Preserving Architectural Choices throughout the Component-based Software Development Process

Chouki Tibermacine, Régis Fleurquin, Salah Sadou

Preserving Architectural Choices throughout the Component-based Software Development Process

Chouki Tibermacine, Régis Fleurquin and Salah Sadou
VALORIA Lab, University of South Brittany
F-56000 Vannes, France
{Chouki.Tibermacine,Regis.Fleurquin,Salah.Sadou}@univ-ubs.fr

Abstract

It is argued that architecture comprehension and regression testing of a software system are the most expensive maintenance activities. This is mainly due to the fact that architectural choices are either not explicit, at every stage of the software development process, or not preserved from one stage to another. In this paper, we present an Architectural Constraint Language (ACL) as a means to formally describe architectural choices at all the stages. This language is based on the UML's Object Constraint Language and on a set of MOF-compliant metamodels. We also present a prototype which validates the proposed approach. It allows the evaluation of ACL expressions at two stages and ensures, by using a transformation mechanism, that the constraints stated at one stage are subsequently preserved.

1 Introduction

Software architectures need to be well documented, so that maintenance activities could be performed easily and safely. Indeed, among the maintenance activities, understanding the software architecture before its evolution, and regression testing after it has evolved, are by far the most expensive. By making explicit and formal all architectural choices, we facilitate on the one hand, the program comprehension and on the other, the automatic regression testing. These architectural choices are often expressed using a constraint language. Many existing Architecture Description Languages (ADLs) provide such languages [8]. However, in the following stages of the software development process, there is often no mean neither to express the new choices nor to preserve those expressed previously. For instance, in component-based software, none of the existing implementation models, e.g. SUN's Enterprise JavaBeans (EJB) [17], OMG's CORBA Component Model (CCM) [11] or Microsoft Component Object Model (COM+) [9], provide such features. The architectural choices are thus often embedded into code and then become implicit at this level. Consequently, if an evolution is made on this level, the architectural choices may be affected (see section 2 for a detailed example).

To resolve this problem, architectural choices made at each stage of the software development process should, not only be explicit but they should be preserved in the following stages as well. In this paper, we present an Architecture Constraint Language (ACL). This predicate language serves to formally document component-based software architectures throughout the different stages of the development process (see section 3). ACL is based on the UML's Object Constraint Language (OCL), associated to several MOF-compliant metamodels. Each metamodel meets the needs of one stage of the software development. The software designer can thus continue to use concepts from his favourite (usual) ADL or component technology. To preserve the architectural choices, from one stage to the others, we use a generic metamodel (ArchMM). This pivot metamodel is the basis of a constraint transformation mechanism. ACL expressions are interpreted by an Architectural Constraint Evaluator (ACE). This tool, presented in section 4, makes possible to evaluate the compliance of the implementation models with the constraints stated during the component design stage.

2 Illustrative Example

Let us consider a simple example of an Access Control System (*ACS*). The different parts of figure 1 provide an overview of its architecture. This architecture is mainly organized as a *pipeline* [15]. The system receives as input the necessary data for user authentication (*Authenticator* component). After identification, the data is sent to the *Access-Controller* component. The latter checks whether the user is authorized to enter to the controlled area or not. If the access

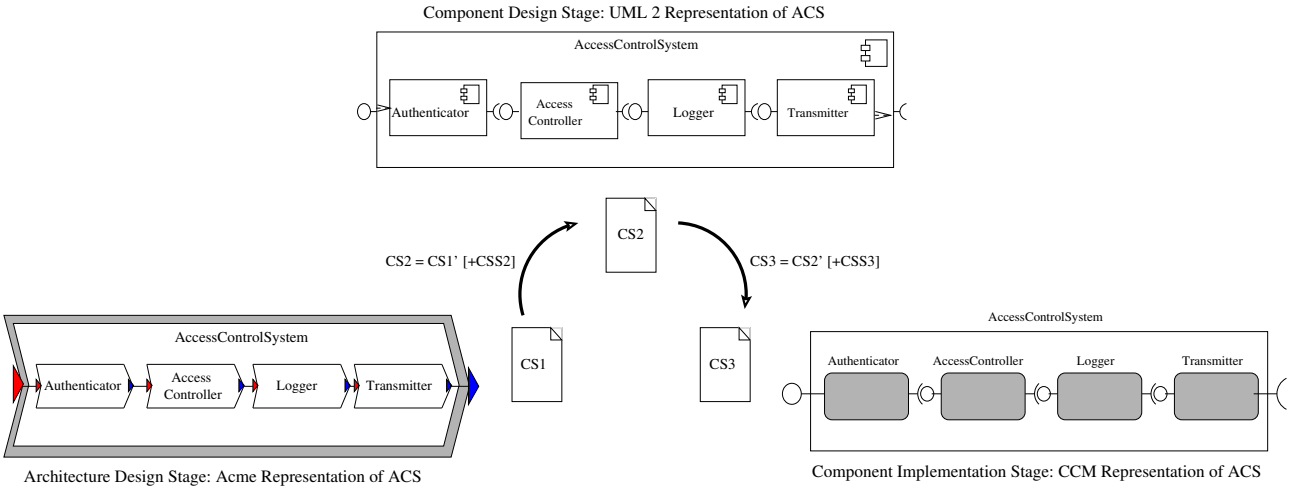

Figure 1. Constraint preservation in a component-based software development process

is authorized, the *Logger* component adds to the data the entrance date and hour, and stores it locally for the controlled area. Then, it sends these logs to the *Transmitter* component, which adds information about the controlled area and transmits them for global archival storage.

To illustrate our purpose, we briefly describe the development process of *ACS*, from architectural design to implementation. Then, we describe the raised problems.

2.1 Architectural Design Stage

The architecture designer of this system chooses the pipeline style aiming at a high level of maintainability of the software. Suppose that the architecture of this system was described using xAcme [19]. Using graph theory, we enumerate the structural constraints that guarantee this architectural style. These constraints are defined using the *Armani* predicate language [10] as follows[1]:

1. The first constraint implies the definition of vertices and arcs, and that each arc must be connected to two vertices:

   (a) A vertex stands for a component with input or output ports:

   ```
   invariant forall comp: Component
   in self.Components |
     forall p: Port in comp.Ports |
       satisfiesType(p,inputT)
       or satisfiesType(p,outputT)
   ```

   (b) An arc represents a connector with exactly two roles (one sink and one source):

   ```
   invariant forall conn: Connector
   in self.Connectors |
     size(conn.Roles) == 2
     and exists r: Role in conn.Roles |
     satisfiesType(r,sinkT)
     and exists r: Role in conn.Roles |
     satisfiesType(r,sourceT)
   ```

   (c) Each connector (arc) binds two components (vertices). the input port to a sink role and the output port to a source role:

   ```
   invariant forall conn: Connector
   in self.Connectors |
     forall r: Role in conn.Roles |
      exists comp: Component
      in self.Components |
        exists p: Port in comp.Ports |
          attached(p,r)
          and ((satisfiesType(p,inputT)
          and satisfiesType(r,sinkT))
          or (satisfiesType(p,outputT)
          and satisfiesType(r,sourceT)))
   ```

2. The second constraint implies two sub-constraints:

   (a) The graph should be connected:

   ```
   invariant forall c1, c2: Component
   in self.Components |
     c1 != c2 -> reachable(c1,c2)
   ```

   (b) And, it contains n-1 arcs (connectors), n being the number of vertices (components):

   ```
   invariant size(self.Connectors)
   == size(self.Components)-1
   ```

3. The last constraint stipulates that the tree must be a list. It may be expressed as following:

   ```
   invariant forall comp: Component
   in self.Components |
     size(comp.Ports) == 2
   ```

[1] Concretely in xAcme constraint descriptions are decomposed in many XML elements. For the sake of brevity, we present the constraints as they are described traditionally in the Armani Language.

```
and exists p: Port in comp.Ports |
satisfiesType(p,inputT)
and exists p: Port in comp.Ports |
satisfiesType(p,outputT)
```

The constraints, described above, guarantee the structural compliance with the pipeline style at this stage. During the following stages, we should also be able to check whether this style is still respected.

2.2 Component Design Stage

Before implementing ACS, using CORBA components, we decided to establish an intermediate UML model for a smooth transition[2]. In this case, the constraint specification *CS1* should be transformed into *CS1'* (see figure 1). Additional constraints specific to this UML diagram may be added (*CSS2*, in figure 1). For example, to meet a maintainability requirement, we may specify in *CSS2* that the *ACS* component should provide no more than two interfaces. At this stage, the overall constraint specification (noted *CS2* in the figure) consists of *CS1'* union *CSS2*. Unfortunately, neither the *CSS2* constraint nor those expressed at the previous stage (*CS1'*) can be specified and checked using the UML language. This is due to the fact that its predicate language (OCL) cannot express this type of constraints, which imply restrictions on UML meta-classes, such as *Component* or *Interface*. Consequently, there is no means: i) to guarantee that *CS1'* is respected and ii) to document *CSS2*, in order to be explicit and checkable.

2.3 Component Implementation Stage

At this stage, we should also be able to transform *CS2* into *CS2'* and possibly add other constraints (*CSS3*). *CS2'* represents here the mapping of the *CS2* constraints into the CORBA Component Model. The overall constraint specification (noted *CS3* in the figure) consists of *CS2'* union *CSS3*. As in the previous stage, there is no constraint language in this technology which allows the definition of *CSS3*. In addition, there is no means to transform *CS2* into *CS2'*.

2.4 An Evolution Scenario

Let us suppose the following evolution scenario:

To gain performance in the application processing, we decide to send directly some data, which are not useful for local logging, to the central server.

One solution to meet the requirement above is to add a direct connection between the components *AccessController* and *Transmitter*. Unfortunately, this change breaks the pipeline structure of the system architecture and consequently weakens its level of maintainability. If this change is made on the architecture design, the system maintainer is notified of the consequences, because the pipeline style was formally described (*CS1* specification) and thus checkable. However, if it is made on the UML diagrams, or on implementation models, which is often the case, there is no mean to notify the system maintainer.

The next section introduces ACL, a constraint language which allows the specification of structural constraints at each stage of a component-based software development process (*CS1*, *CSS2* and *CSS3* in the previous example). In addition to the specification of architectural constraints, ACL associated to its interpreter (ACE) allows a transformation process of constraints from one stage to another (in the example above, transform *CS1* into *CS1'* and *CS2* into *CS2'*). This implies the persistence of architectural choices.

3 Architectural Constraint Language

Because many aspects of a software might be of interest, we can use various modeling languages to highlight one or more particular perspectives or views of that system depending on what is relevant at every point of the development process. A language is used during such or such stage of the process for its effectiveness at solving the problems arising at this particular stage. Each language uses concepts defined in its metamodel. Because two languages can be based on different concepts, it is frequent to have to manage many concepts during the software development process.

In order to propose a constraint language usable at all the stages of the software development process and to support this diversity, we chose a two-level solution. The first level is based on a constraint language, easy to grasp and standardized by the OMG, namely OCL [13]. The second level takes the form of a set of metamodels. Each metamodel enables us to adapt ACL, without making changes on its grammatical part, to express predicates on models written in a specific language. For instance, we propose a metamodel for xAcme architecture descriptions, UML 2 component models or CCM component descriptions. To specify a constraint on an xAcme model, we write an OCL statement in the context of the proposed xAcme metamodel (see figure 2). This two-level structure guarantees that at each stage of the software development process, the designer uses the same notation (OCL), but applied to the concepts that he is accustomed to handle usually at this stage.

Transforming an architectural constraint from one model to another in order to check it, is often difficult because there is not a one-to-one concept mapping between two meta-

[2]Recent experiments [14] showed also that some ADLs and the UML can be used in a complementary fashion, in order to make better analysis of software architectures.

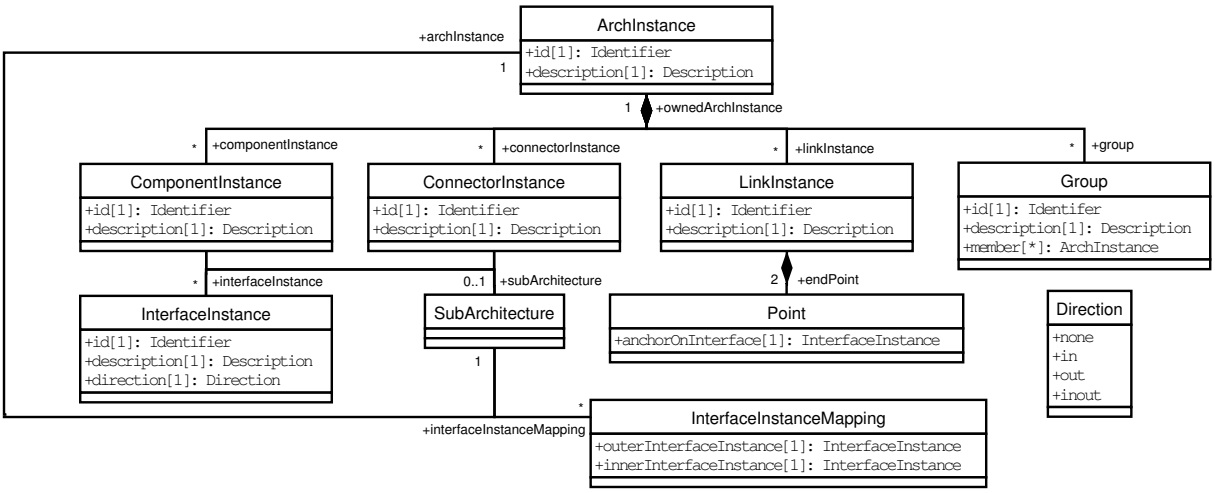

Figure 2. The xArch metamodel

models. For example, xAcme connectors or component hierarchical descriptions have not direct equivalents in CCM. To solve this problem we proposed a two-step transformation process using an intermediate representation. We introduced a generic metamodel, called ArchMM. Expressions written on a given metamodel are systematically translated internally into their equivalents defined on ArchMM. The latter plays the role of a pivot between the different metamodels.

In the sequel, we will describe the OCL part of ACL, the different metamodels that enable us to cover the development process, how ArchMM was designed and finally how it is used to translate constraints written from one metamodel to another.

3.1 Constraint Language

OCL is originally used to add more semantics to UML semi-formal diagrams. It has been standardized by the OMG as version 1.5 and is currently available as the UML 2 final adopted specification [12]. In addition, Briand et al. [2] has recently experimented the OCL language in the maintenance of UML models. They concluded that it improves significantly the comprehension and the maintainability of these models.

Usually OCL is used to express constraints on a UML model, whereas, we use it to express constraints on a metamodel. Thus, constraints such as "*components in the model must have less than 10 required interfaces*" can be expressed. Unfortunately they have a global scope. They apply to all components and not to a particular one as we would like it to be. To limit the scope of such constraints to a particular component, we propose to slightly modify the syntax and semantics of the context part in OCL. At the syntactic level, every constraint context should introduce an identifier. This identifier must be the name of a particular instance of the meta-class cited in the context. At the semantic level, we interpret a constraint with the meaning it would have in the context of the metaclass but limiting its scope only to the instance cited in the context.

According to this principle, the following constraint, applied to the UML metamodel (see figure 3), states that the required interface *ArchivalStorage*, of the primitive component of name *Logger* (introduced in the example of section 2), must be bound to one and only one component.

```
context Logger:Component inv:
Logger.ownedPort->select(p:Port|
p.required.name = 'ArchivalStorage')
.end->size() = 1
```

Thus, with a tiny modification of OCL syntax and semantics, we succeed in describing constraints that present an introspection mechanism, by using a 'bicephalous' linguistic structure OCL/Metamodel. A component is able to define a constraint on its own structure. Note that only invariants can be expressed in ACL. Pre- and post-conditions are dismissed.

3.2 Architecture Metamodels

In order to cover the development process, we propose different metamodels.

3.2.1 Architectural Abstractions in the ADLs.

In the literature, there exists a plethora of domain-specific and general purpose ADLs. We studied many languages and we established a MOF metamodel for each one. For the sake of brevity, we focus only on those of general purpose,

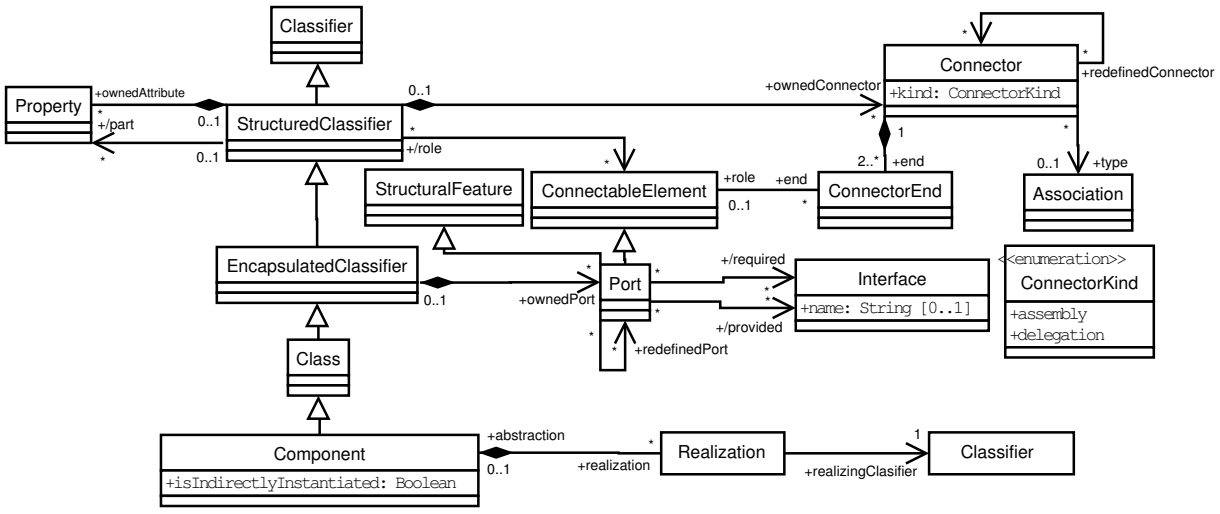

Figure 3. A flattened excerpt of the UML 2 component metamodel

like xAcme and xADL 2.0 [5]. xAcme and xADL 2.0 share the same core representation, namely xArch. xArch is a common representation for architecture description which can be extended to provide XML notation for other existing ADLs. It contains the primitive elements that compose an architecture description from a structural perspective.

In figure 2, we present a metamodel illustrating xArch. An xArch architecture instance is composed of a set of component instances, connector instances, link instances and logical groups of the previous architectural elements. Component or connector instances define a set of interface instances and optionally a subarchitecture for a hierarchical description. The subarchitecture defines a set of architecture instances and a list of mappings between inner and outer interface instances. Link instances bind two end points, each one references an interface instance.

xAcme adds to these elements, properties which can be associated to component instances, connector instances and interface instances. It distinguishes also type definitions of the architectural elements from instance descriptions. In addition, it introduces constraints to the different elements.

xADL 2.0 extends the xArch metamodel with some configuration management concepts, like guards for architectural constraints, optional architectural elements, variants of architectural elements which can be used as alternatives, versions of architecture descriptions, and differences in descriptions.

3.2.2 Architectural Abstractions in the UML 2 Metamodel.

Unlike in the version 1.5 of the UML specification, in the final adopted specification of the UML 2 a component is considered as a first-class modelling element. It inherits from the metaclass *Class*, thus it can participate in associations and in generalizations. It also specializes *EncapsulatedClassifier* and *StructuredClassifier*. So, it can declare ports that group provided and required interfaces, and can be assembled with other components through assembly and delegation (hierarchical) connectors.

3.2.3 Architectural Abstractions in the Component Technologies.

Among existing component technologies, we studied the Enterprise JavaBeans, Microsoft COM+ and CORBA Component Model (CCM). We present only the latter as it is the OMG standard, but also because it aims at synthesising the concepts found in the other technologies. Through the figure 4 we illustrate these concepts.

A CORBA component defines a set of ports. Each one declares either, emitted, published or consumed events, or used or provided interfaces. A component can be derived from existing components, declare many supported interfaces, may declare a set of attributes and should define one home interface. An interface is characterized by a set of operations. A component assembly defines a set of connections. Each one links a consumed port to a published or emitted port, a provided to a used port or two home interfaces together.

3.2.4 ArchMM: A Generic Architecture Metamodel

This metamodel represents a support for all architectural elements that can be constrained. Because constraints should be expressed throughout the software life cycle, the metamodel should contain abstractions present at the same time in analysis/design models (architecture descriptions or

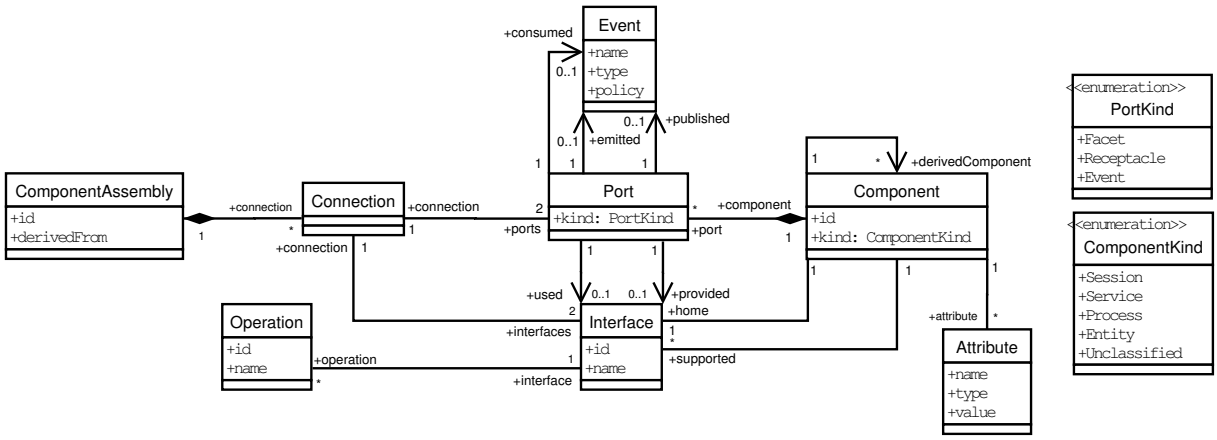

Figure 4. The CORBA component structural metamodel

UML component diagrams) and in component-based implementation models.

Starting from the previous metamodels, we designed a generic MOF-compliant metamodel *ArchMM* which is illustrated in figure 5. It supports the structural description of components, which correspond in the previous metamodels to component instances in xArch metamodel and components in CCM. These are described as a set of ports. These ports define a set of interfaces of different kinds: required (*used interfaces* in CCM and *in interfaces* in xArch), provided (*provided* in CCM and *out interfaces* in xArch), etc. Events in CCM and in many other ADLs are considered in ArchMM as interfaces. So, ArchMM interfaces can also be of various kinds: emitted, published or consumed like in CCM. Interfaces define a set of services, which correspond to CCM operations.

ArchMM supports also connectors as first class entities. They are described as a set of roles, which correspond to connector interfaces in xArch. The metamodel abstracts hierarchical models of components and connectors. Composite components or connectors have configurations, which correspond to subarchitectures in xArch metamodel. Each configuration defines a set of bindings. A binding, corresponding to a connection in CCM and to a link instance or an interface instance mapping in xArch, links: i) two component interfaces together, ii) a component interface to a connector role, iii) the port of a composite component to the interface of one of its subcomponents, iv) or, two connector roles together.

Properties are associated to components, to connectors and to interfaces. They are named and have a typed value. They correspond to CCM component attributes or xAcme properties.

Note that CCM is a flat component model. That is, no hierarchical descriptions of components or connectors are possible. In order to perform transformations of hierarchical descriptions into CCM, component assemblies are used. These allow us to define flattened hierarchical descriptions.

Some abstractions exist in all the metamodels, such as component and required or provided interface. However, some concepts present in ArchMM do not exist in some other metamodels. These concepts, such as port, binding, connector or role, may be inferred during the transformation to ArchMM. For example, in CCM there are no connectors. When transforming CCM descriptions into ArchMM, connectors are generated with two roles. Depending on the port kind of the connected components *Facet*, *Receptacle* and *Event*, the generated roles are, respectively, *Callee*, *Caller* and *Listener* or *Trigger*. The distinction between events of kind *Published* and *Emitted* of the CCM metamodel is made at *Interface* level in ArchMM.

3.3 Examples of Constraints

In this section, we present some examples of architectural constraints using ACL applied to ArchMM. We start with a simple example which states that no more than two provided interfaces can be described by a component. Despite its simplicity, this constraint is not of less importance. Such constraints can be found in the quality manuals of certain software development organizations. This kind of constraints can be described on all the subcomponents[3] of the component *ACS* as following:

```
context ACS:CompositeComponent inv:
ACS.subComponent->forAll(C:Component|C.port
.interface->select(i:Interface|
i.kind = 'Provided')->size() <= 2)
```

Another type of constraints can be expressed with ACL. In the constraint below, we describe a rule using a quality measure for components, which are cited modules by its

[3]For the moment, we suppose that only direct subcomponents can be accessible from a composite component.

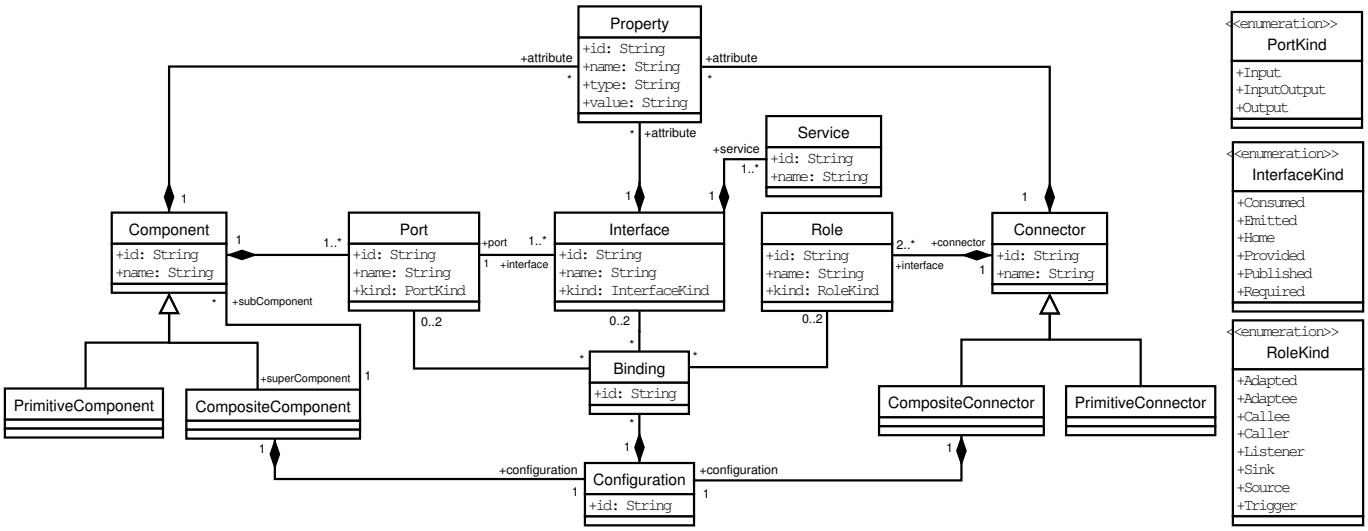

Figure 5. ArchMM: A generic metamodel representing architectural abstractions

authors [6]. This quality measure is CBM, which stands for Coupling Between Modules.

```
context ACS:CompositeComponent inv:
ACS.port.interface.binding->size() < 5
```

This metric is the equivalent of CBO (Coupling Between Objects) [4] for object-oriented applications. It represents the number of non-directional, distinct inter-component references. In order to preserve a certain level of maintainability of the *ACS* component and to limit its dependencies with the other components, we described a constraint stating that CBM should be low (CBM < 5) [6].

The third example introduces a general architectural constraint which prevents the existence of a connector binding ACS subcomponents to an external component.

```
context ACS:CompositeComponent inv:
ACS.configuration.binding
->forAll(b:Binding|ACS.subComponent.port.interface
->includes(b.interface))
```

The following example addresses a typical evolution constraint involving two consecutive versions. This constraint states that only one provided interface can be added to a component between two consecutive versions [18] of the architecture. It can be expressed as follows:

```
context ACS:CompositeComponent inv:
ACS.port.interface
->select(i:Interface|i.kind = 'Provided')@old
->comparedTo(ACS.port.interface
->select(i:Interface|i.kind = 'Provided'))
->size() <= 1
```

Note that we introduced a new syntactic construct to OCL so that we can reference old versions of architecture descriptions: the @*old* mark. Unlike @*pre* which is used, traditionally in OCL, only in postconditions, the @*old* can be used in invariants. The @*old* returns the descriptions of the former version of an architecture. In this example, it returns a collection of the provided interface descriptions of the *ACS* component version before evolution. Note that we have also added a collection operation: *comparedTo(c2:Collection(T)):Collection(T)*. It returns a collection which represents the difference between the collection to which it is applied and the parameter collection (c2). We introduced this new operation in order to not affect OCL semantics of the set difference operator (-). Indeed, traditionally in OCL, the set difference operator compares references of object collections, whereas *comparedTo(...)* uses the value of the *id* attribute. In our situation, we are confronted to a comparison of collections of architectural descriptions. The elements of these collections are objects which have different references from one version to another, but the same value of the *id* attribute.

As stated in section 2, ACS subcomponents are organized as a pipeline. This pattern can be structurally described using ACL as following:

```
--Each subcomponent should have
--input and output ports
context ACS:CompositeComponent inv:
ACS.subComponent.port
->forAll(p:Port|(p.kind = 'Input')
or (p.kind = 'Output'))
and
--Each connector should define two roles
--(one sink and one source)
ACS.configuration.binding.role.connector->AsSet()
->forAll(con:Connector|(con.role->size() = 2)
and ((con.role.kind = 'Source')
or (con.role.kind = 'Sink')))
and
--Each connector binds two components
--(input bound to sink and output to source)
```

```
ACS.configuration.binding.role.connector->AsSet()
->forAll(con:Connector|con.role
->forAll(r:Role|ACS.subComponent
->exists(com:Component|com.port
->exists(p:Port|(r in ACS.configuration.binding)
and ((p.kind = 'Input') and (r.kind = 'Sink'))
or ((p.kind = 'Output') and (r.kind = 'Source'))
))))
and
--The graph should be connected
ACS.configuration.isConnected
and
--The graph should contain the number
--of vertices - 1, arcs
ACS.configuration.binding.role.connector
->AsSet()->size() = ACS.subComponent->size()-1
and
--The graph represents a list
ACS.subComponent->forAll(com:Component|
(com.port->size() = 2)
and (com.port->exists(p:Port|p.kind = 'Input'))
and (com.port->exists(p:Port|p.kind = 'Output')))
```

The *Configuration* meta-class of ArchMM contains some attributes and operations that allow the expression of constraints involving certain graph properties, such as connected graphs (in the constraints above), regular graphs, simple graphs, etc. In addition, we are able to query, for example, for the predecessors and the successors of a vertex (component), the in- and out-degrees of a vertex, the distance between two vertexes or the graph diameter.

3.4 Use Scenario of ACL

As stated previously, the constraint language that we proposed (ACL) is not exclusively associated to ArchMM. In fact, it is associated to many metamodels. Each association ACL/metamodel constitutes an *ACL profile*. So, there are as many profiles as ADLs and component technologies. Each developer uses the profile that corresponds to the language or the technology that he usually uses. In this way, he is not bound to manipulate some concepts, present in ArchMM, and that he is not familiar with, like for example, connectors or composite components for a CCM developer.

For the existing ADLs, having a constraint language, for example xAcme's Armani, ACL can be used as a complementary predicate language. It can describe, in addition to general structural constraints, constraints of pure evolution, stated previously, or constraints using graph operations. However, for the existing specifications written in xAcme, the architect is requested to translate manually the constraints written in Armani into ACL profile for xAcme. Consider the example presented in section 2. The pipeline structural constraints defined in Armani can be expressed in ACL profile for xAcme as follows:

```
context ACS:ComponentInstance inv:
ACS.subArchitecture.archInstance
.componentInstance.interfaceInstance
->forAll(i:InterfaceInstance|(i.direction = 'in')
or (i.direction = 'out'))
and
ACS.subArchitecture.archInstance
.connectorInstance->forAll(con:ConnectorInstance|
(con.interfaceInstance->size() == 2)
and ((con.interfaceInstance.direction = 'in')
or (con.interfaceInstance.direction = 'out')))
and
ACS.subArchitecture.archInstance
.connectorInstance->forAll(con:ConnectorInstance|
con.interfaceInstance
->forAll(iCon:InterfaceInstance|
ACS.subArchitecture.archInstance
.componentInstance->exists(com:ComponentInstance|
com.interfaceInstance
->exists(iCom:InterfaceInstance|(iCon in ACS
.subArchitecture.archInstance.linkInstance
.point.anchorOnInterface)
and ((iCom.direction = 'in')
and (iCon.direction = 'out'))
or ((iCom.direction = 'out')
and (iCon.direction = 'in'))))))
and
ACS.subArchitecture.isConnected()
and
ACS.subArchitecture.archInstance.connectorInstance
->size() = ACS.subArchitecture.archInstance
.componentInstance->size() - 1
and
ACS.subArchitecture.archInstance.componentInstance
->forAll(comp:ComponentInstance|
(comp.interfaceInstance->size() = 2)
and (comp.interfaceInstance
->exists(i:InterfaceInstance|i.direction = 'in'))
and (comp.interfaceInstance
->exists(i:InterfaceInstance|
i.direction = 'out')))
```

In one of the invariants, we make use of the operation *isConnected()*. This attribute is associated, in this ACL profile, to the type *SubArchitecture*. It has the same semantics as for the type *Configuration* in ACL profile for ArchMM. All graph operations are associated to the *SubArchitecture* type.

As we remark through this example, the syntaxes of Armani and of ACL profile for xAcme are quite similar. We believe that the manual translation could be performed easily. The advantage of translating constraints into the ACL profile is to have a homogeneous constraint specification throughout the software development process. In addition, transformations can be done automatically to obtain constraints that apply to implementation models, as shown below. However, the expression of constraints dealing with the behavioral aspect is not yet taken into account in ACL.

The constraints are evaluated each time new changes are made on the architecture or the component description. Upon their evaluation, these constraints are first transformed into constraints expressed in the standard ACL pro-

file for ArchMM. The architecture or the component description is also transformed into a pivot description, which is ArchMM-compliant. This transformation to the standard ACL profile is also used to make transformations from one profile to another (different from the standard one). For example, the pipeline constraints are transformed into the following when implementing the system in CCM.

```
context ACS:ComponentAssembly inv:
ACS.connection.port->forAll(p:Port|
(p.provided->size() = 1)
or (p.used->size() = 1))
and
ACS.connection->forAll(con:Connection|
(con.port.provided->size() = 1)
and (con.port.used->size() = 1))
and
ACS.isConnected()
and
ACS.connection->size()
= ACS.connection.port.component
->AsSet()->size() - 1
and
ACS.connection.port.component->asSet()
->forAll(com:Component|
com.port->size() = 2
and com.port->exists(p:Port|p.provided
->size() = 1)
and com.port->exists(p:Port|p.used->size() = 1)
```

Note that the invariant that checks for the existence of only two roles per connector has not been maintained in this constraint specification, because the connector abstraction does not exist in CCM metamodel. Note also that in this ACL profile, graph operations are associated to the *ComponentAssembly* type. This is the case for the operation *isConnected()* in one of the invariants.

As in the previous case, this constraint is first transformed so a constraint which conforms to ArchMM before evaluation.

In the example presented in section 2, we established a UML model before implementing the system in CCM. Note that the transformation from the ACL profile for xAcme to the one for UML is done in the same manner as if we directly implemented the system in CCM.

4 Prototype Tool

In order to validate our approach, we developed *ACE*: Architectural Constraint Evaluator. This prototype allows the edition and the evaluation of architectural constraints specified in different stages. In its current version, it interprets ACL profile for xAcme at the architecture design stage and ACL profile for the Fractal Component Model [3] at the implementation stage.

ACE uses the XMI format of the metamodel to guide the developer in editing his architectural constraints. Constraints are transformed to be compliant with ArchMM, in order to be evaluated. This transformation is performed on the basis of a list of mappings between xAcme or Fractal metamodel and ArchMM. The generated constraints are evaluated on an intermediate model, which is obtained by a transformation of xAcme or Fractal descriptions. This pivot model is a specific instance of ArchMM which is defined as a set of XML schemas. The module performing constraint evaluation has been built upon a slightly modified version of OCL Compiler [4].

Suppose a constraint defined at design stage using the ACL profile for xAcme. In order to evaluate this constraint at implementation stage (in this case, on Fractal descriptions), the following steps will be performed: i) transform the constraint specified using ACL profile for xAcme into the standard profile; ii) transform the Fractal architecture description into the pivot model; iii) and evaluate the transformed constraint on the pivot model.

During a maintenance process, a new version of an architecture or a component description can be introduced to ACE. The constraints are evaluated for the new description, which is then transformed into the pivot model.

ACE was designed in order to be easily extended to support other ADLs or component technologies, like those presented in this paper (CCM or UML 2). For each new added ADL or component technology, a new metamodel should be established, which would generate a new ACL profile. Only the model and constraint transformers should be enhanced.

5 Related Work

Some existing ADLs provide constraint languages [8]. In WRIGHT [1], the constraint language used is a subset of the Z specification language [16]. In xAcme, architectural constraints are expressed using the Armani predicate language. All these languages have in common the same basis as with OCL. They are based on FOL (First Order Logic), manipulate collections of architectural elements and provide collection operations. In these languages, the architectural elements, to be constrained, are part of the language and are defined as keywords. The approach that we proposed disconnects these elements through the generic architecture metamodel, ArchMM. Thus, it becomes possible to cover all the development process with the same constraint language. ACL allows also the specification of pure evolution constraints, whereas the ADLs focus only on design constraints.

In [7], Medvidovic et al. propose three approaches to support modeling software architectures in UML 1.5. In the second approach, the authors suggest to extend UML through stereotypes, tagged values and OCL constraints in order to define architectural styles. In this work, the con-

[4] Available at http://dresden-ocl.sourceforge.net/

straints are considered as general rules that apply to all modelled architectural elements. However, in our approach the constraints are considered as specific rules that apply only to a component of the current model. They stipulate particular choices of the developer. We think that the two approaches are complementary. The first provides capabilities to define common rules for several projects, while the second can be used to specify rules for a specific project.

6 Conclusion and Future Work

In this paper, we presented a constraint language which allows to make explicit architectural choices at every stage of the component-based development process. This language was designed in such a manner to be extended easily in order to support other design or implementation models. The originality of this language is the separation between the basic constraint expression mode and the manipulated architectural concepts.

Throughout development process, various concepts are handled. The separation that we propose, enables us to use the same syntax to express constraints at different stages of the development process. Moreover, it is easier to identify the mappings between concepts from different metamodels, when they are not embedded into the language. Thus, it becomes easier to define constraint transformation mechanisms. We think that our approach may be helpful in the context of Model Driven Engineering.

The next step of our work will be the extension of our approach on behavioral aspect of architecture descriptions. Thus, we project to enhance ACL to provide such capabilities, by extending ArchMM with behavioral representations.

Acknowledgment

This material is based upon work supported by the Brittany Region Council under contract number 20046839.

References

[1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997.

[2] L. C. Briand, Y. Labiche, H. D. Yan, and M. Di Penta. A controlled experiment on the impact of the object constraint language in UML-based maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'04)*, pages 380–389, Chicago, Illinois, USA, 2004.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quma, and J. B. Stefani. An open component model and its support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering. Held in conjunction with ICSE'04*, Edinburgh, Scotland, may 2004.

[4] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[5] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions On Software Engineering and Methodology*, 14(2):199–245, 2005.

[6] M. Lindvall, R. Tesoriero, and P. Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS'02)*, pages 77–86, Ottawa, Ontario, Canada, June 2002.

[7] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions On Software Engineering and Methodology*, 11(1):2–57, 2002.

[8] N. Medvidovic and N. R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[9] Microsoft. COM: Component object model technologies. http://www.microsoft.com/com/, 2005.

[10] R. T. Monroe. Capturing software architecture design expertise with Armani. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2001.

[11] OMG. Corba components, v3.0, adpoted specification, document formal/2002-06-65. Object Management Group Web Site: http://www.omg.org/docs/formal/02-06-65.pdf, June 2002.

[12] OMG. Meta Object Facility (MOF) 2.0 final adopted specification, document ptc/03-10-04. Object Management Group Web Site: http://www.omg.org/docs/ptc/03-10-04.pdf, 2003.

[13] OMG. UML 2.0 OCL final adopted specification, document ptc/03-10-14. Object Management Group Web Site: http://www.omg.org/docs/ptc/03-10-14.pdf, 2003.

[14] R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. Understanding tradeoffs among different architectural modeling approaches. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 47–56, June 2004.

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

[16] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[17] Sun-Microsystems. Enterprise JavaBeans(TM) specification, version 2.1. http://java.sun.com/products/ejb, November 2003.

[18] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[19] xAcme: Acme Extensions to xArch. School of Computer Science Web Site: http://www-2.cs.cmu.edu/ acme/pub/x-Acme/, 2001.