HAL
archives-ouvertes.fr

A family of languages for architecture constraint specification

Chouki Tibermacine, Régis Fleurquin, Salah Sadou

▶ **To cite this version:**

Chouki Tibermacine, Régis Fleurquin, Salah Sadou. A family of languages for architecture constraint specification. Journal of Systems and Software, Elsevier, 2010, 83 (5), pp.815-831. <hal-00498761>

HAL Id: hal-00498761
https://hal.archives-ouvertes.fr/hal-00498761

Submitted on 8 Jul 2010

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.

A Family of Languages for Architecture Constraint Specification

Chouki Tibermacine [a] Régis Fleurquin [b,c] and Salah Sadou [c]

[a] *LIRMM, CNRS and Montpellier-II University, France*
Chouki.Tibermacine@lirmm.fr

[b] *IRISA, INRIA Rennes, France*
Fleurquin@irisa.fr

[c] *VALORIA, University of South Brittany, France*
Salah.Sadou@univ-ubs.fr

Abstract

During software development, architecture decisions should be documented so that quality attributes guaranteed by these decisions and required in the software specification could be persisted. An important part of these architectural decisions is often formalized using constraint languages which differ from one stage to another in the development process. In this paper, we present a family of architectural constraint languages, called ACL. Each member of this family, called a profile, can be used to formalize architectural decisions at a given stage of the development process. An ACL profile is composed of a core constraint language, which is shared with the other profiles, and a MOF architecture metamodel. In addition to this family of languages, this paper introduces a transformation-based interpretation method of profiles and its associated tool.

Key words: Architecture Constraint, Constraint Language, ADL, Software Component, MOF, OCL, Constraint Transformation

1 Introduction

Software architectures deal, at a relatively coarse granularity, with the decomposition of a system into its major components, the mechanisms and rules by which these components interact and the global properties of the system that emerge from the composition of its pieces. Documenting software architectures produces major benefit such as early analysis, system visibility and complexity management, design discipline and global conceptual integrity management. It received in the last two decades a lot of attention by the software engineering

community. With the ever more widespread use of components, these high-level models have now a central role in all the stages of most software development processes. This led to the emergence of development processes driven by architectures supported by numerous languages, tools and methods (20; 44). Nowadays, at analysis/design time, one can describe software architectures using an Architecture Description Languages (ADLs). After that, she/he can produce straightforwardly component implementations (46) using an existing industrial technology, like OMG's CORBA Component Model (CCM (37)), Sun's Enterprise JavaBeans (EJB (45)) or Microsoft's COM+ (29). For a smooth transition, she/he can first shift to a UML 2 component model (35) and then to one of its profiles for the technologies previously cited (UML profile for EJB (40) or UML profile for CCM (39)).

Any time we need to evolve a given architecture model, and before applying changes, we must gain a deep knowledge of the rationale behind the design decisions upon which the model was created. This can help, for instance, in understanding the reasons of the choice of a particular architectural style (43) or design pattern (12). Usually, this knowledge takes the form of a set of what we call architecture decisions. An architectural decision identifies, at least, some of the key structural elements in the system and their externally visible related properties (their rationale). But, properly documenting and using decisions may involve many other fields (53; 16; 19). Building this necessary knowledge is a big time-consuming task because most models do not capture it. In fact, at certain stages of the development process there is no mean allowing to document that knowledge. Even if it exists, often developers have to deal with a variety of languages that make the task complex and cumbersome. Under cost and schedule constraints, they generally choose to not use them. Without any documentation of architectural decisions any evolution of the system becomes risky. Indeed, the evolution maybe in contradiction with some previously taken decisions, which lead the system to loose some of its quality attributes (maintainability, portability, performance, etc.). When a problem appears, a rework is needed. It is a sequence of iterations, that undoubtedly increases the costs.

There are some approaches that help the practitioners to automatically gather some parts of design knowledge (using for instance pattern detection) and present them in a manner that highlights relevant information (18). But these approaches cannot find all important knowledge behind the design. Even if we identify a particular pattern, it is not obvious to know its rationale. So, an explicit documentation remain the best way in order to preserve the link between the choices made by the designer and their rationale. For this aim, we propose an approach which encourages the documentation of the taken decisions directly in the models and throughout the development process. This requires to provide developers with a complete set of dedicated languages covering all the software process. These languages must be both simple to

learn and powerful enough to automate some tasks of a high value added (such as, the detection of the violation of some properties during evolution, the detection of the lost of consistency between the documentation and the architecture designs/code, the decreasing of the costs of regression testing, etc.). This is vital for ensuring a sufficient return on investment to motivate developers to adopt them.

In this paper, we present a family of languages designed to help for an important and time consuming part of the decision documentation process: the description of the structural elements of a model related to a given decision. We call this part of a decision an architectural choice (49). This includes the description of coarse-grained structures (such as styles or patterns) or more fine-grained ones (such as design rules). In this paper, we do not deal with the other parts of an architectural decision such as the documentation of the context and assumptions. In fact, there is no standard recommending what to document and especially how to document (15). Our approach can be used, as it is or with extensions, in order to document any specific architectural decisions. In (50; 52), we provided an example of the use of this family of languages to document certain kind of architecture decisions for the purpose of software evolution assistance. In these languages, an architectural choice is expressed defining an invariant condition that must hold for the system being modeled over elements described in a model. In these languages, an architectural choice is expressed using invariant conditions on elements describing the model. The formal aspect of theses languages makes it possible to provide a tool which automatically analyze and/or evaluate the invariants. It can be used throughout the development process to check if the architectural decisions remain preserved.

In order to reduce their learning cost, all these languages use the same core constraint-level description language (CCL) based on the well-known UML Object Constraint Language (OCL). The choice of OCL is based on the fact that traditional formal languages are useable by persons with a strong mathematical background, but difficult to use by average business or system modelers. OCL has been developed to fill this gap. OCL is a formal language, which remains easy to learn, read and write, so is CCL (our constraint-) language. The variable part of each language is encapsulated in a MOF (Meta-Object Facility (38)) model. These (meta-)models encapsulate the concepts issued from the modeling domain of the concerned process stage. Thus, an expert of the domain uses the same concepts when she/he specifies architectural choices in the same way as when she/he defines architecture/component descriptions.

The outline of the remaining of the paper is as follows. In the next section we illustrate the problem that is dealt with in this paper in more depth by using an example. In section 3, we present ACL, the family of languages introduced above to solve this problem. Section 4 details a method for the evaluation

of constraints described by ACL. The dedicated tool using this method is presented in section 5. It allows the preservation of architectural choices during a model evolution. Before concluding and highlighting the perspectives in the last section, we make a state of the art of the related work in section 6.

2 Problem Situation by an Example

To better situate the problem, let us consider a fictional example illustrating the development of a simplified Access Control System (*ACS*). The different parts of figure 1 provide an overview of its architecture, which is organized as a *pipeline* (43) of four components. This system receives as input the necessary data for user authentication (*Authenticator* component). After identification, the data is sent to the *AccessController* component which checks whether the user is authorized to enter into the controlled area or not. If the access is granted, the *Logger* component adds to the data the entrance date and hour, and stores it locally (for the controlled area). It then sends these logs to the *Transmitter* component, which adds information about the controlled area and transmits all this data for a global storage.

The development of such an application may follow a classical process in three stages: architectural design, component design and component implementation. Transiting from one stage to another can be done manually or automatically by using model transformation techniques. Indeed, the choice of the means to transform from one model to another has no effect on our approach. Below we briefly present the development process of *ACS* in order to describe the raised problems.

2.1 Architectural Design Stage

Suppose at this stage that the architecture of the system was described using xAcme (56) [1] . The architecture designer chooses a pipeline style aiming at a high level of maintainability. To make the evolution of the model easier, this important decision should be documented. The need for this documentation is represented in Figure 1 by the label *AD1* (*AD1* stands for Architectural Decisions specific to the stage 1).

In this stage, it is possible to precisely document an important part of this decision: its associated architectural choice (the compliance with the pipeline style). One can use a dedicated language: the Armani language (32). This language provides constructs for capturing architectural design expertise such as design rules and architectural styles. Moreover, this is a formal language. Thus,

[1] xAcme is the XML representation of Acme ADL (14)

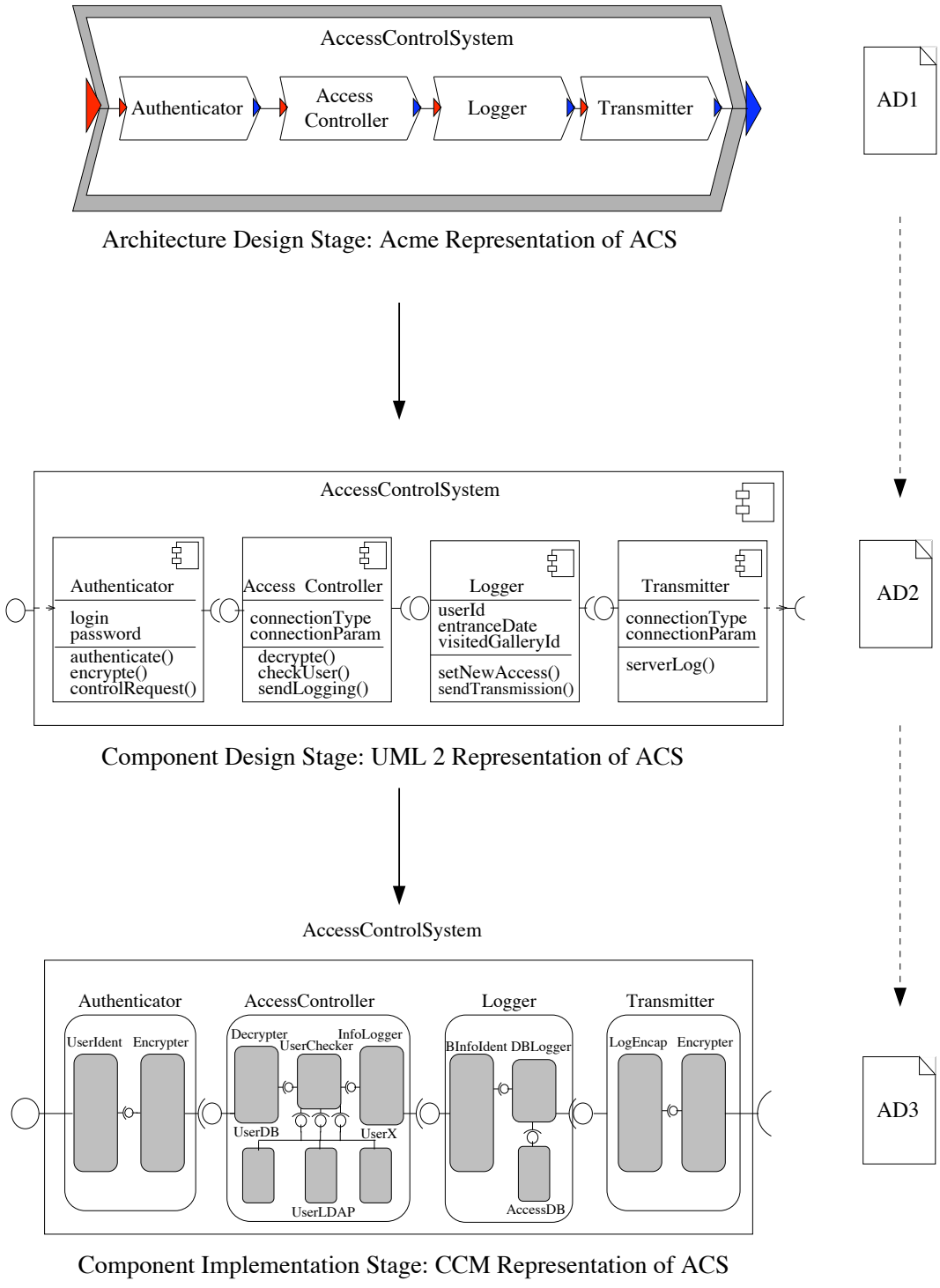

Figure 1. Decisions documentation in a classical component-based software development process

one can expect to benefit from an automated design rule checking engine, embedded in the used modeling tool, such as the AcmeStudio environment.

Using graph theory, we enumerate the structural properties that guarantee this architectural style. These properties are defined using the *Armani* language as follows [2]:

(1) The first constraint introduces the definition of vertices and arcs, such that each arc must be connected to two vertices:
   (a) A vertex is a component with input or output ports (represented by the keywords inputT and outputT):

```
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:

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

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

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

Henceforth, the pipeline architectural choice is documented. This documentation will facilitate in the future the understanding of the Acme model and therefore its evolution. Moreover, using the constraints described above, some tools can automatically check the compliance of the model with this choice after its evolution. Suppose that for performance reasons, during an evolution, a developer decides to send some data (whose logging is not necessary) directly from the *AccessController* component to the *Transmitter* component. Under time pressure, the developer does not take the necessary time in order to read and understand this somewhat complex documentation. Thus, she/he generates a direct link between two non-adjacent components. This change and the resulting component configuration affect the pipeline style chosen previously, and this consequently weakens the maintainability of the system. Using the constraint above, a tool can notify the attempt of breaking the architectural choice. It is the developer's responsibility, fully aware of the consequences, to maintain or not the modification.

Although the example provided here is simple and obvious, the same problems emerge in complex real-world systems as discussed in (3). One can notice that the documentation of this architectural choice, although possible using the Armani language, is far from being straightforward. Accordingly, it is not sure that developers make such an effort. This may discourage them, despite the gains that this documentation will provide later. The existence of a dedicated language is necessary but not a sufficient condition. The language must provide sufficient power while allowing descriptions, as simple and concise as possible. The previous example shows that sometimes the Armani language does not lead to an easily readable solution.

2.2 Component Design Stage

Before implementing ACS, using CORBA components, developers decided to establish an intermediate UML model for a smooth transition [3]. Additional decisions, specific to this UML diagram, may be added at this stage. This is represented by the *AD2* label on Figure 1. For example, to meet a maintainability requirement, we may specify in *AD2* that the *ACS* component should provide no more than two interfaces.

Unfortunately, the *AD2* decision cannot be specified using the UML language. This is due to the fact that its constraint language (Object Constraint Language (OCL)) cannot express this type of information. OCL is a formal language used to describe invariant conditions associated with a type appearing in a model (such as a particular Component or Class). But an invariant is a condition that must be true for all instances of that type at any time. Thus, an invariant is evaluated on the model's instances, not on the model itself. But, writing AD2 directly on the UML metamodel implies its evaluation on all UML components in all models and not just on the ACS component.

At this stage, there is no means to precisely document the architectural choice associated to *AD2*, in order to be explicit and possibly checkable. The existing constraint language (OCL) is not fully adapted to document some architectural choices. Consequently, there is an important risk to affect some important architectural decisions during an evolution of the model.

2.3 Component Implementation Stage

As in the previous stages, developers could add other decisions on the CORBA Component Model (CCM) architecture at this stage. These decisions are noted as *AD3*. These could include, for example, a restriction on the number of *AccessController*'s sub-components. In contrast to the previous two stages, there is no language dedicated to the description of all or part of an architectural decision in CCM models. For this purpose, programmers may use comments directly in the source code. Good comments can make the source code easier to understand. But, the flexibility provided by comments often allows for a wide degree of variability, and potentially useless information inside source code. Moreover, architecture decisions involve, by definition, several regions of the source code. This raises the problem of the localization of these comments. Anyway, informal comments limit the possibilities for an automatic control. Thus, at this stage, there is no dedicated means to help developers to

[3] Recent experiments (41) showed that some ADLs and UML can be used in complementarity, in order to make better analysis of software architectures

document or check architectural decisions. Consequently, when shifting from one version of a model to another, modifications may affect the architectural decisions.

This example highlighted the importance of documenting architectural decisions. It also shows that it is not always possible to document them throughout a development process. Depending on the case, this difficulty is related to either the lack of dedicated languages, or the non suitability of the offered languages, in terms of expressiveness and/or simplicity of use.

3 ACL: a Family of Architectural Constraint Languages

This section introduces ACL, a family of languages which allows the specification of architectural choices associated to decisions at any stage of a component-based software development process, in a convenient way. We will first introduce the requirements that our proposal aims to fulfill. Then, we will present the two-level structure that all these languages share. The first level allows the expression of basic first-order logic predicates in the context of MOF-based models. The second level takes the form of a set of MOF metamodels. We will describe successively each of these levels. We will conclude this section by giving two examples of ACL languages: one for xAcme models and the other for CCM models.

3.1 Requirements for an Architectural Constraint Language

In our approach, an architectural choice is described as a set of constraints that must be respected by a model. An architectural choice is respected by a model if all its related constraints are evaluated to be true on the model. Of course, it is difficult to identify all types of constraints that developers would have to express. Nevertheless, it is obvious that constraints enforcing architectural styles, design patterns, modeling and coding rules stipulated in quality plans should be expressed.

Besides, we should be able to define "scalable" or simple constraints. Two different constraints are given here for a better understanding of the difference between these two kinds of constraints: "the three sub-components should respect a pipeline style" and "whatever the number of sub-components, they should conform to a pipeline style". The first constraint cannot be checked (and should be adapted), if we add a fourth sub-component to the architecture, the second one remains checkable and valid. The last constraint is a scalable constraint, because it can be checkable for any number of architectural elements (we can check the presence of the pipeline style within a configuration of three components or more).

Architectural constraints can also relate to, either the current version (the most frequent case), or both the previous and the current versions. In the second case, we deal with "pure" evolution constraints which are generally present when some architectural constraints would regulate the acceptable structures in a differential mode. For example, for development reliability reasons, we would like to prohibit that a given component can be changed by adding to it more than one interface during an evolution. We should thus be able to evaluate differences between two consecutive versions of this component. These constraints which are real evolution constraints are of great interest as precised in (54).

The other difficulty encountered when proposing a language for describing architecture constraints is related to the fact that this language should be able to be used at each stage of an application life-cycle (need introduced in the previous section). We should have a language which can be applied either on high-level models or on implementation ones. A high-level model represents an abstract definition of the concrete runnable software system. In practice, it is a model provided by an Architecture Description Language (ADL). However, an implementation model represents a concrete definition for the software which corresponds to a model depending on a particular technology, such as EJB, COM+/.net or CCM. It is not desirable that developers have to deal with a variety of too different languages. The use of numerous different syntaxes and semantics make the documentation task quite challenging and if these languages are complex it becomes even more complicated. Under cost and schedule constraints, developers generally choose not to use them. To respond to this problem, it is important to propose a family of languages as simple as possible, that share the maximum syntactical and conceptual elements.

3.2 ACL Language Structure

ACL has a bicephalous structure. The first level allows the expression of basic first-order logic predicates in the context of MOF-based models. It thus provides navigation operations required in this type of models, set operators and usual quantifiers. It is ensured by a slightly modified version of UML's Object Constraint Language (36), called CCL (Core Constraint Language). The second level takes the form of a set of MOF metamodels. These metamodels represent the structural abstractions to be constrained, found in the main modeling languages used at each stage in the life-cycle. These abstractions are introduced during upstream stages by architecture description languages (ADLs and UML), and in coding stage by component technologies.

Each couple composed of CCL and a particular metamodel represents what we called an ACL profile. Each profile is used in a particular stage in the

development process. For example, we can use the ACL profile for xAcme in order to specify architectural constraints in the architectural design stage where we used xAcme ADL. After that, we can formalize other architectural constraints in the coding stage, where we implement the application in CCM, using the profile dedicated to CCM. The xAcme profile is composed of CCL and the xArch metamodel established for xAcme. The CCM profile is composed of CCL and our CCM MOF metamodel. In this way, at each stage in the life-cycle, a developer uses a particular profile to specify her/his architectural constraints. The expression of these constraints is made easy because the same abstractions, as those present in the language the developer is accustomed to use at this stage, are manipulated in this language.

ACL separates clearly two aspects which are interlaced in the architecture constraint languages existing in literature (like Armani). Indeed all these languages have grammars with terminal vocabulary which mixes constraint operators (navigation, predicate, etc.) and architectural abstractions (component, connector, interface, etc.). These languages can be used only in the context related to the architecture modeling language on which they apply. However the separation of these two aspects allowed us to have a modular language, of a reduced size, which can be applied in all stages of the life-cycle. In addition, the expressiveness of the language can be enhanced by simple modifications on the concerned metamodel. Indeed, these metamodels are parameters provided on input to the ACL compiler. They can be changed without rewriting the compiler.

To understand how this two-level structure is articulated, it is necessary to go back to the usual mode of expression of OCL constraints in a class diagram. In this type of diagrams, OCL is generally used to specify class invariants, pre/post conditions of operations, constraints on cycles between associations, etc. These constraints restrict the number of valid object diagrams instantiable from a class diagram. They mitigate the lack of expressiveness of UML graphical notation which, employed alone, can authorize in some cases the instantiation of object diagrams incompatible with reality that we wish to model. OCL constraints are described relative to a context. This context is an element of the class diagram, generally a class, an operation or an association present on this diagram. The followings are two examples of OCL constraints.

```
-- Constraint on the paper size
context Journal_Article inv:
  self.size > 10

-- Constraint on the number of authors
context article: Journal_Article inv:
  article.writtenBy->size() >= 1
```

In both cases, the context is a class (`Journal_Article`). The two constraints

are related to any instance of the context (here, an object instance of the class `Journal_Article`). It is the approach adopted for every OCL constraint. The first constraint references this instance using the keyword `self`, while the second introduces an ad-hoc identifier `article`. These two modes for referencing instances allowed by OCL are semantically equivalent. Every OCL constraint is made up of elements present either in the OCL language (`->`, `size()`, etc), or in the class diagram reachable from the context (the attribute `size` and the association-end `writtenBy`).

It is interesting to consider what might be the meaning of OCL constraints written on a metamodel rather than on a model. A metamodel exposes the concepts of a language and the links between them. It describes an abstract syntax. Thus, a constraint having for context a meta-class limits the expression power of the grammar's production rules and thus the number of the derived phrases (i.e. models). Some phrase structures are dismissed. If this metamodel describes the abstract grammar of a language dedicated to architecture description, then, a constraint expresses that only some architectures (i.e. models) are derivable (i.e. can be instantiated) in this language. So, the language is voluntarily restricted in its capacity of expression because we do not allow the description of some types of architectures. For example, we can impose that every component in the model must have less than 10 required interfaces, by putting this constraint in the context of the meta-class `Component`. This constraint is exactly of the type we wish to express. Unfortunately it has a global scope. It applies to all components and not to a particular one, as we would wish it to be. To achieve our goals, we slightly modified OCL syntax and semantics as explained in the following section.

3.3 ACL Predicate-Level: the CCL language

CCL language is an OCL dialect; more particularly a dialect of the 1.5 version whose grammar and semantics are available on OMG's website. OCL presents in fact many advantages. It is with UML the well-known standard adopted in industry and academia, and is implemented by many commercial and/or open-source tools. It provides besides a satisfactory expressiveness in a simple and intuitive format. Moreover, its efficiency has been proved in the context of maintenance (4). Briand et al. demonstrated in their work that OCL has great benefits in understanding and maintaining object models. In addition, the reuse of an existing standard language allows us to avoid the phenomenon of "yet another language" and thus makes easy ACL training for new users.

At the syntactic level, CCL is hardly a subset of OCL. The only production

rules of OCL language that are not supported by CCL are those related to pre- and post-conditions. Moreover, in order to limit a constraint scope to a particular architectural element, we modified the syntax and semantics of the context part in OCL. On the syntactic level, we impose that each context introduces necessarily an identifier (as the word `article` shown in the previous example). This identifier should be the name of one instance or many particular instances of the meta-class cited in the context. At the semantic level, we interpret the constraint with the meaning it has in the context of a meta-class, but limiting its scope to the instance cited in the context.

According to this principle, the constraint which follows, applied to a given MOF architecture metamodel (presented in the following sections), states that in the context of the ACS component (and only in this component) the primitive component of name `AccessController` must be bound to only one component using its required (out) interface `Archiving`.

```
context ACS:ComponentInstance inv:
  ACS.subArchitecture.archInstance.linkInstance
    ->select(l|l.endPoint.anchorOnInterface
               .componentInstance.description = 'AccessController'
               and l.endPoint.anchorOnInterface.direction = 'out'
               and l.endPoint.anchorOnInterface
               .description = 'Archiving')
    ->size() = 1
```

One addition to the OCL syntax is a mark which allows us to reference the previous version of an architectural element to which it apply: the `@old` mark. Consider the following example applied on a given metamodel.

```
context AccessController:ComponentInstance inv:
  AccessController.interfaceInstance->size() <
  (AccessController@old.interfaceInstance->size()+2)
```

This constraint stipulates that we could not add to the set of interfaces of the component `AccessController` more than one unit from one version to another. Note that this constraint does not tell anything about the possible changes undergone by the old interfaces of this component. In addition to the previous constraint, if we would like to impose that at most only one of its old interfaces could be changed from one version to another, we should compare the structures before and after evolution of all these interfaces. This type of constraints is hard to write only with OCL and the @old mark. Thus, we introduced the collection operators: `modified:Collection(T)` which returns the elements from the collection which have undergone changes between the current and the previous version (only to those which exist in the two versions), `added():Collection(T)` which returns the elements which do not exist in the previous version, and `deleted():Collection(T)` which returns the elements

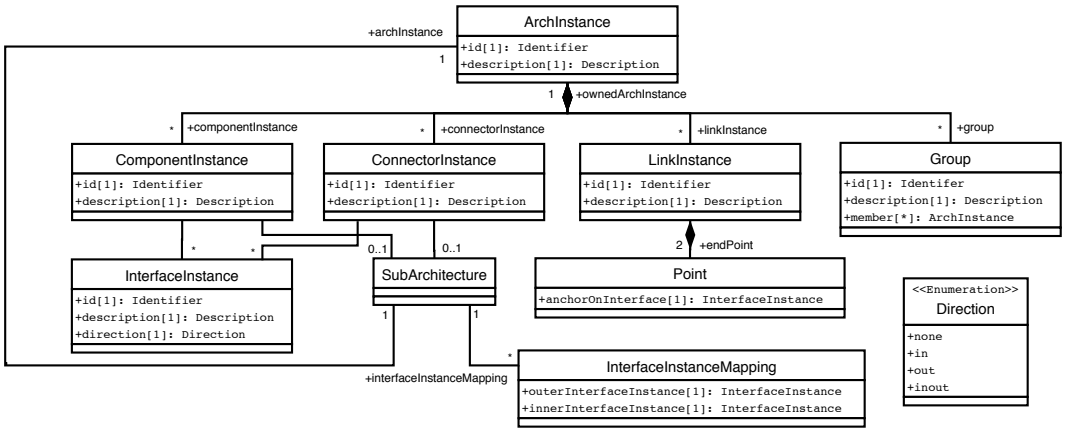

Figure 2. A MOF metamodel of xArch

from the collection omitted from the previous version. The previous constraint could be written as following:

```
context AccessController:ComponentInstance inv:
  (AccessController.interfaceInstance
  ->added()->size()<2)
  and
  (AccessController.interfaceInstance
  ->modified()->size()<2)
```

3.4 Architecture-Level Description

As stated previously an ACL profile is composed of CCL and a given metamodel. A metamodel embeds all architectural abstractions to be constrained by ACL and the relationships between these abstractions. The metamodel is described using MOF. MOF is the OMG's standard language which allows the specification of metamodels.

For instance, Figure 2 represents a MOF metamodel of xArch. xArch is an XML representation, which contains the primitive elements that compose an xAcme architecture description from a structural perspective. An xArch architecture instance is composed of a set of component instances, connector instances, link instances and logical groups of these architectural elements. Component or connector instances define a set of interface instances and optionally a sub-architecture for a hierarchical description. The sub-architecture 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.

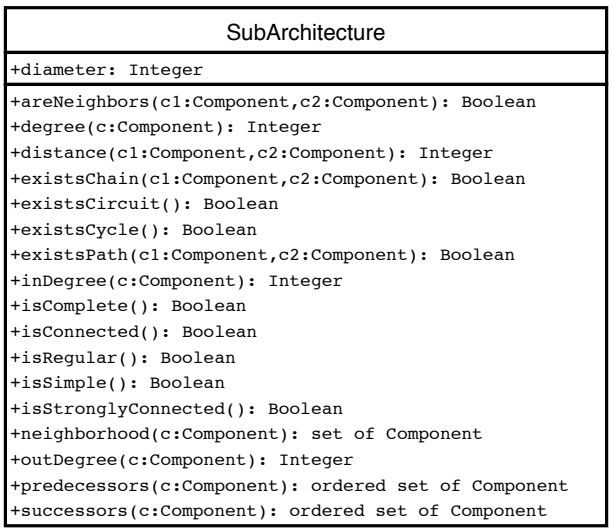

Figure 3. Graph properties on the `SubArchitecture` metaclass of the Xarch metamodel

We defined many profiles of ACL in order to provide a constraint language for several technologies used at design or implementation time in the development process. The profiles we developed are the following: ACL profile for xAcme (AP_{xAcme}), for UML 2 (AP_{UML2}), for CCM (AP_{CCM}), for EJB (AP_{EJB}) and for Fractal (6) ($\mathrm{AP}_{Fractal}$).

3.5 Examples of architectural choices in two ACL Profiles

Among these profiles, we present here some architectural choices expressed using two representative profiles: AP_{xAcme} and AP_{CCM} profiles. We choose the first profile because xAcme makes a synthesis of abstractions from several ADLs. The second language is a representative of the class of implementation technologies.

3.5.1 Architectural choices in xAcme

The previous examples of ACL constraints have been described in AP_{xAcme}. We present here another example of a constraint expressed also in AP_{xAcme}. Consider the example presented in section 2 which involved a pipeline architecture. The pipeline structural constraints defined in Armani can be expressed in AP_{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()`. In this ACL profile, this operation is associated to the type `SubArchitecture` of the xArch metamodel (as shown in Figure 3). Indeed, we introduced many graph operations, in the different metamodels, which make the constraint expression easier. Actually, we need only their signature, because their implementation is made only once for an intermediate meta-model (see section 4 for more details).

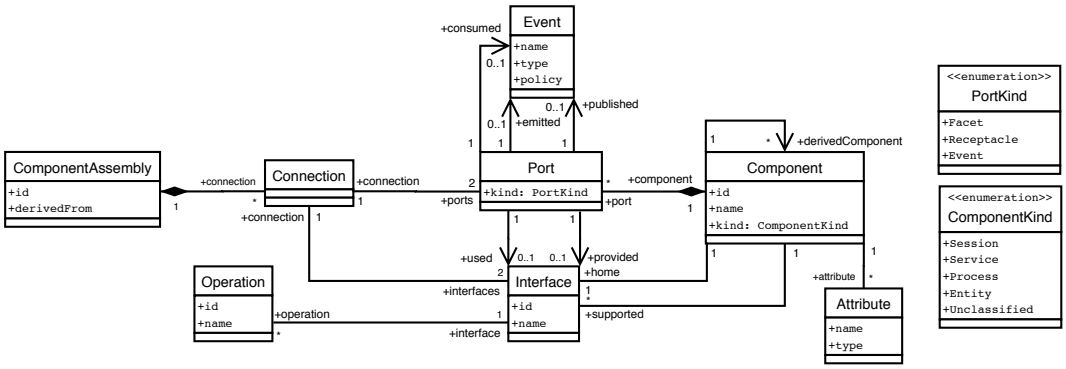

Figure 4. A MOF metamodel of CORBA Component Model (CCM)

With ACL, we can thus express constraints involving some 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.

3.5.2 Architectural choices in CCM

AP_{CCM} is composed of CCL and a MOF metamodel of CCM. The latter is illustrated in Figure 4. This metamodel represents the elements used to describe the structure of CCM applications. A ComponentAssembly defines Connections between Ports and/or Interfaces. A Port can represent a Facet, a Receptacle or one or many Events. A Facet defines the set of services provided by the component. A Receptacle describes the services required by the component. Events can be emitted, consumed or published for many clients. A Port can be represented in the form of required or provided interfaces, which export a set of operations. A Component can define attributes and be of different types: session, entity, service-oriented, process-oriented or none of these types. However, CCM is a flat component model: a component could not be hierarchically designed on the basis of other components.

The first constraint introduced previously in section 3.3 can be described using AP_{CCM} as following:

```
context ACS:ComponentAssembly inv:
  ACS.connection->select(c|c.component
                           .name = 'AccessController'
                           and c.component.port.used
                           .name = 'Archiving')
  ->size() = 1
```

In this constraint, we navigate to a set of all connections which bind the component `AccessController` to other components by the means of its used (out) interface `Archiving`. This set should contain only one element.

The following example represents the constraint, which illustrated the use of the @old mark in AP_{xAcme} in the previous section. This time, we describe it in AP_{CCM}:

```
context AccessController:Component inv:
  AccessController.port.interface->size()
  <
  (AccessController@old.port.interface->size()+2)
```

We navigate in this ACL constraint to all the interfaces of the component `AccessController` before and after evolution (in the previous and the current architecture description). When comparing these two collections of interfaces, the difference of sizes should be less than 2 from the previous to the current version.

Constraints formalizing the pipeline style which were expressed above in AP_{xAcme} can be described using AP_{CCM} as following:

```
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))
```

As in the previous constraints specified in AP_{xAcme}, we use the graph operation `isConnected()` in the constraints above. This operation and all graph properties are associated to the `ComponentAssembly` type in this profile.

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.

4 Evaluation of Architectural Constraints

In this section, we show how to evaluate the constraints whatever the chosen profile. In this evaluation approach, each software architecture model and its associated constraints (written in the profile dedicated to the used modeling language) are first transformed to an intermediate representation. This intermediate representation is defined by a particular ACL profile we have developed. This profile is called the Standard ACL Profile (AP_{Std}). It is composed of CCL and a generic architecture metamodel, called ArchMM. The software architecture model is transformed into an equivalent abstract syntax which conforms to ArchMM. Constraints are also transformed in AP_{Std}. This intermediate representation is then used to evaluate the obtained constraint specification on the transformed software architecture model. The interest of this approach is that we need only one evaluator (for AP_{Std}) instead of one for each profile. Of course, for each profile we have to define transformation rules to generate intermediate models and constraints. But, we have observed that the used modeling languages share many common points. Subject to choose a correct standard profile, it is often easier to write such transformations than coding one evaluator for each specific profile.

We will first present the chosen standard ACL Profile, then we introduce the used transformation mechanisms.

4.1 Standard ACL Profile

As stated previously, AP_{Std} is composed of CCL and ArchMM. We developed ArchMM on the basis of three types of metamodels. The first is a representative of architecture description languages, the second is related to the Unified Modeling Language and the last type of metamodels represents component implementation technologies.

We studied many ADLs such as Acme, Koala, xADL (8), and others. This state of the art [4] conducted us to the elaboration of a set of metamodels which abstract each architectural model described by these ADLs. These metamodels take into account the static aspect of architectures modeled using these languages. They thus represent the manner in which these languages struc-

[4] We started our study from the state of the art of Medvidovic and Taylor established in the end of the nineties (27).

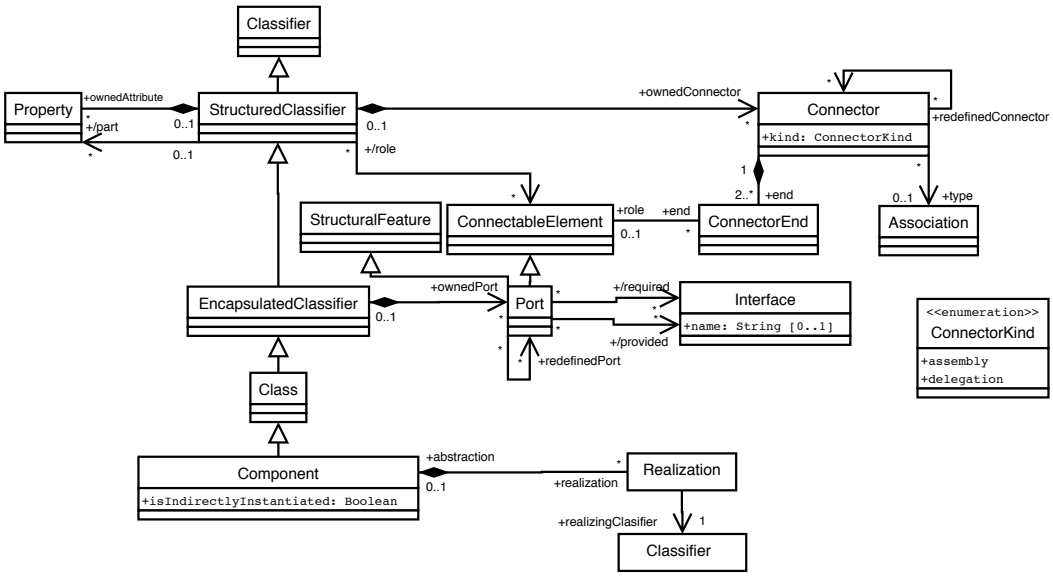

Figure 5. A flattened excerpt of the UML 2 component metamodel

turally describe software architectures [5] . The second category of metamodels was developed starting from the UML 2 superstructure specification. The last type of metamodels was designed for the Sun's EJB (*Enterprise JavaBeans*) and OMG's CCM (*CORBA Component Model*) component technologies. We considered that it is interesting to extend our study to the "hierarchical" component model Fractal of the ObjectWeb Consortium (6) and to define a metamodel for this technology.

In the first step, we elaborated MOF metamodels for each stage (architecture design, component design and component implementation). This task was relatively simple since standards have already emerged at all levels, like ADML for ADLs and CCM for component implementation technologies. Then, we designed ArchMM which summarizes all concepts present at the same time in component-based software abstract and concrete infrastructures. The intersection of abstractions represented by the different metamodels provides a minimal metamodel that does note take into account all architectural concepts and properties. So, ArchMM represents some abstractions which unify abstractions from other metamodels. More details are given in section 4.1.4. Before that, let us see what are the architectural abstractions represented in the different ADLs, in the UML 2 metamodel and in component technologies.

4.1.1 Architectural Abstractions at Architecture Design Level.

In theory, an ADL should be able to describe a software architecture in the

[5] Note that the behavioral aspect is not taken into account in these metamodels.

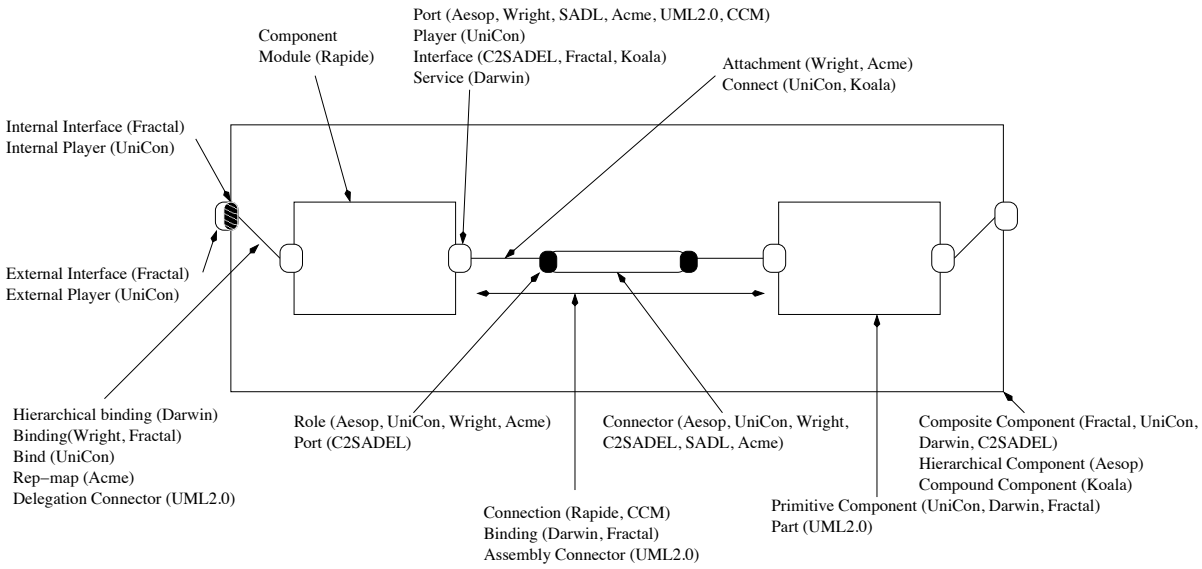

Figure 6. Graphical representation of architectural abstractions in ADLs, in UML 2 and in component technologies

form of the three Cs: Components, Connectors and Configurations (27). Components represent units of processing and storing data. The interaction between these components is encapsulated in connectors. In a configuration, the architect instantiates a number of components and a number of connectors. She/He binds them together to build the system. A part of ADLs answers to this description. However, the other part diverges. Some ADLs like Rapide (24), Darwin (25) or Koala do not make connectors as explicit entities. Others allow hierarchical descriptions of components. In this case, components are seen as white boxes and can have an explicit internal structure. In some ADLs, like Rapide, components are considered as black boxes. In UniCon (42), WRIGHT (2) or Acme we can define composite connectors, whereas this is impossible in the other ADLs.

The majority of ADLs have particular goals and answer to the concerns of their designers. Darwin and Koala target the dynamic reconfiguration of architectures. Rapide aims at describing architectures of event-based systems. SADL (33) focuses on the refinement of architectures. C2SADEL models architectures in the C2 style (26), etc. However, there are some ADLs, like ADML, Acme or WRIGHT who have a general objective. Indeed, WRIGHT metamodel is generic enough to be mapped on several other ADLs in spite of its objective to formalize software architectures. It is thus completely justified to take this metamodel like a reference for ADLs. However this metamodel contains several meta-classes which are not interesting for describing structural constraints. For example WRIGHT distinguishes types of components or connectors from their instances. This distinction is not interesting in our case for the main reason that a particular occurrence of a component or connector can be identified by its name using only one meta-class with a simple attribute

`name`. The same observations can be made on Acme and its extensions ADML and xAcme. We presented an example of a MOF metamodel of one of these ADLs in section 3.4.

4.1.2 Architectural Abstractions at Component Design Level.

UML provides a unified vocabulary for modeling object-oriented software and a standard supported by many CASE tools. In its 2.1.1 version (35), it provides the necessary elements to describe at the same time, the abstract architectures and component-based implementations [6] . A component is considered as a real modeling element (like a class) and not just a deployable entity. However, UML is a general purpose modeling language. Thus, there exist several concepts in its metamodel, which make it relatively complex. Our aim is to provide a metamodel which serves to describe architectural constraints. This metamodel should contain simple concepts known by developers.

Figure 5 presents an excerpt of the UML 2 metamodel for the description of components. A component inherits from the meta-class `Class`. So, it can define attributes, operations and participate in generalizations. It also inherits from `EncapsulatedClassifier`. It can thus define ports, typed by their required and provided interfaces. The meta-class `EncapsulatedClassifier` inherits from `StructuredClassifier`. Consequently, a component can have an internal structure and can define connectors between its internal parts.

For the description of constraints, the meta-classes `EncapsulatedClassifier` and `StructuredClassifier` and many others are not necessary. There exist several abstract classes, whose presence is particularly relevant to the unification aspect of the UML 2, and which make the description of constraints complicated (we should make many navigations to reach a given architecture concept). The same remarks are made on UML profiles [7] for the description of software architectures (17) and those for EJB and CCM.

4.1.3 Architectural Abstractions at Component Implementation Level.

Component technologies provide abstract models for developing component-

[6] Chapter 8: Components. UML 2.1.1 superstructure specification (35), page 159.
[7] A UML profile is a well-formed extension of the UML metamodel. A well-formed extension is an extension which conforms to the UML specification. More concretely, it is an extension based on stereotypes, tagged values and constraints.

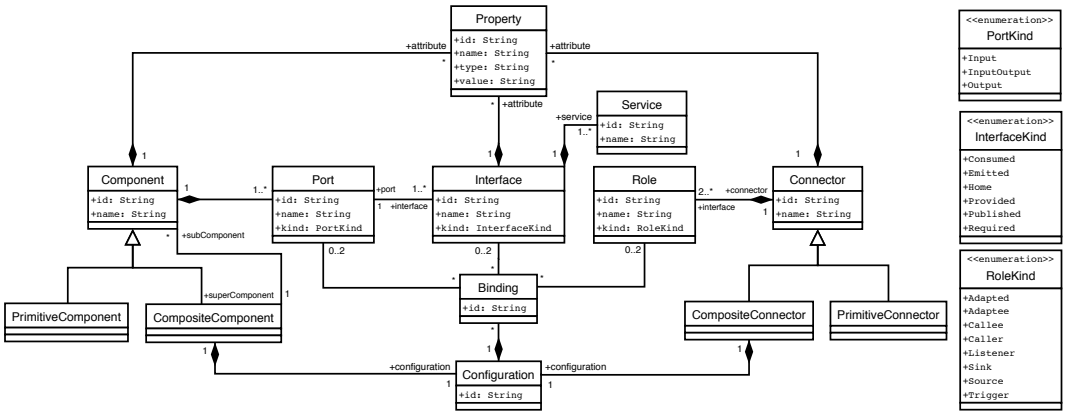

Figure 7. ArchMM: A generic architecture MOF metamodel

based applications and concrete infrastructures for their deployment. These infrastructures provide the necessary framework for the execution of applications, in terms of transactions, security, distribution services, etc. The abstract models defined by these technologies allow the definition of an application as a set of components providing some services (interfaces) and expliciting their dependencies with their environment. On the basis of the UML profiles for EJB and CCM, we established our own MOF metamodels for these two technologies. As stated in the previous sub-section, we defined our own metamodels because the original metamodels are not the ideal solution.

We presented in section 3.5 the CCM metamodel. We note that this metamodel includes all concepts found in the EJB metamodel. In other words, an EJB model can easily be mapped into CCM. We can deduce from this observation that the CCM metamodel can play the role of a generic metamodel for implementation component technologies. However, CCM is a flat component model where it is impossible to describe hierarchical components. Nevertheless, there exist some models of hierarchical implementation components, like Fractal. The latter allows the definition of composite components having an explicit internal structure. The resulting metamodel of this study abstracts component-based implementations like those in CCM, but with hierarchical features like in Fractal.

4.1.4 ArchMM: A Generic MOF Architecture Metamodel.

A synthesis of the architectural elements supported by the models discussed above is illustrated in Figure 6. In some of these models, we distinguish between internal and external interfaces. An internal interface is the interface of a composite component to which a hierarchical connector is attached. An external interface is an interface to which an assembly connector of the same level is attached, or the interface of a sub-component to which a hierarchical connector is attached.

Figure 7 represents the generic metamodel (ArchMM) described in MOF. In this metamodel, a system is described by a set of components (`ComponentInstance` in xAcme and `Component` in CCM), which represents units of data processing and storage. The communication and coordination modes between these components are encapsulated by connectors (`ConnectorInstance` in xAcme and `Connector` in CCM). The configuration of the system (`SubArchitecture` in xAcme and `ComponentAssembly` in CCM) is represented by an assembly of components by the means of connectors. This metamodel proposes the following concepts:

- A component defines one or many ports. These ports represent the interaction points of components with their environment. A port can be of several kinds: i)`Input`, when it receives incoming calls, ii) `Output`, when it receives out-coming calls, iii) or `InputOutput`, when it receives the two kinds of calls (incoming and out-coming). A port can support one or many interfaces of different kinds: i) `required` by the component (`in interfaces` in xAcme and `used interfaces` in CORBA Component Model), ii) provided (`out interfaces` in xAcme and `provided interfaces` in CCM), etc. Events in CCM and other ADLs are considered as interfaces in ArchMM. These interfaces can be of different kinds: `emitted`, `published` or `consumed`. These interfaces define a set of services, which correspond to CCM operations.
- A connector defines a number of roles. These roles correspond to connector interfaces in xAcme. A role can be of different kinds: i) `Adapted` or `Adaptee`, when the connector is an adapter, ii) `Callee` or `Caller` in the case of a connector of type method invocation, iii) `Listener` or `Trigger`, when the connector is of type event broadcast (or multicast), iv) and `Sink` or `Source` in the case of a pipe connector. If a connector is empty, it represents a simple association between two components (`Binding` in Fractal).
- A configuration represents several bindings. These attachments which correspond to `Connections` in CCM and `LinkInstance` or `InterfaceInstanceMapping` in xAcme are defined between: i) interfaces of two components required and provided), ii) the interface of a component and the role of a connector, iii) the port of a composite component and the interface of one of its sub-components, iv) or the roles of two connectors. We note that, in order to be generic, an interface or a role in this metamodel can participate in multiple bindings.
- A hierarchical description associates to a composite component or connector an internal structure. This internal structure is described by a configuration of sub-components and connectors. Delegations from ports of the components to their sub-components are performed by delegation connectors. In this manner, no direct association between interfaces of components' ports and interfaces of their sub-components is allowed.
- Properties are associated to components, connectors and interfaces. These properties are named and have typed values. The attributes of CCM components and the xAcme properties are mapped to these properties.

Some abstractions exist in several metamodels, like components and provided or required interfaces. However, some concepts present in ArchMM do not exist in the other metamodels. These concepts like ports, bindings, connectors and roles can be inferred during the transformation to ArchMM. For example, in CCM there are no connectors. During the transformation of a CCM description to ArchMM, connectors are generated with two roles. The generated roles are `Caller`, `Callee` and `Listener` or `Trigger` depending on the types of the connected components (`Facet`, `Receptacle` or `Event`). The distinction between events of type `Published` or `Emitted` is done at the interface level.

4.1.5 Examples in the Standard Profile.

The first example presented in sections 3.3 and 3.5 can be expressed in AP_{Std} as in the listing below. This example states that the component `AccessController` should have only one binding through its required (out) interface `Archiving`.

```
context ACS:CompositeComponent inv:
  ACS.configuration.binding
    ->select(b|b.interface.port.component.name = 'AccessController'
               and b.interface.kind = 'Required'
               and b.interface.name = 'Archiving')
  ->size() = 1
```

The following example represents the constraint, which illustrated previously the use of the @old mark in AP_{xAcme} and AP_{CCM}. We describe it now in AP_{Std}.

```
context AccessController:PrimitiveComponent inv:
  AccessController@old.port.interface->size <
  (AccessController.port.interface->size()+2)
```

As illustrated in section 2, ACS subcomponents are organized as a pipeline. This pattern can be structurally described using AP_{Std} as following:

```
context ACS:CompositeComponent inv:
  ACS.subComponent.port
  ->forAll(p:Port|(p.kind = 'Input')
                  or (p.kind = 'Output'))
  and
  ACS.configuration.binding.role.connector->AsSet()
  ->forAll(con:Connector|(con.role->size() = 2)
                         and ((con.role.kind = 'Source')
                         or (con.role.kind = 'Sink')))
  and
  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
  ACS.configuration.isConnected
  and
  ACS.configuration.binding.role.connector
  ->AsSet()->size() = ACS.subComponent->size()-1
  and
  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')))
```

We notice that this time the graph operation `isConnected()` is associated in this profile to the `Configuration` meta-class. All the other graph properties are associated to this meta-class.

We remark, in the examples presented in sections 3.5.1, 3.5.2 and those above, that the CCL part of these constraints remains mainly unchanged. The differences are globally observed at the architectural level. So, whatever the stage, the designer needs to deal with only CCL and the current meta-model (the language used to design one's model). Thus, this approach greatly simplifies the documentation of architectural choices throughout the development process (51).

4.2 Transformation of Architectural Constraints

In order to transform constraints written in the different profiles to AP_{Std}, we defined necessary mappings between the different metamodels and ArchMM. We present first what are the architectural concepts that are concretely mapped from the different metamodels to ArchMM. After that we introduce the method proposed to describe and evaluate transformation rules.

4.2.1 Mappings from Profiles to AP_{Std}

Transformation rules between profiles are classified into three categories:

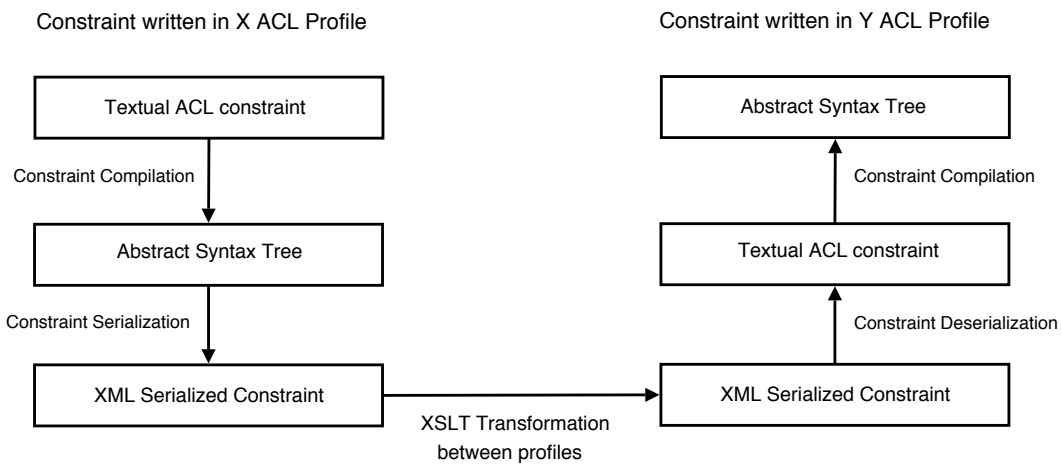

Figure 8. XML-based solution to constraint transformation

(1) Mappings between architectural abstractions, which are represented at the metamodel level by meta-classes and their properties (attributes and operations).

The following itemization lists some mappings between xArch metamodel and ArchMM in this category.

- `ComponentInstance` in xAcme represents `Component` in ArchMM,
- `ConnectorInstance` in xAcme represents `Connector` in ArchMM,
- `InterfaceInstance` in xAcme represents `Interface` in ArchMM,
- `SubArchitecture` in xAcme represents `Configuration` in ArchMM,
- `LinkInstance` in xAcme represents `Binding` in ArchMM,
- `Direction` in xAcme represents `PortKind`, `RoleKind` or `InterfaceKind` in ArchMM (this special case will be detailed at the end of this section),

(2) Mappings of relationships between architectural abstractions, which are represented at the metamodel level by navigation between meta-classes.

The mappings below represent projections between xArch and ArchMM abstractions.

- In xAcme, `ComponentInstance.interfaceInstance` represents the interfaces of a given component instance. In ArchMM it is represented by `Component.port.interface`,
- `ConnectorInstance.interfaceInstance` represents all interfaces of a given connector. It is mapped in ArchMM as `Connector.role`,
- `ComponentInstance.subArchitecture.archInstance.componentInstance` in xAcme navigates to all the subcomponents of a given component. In ArchMM, we write it as `CompositeComponent.subComponent`,
- `ComponentInstance.subArchitecture.archInstance.connectorInstance` in xAcme represents all connectors defined in the sub-architecture of a component. It is mapped in ArchMM as `CompositeComponent.configuration.binding.role.connector`

(3) Mappings between complex architecture queries which represent patterns of navigation in the metamodel. For example, `Component.interfaceInstance->select(i:InterfaceInstance | i.direction = in )` represents all provided (in) interface instances defined for a given xAcme component instance. This is translated into `Component.port.interface ->select(i:Interface | i.kind = 'provided')` in ArchMM.

4.2.2 Transformation Method

In order to transform constraints, we defined an XML-based method which is illustrated in Figure 8. This method is composed of the following primitives:

(1) **Constraint Compilation:** We first get ACL constraints in their textual form to compile them. This compilation generates an Abstract Syntax Tree (AST) for each constraint.
(2) **Constraint Serialization:** At this level, we serialize the generated ASTs into an XML document. This XML document contains mainly the information extracted from the leaves (terminal tokens) of the AST and their types. This represents a sufficient information, which can be used in the next primitive.
(3) **XSL Constraint Transformation:** Mappings illustrated in the previous section are implemented in the form of XSLT (55) statements. These style-sheets are used to transform the XML documents resulted from the previous primitive into other XML documents representing the same constraints in AP_{Std}.
(4) **Constraint Deserialization:** The XML documents resulted in the previous primitive are then deserialized to extract the constraint (in its textual form) in the AP_{Std}. This constraint is ready to be compiled and then evaluated.

During the transformation, first navigation patterns are sought in the source constraint in order to apply the corresponding transformation rule. If a pattern is matched, this part of the constraint is replaced by its equivalent (specified in the transformation rule) in ArchMM. If no pattern is found, relationships between abstractions are looked for. In this case, if a relationship is found, it is replaced by the corresponding part of the constraint in AP_{Std}. If no relationship is found, during analysis, researching architectural abstractions starts. Each abstraction found is replaced by the equivalent concept in ArchMM. If no case of the previous ones is found, no rule is applied and the syntactic unit remains unchanged. It is the case, for example, for the context identifier `AccessController` or the collection operation `size()`.

Note that some of the mappings mentioned previously are ambiguous. They have the same condition part, like `Direction` in the last item of the mappings

between abstractions. Concretely in transformation rules, we use the application context of the mapping to distinguish the different cases. In the previous example, the application context can be `Port`, `Role` or `Interface`. The rule generates respectively, `PortKind`, `Rolekind` and `InterfaceKind`.

Some type checking is performed on collection operations during the transformation. If a collection operation is found in the transformed constraint and if this operation cannot be applied to the generated collection type, the operation is transformed into its equivalent for that type or is removed. For example, if a transformation rule generates a set starting from a bag, and that the transformed operation is `asSet()`, this operation is replaced by an empty token. Indeed, this operation serves to transform a bag into a set and thus has no meaning in this case.

5 Tool Support & Underlying Technologies

In order to interpret ACL expressions, we developed `ACE` (Architectural Constraint Evaluator). This prototype tool allows the edition and the evaluation of architectural constraints specified in different stages. To validate the feasibility of our approach, we implemented in ACE the evaluation of two profiles: xAcme and Fractal.

5.1 ACE Architecture

The core component of ACE (the component `AD_DescEvaluator`) interprets only the AP_{Std}. This means that, it evaluates the intermediate description of the architectural constraints on the intermediate architecture description. Before performing this task, ACE automates the following transformations:

- from one specific architectural description (eg. xAcme) to the intermediate model,
- from ACL profile architectural constraints (eg. AP_{xAcme}) to intermediate architectural constraints (AP_{Std}).

If we want to change the ADL (in order to change the stage), we only have to provide ACE with the descriptor files of the new architecture description language or component technology (eg. Fractal).

Structurally, ACE is organized as in figure 9 and is composed of an:

- AD_Description Editor: Once the profile chosen (eg. xAcme or Fractal metamodel), this component uses the XMI format of the metamodel to guide

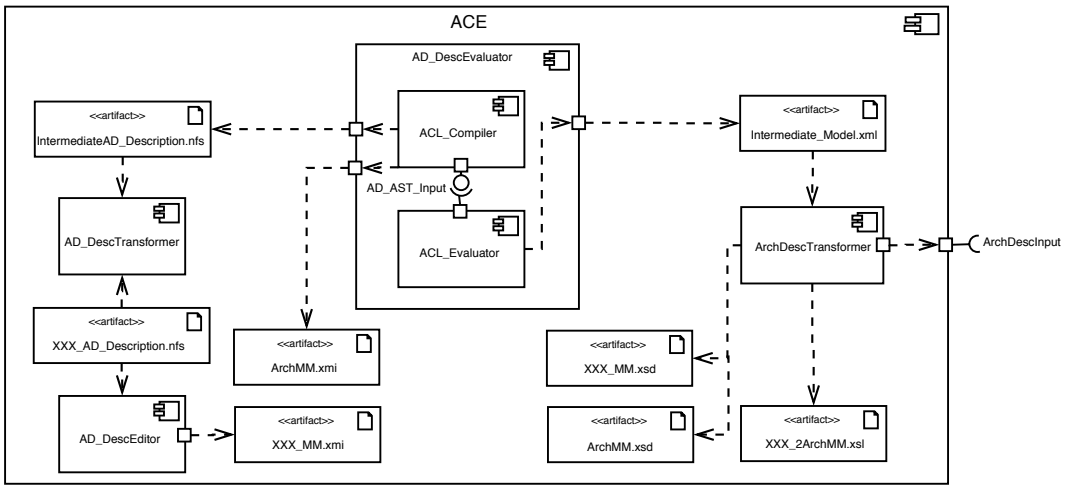

Figure 9. ACE: A prototype tool for editing and evaluating ACL constraints

the developer in editing his/her architectural constraints, by proposing the different navigation alternatives (through an auto-completion mechanism).

- AD_Description Transformer: Once the constraints are edited using the ADL corresponding profile [8], they are transformed to be compliant with ArchMM, in order to be evaluated. This component produces constraints that navigate in ArchMM expressed in the AP_{Std}. The transformation is performed on the basis of the mapping rules between the used ADL metamodel and ArchMM.
- ArchDesc Transformer: Using XSL scripts, this module generates an intermediate model from the used ADL descriptions. This pivot model is a specific instance of the metamodel ArchMM. We defined ArchMM as a set of XML schemas that extend xArch schemas. Thus the intermediate model is an XML-based ADL.
- AD_Description Evaluator: This component uses the intermediate model and ArchMM to evaluate the intermediate architectural constraints (see figure 9). It is composed of an `ACL Compiler`, which is an extension of the OCL Compiler (10). It generates abstract syntax trees for each architectural constraint. The metamodel is used to check if these constraints navigate well in its structure. If there are no syntactic or type errors, then the evaluation process starts. If the constraints are evaluated to be true, the architecture or component description is considered as in conformity with the constraints. Otherwise, the developer should correct either his/her architectural constraints or the architecture description. The latter must conform to the constraints so that ACE validates them. Note that, at this stage, only constraints involving one version of the architecture are evaluated (not those with the `@old` mark). Constraints involving two versions of the same architecture are evaluated once the descriptions of the two versions are available.

[8] It is also possible to edit constraints directly in AP_{Std}.

Following this architecture, ACE can be extended easily to support other ADLs or component technologies, like those discussed in this paper (CCM or UML 2).

5.2 Handling New ADLs or Component Technologies

To handle a new ADL or component technology, the following actions should be performed:

- a new ACL profile should be defined which includes only the MOF metamodel of this language or technology,
- the transformation rules for architectural descriptions should be elaborated (XSL transformation scripts between the new metamodel and ArchMM),
- and the transformation rules for architectural constraints should be enumerated (XSL scripts).

On the tool-level no changes should be performed. One should only create a new metamodel by precising the different elements composing the new profile (the XMI and XSD documents of the metamodel, and the two XSL scripts to transform architecture descriptions and architecture constraints). The necessary changes are then done automatically to take into account the new features.

6 Related Work

Some existing ADLs provide constraint languages, which allow the formalization of architectural decisions. In WRIGHT (2) as in Acme, constraint languages are provided. In Acme (or xAcme) architectural constraints are expressed using the Armani predicate language. All these languages have in common the same basis as with ACL. 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 as syntactic constructs. The approach that we proposed disconnects these elements through the architecture metamodels. Thus, it becomes possible to cover all the development process with the same family of constraint languages. ACL also allows the specification of pure evolution constraints, whereas the ADLs focus only on design constraints.

In (28), 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 constraints are considered as general rules that apply to all modeled architectural elements. However, in our approach the constraints are considered as specific rules that apply only to specific architectural elements 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.

In addition to these languages some other ADLs provide capabilities to define some other kinds of decisions like the choice of architectural styles. In Aesop (13) it is possible to graphically represent architectures according to a particular style. This environment provides a set of tools (for edition and analysis) related to the style in question. Behind the architecture descriptions, the formalization of styles is represented by a library of modules in the C language. The main difference between the approach presented in this paper and the Aesop system is that the first provides capabilities to formalize styles as high-level constraints and the second provides the same capabilities but at the code level. In addition, the goal of Aesop is to introduce an ADL which allows the design of systems in particular styles. The focus in our work is on providing a set of languages for architecture constraint description and not on designing architectures and on supporting style analysis and evaluation.

In the literature, there are many works on the documentation of architecture design decisions. Clements et al. in (7) present an approach which provides a framework for documenting different views of a software architecture. The authors propose a template for architecture description encompassing the documentation of architectural decisions. In (53), Tyree and Akerman discuss the importance of documenting architecture decisions and their specification as first-class entities in architecture description. They present a template specifically designed for architecture decision documentation, which embeds interesting information characterizing architecture design decisions (`status`, `assumptions`, `implications`, `related artifacts`, `constraints`, ...). Philippe Kruchten in (19) introduced a taxonomy of design decisions. He presents a model for describing architecture decisions, including `rationale`, `scope`, `state`, `history of changes`, `categories`, `cost` and `risk`. He identifies the different possible relationships between design decisions and links between design decisions and design artifacts in this ontology. In (23), the authors present a tool support for visualizing all these elements. In (16), Jansen and Bosch present a new way of building software architectures. They propose to define these design models as a composition of architecture design decisions. The authors introduce a model for architecture design decisions, including a `description`, the `rationale`, the `design rules`, the `design constraints`, the `consequences`, the `pros` and `cons`. Archium is a method proposed in conjunction with this model in order to connect this model to software architecture descriptions. Starting from a given documented design problem, a

solution should be chosen among multiple other solutions. Once the decision is made, the solution is defined as an architectural change, represented by deltas to be integrated to the software architecture model through a composition model. de Boer et al. presented in (9) a core model of architectural knowledge. This complete and voluntarily minimalistic model is based on the ISO/IEC 42010:2007 and ANSI/IEEE Std 1471-2000 standard (15). It enriches this standard and adapts it by taking into account the results of some experiments conducted with four industrial partners. An alignment of the different organizations' models of architectural knowledge has been performed to extract the common concepts and build the core model. This core model embeds the concept of Architectural Design Decision, which is a specialization of the concept of Decision that influences the Architectural Design. Rationale is not an explicit concept in the model, but is orthogonal. An architecture design is the result of all architectural decisions. The validated model introduces the concepts of Role and Activity, inspired from SPEM.

Lago and van Vliet (21) introduced the concept of *Design Assumption*. They presented an approach to make explicit assumptions made during design in order to enrich architecture models. The authors defined design assumptions as the rationale of decisions made on-the-fly during architecture design (issued, for example, from the developer's experience). These elements, which are often implicit, represent invariabilities that should be documented to better understand a given software architecture before its evolution. They proposed a tool to describe these assumptions and to define their relationships to the corresponding architectural elements (`F-impacts` for the functional impact of an assumption on an architectural element, `S-impacts` for structural impact, and `Realizes` for a realization relationship, (reversely) between an architectural element and a design assumption). It is also possible to define relationships between assumptions in order to build hierarchies of assumptions.

In all of these works, the authors proposed models to document design decisions (assumptions), with a textual or graphical notations. The authors identified fine-grained elements to specify decisions or relationships between these decisions or their links to architectural design elements. In our paper, the focus was on the specification and interpretation of a part of this documentation, which corresponds to design rules and constraints in (16), to existence decisions ("ontocrises") in (19) and to constraints in Tyree and Akerman's model. This is documented using a formal language (ACL), for which an interpretation tool-support has been developed. Our approach is complementary to all of these approaches. As pointed out in this paper's introduction, ACL can be used to document the most important decisions, related to the critical parts of the system, and for which automatic checking is indispensable. The formal specification of (ACL) constraints can be enriched with information related to the decisions (rationale, state, cost, ...) issued from the models previously cited. The other (less critical, but not less important) decisions could comple-

ment formal decisions and be textually or graphically documented using the previous models.

Other research works focus on the specification of some particular kinds of design decisions, which are the choices of design patterns. In (22), the authors present an approach which aims at modeling design patterns in a formal manner by proposing extensions to the UML metamodel [9]. They use OCL to describe (structural and behavioral) constraints representing two examples of design patterns (the visitor and the observer (12)). These constraints are defined at the metamodel level and are specified in the form of collaboration meta-diagrams. Some association mechanisms of these diagrams of meta level and their instance (occurrences of design patterns) are then defined. This allows to model design patterns with the UML language in a precise manner. Like in this work, we use the OCL language at the meta-model level. However, in our case, the constraints' context is not an element of the metamodel, but an entity at the model level. The goal of the work referenced here is to provide UML developers with a means to describe a design pattern in a precise way. In our work, we focus on the documentation of a design decision which can be eventually a design pattern that we would like to preserve throughout a development process.

There are many other works on formal description of design patterns. The goal of these approaches vary from pattern application (31) to code generation (1). Some approaches aim at detecting patterns (1) in design documents or in the code, whereas some others target a better comprehension of patterns in order to choose the one that better resolves the encountered design problem (47; 22). Other works target pattern composition and deal with their temporal aspects (30). The aims of these works are different from ours. However, our approach is complementary to these works in order to build a safe software development process.

In the literature, only functional constraints have been addressed in component implementation technologies, like for CCM in (48) and for EJB in (5). Architectural issues have not been explored yet. We think that the work presented in this paper has a main contribution in this part of the development process.

[9] The UML version taken into account is the 1.3 one. The authors also discuss the applicability of the approach in the UML 2 metamodel.

7 Conclusion & Future Work

Bridging the gap between software architectures and component implementations (defined by component technologies) can be dealt with in an MDA-like approach. These two models can be treated as complementary first-class entities within a component-based software development process. In this paper, this was the starting point. Then the discussion focused on the specification and evaluation of the constraint part of architecture decisions made on these models. These constraints are defined, throughout the development process, using a FOL-based family of languages called ACL, proposed in our work. At each stage, it is possible to describe and then evaluate architectural constraints using the different members of the ACL family. This family of languages is based on well known and widely used technologies, which are OCL and MOF ("UML-like language").

Using ACL profiles to define architectural constraints provides a means to make available one of the important parts of architecture documentation throughout the software life-cycle. This ensures easy and safe evolution (maintenance) activities on the software. Indeed, among the maintenance activities, understanding the software architecture before its evolution, and iterations between regression testing and applying concrete changes on the software, are by far the most expensive (52). By making the most important architectural choices explicit and formal, on the one hand, we facilitated the architecture comprehension (through knowledge documentation) and on the other, we automated some checking that allows an assistance to the evolution activity. This assistance notifies developers on-the-fly, whenever an architectural change affects the documented architectural choices. This contributes in reducing the costs related to the software evolution activity by anticipating some regression tests, as detailed in (52).

It is true that often developers performing maintenance and evolution tasks are less skilled than developers performing the initial design. Indeed, the violation of a constraint alerts the maintainer, but it is up to another person (architect) to rewrite (if it is needed) the constraint if the maintainers are unable to do it. The definition of this role is a methodological problem within the development team. Also, it is observed that frequently the design models and code are not effectively synchronized; only the code is evolved by the system evolvers, and updates are not made on design artifacts. Indeed, we made here a technological assumption where we consider that developers use existing tools performing round-trip engineering to ensure model-code consistency.

Even if there exist many tools for round-trip engineering, we would like to be able to detect constraint violations on the source code. To do this, we are adopting a component-oriented programming language, called SCL (11), for

which we are adding reflection capabilities. We are implementing the transformation of ACL constraints to SCL (reflective) code. This will inevitably help the developers (performing maintenance and evolution tasks) to better understand the architecture constraints.

Besides, we started recently an investigation on model constraint transformation. This work aims at using MOF/QVT-compliant (34) languages dedicated to model transformations to transform constraints. The final goal is to transform architecture descriptions using rules defined by these languages, and then study the possible reuse of the same rules to transform architecture constraints associated to these models (descriptions).

We validated one ACL profile with an industrial partner in (52). We experienced the use of this profile in the development of a part of a GIS (Geographic Information System). The main goal of this work was to test the documentation of architecture choices and its potential benefit during software evolution. For this aim, we developed an Eclipse plugin version of the ACE tool in order to ease its use by the company's developers. The company implements its information system projects in Eclipse and uses its own ADL (close to UML) which is specific to GIS. We had thus described its metamodel and the necessary transformation files to ArchMM (model + constraints).

The main lesson learned from this experimentation is that we should not try to describe all the knowledge using architectural constraints. Otherwise, the amount of information is detrimental to its usability. Some architectural choices are obvious and do not require formal documentation. The best strategy is that the developers first identify the most significant choices which are fundamental and important in the domain and then document only these choices.

Some interviews conducted after the experiment, for a qualitative goal, showed that the developers found the profile easy to grasp. This is mainly due to the fact that the profile is based on OCL, which is a standard that respects the well-known object concepts. In addition, we obtained a good feedback on the usefulness of this language in the maintenance activities. So, we are convinced that the use of the different ACL profiles in different contexts can be an added value for a reliable software development.

References

[1] Albin-Amiot, Hervé and Guéhéneuc, Yann-Gaël: Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis. In proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods (2001)

[2] Allen, Robert: A Formal Approach to Software Architecture. PhD thesis of Carnegie Mellon University, Pittsburgh, PA, USA (1997)
[3] Bass, Len and Clements, Paul and Kazman, Rick: Software Architecture in Practice, 2nd Edition. SEI Series in SW Eng. Addison-Wesley (2003)
[4] Briand, Lionel C. and Labiche, Yvan and Di Penta, Massimiliano and Yan-Bondoc, Han (Daphne): An Experimental Investigation of Formality in UML-Based Development. In IEEE Transactions on Software Engineering, vol. 31, num. 10, IEEE Computer Society (2005) 833–849
[5] Brucker, Achim D. and Wolff, Burkhart: Checking OCL Constraints in Distributed Systems Using J2EE/EJB. Technical report of Albert-Ludwigs-Universität Freiburg (2001) 1–46
[6] Bruneton, Eric and Coupaye, Thierry and Leclercq, Matthieu and Quéma, Vivien and Stefani, Jean-Bernard: An Open Component Model and its Support in Java. In Proceedings of the International Symposium on Component-Based Software Engineering (CBSE'04), Edinburgh, Scotland (2004)
[7] Clements, Paul and Bachmann, Felix and Bass, Len and Garlan, David and Ivers, James and Little, Reed and Nord, Robert and Stafford, Judith: Documenting Software Architectures, Views and Beyond. SEI Series in Software Engineering. Addison-Wesley (2003).
[8] Dashofy, Eric M. and van der Hoek, André and Taylor, Richard N.: A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM Transactions On Software Engineering and Methodology, vol. 14, num 2 (2005) 199–245
[9] de Boer, Remco C. and Farenhorst, Rik Lago, Patricia and van Vliet, Hans and Clerc, Viktorand Jansen, Anton: Architectural Knowledge: Getting to the Core. A book chapter in Software Architectures, Components, and Applications. Springer LNCS, vol. 4880, (2008) 197–214
[10] Technische Universitat Dresden: OCL Compiler web site. http://dresden-ocl.sourceforge.net/ (2002)
[11] Fabresse, Luc and Dony, Christophe and Huchard, Marianne: Foundations of a Simple and Unified Component-Oriented Language. In Journal of Computer Languages, Systems & Structures, vol. 34, num. 2-3, Elsevier (2008) 130–149.
[12] Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John: Design Patterns: Elements of Reusable Object-Oriented Sofware. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc. (1995)
[13] Garlan, David and Allen, Robert and Ockerbloom, John: Exploiting Style in Architectural Design Environments. In proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering (1994) 175–188
[14] Garlan, David and Monroe, Robert T. and Wile, David: Acme: Architectural Description of Component-Based Systems. In Foundations of Component-Based Systems, Cambridge University Press, Gary T. Leavens and Murali Sitaraman (2000) 47–68

[15] ISO/IEC 42010:2007 and ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems (2007)

[16] Jansen, Anton and Bosch, Jan: Software Architecture as a Set of Architectural Design Decisions. In Proceedings of of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05), Pittsburgh, Pennsylvania, USA (2005)

[17] Kandé, Mohamed Mancona and Strohmeier, Alfred: Towards a UML Profile for Software Architecture Descriptions. In proceedings of UML'2000 - The Third International Conference on the Unified Modeling Language: Advancing the Standard - (2000)

[18] Karahasanovic, Amela and Levine, Annette Kristin and Thomas, Richard: Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. In Journal of Systems and Software Volume 80 , Issue 9 (Sept 2007) 1541–1559

[19] Krutchen, Philippe: An Ontology of Architectural Design Decisions in Software Intensive Systems. In Proceedings of the 2nd Groningen Workshop Software Variability (2004) 54–61

[20] Kruchten, Philippe and Obbink, Henk and Stafford, Judith: The past, present, future of Software Architecture. In IEEE Software, vol. 23, num. 2 (2006) 22–30

[21] Lago, Patricia and van Vliet, Hans: Explicit Assumptions enrich Architectural Models. In Proceedings of the 27th International Conference on Software Engineering (ICSE'05), St. Louis, Missouri, USA, (2005) 206–214

[22] Le Guennec, Alain and Sunyé, Gerson and Jézéquel, Jean-Marc: Precise Modeling of Design Patterns. In proceedings of the third International Conference on the Unified Modeling Language (2000)

[23] Lee, Larix and Kruchten, Philippe: A Tool to Visualize Architectural Design Decisions. In Proceedings of the Fourth International Conference on the Quality of Software Architectures (QoSA'08), Karlsruhe, Germany, Springer-Verlag LNCS (2008) 43–54

[24] Luckham, David C. and Kenney, John L. and Augustin, Larry M. and Vera, James and Bryan, Doug and Mann, Walter: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, vol. 21, num. 4 (1995) 336–355

[25] Magee, J. and Kramer, J.: Dynamic Structure in Software Architectures. In proceedings of the fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, USA (1996) 3–14

[26] Medvidovic, Nenad: Architecture-Based Specification-Time Software Evolution. PhD thesis of the University of California, Irvine (1999)

[27] Medvidovic, N. and Taylor, N R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, vol. 26, num. 1 (2000) 70–93

[28] Medvidovic, Nenad and Rosenblum, David S. and Redmiles, David F. and Robbins, Jason E.: Modeling Software Architectures in the Unified

Modeling Language. In ACM Transactions On Software Engineering and Methodology, vol. 11, num. 1 (2002) 2–57
[29] Microsoft: COM: Component Object Model Technologies. Microsoft Web Site: http://www.microsoft.com/com/ (2005)
[30] Mikkonen, Tommi: Formalizing Design Patterns. In proceedings of the 20th International Conference on Software Engineering (1998) 115–124, IEEE Computer Society Press
[31] Mili, Hafedh and El-Boussaidi, Ghizlaine: Representing and Applying Design Patterns: What is the Problem. In proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (2005) Springer-Verlag
[32] Monroe, Robert T.: Capturing Software Architecture Design Expertise with Armani. PhD thesis of the School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (2001)
[33] Moriconi, Mark and Qian, Xiaolei and Riemenschneider, R. A.: Correct Architecture Refinement. IEEE Transactions on Software Engineering, vol. 21, num. 4 (1995) 356–372
[34] Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation, version 1.0: formal/08-04-03. Object Management Group Web Site: http://www.omg.org/spec/QVT/1.0/PDF/ (2008)
[35] Object Management Group: Unified Modeling Language: Superstructure, version 2.1.1, formal/2007-02-03. Object Management Group Web Site: http://www.omg.org/docs/formal/07-02-03.pdf (2007)
[36] Object Management Group: OCL 2.0 Specification, version 2.0, Document ptc/2005-06-06. Object Management Group Web Site: http://www.omg.org/docs/ptc/05-06-06.pdf (2005)
[37] Object Management Group: CORBA Components, v3.0, Adopted Specification, Document formal/2002-06-65. Object Management Group Web Site: http://www.omg.org/docs/formal/02-06-65.pdf (2002)
[38] Object Management Group: 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)
[39] Object Management Group: UML Profile for Corba Components Final Adopted Specification, Document ptc/04-03-04. Object Management Group Web Site: http://www.omg.org/docs/ptc/04-03-04.pdf (2004)
[40] Rational Software Corporation: UML/EJB (TM) Mapping Specification. Java Community Process, JSR-000026: http://jcp.org/aboutJava/communityprocess/review/ jsr026/ (2001)
[41] Roshandel, Roshanak and Schmerl, Bradley and Medvidovic, Nenad and Garlan, David and Zhang, Dehua: Understanding tradeoffs among different architectural modeling approaches. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'04) (2004) 47–56
[42] Shaw, Mary and DeLine, Robert and Klein, Daniel V. and Ross, Theodore L. and Young, David M. and Zelesnik, Gregory: Abstractions for Software

Architecture and Tools to Support Them. IEEE Transactions on Software Engineering vol. 21, num. 4 (1995) 314–335
[43] Shaw, Mary and Garlan, David: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
[44] Shaw, Mary and Clements, Paul: The Golden Age of Software Architecture. In IEEE Software, vol. 23, num. 2 (2006) 31–39
[45] Sun-Microsystems: Enterprise JavaBeans(TM) Specification, version 2.1. Sun-Microsystems Web Site: http://java.sun.com/products/ejb (2003)
[46] Szyperski, C.: Component Software: Beyond Object-Oriented Programming (2002)
[47] Taibi, Toufik and Ngo Chek Ling, David: Formal Specification of Design Patterns - A Balanced Approach. In the Journal of Object Technology, vol. 2, num. 4 (2003)
[48] Teiniker, Egon and Lechner, Robert and Schmoelzer, Gernot and Kreiner, Christian and Kovacs, Zsolt and Weiss, Reinhold: Towards a Contract Aware CORBA Component Container. In proceedings of the 29th Annual International Computer Software and Applications Conference (2005) 545–550, IEEE Computer Society Press
[49] Tibermacine, Chouki and Fleurquin, Régis and Sadou, Salah: Preserving Architectural Choices throughout the Component-based Software Development Process. In proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05), Pittsburgh, Pennsylvania, USA (2005) 121–130
[50] Tibermacine, Chouki and Fleurquin, Régis and Sadou, Salah: NFRs-Aware Architectural Evolution of Component-based Software. In proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), Long Beach, California, USA (2005) 388–391
[51] Tibermacine, Chouki and Fleurquin, Régis and Sadou, Salah: Simplifying Transformations of Architectural Constraints. In Proceedings of the ACM Symposium on Applied Computing (SAC'06), Track on Model Transformation (2006) 1240–1244
[52] Tibermacine, Chouki and Fleurquin, Régis and Sadou, Salah: On-Demand Quality-Oriented Assistance in Component-Based Software Evolution. In Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06) (2006) 294–309
[53] Tyree, Jeff and Akerman, Art: Architecture Decisions: Demystifying Architecture. IEEE Software, vol. 22, num.2 (2005) 19–27
[54] van Ommering, Rob and van der Linden, Frank and Kramer, Jeff and Magee, Jeff: The Koala Component Model for Consumer Electronics Software. In IEEE Computer, vol. 33, num. 3 (2000) 78–85
[55] The Extensible Stylesheet Language Family (XSL). W3C Web Site: http://www.w3.org/Style/XSL/ (2005)
[56] xAcme: Acme Extensions to xArch. School of Computer Science Web Site, Carnegie Mellon University: http://www-2.cs.cmu.edu/∼acme/pub/xAcme/ (2001)