), 422–433 (1997)
28. Tibermacine, C.: Software Architecture 2, chapter Architecture Constraints. John Wiley and Sons, New York (2014)
29. Tibermacine, C., Fleurquin, R., Sadou, S.: On-demand quality-oriented assistance in component-based software evolution. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 294–309. Springer, Heidelberg (2006)
30. Tibermacine, C., Fleurquin, R., Sadou, S.: Simplifying transformations of architectural constraints. In: Proceedings of the ACM Symposium on Applied Computing (SAC 2006), Track on Model Transformation, Dijon, France. ACM Press. April 2006

31. Tibermacine, C., Sadou, S., Dony, C., Fabresse, L.: Component-based specification of software architecture constraints. In: Proceedings of the 14th ACM Sigsoft Symposium on Component Based Software Engineering (CBSE 2011). ACM (2011)
32. That, T.M.T., Tibermacine, C., Sadou, S.: Catalogue of architectural patterns characterized by constraint components, Version 1.0. Technical report, July 2013, 53p
33. Zdun, U., Avgeriou, P.: A catalog of architectural primitives for modeling architectural patterns. Information and Software Technology **50**(9) (2008)