Simplifying Transformation of Software Architecture Constraints*

Chouki Tibermacine
VALORIA Lab.
University of South Brittany
F-56000, Vannes, France
tibermac@univ-ubs.fr

Régis Fleurquin
VALORIA Lab.
University of South Brittany
F-56000, Vannes, France
fleurqui@univ-ubs.fr

Salah Sadou
VALORIA Lab.
University of South Brittany
F-56000, Vannes, France
sadou@univ-ubs.fr

ABSTRACT

The heterogeneity of architectural constraint languages makes difficult the transformation of architectural constraints throughout the development process. Indeed they have significantly different metamodels, which make the definition of mapping rules complex. In this paper, we present an approach that aims at simplifying transformations of architectural constraints. It is based on an architectural constraint language (ACL), which includes one core constraint expression language and different profiles. Each profile is defined upon a metamodel, which represents the architectural abstractions manipulated at each stage in the development process.

General Terms

Design, Languages

Keywords

Architecture Constraint Languages, Constraint Transformation, OCL, Component-Based Software Development

1. INTRODUCTION

Software architecture models define the global structure of a software. It is argued that these models should be associated with a documentation that explains the architectural decisions made (called `Rationale` in Perry's architectural model [12]). This documentation aims at better understanding the architecture of the system, and at controlling its future evolution [15]. It is often defined using a formal constraint language. The constraints composing this documentation relate on the structural aspect of a model and stipulate that only certain architectures (i.e. models) are allowed (for instance, in an architectural model all components should have less than 10 provided interfaces). They have a different concern than traditional constraints, which serve as guards on instances of a model, like OCL invariants on a UML model. The latter constraints are of model-level, while architectural constraints are of metamodel-level. In general, architectural constraints define some architectural styles in order to meet non-functional requirements. We notice that only structural constraints are discussed in this paper, the behavioral aspect of software architecture will not be dealt with.

Let us consider the component-based software development (CBSD) process. In an MDE approach a software architecture model can be transformed into another architecture model (for instance a design-level model into an implementation-level one). It is important that we transform not only the software architecture model, but also its associated documentation (architectural constraints). This makes the documentation available and consistent throughout the different stages of the CBSD process. In addition, constraints can be checked not only when evolving the design-level model, but also when evolving the implementation-level model [14]. However, transformation of architectural constraints is complex due to the fact that constraint languages, used at different stages in the CBSD process, are based on metamodels which are completely different (see section 2). In this paper, we present an approach which aims at simplifying architectural constraint transformation. This approach introduces a bicephalous architectural constraint language, named ACL (section 3). It is based on OCL, as a core constraint language, and on a set of MOF architecture metamodels. We show how we can use this language to describe uniformly architectural constraints throughout the CBSD process in order to facilitate the description of transformation (section 4).

2. IDENTIFIED PROBLEM

Consider that at design-level, we describe a software architecture of a system using xAcme [16], an XML representation of Acme Architecture Language [6]. The corresponding architectural constraints are specified using the constraint language of this ADL, namely Armani [9]. For example, let us suppose a simple architectural constraint that ensures in a given architecture model the existence of less than 5 provided (in) interfaces for all subcomponents of a specific component. This constraint is the translation of a rule stating that the quality measure CBM[1] (Coupling Between Mod-

*This work is supported partially by the Brittany Region Council under contract number 20046839.

[1]CBM metric is the equivalent of CBO (Coupling Between

.

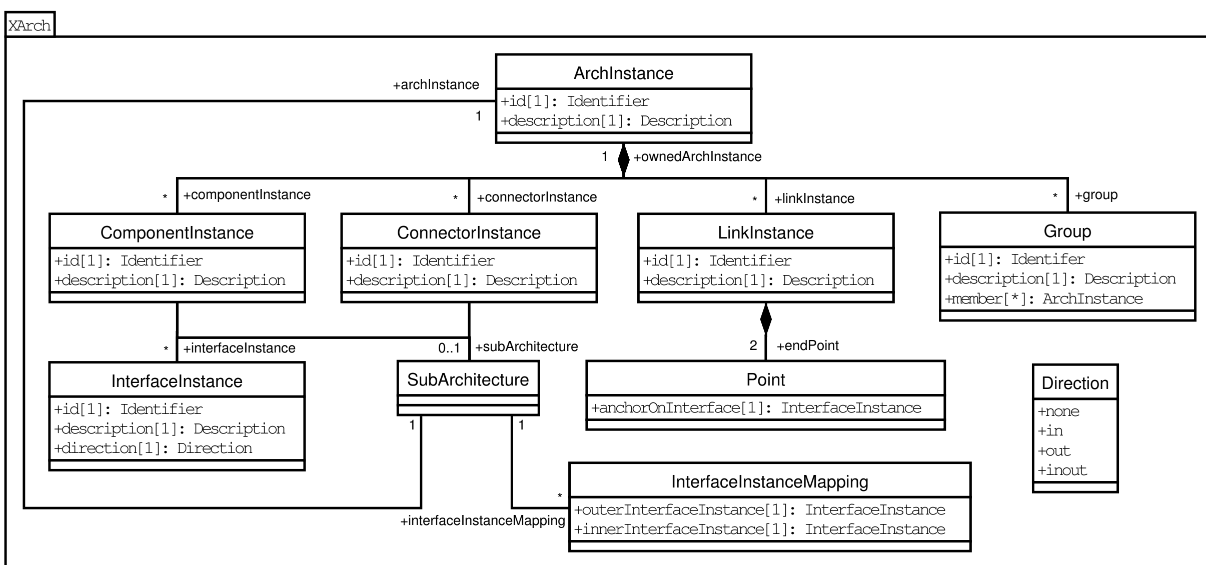

Figure 1: The xArch metamodel

ules) should be less than 5 [8]. This constraint helps to guarantee the maintainability of the architecture and can be formalized as following:

```
invariant forall com:Component in self.Components|
  forall p:Port in com.Ports
    size(select p:Port in com.Ports |
         satisfiesType(p,inputT)) < 5
```

Suppose that we want to transform this architecture model, which conforms to the xAcme metamodel, into a model which conforms to another metamodel. There are two possible cases: i) the targeted technology does not provide a constraint language (CORBA Component Metamodel [10], for example[2]); ii) the target technology provide a constraint language (for example the WRIGHT language [2]).

In the first case, a new language should be provided in order to maintain the architectural decisions throughout the CBSD process. This language should be designed in a way to minimize the transformation cost. In the second, metamodels of constraint languages at different stages of the CBSD process are often different. This is logically due to the distinct nature of concerns at each stage of the development process. Because constraint languages include in their grammars the architectural abstractions to be constrained (for example, Component and Ports, which are keywords of Armani language), the definition of transformation rules between languages at different stages is made complex. But even at the same stage constraint languages can be significantly different. For example, consider the same constraint written using WRIGHT constraint language:

$$\forall c : Components; p : Port|$$
$$p \in Ports(c) \bullet Type(p) = DataInput \wedge \#Ports(c) < 5$$

It is clear that although the two languages have several similar architectural concepts, they differ in the predicate language they use. Transformation of architecture constraints is more difficult than architecture model transformation because we have not only to deal with architectural concepts mapping[3] (as in architecture models transformation) but also with predicate language mappings. Consequently, we propose an approach based on a family of architecural constraint languages, which facilitates the transformation of architectural constraint.

Objects) [4], which is used for object-oriented applications.

[2]In the best of our knowledge, there is no implementation-level architectural constraint language.

3. ACL AS A CONSTRAINT LANGUAGE

To resolve the problem mentioned above, we propose a new architectural constraint language, named ACL. This language is structured on two levels: the first level takes the form of a predicate language, and the second represents the manipulated architectural concepts.

The predicate language chosen is OCL [11]. We slightly modified it to achieve our goals. This part of ACL has been called, the Core Constraint Language (CCL). The second level is represented by a set of MOF-compliant metamodels. Each metamodel defines the architectural abstractions manipulated at a given stage and the relationships between these abstractions. At each stage in the development process a couple composed of CCL and a metamodel, called an **ACL profile**, can be used. For example, we defined an ACL profile for xAcme, UML 2 and CCM.

In the following subsections, first we present CCL and its differences with OCL. Next, we introduce ACL profiles. At last, we illustrate the example specified previously in Armani, in the ACL profile for xAcme.

3.1 Core Constraint Language

Like OCL, CCL is a FOL[4]-based language. It supports set and graph operations, and provides capabilities for specifying navigations in structures. In addition and at the difference of OCL, CCL is characterized by:

- Context declarations with mandatory identifiers. These

[3]In this parer, the word mapping is used to stipulate a relationship between two entities.

[4]First-Order Logic

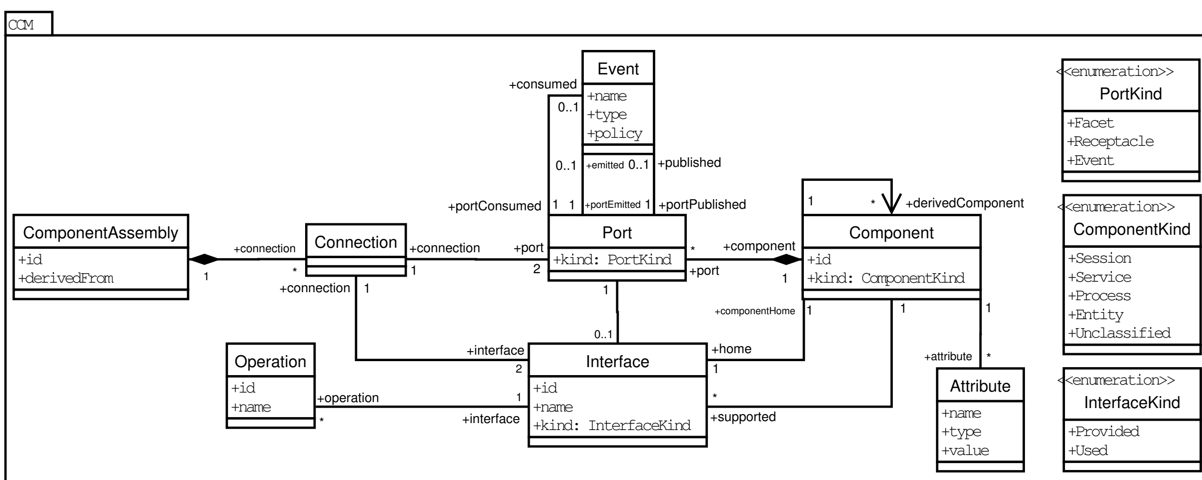

Figure 2: A CCM structural metamodel

identifiers represent the names of specific instances of an architectural abstraction.

- Only invariants can be specified using ACL. Pre- or post-conditions are not supported because not necessary.

- Additional operations and marks have been introduced in order to facilitate the description of certain types of constraints. For example, in order to describe constraints comparing different versions, we have introduced the `@old` mark.

- Constraints navigate on the metamodel, but are evaluated on a specific instance specified in the context.

To better understand the first and last items, it is necessary to go back over 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 the valid object diagrams instanciable from a class diagram. They mitigate the lack of expressivity of the UML graphical notation which, employed alone, can authorize in certain cases the instanciation 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 following are two examples of OCL constraints.

```
context MTProceedingsPaper inv:
self.size <= 8
context paper:MTProceedingsPaper inv:
paper.writtenBy->size() >= 1
```

In both cases, the context is a class (`MTProceedingsPaper`). The two constraints concern any instance of the context (here, an object instance of the class `MTProceedingsPaper`). It is the approach adopted for every OCL constraint. The first constraint references this instance using the keyword `self`, while the second one introduces an ad-hoc identifier `paper`. 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 not on a model, but on a metamodel. A metamodel exposes the concepts of a language and the links between them. It describes an abstract grammar. Thus, a constraint having for context a metaclass limits the expression power of the grammar's production rules and thus the number of the derived phrases (i.e. models). Certain phrase structures are dismissed. If this metamodel describes the abstract grammar of a language dedicated to architecture description, a constraint expresses, then, that only certain architectures (i.e. models) are derivable (i.e. can be instanciated) in this language. The language is restricted voluntarily in its expressiveness because we do not allow the description of certain 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 metaclass `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 limit the scope of such a constraint to a particular component, we propose to slightly modify the syntax and semantics of the context part in OCL. At the syntactic level, we impose that every context introduces an identifier. Furthermore, this identifier must be the name of a particular instance of the metaclass cited in the context. At the semantic level, we interpret the constraint with the meaning it would have in the context of the metaclass but by limiting its scope to the instance cited in the context. Illustrative examples are presented in the next subsection.

3.2 ACL profiles

As mentioned previously, ACL has several profiles. Each profile has two levels of expressiveness. The first level is defined by CCL and the second by a specific MOF metamodel. Each metamodel represents architectural abstractions present in a given Architecture Description Language (ADL) or a component technology, and relationships be-

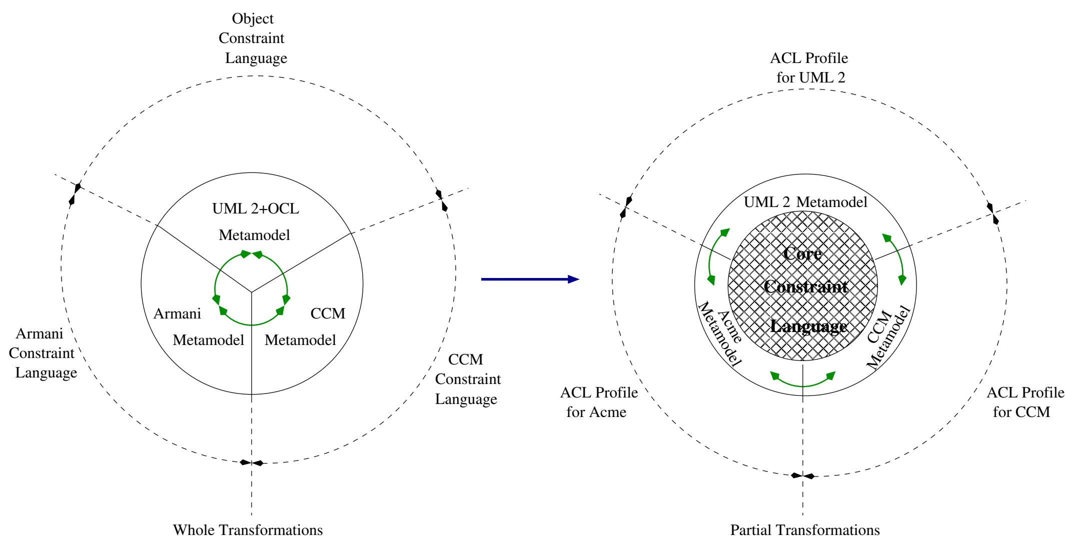

Figure 3: Shifting from whole to partial transformations

tween these abstractions. Each profile can be used, at a given stage in the CBSD process with the corresponding developer's language or technology to describe architectural decisions.

Among the ACL profiles, we present here the profile for xAcme ADL. As stated previously, this ACL profile is composed of CCL and the xAcme metamodel. Note that in our approach, only the structural aspect is represented in the metamodels. That is why we illustrate in figure 1 an xArch metamodel, which contains only the primitive structural elements that compose an xAcme architecture description[5]. An xAcme architecture instance is composed of a set of component instances, connector instances, link instances and logical groups of the previous architectural elements. Component or connector instances define a set of interface instances and optionally a subarchitecture for a hierarchical description. The subarchitecture defines a set of architecture instances and a list of mappings between inner and outer interface instances. Link instances bind two end points, each one references an interface instance.

The constraint presented in the previous section can be specified using xAcme profile as following:

```
context ACS:ComponentInstance inv:
ACS.subArchitecture.archInstance.componentInstance
.interfaceInstance->select(i:InterfaceInstance|
i.direction = 'in')->size() < 5
```

4. PROFILE TRANSFORMATIONS

Let us consider that we transform the xAcme architecture model, to which is associated the previous constraint, into a component description in CCM. Following the proposed approach, the constraint will be transformed to the ACL profile for CCM. As for the other ACL profiles, the CCM one is composed of CCL and the CCM metamodel (see figure 2). The constraint above becomes as follows:

[5]More details about xArch can be found in http://www.isr.uci.edu/architecture/xarch/

```
context ACS:ComponentAssembly inv:
ACS.connection.port.component.port
.interface->select(i:Interface|
i.kind ='Provided')->size() < 5
```

Having CCL in common, the two previous constraints have the same structure and basic predicate constructs with the same semantics (like, `context`, `inv`, `select()` or `size()`. However it navigates in different metamodels.

When transforming xAcme profile to the CCM one, only the part of the constraint related to the architectural aspect needs to be transformed. This is illustrated by the right side of figure 3.

Now that we identified clearly the different parts between the constraints written in the two profiles, the transformation problem can be restricted to translation of the architectural part. In order to resolve this problem, we can use transformation languages, like MTL [13] or TRL [1]. For example, in order to perform the transformation of the constraint defined with the xAcme profile into the CCM profile, we need to have the following mappings:

ComponentInstance ▷ ComponentAssembly
ComponentInstance.subArchitecture
  ▷ ComponentAssembly.connection.port.component
ComponentInstance.interfaceInstance
  ▷ Component.port.interface
InterfaceInstance ▷ Interface
InterfaceInstance.direction ▷ Interface.kind
Direction.in ▷ InterfaceKind.Provided

This information can be derived from possible existing mappings between metamodels. These mappings are supposed to have been used in transforming models and not specially defined for constraint transformation. For example, the first mapping can be deduced from the MTL relation below.

```
Relation ComponentInstance2ComponentAssembly{
  domain {
    (ComponentInstance)
    [id = i, subArchitecture = s]
    when s->notEmpty()
```

```
  }
  domain {
    (ComponentAssembly)
    [id = i,connection = (Connection)
                         [port ={(Port)[// ...
                                       ]}]
    ]
  }
}
```

We have not detailed here all transformations, because of the size of the two metamodels and the paper's page limitations. However, what we showed through this example is that once a constraint transformation is restricted to equivalences between elements of two different metamodels, it exists several means to elaborate a solution.

5. RELATED WORK

In the literature, it exists several works on software architecture transformation [3, 5], whose focus was on architecture refactoring. In these works, transformation of architectural constraints has not been dealt with. In [7], the proposed approach consists in transforming OCL constraints in order to remove redundancies and simplify them. We think that their approach is complementary to ours. It can be used once the constraints are generated in the target ACL profile, in order to simplify them.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a bicephalous architectural constraint language, which clearly separates the invariable part from the variable one in order to simplify constraint transformations. Thus, in an architectural constraint expression we isolated the part that concerns architectural aspect in order to perform the transformation only on this part.

We think that the proposed approach promotes the reuse of mappings between metamodels established for architectural model transformation. Indeed it becomes possible to transform architectural constraints at the same time we transform the concerned models.

Our objective in the short run is to define MTL relations between xAcme, UML 2 and CCM metamodels. Then, we apply our architectural constraint transformation approach between different couples of metamodels. This will validates the proposed approach.

7. REFERENCES

[1] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen Technologies Corp et al.: Response to the MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10), Revised Submission, version 1.0. OMG Document: ad/2003-08-08 (2003)

[2] Abowd, Gregory D. and Allen, Robert and Garlan, David: Formalizing Style to Understand Descriptions of Software Architecture. ACM Transactions on Software Engineering and Methodology, vol. 4, num. 4 (1995) 319–364

[3] Carrière, S. Jeromy and Woods, Steven G. and Kazman, Rick: Software Architectural Transformation. In Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE'99), Atlanta, Georgia, USA (1999) 13–23

[4] Chidamber, Shyam R. and Kemerer, Chris F.: Towards a Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering, vol. 20, num. 6 (1994) 476–493

[5] Fahmy, Hoda and Holt, Richard C.: Software Architecture Transformations. In Proceedings of the International Conference on Software Maintenance (ICSM'00), San José, California, USA (2000) 88–96

[6] 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

[7] Giese, Martin and Larsson, Daniel: Simplifying Transformations of OCL Constraints. In Proceedings of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS/UML 2005), Montego Bay, Jamaica (2005)

[8] Lindvall, Mikael and Tesoriero, Roseanne and Costa, Patricia: Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture. In Proceedings of the Eighth IEEE Symposium on Software Mertrics (METRICS'02), Ottawa, Ontario, Canada (2002) 77–86

[9] Monroe, Robert T.: Capturing Software Architecture Design Expertise with Armani. Technical Report of the School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (2001)

[10] 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)

[11] Object Management Group: UML 2.0 OCL Final Adopted Specification, Document ptc/03-10-14. Object Management Group Web Site: http://www.omg.org/docs/ptc/03-10-14.pdf (2003)

[12] Perry, Dewayne E. and Wolf, Alexander L.: Foundations for the Study of Software Architecture. Software Engineering Notes, vol. 17, num. 4, ACM SIGSOFT (1992) 40–52

[13] Tata Consultancy Services et al.: Revised submission for MOF 2.0 Query / Views / Transformations RFP, version 1.1. OMG Document: ad/2003-08-08 (2003)

[14] 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)

[15] 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)

[16] xAcme: Acme Extensions to xArch. School of Computer Science Web Site, Carnegie Mellon University: http://www-2.cs.cmu.edu/ acme/pub/xAcme/ (2001)