Generating Reusable, Searchable and Executable "Architecture Constraints as Services"

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

▶ **To cite this version:**

Sahar Kallel, Bastien Tramoni, Chouki Tibermacine, Christophe Dony, Ahmed Hadj Kacem. Generating Reusable, Searchable and Executable "Architecture Constraints as Services". Journal of Systems and Software, 2017, 127, pp.91-108. 10.1016/j.jss.2017.01.032 . lirmm-01706634

HAL Id: lirmm-01706634

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01706634

Submitted on 12 Feb 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Generating Reusable, Searchable and Executable "Architecture Constraints as Services"

Sahar Kallel[a,b], Bastien Tramoni[a], Chouki Tibermacine[a], Christophe Dony[a], Ahmed Hadj Kacem[b]

[a] *Lirmm, CNRS and University of Montpellier, France*
[b] *ReDCAD, University of Sfax, Tunisia*

Abstract

Architecture constraints are components of design documentation. They enable designers to enforce rules that architecture descriptions should respect. Many systems make it possible to associate constraints to models at design stage but very few enable their association to code at implementation stage. When possible, this is done manually, which is a tedious, error prone and time consuming task. Therefore, we propose in this work a process to automatically generate executable constraints associated to programs' code from model-based constraints. First, the process translates the constraints specified at design-time into constraint-components described with an ADL, called CLACS. Then, it creates constraint-services which can be registered and later invoked to check their embedded constraints on component- and service-based applications. We chose to target components and services in order to make architecture constraints reusable, searchable in registries, customizable and checkable at the implementation stage. The generated constraint-services use the standard reflective (meta) layer provided by the programming language to introspect elements of the architecture. We experimented our work on a set of 15 architecture constraints and on a real-world system in order to evaluate the effectiveness of the process.

Keywords: Architecture Constraint, OCL, Constraint-component, Constraint-service, Automatic Translation, Introspection, OSGi

Email address: sahar.kallel@lirmm.fr (Sahar Kallel)

1. Introduction: Context and Problem Statement

Software architectures play an important role in the software development process. They are specified or reconstructed and maintained throughout the life cycle in order to make persistent user requirements and to ease development. Documenting architectures provides a preliminary comprehensive view of the software structure and behavior of the software. This documentation may include various kinds of constraints, such as: i) functional constraints, which are predicates on the states of the running components constituting the architecture, and ii) architecture constraints, which are specifications of invariants on the structure of these components.

For example, if we consider a UML model (an architecture description) containing a class `Employee` (a component in that architecture) which defines an integer attribute `age`, a functional constraint representing an invariant in this class could impose that the values of this attribute be included in the range [16-70] for all instances. Such a constraint is said to be dynamic, it can only be verified at runtime. Architecture constraints deal with architecture descriptions and not with component states. As an example, a constraint representing the layered architecture style [1], states that "components in non-adjacent layers must not be directly connected together".

Both kinds of constraints can be specified using standard languages, like OCL (*Object Constraint Language*), which is an OMG's (*Object Management Group*) standard [1]. In this case, functional constraints can be written as predicates that navigate at the model level (M1) in the OMG's modeling stack[2]. Architecture constraints navigate however at the metamodel level (M2).

Many existing works [2, 3, 4] propose solutions to express at design time constraints representing architectural patterns. But unfortunately, these constraints are generally ignored at the implementation stage. They are statically checked on design artifacts (models). The question of translating architecture

[1] OCL specification: `http://www.omg.org/spec/OCL/2.4/`

[2] MDA (Model-Driven Architecture) Website: `http://www.omg.org/mda/`

constraints to become checkable at runtime is globally open. Architecture constraints should be associated with the architectures' representation available in programs codes and at runtime. Any modification of an architecture on the code or at runtime entails that the constraints should be checked again.

Furthermore, in a previous work [5], we have demonstrated that certain quality attributes may be weakened due to architecture constraint violation during software evolution. In other works [6, 7], the authors exemplify how structural constraints or design rules are violated in source code level. They also detail the consequent effects of these violations like technical dept, quality attribute losing, etc. It is thus important to be able to check them at that (implementation) level of the software's life-cycle.

Manually writing all the constraints defined at design time into executable programs is a tedious and error prone task. Besides, implementing a new interpreter for the architecture constraint language (like OCL), making it able to analyze (source code) programs, is obviously not a natural solution since it is a time-consuming task. In addition, this solution would require programmers to learn another language (the one used to specify constraints in the design phase) to specify new architecture constraints, in the implementation phase. For these reasons, we propose in this paper a process to automatically generate executable programs from architecture constraint specifications.

Instead of generating monolithic blocks of code that do not offer any reuse or customization possibilities, the proposed process transforms architecture constraint specifications, before code generation, into more structured assets in order to facilitate their reuse. Our process decomposes therefore architecture constraints into entities embedded in a special kind of software components. These components can be reused, assembled, composed into higher-level ones and customized using standard component-based techniques. These "constraint-components" are defined using an ADL (Architecture Description Language), called CLACS that we have developed in the past [8]. This ADL allows specifying the constraints in this new kind of components in order to reinforce reuse and composition. After that, the process translates automatically these components

into "executable programs" at the implementation stage. These programs take the form of components defined using the OSGi framework[3]. The generated "bundles" (components) provide services, "constraint-services", that can be invoked to check architecture constraints. In this way, architecture constraints become not only reusable, but also searchable in a service registry. Constraint-services publish operations that are able to check architecture constraints. These operations are implemented using the reflective (introspection) mechanism provided by the programming language (Java, in the current implementation) and the OSGi runtime, in order to analyze architecture descriptions and to examine the structure of "business" (functional) bundles at runtime. Architecture constraints can thus be checked after a dynamic reconfiguration of the architecture. We used this reflective capabilities, in order to exploit a standard mechanism provided by the programming language and the framework runtime, without having to use external libraries or tools.

An alternative solution to our method can be designed without transforming constraints: models of the analyzed application should be recovered or reconstructed and an OCL compiler is simply used to check architecture constraints. There are many drawbacks to this solution. First, each time the constraints should be checked, models have to be recovered, which is a costly task. Second, these models have to be always compliant with what the OCL compiler requests for the evaluation of constraints. Third, the OCL checking should be upgraded with the challenging task of dynamically evaluating constraints on the running system. At last, reuse and search of architecture constraints become difficult without the additional support provided by the solution that we propose in this paper.

This paper is an extension of a previous communication [9] at ECSA (the *European Conference on Software Architecture*) 2015. In this paper, we have particularly :

[3]OSGi Alliance Website: `https://www.osgi.org/`

- added a new step to the process, for generating executable programs, making thus possible the checking of architecture constraints on programs and at runtime
- extended the process by generating constraint-services
- illustrated the process with other richer examples
- conducted a new experiment and made additional measurements
- applied our process in a real-world system and showed the usability of our approach
- largely extended the related works

This paper is accompanied by appendices [4]. In Section 2, we give an illustrative example of the inputs and the outputs of the proposed process. This will serve as a running example throughout the paper. In Section 3, we expose our approach in a nutshell. Sections 4 to 7 describe the steps of the approach in detail. In Section 8, we present an evaluation of the approach. Before concluding and presenting the future work, we discuss the related work in Section 9.

2. Illustrative Example

To better understand the context of this work, we introduce an example of an architecture constraint (Listing 1) enabling to check the topological conditions imposed by the "Service Bus Pattern". The constraint is originally written by an architect according to her/his architecture description. The architecture imposes the existence of three kinds of components: the customers (cust1, cust2, cust3), the producers (prod1, prod2, prod3) and the bus. This later is defined as an adapter that establishes the communication between customers and producers which may have mismatching interfaces. The architecture constraint which specifies the conditions imposed by this pattern is expressed in OCL using the UML metamodel (Figure 1) in Listing 1.

[4] These appendices are available here `https://seafile.lirmm.fr/f/2faaa66069/`

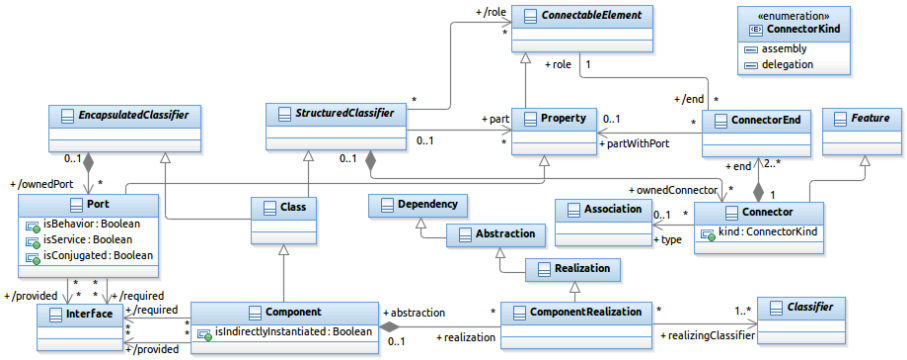

Figure 1: An Excerpt from the UML metamodel

The UML metamodel in which the constraint navigates is depicted in Figure 1. In UML, a component is a specialization of a class. It inherits all class capabilities, it can own attributes (properties), declare operations, participate in associations or inheritance relations, etc. In addition, it can have ports, with required and provided interfaces, and can define connectors. For more details, see the UML specification: `http://www.omg.org/spec/UML/2.5/`.

```
1  context Component inv :
2  let bus:Component
3  self.realization.realizingClassifier
4  ->select(c:Classifier|c.oclAsType(Component).name='esbImpl')
5  ->collect(oclAsType(Component))->asOrderedSet()->first()in
6  let customers:Set(Component)
7  = self.realization.realizingClassifier
8  ->select(c:Classifier|c.oclAsType(Component).name='cust1'
9    or c.oclAsType(Component).name='cust2'
10   or c.oclAsType(Component).name='cust3')
11   ->collect(oclAsType(Component))->asSet()in
12 let producers:Set(Component)
13 = self.realization.realizingClassifier
14 -> select(c:Classifier|c.oclAsType(Component).name='prod1'
15 or c.oclAsType(Component).name='prod2'
16 or c.oclAsType(Component).name='prod3')
17 ->collect(oclAsType(Component))->asSet()
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 only output ports
25 customers->forAll(c:Component|
26   c.ownedPort->forAll(required->notEmpty() and provided->isEmpty()))
27 and
28 --Customers should be connected to the bus only
29 customers->forAll(com:Component|com.ownedPort
30   ->forAll(p:Port|p.end->forAll(con:ConnectorEnd|bus.ownedPort
31   ->exists(pb:Port|con.role->includes(pb)))))
32 and
33 --Producers should have only input ports
34 producers->forAll(c:Component|
35   c.ownedPort->forAll(provided->notEmpty() and required->isEmpty()))
36 and
37 --Producers should be connected to the bus only
38 producers->forAll(com:Component|com.ownedPort
39  ->forAll(p:Port|p.end->forAll(con:ConnectorEnd|bus.ownedPort
40   ->exists(pb:Port|con.role->includes(pb)))))
```

Listing 1: Service Bus Pattern Constraint in OCL/UML

This constraint searches first for components representing the bus, the customers and the producers (let expressions, in Lines 2 to 17 in Listing 1). This search is performed by analyzing the architecture description of their encompassing component, which is the context of the constraint. This analysis is performed by navigating in the metamodel of Figure 1 (by following the relations between meta-classes, using the `"."` OCL operator for example). The topological conditions of the pattern are presented as comments in Listing 1. The same OCL navigation mechanism is used here to analyze the architecture description. It is obvious that this is a simple variant of the `Service Bus Pattern` (where customers and providers should be connected to the bus only, and not to other components). We can choose other more complex (potentially more realistic) variants to illustrate our work, but we used this example for simplicity reasons, to focus more on our contributions.

The result of the transformation of this constraint using the proposed process are two OSGi bundles. The first is a `query-bundle`. It provides services

generated from the let expressions. The second one is a `constraint-bundle`. It provides services to check the 5 sub-constraints that compose the OCL constraint. Two of these services are presented in Listings 2 and 3:

```
1 public class BusIdentificationImpl implements IBusIdentification{
2  public Bundle getBus(String busName){
3   Bundle[] bundles=Activator.bc.getBundles();
4   for(Bundle aBundle: bundles){
5     if(aBundle.getSymbolicName().equals(busName))
6       return aBundle;    }
7   return null; }
8 }
```

Listing 2: A sample of OSGi code generated for the Service Bus Pattern Constraint (query-bundle)

```
1 public class BusStructureImpl implements IBusStructure{
2  private IBusIdentification bi;
3  public boolean isBusStructure(String busName){
4        Bundle b= bi.getBus(busName);
5        ServiceReference[] refs1=b.getRegisteredServices();
6        ServiceReference[] refs2=b.getServicesInUse();
7        if(refs1!=null && refs2!=null) return true;
8         return false;
9  }
10 public setService(IBusIdentification b){    bi=b;  }
11 }
```

Listing 3: A sample of OSGi code generated for the Service Bus Pattern Constraint (constraint-bundle)

Listing 2 presents the implementation of a service provided, among others, by the `query-bundle` component. This service implementation is defined as a class which implements the interface `IBusIdentification`. The method `getBus` (operation of the service and whose signature is part of the aforementioned interface) returns a Bundle object representing the bus (an OSGi reification of the Bus component). In this code generation, we rely on the introspection mechanism provided by the OSGi runtime by using `getBunldes()` and `getSymbolicName()` to introspect the architecture of the business bundles, on which the constraint is checked, and to select the Bundle whose name corresponds to the value of `busName` parameter.

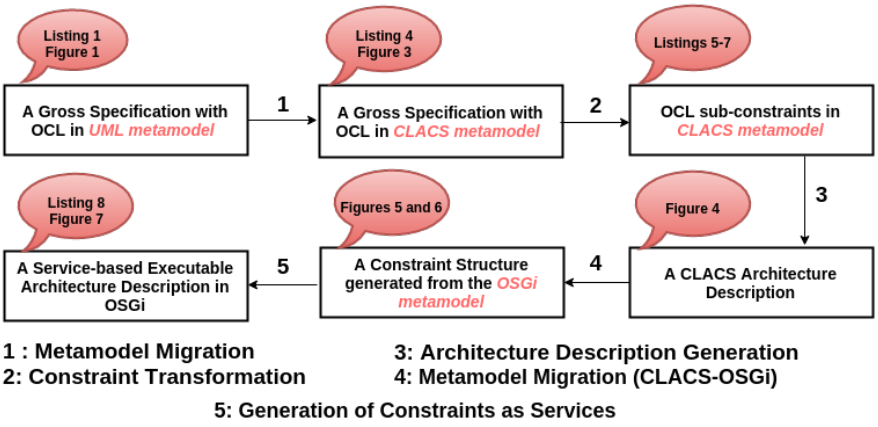

Figure 2: The Generation Process

Listing 3 shows an example of a service provided, among others, by the `constraint-bundle` component. This service provides an operation (`isBusStructure`) which checks if the bus has input and output ports. It uses `getServiceInUse()` and `getRegisteredServices()` methods from the OSGi runtime to analyze the architecture of the bus component. The Bundle object representing the bus component is obtained using the `getBus` operation invoked on an object, of type `IBusIdentification`, whose reference (assigned to `bi` field) is injected (*via* the `setService` method) by the declarative services mechanism of the OSGi runtime.

In the following section, we describe the process that we propose for generating these OSGi bundles from OCL architecture constraints.

3. The Process in a Nutshell

Figure 2 depicts the main steps of this generation process. The input of the process is an architecture constraint specified with standard languages: OCL and UML (the constraint is written with OCL, and navigates in the UML metamodel to analyze architecture descriptions defined with UML components). In the first step, we transform the input constraint into another constraint which navigates in the CLACS metamodel. This transformation is needed to make the

constraint checkable at design-time. In the second step, the constraint is decomposed from a textual "gross" specification [5] (see Listing 1) into sub-constraints in order to make them parameterized and reusable. The third step consists in changing the format of these sub-constraints into an architecture description made of "constraint-components". In the fourth step the constraints embedded in CLACS components are transformed into constraints specified on the OSGi metamodel. This transformation facilitates code generation. Finally, we obtain a service-based executable architecture description made of OSGi components. All these steps are detailed in Sections 4 to 7.

We did not perform a direct translation from OCL/UML to OSGi since this translation includes several transformations at the same time: changing the syntax of constraints, decomposing them, shifting to a new metamodel, introducing a structure around the constraints, among others.

In our approach, we use two main languages: CLACS, an architecture description language, and Java together with its component-oriented programming framework, OSGi. In the literature, there are many languages enabling the specification of architecture constraints (see [10] 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 and navigate in CLACS metamodel. The choice of UML is motivated by the fact that it is an industrial standard [6], and that OCL is its original constraint language. We consider here a repository of architecture constraints that can be fed by the software architecture community, by using these general modeling languages (easy to learn, as was experimented in [12]), which are UML

[5] By "gross" specification, we mean a specification that does not offer enough structure, reusability and parameterization.

[6] Even if a recent empirical study [11] 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.

and OCL.

OCL (we used version 2.4) has a simple and intuitive concrete syntax which enables to write expressions using first order logic, and set operations. 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.

The choice of OSGi is motivated by the fact that it provides a concrete component-oriented programming framework with a support for service-oriented architectures. It includes a service registry and a simple way of publishing and consuming services, with the declarative services mechanism. It is nowadays a specification adopted by a large number of industrials, and there are many implementations of this specification which are widely used in practice.

4. Metamodel Migration

The first step in our process consists in transforming constraints written in OCL/UML into OCL/CLACS. This is performed using a set of declarative mappings that we have specified between the two metamodels (UML – Figure 1 and CLACS – Figure 3). This mapping is shown in Appendix A.1. A CLACS component is an instance of a component descriptor (a dichotomy like in object-oriented development, where an object is an instance of a class). A component declares ports, which are characterized by a direction and a visibility. Each port has an interface which specifies a set of service signatures. Ports are linked via connectors. A connector receives service invocations through its source port and transmits them through its target port.

OCL transformation is based on the Abstract Syntax Tree (AST) generated from the initial constraint. The transformation of the OCL constraint (meta-model migration) is automatically performed with an ad-hoc manner. This means that we "programmatically" analyze the OCL's AST in depth and for each matched node (the meta-class to map) we transform the corresponding

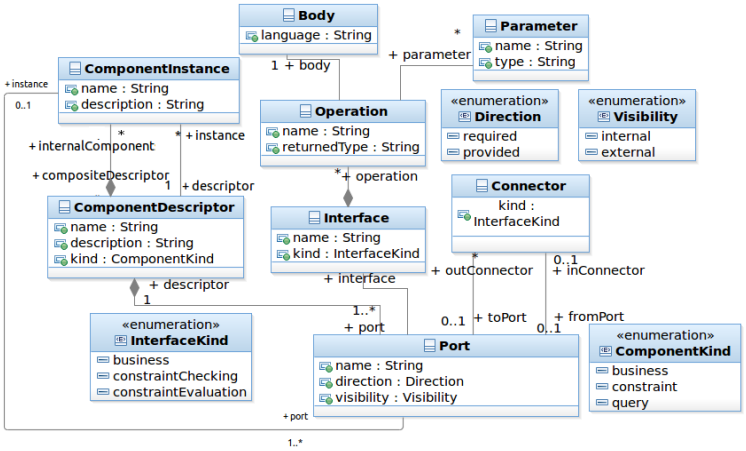

Figure 3: CLACS metamodel

part of constraint into the appropriate part based on the predefined mapping between the two meta-models. We start with the navigation patterns, then the roles and finally the meta-classes. We modify step-by-step the AST and then we generate from it the new constraint that navigates in the target metamodel. After each modification, we evaluate the constraint with an OCL compiler that validates it according to the two metamodels (UML and CLACS).

We have used XML to implement the mappings, and we have written a program for implementing the transformation instead of using an existing model transformation language like Kermeta [7] or ATL [13]. In fact, architecture constraints are not models. We might have generated models from constraints, but this process is complex to implement. It requires to transform the text of the constraint in models, to use a transformation language to transform these models and then generate again the text of the new constraint from the newly generated model. We opted for a simple solution that consists in exploiting an OCL compiler.

After applying the mappings, the resulting constraint of our example is pre-

[7] `www.kermeta.org`

sented in Listing 4:

```
context ComponentDescriptor inv :
let bus:ComponentInstance
= context.internalComponent
->select(c:ComponentInstance |
 and c.oclAsType(ComponentInstance).name='esbImpl')
 ->collect(oclAsType(ComponentInstance))->asOrderedSet()->first()in
let customers : Set(ComponentInstance)
= context.internalComponent
->select(c:ComponentInstance |
 and c.oclAsType(ComponentInstance).name='cust1'
  or...)->collect(oclAsType(ComponentInstance))->asSet()in
let producers : Set(ComponentInstance)
....
in
-- The bus should have at least one input port and one output port
bus.port->exists(p1,p2:Port|
   p1.direction=Direction::provided and
   p2.direction=Direction::required)
and
--Customers should have only output ports
customers->forAll(ci:ComponentInstance|
  ci.port->forAll(p:Port|p.direction=Direction::required
    and not(p.direction=Direction::provided)))
and
--Customers should be connected to the bus only
customers.port.inConnector->union(outConnector)
      .toPort->union(fromPort).instance->asSet()=Set{bus}
and
.....
```

Listing 4: Service Bus Pattern Constraint in OCL/CLACS

5. Constraint Transformation

At this level, our process is composed of three steps. The first step consists in extracting variable declarations from the constraint. The second one consists in decomposing the invariant of the constraint into sub-constraints. In the third step, these sub-constraints are specified as parameterized OCL `definitions`.

5.1. Variable declaration extraction

In our process we extract **let** expressions from our textual constraint specification and define them as definitions (constraints stereotyped with `def`). In our case, these OCL `definition` constraints return a value whose type is different from Boolean. At the same time, we modify the textual constraint, i.e, the constraint undergoes changes to call these generated OCL `definitions` in the appropriate places. An excerpt of the result of the transformation is presented in Listing 5:

```
1  context ComponentDescriptor
2  --let expressions extraction
3  def: getBus(): ComponentInstance = context.internalComponent
4  ->select(c:ComponentInstance | c.oclAsType(ComponentInstance)
5       .name='esbImpl')->collect(oclAsType(ComponentInstance))
6      ->asOrderedSet()->first()
7  def: getCustomers(): Set(ComponentInstance) =
8  context.internalComponent->select(c:ComponentInstance |
9   c.oclAsType(ComponentInstance).name='cust1'
10   or c.oclAsType(ComponentInstance).name='cust2'
11   or c.oclAsType(ComponentInstance).name='cust3')
12   ->collect(oclAsType(ComponentInstance))->asSet()
13 def: getProducers(): ...
14 inv:
15 ...
```

Listing 5: Constraint after extracting let expressions

5.2. Decomposition and Refactoring

In this step, we first extract the sub-constraints as OCL `definitions` and then we identify potential parameters for them to 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. This step uses as input the abstract syntax tree of the initial constraint.

We decompose automatically the obtained constraint into a set of sub-constraints. This decomposition is primarily based on logical operators used at the top level. 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 recursive process will stop when no logical operator is found in the sub-constraint. All these sub-constraints are represented as OCL `definition` constraints. The refactoring of the constraint is performed every time we generate a new `definition`. At this level, we obtain a bag (multi-set) of OCL `definition` constraints that return a Boolean value. Listing 6 represents an excerpt of our constraint during the decomposition stage.

```
1  context ComponentDescriptor
2  def: getBus(): ...
3  ...
4  def: def1(p:Port):Boolean= p.direction=Direction::provided
5  def: def2(p:Port):Boolean= p.direction=Direction::required
6  def: part1(): Boolean = getBus().ownedPort
7  ->includes(p1, p2 : Port | def1(p1) and def2(p2))
8  def: def3(p:Port):Boolean= p.direction=Direction::provided
9  def: def4(p:Port):Boolean=not(p.direction=Direction::required)
10 def: part2(): getCustomers()->forAll(ci:ComponentInstance|
11   ci.port->forAll(p:Port|def3(p) and def4(p)))
12 def: part3(): ...
13 ....
14 inv:
15 part1() and part2() and part3() and part4() and part5()
```

Listing 6: Service Bus Pattern constraint during the decomposition stage

In Listing 6, 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 with this recursive process [8]. For instance `getCustomers()`(see Listing 5) contains the operator `or`, so it will be decomposed again. All these sub-constraints are defined as OCL `definitions` presented before the `inv:` stereotype (Line 14). We can observe that there are some OCL `definitions` that have parameters. The reason to declare these parameters at this stage (of decomposition) is to have the possibility to define all the generated OCL `definitions` with the same context as that of the constraint (Line 1).

[8] In Listing 6, the decomposition is stopped in part3().

After the constraint decomposition, we obtain a bag of OCL definitions. We remove then all redundant `definitions` and we update the constraint. For instance, in Listing 6 `def1()` and `def3()` are syntactically identical.

5.3. 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, we obtain the following `getBus()` definition:

```
1 context ComponentDescriptor
2 def: getBus(name:String): ComponentInstance =
3 context.internalComponent->select(c:ComponentInstance |
4 c.oclAsType(ComponentInstance).name=name)
5 ->collect(oclAsType(ComponentInstance))->asOrderedSet()->first()
```

Listing 7: Parameterized OCL definition constraint

In this stage, we need to measure the similarity between the OCL `definitions`. This measure enables us to optimize the process, i,e. remove some redundant OCL `definitions` (obtained in the parameterization stage). An example is presented in Appendix B.1.

Concerning how we measured the similarity between OCL `definitions`, we implemented a simple solution which consists in 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 sub-tree, an edit distance [14] is measured between each pair of sub-trees. If this measure is less than a threshold[9], we consider that the two `definitions` are similar.

At the end of this step, our invariant is completely decomposed in OCL `definition` constraints that can be reused to create other invariants (a first

[9]The value of this threshold is fixed empirically.

step towards reuse).

6. Generation of CLACS components

After constraint transformation, we generate our CLACS architecture description that corresponds to the initial constraint. We generate the component operations that wrap the extracted OCL definitions and then we create the component descriptors and their connected instances.

6.1. Operation grouping

First, we describe the transformation of OCL `definitions` generated in the first step into CLACS components. 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 a Boolean value. In addition, among the generated OCL `definitions`, each one that corresponds to a `let` in the constraint will be embedded in a query-component descriptor and the others will be embedded in a constraint-component. In this case, we can obtain a large number of components. Therefore, we 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 than previously. 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 `componentInstance` and a common sub-tree generated from the expression `.port->forAll(p: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.

6.2. 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 [10] 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 checked.

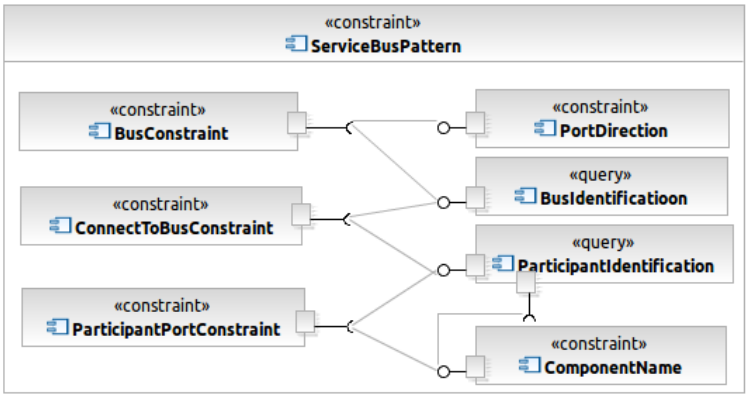

Figure 4: Sample of CLACS architecture description generation

Figure 4 shows the CLACS architecture description generated from `Service Bus Pattern` constraint. The query-components `BusIdentification` and `ParticipantsIdentification` encompass the let expressions. Besides, there are two constraint-components on the right of the figure. These components represent the OCL `definitions` that are extracted from our initial constraint and then parameterized. These `definitions` are called throughout the constraint and will potentially serve other constraints.

There are in total five sub-constraints in the architecture constraint. Each

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

one is supposed to be defined basically in a separate component descriptor. But in this example, sub-constraints 2 and 4 (part 2() and part4() in Listing 6) can be grouped in the same component descriptor (`ParticipantsPortConstraint`) because they check similar "aspects"(Port kind). `ParticipantsPortConstraint` 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, except that they apply on different sets of components. 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. (See Appendix B.2 for constraints' bodies)

Through this "componentization", constraint and query components 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), parameterizable and checkable at design time.

To make these constraints checkable at the implementation stage, we translate them into constraint-services. Section 7 presents how we generate automatically services provided by OSGi bundles from CLACS constraint and query components.

7. Generation of Constraint-services

We translate the result of the previous step into a set of services published in an OSGi registry. The business bundles (that constitute the components on which the constraint is checked) can lookup for these services in order to verify the constraint after customizing it (i.e. passing the appropriate arguments). At this level, we have a multi-step micro-process to build `constraint-services`.

7.1. Generation of Constraint-service Structure

In this step, we generate the configuration of the bundles which correspond to the constraint-service structure. Indeed, we prepare all the necessary elements

for the implementation of a bundle (packages, Java interfaces and classes, bundle configuration (XML and Manifest) documents, among other elements). Figure 5 shows the outputs of this step.

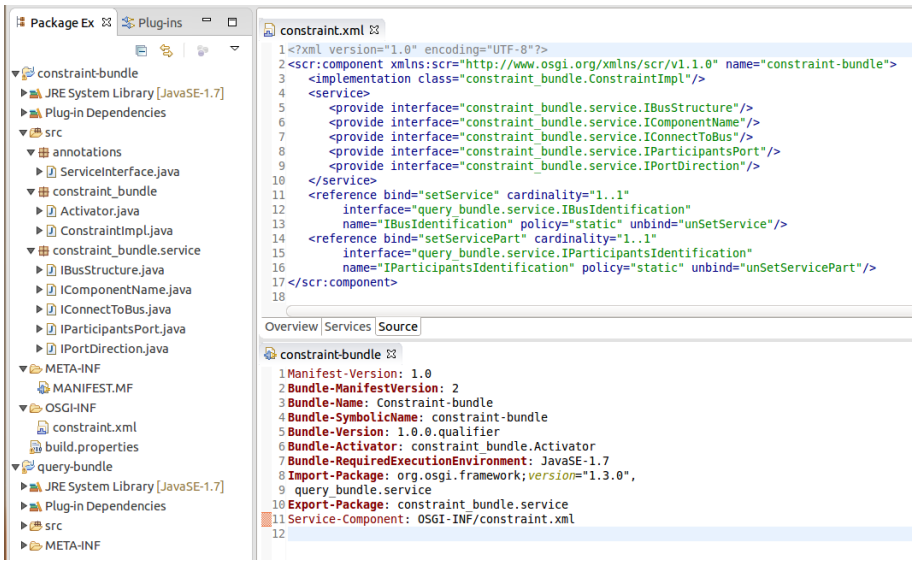

Figure 5: Structure Generation

Two packages are generated in the `constraint-bundle` . The package "constraint_bundle.service" which contains all the generated interfaces provided by all the constraint-components (Input). The parent package contains the implementation of these interfaces, which is hidden (not exported) to the other components. The `constraint-bundle` requires the interfaces of the `query-bundle`. In addition, configuration files are generated for each bundle. The Manifest file includes the imported and the exported packages (on the bottom right of the Figure) and the file constraint.xml (top right) as a component definition in the OSGi's `Declarative Services` mechanism. In this definition we specify the provided and required interfaces.

We have used a parser to implement this structure generation. All the needed information is extracted from the obtained CLACS architecture description as an XMI document. We parse this document and we generate automatically the

necessary configuration documents that compose the structure of our bundles.

After the structural translation, we generate Java code from the component operations that wrap the constraints.

7.2. Generation of Constraint-service Implementation

As done previously, we begin by translate the constraints from the CLACS metamodel into the OSGi one. Then, we generate the corresponding code.

7.2.1. Metamodel Migration

In order to make a smooth transition towards OSGi, OCL definitions obtained from the previous steps are first transformed into constraints that navigate in the OSGi metamodel. This metamodel is shown in Figure 6.

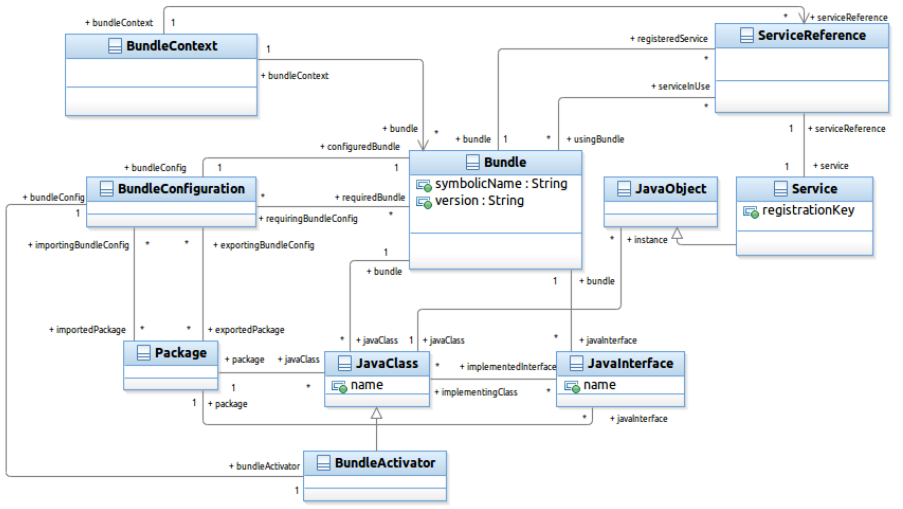

Figure 6: OSGi metamodel

A `bundle` is the main element in the OSGi metamodel. It represents a component in an OSGi application. One bundle can publish a set of services (its `registered services`). A bundle which requires services holds `ServiceReference` objects of its `services in use` (the two roles between `Bundle` and `ServiceReference`

metaclasses in Figure 6). Using a `BundleContext`, one can obtain all the running bundles and can also register a `service` and get a reference to an existing one. A service is an object, which is instantiated from an existing class which is hidden (whose package is not exported) in the bundle. It can be identified using a `registrationKey`, which is the (fully qualified) name of the main interface implemented by the class of the object. Besides, each bundle has a `configuration`, in which are declared a set of (imported and exported) `packages` that contain Java interfaces and classes.

We have implemented this translation using the same process as explained in Section 4 (UML to CLACS migration) and we have defined mappings between the two metamodel elements. Appendix A.2 details this mapping and an example is shown in Appendix B.3.

7.2.2. Code generation

For `constraint-service` code generation, we have implemented an automatic process which relies on `String Templates` [11]. The generated code makes invocations to introspection methods offered by the OSGi runtime. We used String Templates because of their flexibility (easy evolution), simplicity and the existence of a good tool support.

Figure 7 presents the mechanism used for the generation of the constraint services' code.

The starting point to generate the code is the Abstract Syntax Tree (AST) of the OCL definitions (which navigate in the OSGi metamodel).

The `CodeGenerator` is the central element in this mechanism. `Environment` and `CodeStacker` are simple elements, which are used to save information (respectively, variables and filled templates). The `CodeGenerator` reacts only to the node that it must process. For every type of node (ex. ArrowrightIteratorPostfixExp, FormalParameter, DotPropertyCallPostfixExp, etc.) we have defined a common default processing. There is a small set of nodes (compara-

[11] `http://www.stringtemplate.org/`

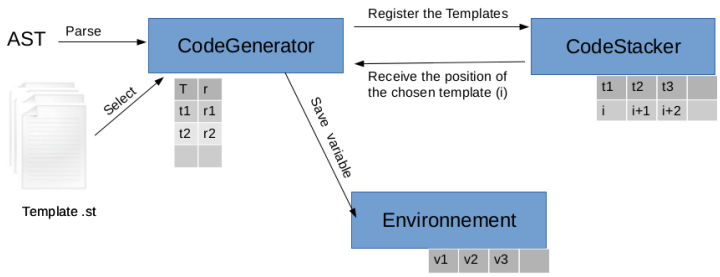

Figure 7: Code Generation Process

tively to 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, 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 it in the `CodeStacker`. The templates obtained 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, the `CodeGenerator` 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. An excerpt of the `CodeGenerator` implementation when it processes the node whose type is `InitializedVariable` is detailed in Appendix C.1. This mechanism is based on the DepthFirstAdapter pattern proposed in the DresdenOCL parser. The OCL parser 2.0 and the graphical user interface we have used are from Dresden OCL. The templates created have been created using StringTemplate 4.0.8.

One of the limitations of our approach is the fact that it does not consider

all OCL expressions such as OCLIsNew, OclAny, OclVoid and OclInvalid. From the one hand, these operations are mainly used in OCL post conditions and not in OCL invariants adopted by our approach. From the other hand, our approach currently covers the mostly used OCL expressions in invariants. Besides, our tool is flexible, in order to integrate a new OCL expression. We just need to write a specific String Template and to implement a Java method that initializes the String template.

Listing 8 shows an excerpt of the generated code for the implementation of the service `IBusStructure`. The remaining of the code is illustrated in Appendix C.2. Note that this service has references to `IBusIdentification` and `IPortConstraint` to invoke respectively `getBus()`, `isProvided()` and `isRequired()` operations.

```
1  public class BusStrutureImpl implements IBusStructure{
2   // Reference to IBusIdentification
3   IBusIdentification ibi;
4   // Reference to IPortDirection
5   IPortDirection ipd;
6   public void synchronized setService(IBusIdentification bi){
7     ibi=bi;   }
8   public void synchronized bindService(IPortDirection pd){
9     ipd=pd;   }
10  public boolean isBusStructure(String name){
11   Bundle bus=ibi.getBus(name);
12   boolean bool1=ipd.isProvided(bus);
13   boolean bool2=ipd.isRequired(bus);
14   boolean bool3=bool1 && bool2;
15   return bool3; }
16 }
17 public class BusIdentificationImpl implements IBusIdentification{
18   public Bundle getBus(String name){
19   Bundle bus=null;
20     Bundle[] bndl = Activator.bc.getBundles();
21     ArrayList<Bundle> bndls = new ArrayList<Bundle>();
22   for(Bundle b : bndl) {
23     boolean bool = b.getSymbolicName().equals(name);
24     if(bool)  bndls.add(b);  }
25   Bundle[] bndls2 = new Bundle[bndls.size()];
26   int selectiterator = 0;
27   for(Bundle b : bndls) {
28     bndls2[selectiterator] = b;
```

```
29      selectiterator++; }
30    bus = bndls2[0];
31    return bus;
32    }
33  }
```

Listing 8: Example of a generated code

It is worth noting that this code is syntactically different from the optimal code presented at the beginning of the paper (see Listings 2 and 3) but they are semantically equivalent. It is obvious that the automatic translation does not allow to obtain a code having an optimal complexity. However, it is a valuable tool for developers who will rather focus on implementing the business logic of their application.

7.3. Registering and Looking-up Constraint-Services

We have added a set of properties for each service to be published in the registry. These properties contain all the OCL constraints that are embedded in the CLACS component operations which correspond to this service. This is done by generating a set of "property" tags in the component definition file. Besides, when we generate the OSGi code associated to this CLACS component, we annotate each operation, in each interface, with the corresponding OCL constraint as a string value, in order to know what is the operation that should be invoked by the business bundle.

To illustrate these modifications, we present an example of an architecture description and the associated architecture constraints of the layered architecture style [1]. An architecture constraint, among others, of this architecture style is that, "components in non-adjacent layers should not be directly connected together".

We would like to check this architecture constraint, embedded in the constraint-component, at the implementation stage of an OSGi component/service-based application. Therefore, we follow the proposed process of constraint-service generation. But before generating code as described previously, our process checks

if there is a service, registered in the service registry, which checks the same layered architecture constraint or a part of it. For that, it searches all the services that are registered in the OSGi service registry. It looks for the properties of these services and it compares the layered architecture constraint with the OCL constraints that exist in the properties. If it finds one which is equivalent to the constraint, then it is not necessary to generate the corresponding service. The name of the interface of the service and the signature of the operation which is annotated with the searched OCL constraint are retrieved. If not, then the process follows the previous steps of constraint-service generation.

The layered architecture constraint above is the same as the one which is embedded in the `ConnectToBusConstraint` component-constraint. In order to lookup the registered service (generated from `IConnectToBus` interface), the process adds automatically a `reference` tag in the Component Definition file in the constraint-bundle which corresponds to the layered architecture pattern. This tag needs the name of the service interface (`IConnectToBus`) and also two operation names to bind and unbind the service. The bind operation has as a parameter IConnectToBus ictb. Then we are able to invoke ictb.areConnectToBus(...) in the implementation of the newly generated constraint-bundle for the layered architecture pattern.

8. Process Evaluation

The experimentation presented in this paper complements the one we exposed in our previous work [9]. In the latter experimentation, we evaluated the reuse brought by the decomposition of constraints and their parameterization. We focused on measuring reusability in the generated CLACS components, using a well-know metric [15].

In this paper, the experimentation's goal is to answer the following research question: *What is the performance of the generation process and to what extent the output of this process (the concept of "reusable, searchable and executable constraint as a service") is useful in a real-world scenario?*

We decompose this research question in two sub-questions:

- **RQ1**: What is the performance of our constraint-component and constraint-service generation process, compared to a manual design and coding of these artifacts?
- **RQ2**: How can we use our approach in a real-life scenario and what is the overhead when applying it in such a scenario?

8.1. Comparison of manual constraint specification and automatic generation:

To answer **RQ1**, we need to compare the automatic process presented in this paper with a "traditional" manual specification of constraints. For this purpose, we invited some external users.

Data Collection. We invited 8 Ph.D students to collect our experimentation data and to evaluate the process. All of these students work on software architecture in their thesis.

From the literature [2, 3, 4] a set of architecture patterns and styles has been collected. Only those related to the structural aspect of the architecture have been selected. 15 patterns including their variants have been identified. We choose 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. For each pattern, the group of students involved in the evaluation specified their architecture constraints [12] as the topological conditions that an application's design and implementation should respect. These constraints have been specified on the UML metamodel.

We have also used in this experimentation a set of CLACS constraint-components which have been manually designed in [16] by another Ph.D student, who did not participate in this experimentation.

[12] These constraints are available in the following website: `https://seafile.lirmm.fr/d/2b7cf7c85c`

Table 1 shows the description of the experimentation data. The first column presents the architecture patterns while the second column shows the size (in terms of number of tokens in the AST) of the architecture constraints that formalize them. We have chosen constraints with different sizes, ranging from 434 tokens for the smallest to 2511 for the largest one.

Table 1: Size of pattern architecture constraints

Pattern	**Size (# tokens in AST)**
Service Bus	1423
Layered Architecture	503
Client-Server	507
Broker	733
Layered Architecture - Hybrid	1426
Pipe-Filter	512
Pipe-Filter - Group not Layered	2511
Pipe Filter - Layered	834
Pipe Filter - Layered (2)[13]	1810
Pipeline	869
Pipeline (2)	955
MVC	434
Facade	650
Microkernel	826
Legacy Wrapper	518

Each Ph.D student was asked to write all the constraints with OCL and give a difficulty coefficient in a scale ranging from 1 to 5. We measured the time spent to write them.

Figure 8 depicts the average time in minutes spent to write OCL architecture constraints by following the chronological order of the patterns appearance (from left to right). We can observe a downward trend, the first constraint took more time to be specified than the others (more than 3 hours), the overage time decreases when specifying more constraints despite of their size variance (For instance, Pipeline(2) and Microkernel). Indeed, the Ph.D students have naturally acquired experience when specifying each time a new constraint. This

[13] Another variant of the pattern.

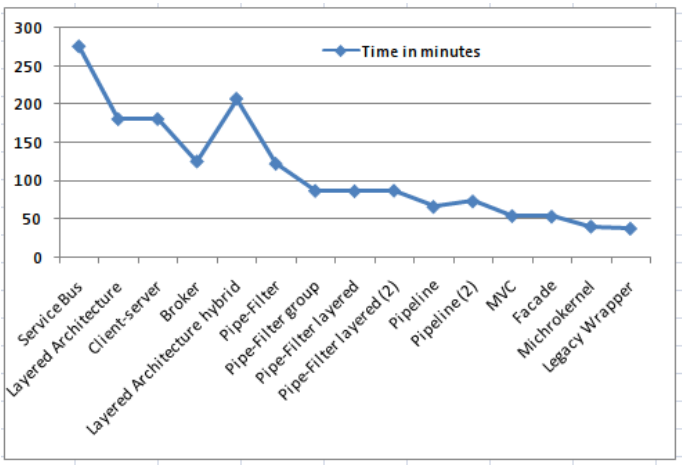

Figure 8: Average time spent to write OCL architecture constraints

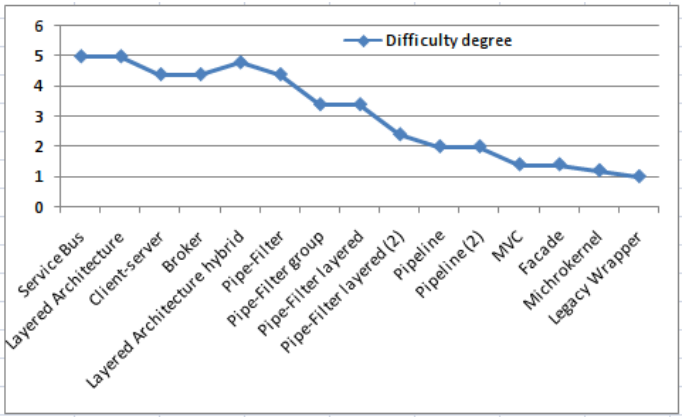

Figure 9: Average difficulty coefficient to specify OCL architecture constraints

natural acquisition of knowledge is explained also by the decrease of the average of difficulty coefficient depicted in Figure 9. OCL is a language easy to learn and to use [12]. The students need only to know for each constraint the appropriate navigation in the UML metamodel and use always the same OCL expressions that are naturally clear like *forAll*, *exists* and *select*.

Protocol. The experimentation protocol consists of two steps. In the first step, we measured precision and recall by comparing CLACS constraint-components

manually designed in [16] with those generated using our process. This has been performed on a subset of the catalog of constraint-components in [16]. The selected constraint-components are those which formalize the patterns presented in Table 1.

In the second step of this experimentation, we defined manually for each constraint its corresponding constraint-services, and then we generated them using our process. In the manual coding of these constraint-services, we used the same service interfaces (their qualified names and operation signatures) than in the generated ones. The reason behind this choice is to focus on the comparison of the generated code and not on the structure around it. At the end, we compared the two source code artifacts (the manually written one and the generated one). It is obvious that an exact matching of these two types of code artifacts provides false results (it is impossible to produce exactly the same code following the two procedures). Then, we decided to use "Clone Detection" techniques. We relied in this evaluation on some metrics presented in [17]. These metrics measure the distance between two portions of source code and enable us to calculate the recall and the precision (by considering the relevant code, the one which was manually written).

Metric used in the First Step. Precision and recall metrics are calculated as follows:

$$(1)\ \text{Precision} = \frac{tp}{tp+fp} \quad \Big| \quad (2)\ \text{Recall} = \frac{tp}{tp+fn}$$

where: **tp** (true positives): are the generated constraint-components which are equivalent to the constraint-components which have been manually designed in the catalog or which are reusable, **fp** (false positives): are the constraint-components which do not exist in the catalog manually designed and which can not be reusable and **fn** (false negatives): are the constraint-components designed manually but which have not been generated by the proposed process.

We observed that all the generated constraint-components exist in the catalog. This means that the false negatives are equal to 0. So, recall is always

equal to 1. (This will not be taken into consideration in the results later.)

Metric used in the Second Step. In this step, we use a metric for clone detection defined in [17]. We call each Java program in the OSGi source code written manually, a `reference` R. Besides, we call the generated program a `candidate` C. A pair of clones here is a couple formed by a candidate and a reference, known a priori to be the programs that correspond to the same constraint. To compare R and C, we decompose each program into fragments. The size of a Java fragment do not exceed 6 lines[14]. During the analysis of our data, we observed in some cases that we can not decompose programs into fragments respecting the size condition because the "cutting" of constraints did not occur at the right place. So, we decided to manually adjust the size of fragments, case by case. Before comparison, C and R should be normalized [17].

The metrics used in this evaluation process are:

$$(3)\ Contained\,(fR, fC) = \frac{|lines\,(fR) \cap lines\,(fC)\,|}{|lines\,(fR)\,|}$$

$$(4)\ Ok\,(R, C) = min\,(Contained\,(fRi, fCi)\,, i < numberOfFragmentsIn(R, C))$$

where:

fR: a code fragment in a manually written program

fC: a code fragment in an automatically generated program

For the same OCL constraint, the size of a generated program is higher than the size of the manual code. For this reason, we calculated the ratio of code of each manual code fragment contained in the automatic one. We evaluated Metric (3) above in calculating the number of common lines in each code fragment in R and C and then we divided the result by the total number of code fragments in R. These C and R belong to clones of type 2. Then, we calculated the minimum of these values in the same pair of clones (R,C) to

[14]The choice of this size was taken after having analyzed the generated code of constraints. Indeed, this size corresponds to the smallest generated code. The generated code have a size which is roughly a multiple of 6.

obtain the Ok values (Metric (4)). In other words, the $Ok(R, C)$ is the minimum of the *Contained* values for the fragments which constitute R and C. After that, we measured precision and recall as previously.

To calculate the precision and recall in this step, we have identified the true positives, the false positives and the false negatives as follows:

tp: true positives are candidates that are correct and which have $Ok >= 0.7$[15]. Candidates are correct when they return correct results in the test cases applied on them. Each candidate is checked on several variants of applications we have developed. In these applications, patterns are instantiated (in one variant) and partially invalidated (in each other). By partial invalidation, we mean that the evaluation of one of the sub-constraints returns false.

fp: false positives are candidates that are correct and which have $Ok < 0.7$

fn: false negatives are candidates that are not correct.

Results and Discussion. For the first part of our experimentation, we obtained the results which are presented in Table 2

Table 2: Precision values in the first step

Pattern	**tp**	**fp**	**Precision**
Service Bus Pattern	9	0	1
Layered Architecture Style	5	0	1
Client-Server	5	1	0.83
Broker	6	3	0.66
Layered Architecture-Hybrid	4	1	0.8
Pipe-Filter	4	2	0.66
Pipe Filter- Group not Layered	11	10	0.52
Pipe Filter- Layered	7	3	0.7
Pipe Filter- Layered (2)	7	0	1
Pipeline	11	1	0.91
Pipeline (2)	15	5	0.75
MVC	3	0	1
Facade	4	0	1
Microkernel	6	2	0.75
Legacy Wrapper	8	5	0.61

[15]The choice of the threshold is explained in [17]

We observe that there are 5 patterns among 15 that have precision equal to 1 and 10 patterns that have precision > 0.7. The decrease of the precision value for the `Pipe Filter-Group not Layered` pattern is due to the decomposition in depth of this very large constraint, which provides a large number of low-level constraint-components that are unexploited in terms of reuse. Pattern `Pipe Filter- Layered (2)` has also a large size (1800 tokens) but it has precision equal to 1. This is explained by the fact that all of the obtained OCL constraints after decomposition are reusable in other constraints in our data set.

For the second step of our experimentation, we present in Table 3 the measures obtained after applying the evaluation protocol on the selected architecture constraints. Note that the number of the interfaces is not included in the number of candidates. All the generated interfaces are identical to those written manually.

Table 3: Experimentation values in the second step: Precision and Recall

Patterns	**#C**	**tp**	**fp**	**fn**	**Precision**	**Recall**
Service Bus Pattern	9	6	3	0	0.66	1
Layered Architecture	5	3	2	0	0.6	1
Client-Server	5	4	1	0	0.8	1
Broker	9	4	3	2	0.57	0.66
Layered Architecture - Hybrid	5	3	2	0	0.6	1
Pipe Filter	8	4	2	2	0.66	0.66
Pipe Filter - Group not Layered	21	7	11	3	0.388	0.7
Pipe Filter - Layered	10	3	3	4	0.5	0.42
Pipe Filter - Layered (2)	7	2	2	3	0.5	0.4
Pipeline	12	7	4	1	0.63	0.87
Pipeline (2)	19	9	8	2	0.53	0.81
MVC	7	5	2	0	0.71	1
Facade	9	7	2	0	0.77	1
Microkernel	9	6	1	2	0.85	0.75
Legacy Wrapper	9	4	3	1	0.57	0.8

As we can observe in Table 3, the generated constraint-services are pertinent (in half of the cases, Recall > 0.7) compared with the constraint-services manually coded. We have 13 patterns with precision > 0.6 and 6 that have recall $=1$. The high values of precision and recall for some patterns are explained

by the fact that the architecture constraints of these patterns are decomposed into sub-constraints (OCL definitions in our approach) with a small size and they are simple in terms of number of navigations and OCL quantifiers. The pattern `Pipe Filter Group not Layered` has a low precision (≈0.39) because its generated constraint-services have many candidates and some of them are very complex (with many nested quantifiers).

Table 4: Measures of time (in seconds) spent in process steps

	Transformation		**Coding**		**Execution**	
Patterns	$T_{T(M)}$	$T_{T(A)}$	$T_{C(M)}$	$T_{C(A)}$	$T_{E(M)}$	$T_{E(A)}$
Service Bus Pattern	2460	0.118	4205	0.531	0.450	0.583
Layered Architecture	1502	0.087	4380	0.501	0.380	0.433
Client-Server	2456	0.073	3960	0.430	0.200	0.290
Broker	2700	0.106	4380	0.456	0.307	0.468
Layered AH	3402	0.148	7298	0.691	0.506	0.601
Pipe Filter	2400	0.066	4385	0.511	0.248	0.354
Pipe Filter - GNL	3422	0.199	8580	0.861	0.640	0.823
Pipe Filter - Layered	3300	0.076	5526	0.583	0.327	0.478
Pipe Filter - Layered (2)	2400	0.253	8400	0.654	0.654	0.780
Pipeline	2400	0.093	5842	0.431	0.376	0.444
Pipeline (2)	3209	0.166	4231	0.743	0.870	0.996
MVC	1460	0.088	4688	0.467	0.487	0.482
Facade	1202	0.088	4390	0.497	0.621	0.762
Microkernel	1800	0.101	4980	0.670	0.675	0.777
Legacy Wrapper	1760	0.111	4354	0.501	0.544	0.564

Table 4 presents the measures of time of different steps of the process, manually operated (by the Ph.D students) and automatically performed. We present the average of the measures for each step. Column 2 presents the time spent in manually transforming CLACS constraints into constraints that navigate in OSGi metamodel, while Column 3 presents the time spent in the automatic transformation. The fourth column depicts the time in manually coding the constraints into Java/OSGi while the fifth depicts the same step in automatic manner. Finally, the two last columns present respectively the execution time (in milliseconds) of the manually created and the generated source code.

As we can observe, it takes for a developer an average of 2.5 hours to code manually a Java OSGi source code that allows to check an architecture constraint without considering the time spent to configure the OSGi bundles. It is obvious that the manual tasks need more time than the automatic ones (automation reduces time to 98% in code generation, and thus in maintenance in our case (constraint checking)), but the interesting aspect in these results is when we compare the values in the two last columns. We can notice that the execution time of the generated source code is higher than the execution time of the manual one, in all cases. This is explained by the fact that the generated code is longer than the manual one and this latter is more optimal. The average overhead of the generated code is +22,15% (in milliseconds). But this is negligible and does not affect much the overall process.

8.2. Case study:

In this subsection, we present an example of a real system on which we applied our approach. We present first the application of the concept of "reusable, searchable and executable constraint as a service" in this example and then we measure the overhead of producing these constraints and the overall system performance overhead in terms of execution time using these constraints as services.

Our example is a dynamic and extensible ambient assistive living framework called `Ubiquitous Service Management ARchiTecture` (`UbiSMART`) developed by a research team in our research institute (LIRMM). This framework enables to develop applications which allow to detect unusual behavior of senior persons who live alone or in a nursing home [16].

`UbiSMART` is structured as a web application implemented as OSGi services, on a cloud server connected to many assisted houses. It is composed of two essential parts. The first part, Sensing part, is located in the patient residence

[16] `UbiSMART` is deployed in Saint Vincent De Paul Nursing home in Argentan, Normandie, France and in a Peacehaven nursing home in Singapore since 2012 [18]

and is composed of multiple sensors, a gateway, and communication devices. It is in charge of pre-processing the raw data from the sensors, converting it into events that are sent to the server via Internet. The second part is a Web platform, which handles the communication with the Sensing part through the MQTT communication protocol[17]. It also manages the platform storage, reasoning and the service provisioning that will be explained later. The reasoning part determines the activity and acts of senior persons as a trigger for the service provisioning [18].

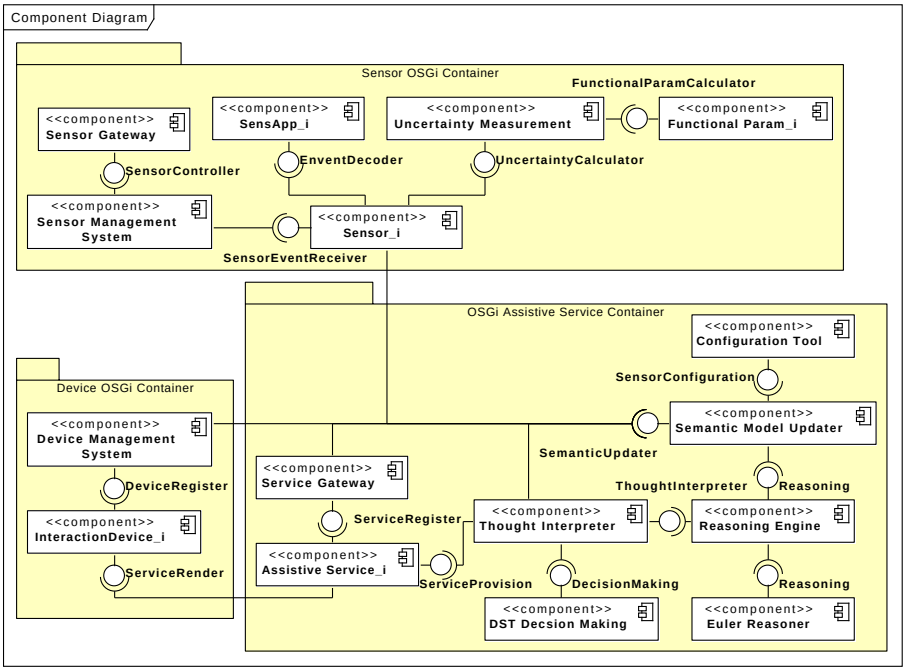

Figure 10: Component Diagram of the UbiSMART framework [18]

The component diagram of `UbiSMART` depicted in Figure 10 represents the different modules of the framework that have been implemented and the interactions between them. In the OSGi implementation, each module exports one or more services (interfaces) which are imported and consumed by other modules.

[17] `http://mqtt.org/`

The main parts of the UbiSMART components are the following:

- "Sensor i" is an abstraction of all the sensors' modules. Each of them has an exposed service "SensorEventReceiver" used by the "Sensor Management System" (SMS) to send new sensors' events.
- "Thought Interpreter" (TI) decodes the reasoning engine output. This module updates the semantic model through the "SemanticUpdater" (SMU) service and starts the selected service through the "serviceProvision" interface.
- "Service Gateway" (SG) is responsible of registering the different assistive services. It uses the "semanticUpdater" interface to register new assistive services to the semantic model.
- "InteractionDevice i" is an abstraction of all the detected interaction devices in the environment.
- "Device Management System" (DMS) uses the "SemanticUpdater" service to update the semantic model with the newly detected devices description.

We have realized an interview with UbiSMART developers to give us the structural conditions which UbiSMART architecture should respect at runtime. Here is the list of the main architecture constraints of the framework:

1. UbiSMART architecture should respect the Layered pattern; the Sensor part should work the first, then the Reasoning part and finally the Device part in order to get low uncertainty results.
2. (SG, SMS and Sensor-i) and (SG, Assistive Service-i and InteractionDevice-i) should respect the Service Bus pattern. This constraint is necessary to safely transmit for each sensor or each device the corresponding data or events.
3. Sensor-i and SMU should be directly connected in order to notify the reasoning engine of a new discovered sensor in the environment. The same

condition should be respected for DMS and SMU to notify the apparition of a new device.

4. A unidirectional connection should be maintained between some `UbiSMART` components such as (DST,TI), (TI,RE), etc

To apply our approach in this real system, we need to specify these textual conditions with OCL on UML metamodel. We asked one of the framework contributors to write the conditions 1 and 4 with OCL knowing that he is a non-specialist in OCL but as a computer scientist he knows first order logic [18]. It took him approximately 230.66 minutes for the specification of the first constraint and 97.85 minutes for the second knowing that the later one is more complex. It is trivial that a non specialist takes more time to specify an OCL constraint (if we compare these values with those of Figure 8). The first value has decreased to 113.97 minutes when we provided some architecture constraints to the contributor.

`UbiSMART` is a dynamic framework allowing to present as a service any sensor or device discovered at runtime in the environment. Even `UbiSMART` allows the integration and representation of sensors and devices as services, they are still not integrated in the reasoning process. Thus, it is not possible to use them in the selection of the end-user service and the interaction device. To solve this problem, the contributors introduce semantic Plug&Play to register these bundles (2-6 in Figure 3.8 in [18]). To ensure the integration of these generated bundles in the system at runtime, the framework needs to respect Constraint 3 to guarantee the connection between each discovered sensor and SMU, and between each discovered device and SMU at runtime. This constraint needs to be `executable` at runtime to check the integration process. Here, we apply our approach to generate the constraint 3 as a service. We choose to generate it as a service to guarantee a loose coupling and easy connection and/or disconnection with the business services (SMU, sensor-i bundles). In addition, Constraint 3

[18]The constraints are presented in Appendix B.4

should be applied for two pairs of bundles. First, we applied our approach's steps on the constraint starting with the decomposition, the generation of constraint-components to reach constraints as services registered in the OSGi runtime. Then, as the generated constraint has already been registered, we search the appropriate service and we reuse it by passing the appropriate parameters during its invocation (here, the names of the DMS and SMU bundles).

Constraint-services production time. The production of the constraint-bundle that provides the constraint-services took 0.887 seconds, i,e the constraint-services structure (the bundle, the configuration files, the interfaces, the classes, ...). Our approach produces 20 constraint-services, i,e the Java code elements generated from the OCL definitions obtained after applying our approach on the `UbiSMART` constraints. These code portions needed 4.943s to be generated. The production of our constraint-services in terms of structure and code took in overall 5.83s. Constraint-service production is statically performed and it does not influence on the system reaction time.

Constraint-service execution time. In `UbiSMART`, in both the static and dynamic configurations, an average time of 0.224s is needed for starting the real sensor and for the communication required to detect the sensor presence in the environment by the framework. In addition, for the dynamic configuration, we observed an additional average time of 0.373s needed to represent an ultrasound sensor as a service in the framework. This is the time required for generating and starting the bundle representing the ultrasound sensor on the gateway and for updating the environment description with the sensor information. After that, the system's reaction time, calculated between the time a service is needed and the time it is delivered in the environment, has an average of 2.713s, which has been refined in 1.226s for the reasoning engine module's processing itself, 0.735s for the communication between modules and 0.752s for the processing due to other miscellaneous bundles [19].

In the other hand, the 20 constraint-services took in overall 2.787s to execute. The overhead is 50.15%. Indeed, using our approach, the `UbiSMART` reaction

time increases to 5.777s instead of 2.770s. But this high percentage is inherently related to the nature of the system (`UbiSMART`), whose services execute in very short times. In other kinds of systems (business applications with data access layers, for instance), we are quite confident that the overhead of constraint checking at execution time is marginal.

8.3. Threats to validity

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

8.3.1. Internal validity

In our evaluation process, we have used architecture patterns that are specified from several sources to mitigate the risk of forgetting some patterns conditions. Besides, in our selected architecture patterns, we can find several variants for a given pattern like the `Pipe-Filter`. 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.

The constraints that formalize our patterns are specified by participants who have experience with OCL. We have involved 8 persons to perform different tasks in the evaluation. We have invited also a non expert participant in OCL language to write architecture constraints and we have compared the results.

8.3.2. 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 component-based architecture patterns or that only constraints written in OCL can be evaluated as input and Java/OSGi as output. Any kind of architecture constraints can be considered, including GoFs object-oriented design patterns or SOA patterns. It suffices to specify them in UML metamodel. For any kind of predicates analyzing architecture descriptions, a parser should exist for their specification language. Constraint transformation

step is applicable with any metamodels because it uses external mapping 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. These String 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.

9. Related Works

Works related to our approach can be classified in different categories: i) languages and tools for the specification of architecture constraints, ii) methods for predicate/constraint transformations, iii) methods for OCL constraint refactoring, iv) methods for constraint reuse, v) methods and tools for code generation from OCL, and vi) works about architecture constraint checking for design patterns/styles.

A state of the art on languages used for the specification of architecture constraints at design and implementation stages is presented in [10]. These languages vary from embedded notations in existing ADLs, like Armani for Acme, FScript[19] for Fractal ADL or REAL for AADL, to notations with a logic programming style, like LogEn or Spine, or notations with an object-oriented programming (OOP) style or for OOP languages, like CDL, DCL or SCL. In practice there are several tools for static code quality analysis that enable the specification of architecture constraints, like Sonar, Lattix, Architexa and Macker, 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.* [20] proposed a method for transforming OCL constraints during UML model refactoring, using model transformations. Their approach

[19] A tutorial for this language is available in the following SVN repository: svn://forge.objectweb.org/svnroot/fractal/tags/fscript-2.0

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 and ad-hoc way without using additional modeling and transformation languages.

In [21], the authors propose an approach for generating (instantiating) pertinent models from metamodels taking into account OCL constraints. Their approach is based on a CSP (*Constraint Satisfaction Problem*) solver. They defined formal rules to transform models and constraints associated to them. Cabot *et al.* [22] 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 [23]) to analyze constraints. In contrast to CSP, our work 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, in [24], Cabot and Teniente proposed a transformation technique of OCL constraints into other simpler OCL constraints semantically equivalent using transformation rules. The paper addresses endogenous transformations, it does not propose constraint transformations expressed in different metamodels. Bajwa and Lee presented in [25] a two-step process for transforming SBVR rules (Semantics of Business Vocabulary and Business Rules) into OCL constraints. The first step consists in defining 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 into an OCL model instance an SBVR model instance using a mapping between the two metamodels (OCL

and SBVR). This paper uses model transformation techniques. Their process is troublesome when the constraints have a gross specification (very large models). The generated constraints are complex, not reusable and parameterizable.

OCL refactoring consists in simplifying the constraints and making them more optimized. In [26], the method proposed by Correa *et al.* has as a goal to improve the readability and the comprehensibility of constraints. Therefore, they prepared a catalog of smells. They proposed refactorings for removing a given smell in a constraint. It is true that this refactoring improves 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 [27], Reimann *et al.* complete the previous work of Correa *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 the practice of model-driven engineering, there exist several tools to translate OCL constraints in Java source code, like Eclipse OCL [20], Octopus [21], and DresdenOCL. They however transform constraints which are functional and not architectural. These tools translate this kind of constraints into object-oriented programs which do not use the introspection mechanism. Other works in the literature deal with code generation for functional constraints too. Briand *et al.* in [28] proposed an approach to transform functional constraints into Java using aspect-oriented programming. Another work [29] proposed a method for translating functional constraints in JML (Java Modeling Language). In a previous work [9], we developed a method for transforming OCL architecture constraints into Java metaprograms. But in this work, the transformation result is not an easily reusable and customizable architecture constraint, that is why we proposed in this paper to translate constraints into constraint-components then

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

[21] `http://octopus.sourceforge.net`

into services.

There are many works ([30] for a survey) that propose methods to validate constraints on several kinds of applications. In most of cases, these constraints are not architectural. We can find functional and reconfiguration constraints. For the few works that consider structural constraints they discuss only dependencies between components and not the conditions imposed by the application architecture. In [31], the authors insert skeleton code in user source code to verify functional constraints. Besides, the user is involved in all the process steps whereas our verification process is fully automatic and user source code is not affected.

The authors in [7] introduced design rule spaces, a new form of architecture representation that uniformly captures both architecture and evolution relations using design structure matrices. They proposed that software architectures should be viewed and analyzed as multi-layered overlapping DRSpaces, because each DRSpace, formed using different types of primary and secondary relations, exhibits meaningful and useful modular structures. They were able to identify structural and evolutionary problems. This work has the same goal as us but our approach focuses on the internal problems within a file, rather than the structure among files.

In the same context of architecture constraints, Fowler [32] describes the concept of a "bad smell" as a heuristic for identifying refactoring opportunities. Others [33] have extended this notion to include architecture-level bad smells. Automatic detection of bad smells has been widely studied. For example, Moha et al. [34] presented the Decor tool and language to automate the construction of design defect detection algorithms. There is a number of proposals for automatically detecting bad smells which may lead to refactorings. For example, Tsantalis and Chatzigeorgious study [35] presented a static slicing approach to detect and extract method refactoring opportunities. Our approach is different. First, it focuses on architecture constraints and not on "bad smells". Our constraints target the structure of the application and are related to the architecture decisions taken on the design stage. Second, existing research on

bad smells has focused on analyzing a single version of the software, while our approach examines the application's evolution.

In [36], the authors propose a method for extracting models from the source code of an application and check functional constraints by using an OCL interpreter. We can not apply this approach for our problem, because architecture constraints use meta-level constructs and not model-level ones. Besides, in our approach, we can check constraints at run-time using the reflective layer and service registry access methods and we obtain reusable and customizable constraints contrary to their method.

PEC [37] is a pattern enforcing compiler for Java. Using interfaces to identify the intended design pattern, the tool combines static testing, dynamic testing (unit testing), and code generation to verify that the pattern is implemented according to a specification. Since it obliges to add some interfaces and more statements in client source code in order to enforce the implementation of design patterns, the code will be difficult to understand and maintain. It is impossible to apply PEC in service-based applications because it is difficult to manage these injected interfaces throughout the code.

Experienced developers apply design patterns in software development to solve design problems and reduce software maintenance cost. However, software systems evolve over time, increasing the chance that the design patterns in their original form will be broken. To verify the original intent of the design patterns, Blewitt, Bundy and Stark [38] presented a pattern specification language Spine that allowed design patterns to be defined in terms of constraints on their implementation in Java. In our work, we have used the same language for coding the application to express constraints at implementation stage. Our goal is to use the standard mechanisms offered by Java, such as introspection.

10. Conclusion and Future Work

Architecture constraints are predicates that bring a valuable help for preserving design rules, like the instantiation of architecture styles or patterns

in a given application, after having evolved its architecture description. We have presented in this paper a process for generating code starting from architecture constraint specifications. Our process is composed of two main steps. The first one consists in generating constraint-components from "gross" textual constraint specifications. These components provide operations for checking constraints. They are specified in an ADL named CLACS. The second step generates services, which can be invoked: i) at the implementation stage to check architecture constraints on source code, ii) at runtime to check these constraints after a dynamic evolution of the architecture, and iii) by any external application to check constraints on its architecture, simply by making a lookup in the service registry of the runtime environment in which the services have been published.

In the near future, we plan to work on lightweight instrumentation of source code, using annotations and aspects, to statically and dynamically check "architecture constraints as services". As a perspective to this work, we envision to generalize this approach, *i.e.* to specify architecture constraints in a paradigm- and language-independent way, by using predicates on graphs and operations on them, and then to make automatic transformations towards object-oriented, component-based or service-oriented architectures, using feature models that specify the variability between these paradigms.

References

[1] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

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

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

[4] F. Buschmann, K. Henney, D. C. Schmidt, Pattern-Oriented Software Architecture, Volume 5, On Patterns and Pattern Languages, Wiley, 2007.

[5] C. Tibermacine, R. Fleurquin, S. Sadou, On-demand quality-oriented assistance in component-based software evolution, in: Proceedings of CBSE'06, Springer LNCS, 2006, pp. 294–309.

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

[7] 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.

[8] C. Tibermacine, S. Sadou, M. T. Ton That, C. Dony, Software architecture constraint reuse-by-composition, In Journal of Future Generation Computer Systems 61 (2016) 37–53.

[9] S. Kallel, B. Tramoni, C. Tibermacine, C. Dony, A. H. Kacem, Automatic translation of architecture constraint specifications into components, in: The 9th European Conference on Software Architecture, Springer, 2015, pp. 322–338.

[10] C. Tibermacine, Software Architecture 2, John Wiley and Sons, New York, USA, 2014, Ch. Architecture Constraints, pp. 37–90.

[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] 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.

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

[14] K.-C. Tai, The tree-to-tree correction problem, Journal of the ACM 26 (3) (1997) 422–433.

[15] J. Favaro, What price reusability?: a case study, in: ACM SIGAda Ada Letters, Vol. 11, ACM, 1991, pp. 115–124.

[16] T. M. Ton That, C. Tibermacine, S. Sadou, Catalogue of architectural patterns characterized by constraint components, Version 1.0, Tech. rep., IRISA, 53 pages (Jul. 2013).

[17] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, Software Engineering, IEEE Transactions on 33 (9) (2007) 577–591.

[18] H. Aloulou, Framework for ambient assistive living: handling dynamism and uncertainty in real time semantic services provisioning, Ph.D. thesis, Evry, Institut national des télécommunications (2013).

[19] T. Tiberghien, M. Mokhtari, H. Aloulou, J. Biswas, Semantic reasoning in context-aware assistive environments to support ageing with dementia, in: International Semantic Web Conference, Springer, 2012, pp. 212–227.

[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 Internationnal Conference on Tools with Artificial Intelligence, 2013, pp. 1044,1051.

[22] J. Cabot, R. Clarisó, D. Riera, 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, ACM, 2007, pp. 547–548.

[23] B. Demuth, The dresden ocl toolkit and its role in information systems development, in: ISD2004, 2004.

[24] J. Cabot, E. Teniente, Transformation techniques for ocl constraints, Science of Computer Programming (2007) 179–195.

[25] I. S. Bajwa, M. G. Lee, Transformation rules for translating business rules to ocl constraints, in: Modelling Foundations and Applications, Springer, 2011, pp. 132–143.

[26] A. Correa, C. Werner, M. Barros, Refactoring to improve the understandability of specifications written in object constraint language, Software, IET 3 (2009) 69–90.

[27] J. Reimann, C. Wilke, B. Demuth, M. Muck, U. Aßmann, Tool supported ocl refactoring catalogue, in: Proceedings of the 12th Workshop on OCL and Textual Modelling, ACM, 2012, pp. 7–12.

[28] L. C. Briand, W. Dzidek, Y. Labiche, Using aspect-oriented programming to instrument ocl contracts in java, Technical Report, Carlton University, Canada.

[29] A. Hamie, Translating the object constraint language into the java modelling language, in: Proceedings of the 2004 ACM symposium on Applied computing, ACM, 2004, pp. 1531–1535.

[30] L. Froihofer, G. Glos, J. Osrael, K. M. Goeschka, Overview and evaluation of constraint validation approaches in java, in: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, 2007, pp. 313–322.

[31] B. Verheecke, R. Van Der Straeten, Specifying and implementing the operational use of constraints in object-oriented applications, in: Proceedings of the Fortieth International Conference on Tools Pacific, 2002, pp. 23–32.

[32] M. Fowler, Refactoring: Improving the design of existing code, in: 11th European Conference. Jyväskylä, Finland, 1997.

[33] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on, IEEE, 2009, pp. 255–258.

[34] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, A domain analysis to specify design defects and generate detection algorithms, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2008, pp. 276–291.

[35] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367.

[36] M. Goldstein, I. Segall, Automatic and continuous software architecture validation, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2, IEEE, 2015, pp. 59–68.

[37] H. C. Lovatt, A. M. Sloane, D. R. Verity, A pattern enforcing compiler (pec) for java: using the compiler, in: Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling-Volume 43, Australian Computer Society, Inc., 2005, pp. 69–78.

[38] A. Blewitt, A. Bundy, I. Stark, Automatic verification of java design patterns, in: Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on, IEEE, 2001, pp. 324–327.