CHAPTER 2

SOFTWARE ARCHITECTURE: ARCHITECTURE CONSTRAINTS

Preamble

In this chapter, we introduce an additional, yet essential, concept in describing software architectures : architecture constraints. We explain the precise role of these entities and their importance in object-oriented, component-based or service-oriented software engineering. We then describe the way in which they are specified and interpreted. An architect can define architecture constraints and then associate them to architectural descriptions to limit their structure, and for ultimately making persistent a certain level of quality. These constraints enable us to enforce adherence to a particular architecture pattern or style so as to ensure a certain level of maintainability. By interpreting these constraints, we are able to check whether these patterns/styles are respected, after the evolution of architecture descriptions. We present a state of the art on the current techniques and languages for expressing these constraints. We then introduce our recent research, where we have developed languages for expressing these constraints on architectures of object-oriented, component-based and service-oriented applications. We will use different examples of architecture constraints representing known patterns and styles, like the *Pipe and Filter* architecture style and the *Service Facade* or *Model-View-Controller* architecture patterns, to illustrate these works. We conclude this chapter with a summary of some of the open questions which have given rise to ongoing research about this concept of architecture constraints.

2.1 Introduction

Since several years, software systems have been ceaselessly evolving, growing in size and complexity. Software architectures therefore play a leading role and have become a central artifact in the life cycle of computer systems, because they provide the various stakeholders with an overview of the organization of those systems. Software architecture is defined in [3] as "the set of structures of a software system, necessary for reasoning about it. It is composed of software entities, the relations between them as well as properties of these entities and relations". This definition brings to the forefront the fact that this artifact makes the components of a software system explicit, as well as the dependencies between these components [1]. This enables us to give a general overview of the organization of this system and reason on it to verify certain properties, like quality attributes.

The activities surrounding software architectures are many and varied. They include, among others, documentation [11], evaluation [12] and reasoning [48]. Of these, the activity which has held the attention of software engineering practitioners considerably these last few years, is architecture documentation. Indeed, in the literature and in practice, a plethora of models, languages and tools have been proposed to document software architectures. This documentation may concern the architecture itself and in this case, we speak of *architecture description*, as it may concern architecture decisions [23, 26, 28, 54] or the (*rationale*) behind these decisions [47].

Documenting architecture decisions is an important activity in software development processes [28]. Indeed, this type of documentation allows for, among other things, limiting the evaporation [6] of architectural knowledge [13]. A multitude of models for defining this type of documentation exists [17]. These models include both textual and (more or less) formal specifications (which can be automatically interpreted by programs). These models include, among other things, the description of the decision itself, its state and its alternative decisions. Among the most important descriptions encountered in the documentation of an architecture decision, we find the architecture constraints.

An architecture constraint represents the specification of a condition which an architecture description must adhere to in order to satisfy an architecture decision. This specification must be defined using a language that facilitates its automated interpretation. For example, an architect may make the decision to use the MVC (*Model-View-Controller* pattern [42]). An architecture constraint allowing the verification of the adherence to this pattern in an architecture description would then consists of checking, among other things, that there are no dependencies between the components representing the model and those representing the view.

This type of constraint does not necessarily have to be expressed in the design phase, accompanying (for example) UML class diagrams. It is quite possible to define them in the implementation phase. Indeed, we can envisage writing and verifying this type of constraint in code, in which we can easily identify the architecture descriptions, as in some object-oriented, component-based or service-oriented applications. Specification of constraints in the implementation phase allows them to be dynamically interpreted, beyond their static verification. It then becomes possible to verify whether they are violated, if the application's architecture evolves during runtime.

1. The term "component" is used in the broadest sense, i.e. an element in the architecture which constitutes a system. They are not components in the sense of component-based software engineering (CBSE).

If we take the case of the architecture constraint verifying a part of the MVC pattern and expressed on an object-oriented application, for example, we can specify a condition stipulating that the objects marked (the classes which have been stereotyped, if we are in UML in the design phase, or annotated, if we are in Java in the implementation phase, for example) as entities of the model must not contain references to the objects marked as entities of the view.

Various languages have been developed to specify this type of constraint. They are mainly used in the design phase and are often associated with architecture description languages [31]. There are, however, some languages which are used in the implementation phase. A state of the art on these different languages is given in the next section of this chapter.

In sections 2.3 to 2.5, we present a language that we have developed, called ACL : *Architecture Constraint Language* [52]. We explore the use of this language in contexts different to that for which it was originally developed, which is that of component-based applications [2] We show how this language could be used for writing architecture constraints on object-oriented applications, described in UML in the design phase, then in Java in the implementation phase. We also explain the use of this language with service-oriented applications, described in the implementation phase with BPEL (*Business Process Execution Language*). All of this is presented in sections 2.3 to 2.5.

We conclude this chapter with a summary of the contribution of this work. We then end the discussion with a presentation of our ongoing research study about this concept of architecture constraints.

2.2 State of the art

In this state of the art, we distinguish two kinds of languages : languages used to specify architecture constraints in the design phase, and which have been jointly proposed with, or directly integrated into, architecture description languages, and languages used in the implementation phase.

2.2.1 Expression of architecture constraints in the design phase

We present architecture constraint expression in the design phase in two steps. First of all, we present the languages and methods for expressing these constraints in architecture description languages (ADLs). Secondly, we show different uses of the OCL language, for specifying this particular type of constraints.

2.2.1.1 Expression of architecture constraints in ADLs In [31], Medvidovic and Taylor propose a classification framework of the architecture description languages developed till 1998-2000. Among the classification criteria, we find architecture constraints. In this article, the authors introduce architecture constraints, introduced in this chapter, as "programmable invariants" specified during architecture configuration modeling. Only certain languages offer this option. These languages are : Aesop [20], SADL [36, 37] and Wright [2].

Aesop allows the writing of "style" or "topology" constraints (also called "configuration rules" by these authors) to force a particular organization of the architecture, so as to

2. Here, the meaning of the term "Component" is that used in component-based software engineering (CBSE).

adhere to an architecture style [44], such as *Pipe and Filter* or "client/server". These constraints are specified in the form of implementations of methods in C++ abstract classes [3] representing the architecture types (*Filter*, *Server*, etc.).

These classes all inherit a programming interface (set of functions), called FAM (*Fable Abstract Machine*). This makes it possible, among other things, to add or remove ports to the components, or to put connectors in place. These classes must be specialized (by sub-typing) to create components, connectors or configurations adhering to the style introduced by these classes. Every description of a new architecture (adding components, ports, connectors, etc.) adhering to a style will pass through the checking of these constraints by invoking the methods which implement them. An example of a constraint in Aesop, taken from [20], is given in the listing below :

```
fam_bool pf_source::attach(fam_port p) {
  if (!fam_is_subtype(p.fam_type(),PF_WRITE_TYPE))
  {
    return false;
  }
  else
  {
    return fam_port::attach(p);
  }
}
```

In this constraint, the port received as a parameter of the `attach` function is verified. If its type is a sub-type of a specific type introduced by the *Pipe and Filter* style, then the connector is established.

SADL is an ADL allowing the specification of constraints (called *Well-formedness Rules* by its authors) enabling adherence to architecture styles. It introduces a syntax for expressing predicates in the first-order logic restricting the types of elements composing an architecture description or style (component, connector, port, etc.). As in Aesop, in SADL, basic types are defined for representing the architecture elements without any constraint. Every new architecture description or architecture style will have to introduce sub-types (by inheritance) with possible constraints. These are verified during interpretation of the instantiation primitives of components and connectors or of their interconnection. A constraint concerning the connectors of the *Dataflow* style, taken from [37], is presented below :

```
connects_argtype_1: CONSTRAINT =
  (/\ x)(/\ y)(/\ z) [ Connects(x, y, z) => Dataflow_Channel(x) ]
connects\_argtype_2: CONSTRAINT =
  (/\ x)(/\ y)(/\ z) [ Connects(x, y, z) => Outport(y) ]
connects\_argtype_3: CONSTRAINT =
  (/\ x)(/\ y)(/\ z) [ Connects(x, y, z) => Inport(z) ]
```

This constraint stipulates that a connection between components in this architecture style involves three architecture elements : a channel (x), which must be of the type `Dataflow_Channel`, a port (y) of the type `Outport` and another port (z) of the type `Inport`.

Wright follows on from the Aesop language, which was developed by the same research team. It enables the formal description of architectures - especially connectors between components in these architecture descriptions. It relies on a process algebra notation, a variant of CSP [24], for modeling the behavior of ports and connectors. This language enables the specification of constraints for formalizing architecture styles. The following

3. Even if an architect writes code in this ADL, it is not considered in the implementation phase. As the architect only writes architecture specifications, it is still considered in the design phase. The implementation of the functionality offered by architecture components is thus not defined with this language.

constraint, for example, stipulates that the configuration must have a star topology :

$$\begin{aligned}
&\exists center : Components \bullet \\
&\quad \forall c : Connectors \bullet \exists r : Role; p : Port \mid ((center, p), (c, r)) \in Attachments \\
&\wedge \forall c : Components \bullet \exists cn : Connectors; r : Role; p : Port \\
&\qquad \mid ((c, p), (cn, r)) \in Attachments
\end{aligned}$$

The first predicate indicates that there is a component (”center”), amongst all the components of the architecture description (the key-word Components) which is attached to all connectors of the description. The second predicate indicates that all the components must be attached to a connector. Hence, this constraint guarantees that every component is connected to the component representing the center of the star.

Besides these languages, the ADL Acme [21] offers a separate constraint language (not cited alongside the others in the Medvidovic and Taylor classification [31]) named Armani [35]. This language allows the expression of what the authors call “design rules”, which correspond to architecture constraints. Armani is a language allowing the writing of predicates in first-order logic. It also introduces among others a number of functions for the verification of the type of an architecture element (satisfiesType(e:Element, t:Type):boolean), or for testing the properties of graphs (for example, attached(con:Connector,comp:Component): boolean). It enables two types of predicates to be defined : “heuristics” and “invariants”. These two entities are defined in the same way, except that heuristics are not intended for type verifications. The example below shows an invariant and a heuristic specified in Armani :

```
Invariant Forall c1,c2 : component in sys.Components |
              Exists conn : connector in sys.Connectors |
                Attached(c1,conn) and Attached(c2,conn);
Heuristic Size(Ports) <= 5;
```

The invariant specifies that all the system components must be connected to each other. The configuration thus forms a strongly connected graph. The heuristic indicates that the total number of ports must be less than or equal to five.

FScript[4] is a scripting language for the reconfiguration of architectures based on Fractal components [8]. It is based on a navigation language in Fractal architecture descriptions, called FPath. This has a syntax inspired by the xPath language, a navigation language for XML documents. This language enables parameterizable architecture constraints to be expressed. An example taken from the tutorial for this language is given below :

```
-- Tests whether the client interface $itf is bound to
-- (an interface of) component $comp.
function bound-to(itf, comp) {
  servers = $itf/binding::*/component::*;
  return size($servers & $comp) > 0;
}
```

The constraint takes the form of a function, i.e. in FScript, the script has no side effects on the architecture description ; the script uses only introspection[5]. This function accepts two parameters : a required interface (client in the terminology of the Fractal component model) and a component. The constraint makes it possible to test if the required interface

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

5. There is another version of the scripts in FScript called “Action”, which allows the modification of the architecture description (realization of the intercession), FScript being an architectural reconfiguration language rather than a constraint language

received in the first parameter (`$itf`) is connected to the component received in the second parameter (`$comp`).

The expression in line 4 makes it possible to get the set of components (stored in a variable `servers`) which have a provided interface (server in the Fractal terminology) connected to the required interface `$itf`. The expression in the following line makes the intersection (operator `&`) between this set of components (`servers`) and the component `$comp` received in the parameter. If the intersection is not empty, the function returns the value "true". Otherwise, it returns "false".

More generally, this language relies on an FPath navigation language to achieve introspection. It provides a number of set operators : intersection (&), union (—), size (`size`), etc. The richness of this language resides in the use of a syntax similar to xPath for writing complex requests, and the possibility of calling functions already previously specified within these requests.

The AADL language [18] (*Architecture Analysis and Design Language*) is an ADL enabling the description of (software and hardware) component-based architectures for embedded and real-time systems. A language named REAL [22] (*Requirements and Enforcements Analysis Language*) was suggested as a constraint language for AADL. REAL makes it possible to express constraints in the form of theorems applying to a collection of architecture elements (components or connections, for example). In the following listing [22], the constraint applies to instances of `Thread`-type components and verifies their periodicity. Their property, the name of which is `Dispatch_Protocol`, must have `periodic` or `sporadic` as a value.

```
theorem task_periodicity
foreach t in Thread_Set do
  check((Get_Property_Value(t,"Dispatch_Protocol") = "periodic")
        or
        (Get_Property_Value(t,"Dispatch_Protocol") = "sporadic"));
end task_periodicity
```

The constraints in this language use an introspection mechanism to, among other things, obtain the set of component instances of type `Thread`, `Data` , etc., to obtain the values of their properties, to test access or connections between instances of components (`Is_Accessing_To(...)`, `Is_Bound_To(...)`, `Is_Subcomponent_Of(...)`, etc.). This language proposes iterators (`foreach`) and set operations (`Cardinal`, `Max`, etc.) and Boolean and comparison operators.

2.2.1.2 Expression of architecture constraints with OCL

The OCL (*Object Constraint Language*) language [41] is the OMG (*Object Management Group*) standard for expressing constraints on UML models. The goal of this language is to provide developers with a means of specifying conditions for refining the semantics of their models. This constraint language was initially suggested for specifying conditions on functional, not architectural, aspects. For example, in a class diagram, where we find a class `Employee`, having an attribute `age` of type integer, an OCL constraint representing an invariant on this class can force the values of this attribute so that they always fall in the interval 16 to 70. This constraint will be verified on all instances of the UML model, and therefore on all instances of the class `Employee`.

There are, however, other uses of the OCL language, which are at the meta-model level and not at the model one. This allows the expression of architectural constraints like those discussed in this chapter. Below, we list some examples of specifications in which the OCL language is used to express architecture constraints.

1. Specification of the UML language [40] : in this specification, the meta-model of the UML language is introduced, and OCL constraints are associated with it, in order to refine the semantics of this meta-model. These constraints navigate through this meta-model and impose their conditions on meta-classes in the meta-model, the values of their attributes, the number of their instances, their interconnections, etc. In Figure 2.1, taken from [40], we show a small excerpt from the UML meta-model, in which the associations between classes are partially specified.

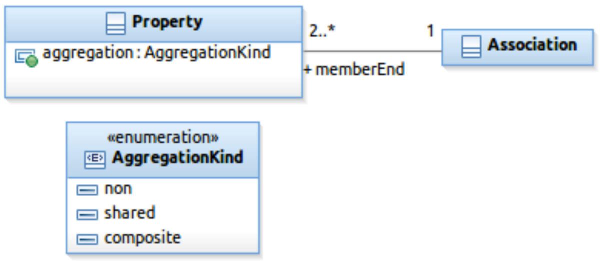

FIGURE **2.1** Excerpt from the UML meta-model

This figure shows that an association has two or several member ends, which are instances of the `Property` meta-class. A property has an attribute `aggregation`, which can take the following three values : `none` (no aggregation), `shared` (aggregation in UML) and `composite` (composite in the sense of UML). Here is an example of a constraint [40] written in OCL on this fragment of the meta-model.

```
1 -- Only binary associations can be aggregations
2 context Association inv:
3 self.memberEnd->exists(aggregation <> Aggregation::none)
4 implies self.memberEnd->size() = 2
```

The first line is a comment indicating the role of the constraint. The line 2 denotes the constraint context. This represents the meta-class in the meta-model on which the constraint is applied. Hence, the constraint will be evaluated on all instances of this meta-class. In the example, we are indicating at the `Association` meta-class shown in Figure 2.1. In line 3, the constraint navigates across the association between the `Association` and `Property` meta-classes in the meta-model to obtain the `Property` instances linked to the `Association` instance on which the constraint is evaluated. The constraint then checks if there is at least one instance among these `Property` instances whose `aggregation` attribute value differs from `none` (to check if there is at least one association which is an aggregation or composition). In the next line, the constraint checks that, in this case, the number of member ends of the association is equal to two (binary association).

2. UML profiles : a UML profile is a standard extension to the UML language for handling a particular domain - real-time systems, telecommunications, systems on a chip, system tests, etc. [6].

   Let us take the example of the UML profile for CORBA and CORBA components (CCCMP [39]). An excerpt from the meta-model implemented by this profile is given in Figure 2.2.

6. See a complete list of UML profiles adopted by the OMG on the following website : www.omg.org/technology/documents/profile_catalog.htm.

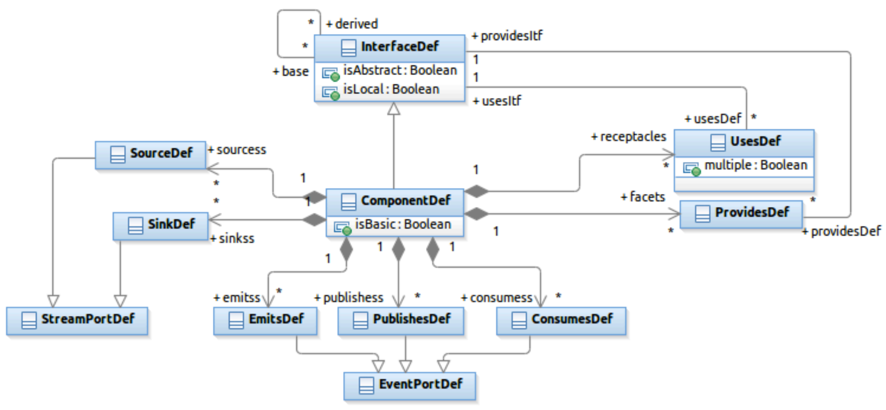

FIGURE **2.2** Excerpt from the meta-model of the UML profile for CORBA and CCM

In this meta-model, it is specified that a component definition declares a certain number of ports, which can be *receptacles* (required ports of type `UsesDef`), *facets* (provided ports of type `ProvidesDef`), event ports (`EmitsDef`, `PublishesDef` and `ConsumesDef`) or flow ports (`SourceDef` and `SinkDef`).

The following constraint stipulates that a basic component definition does not have to declare ports or inherit other component definitions :

```
1 context ComponentDef inv:
2 self.isBasic implies
3 facets->isEmpty and receptacles->isEmpty and
4 emitss->isEmpty and publishess->isEmpty and consumess->isEmpty and
5 sinkss->isEmpty and sourcess->isEmpty and
6 base->isEmpty
```

Lines 3 to 5 indicate that ports should not have as a type one of the types cited above. The last line specifies that the component definition does not have to be derived from another component definition (the `base` role being inherited from the `InterfaceDef` meta-class). In this case, the basic Corba component definition can only have a `HomeDef` interface being able to declare operations of type `FactoryDef` or `FinderDef` (which are not shown in the meta-model in Figure 2.2 for simplicity's sake.)

3. Software architecture description in UML is proposed by Medvidovic *et al.* in [32] : the authors introduce three different methods for using UML as an architecture description language in this article. The first method consists of using UML as it is. The second recommends the establishment of architecture constraints in OCL on UML meta-classes. The third method suggests extending the UML language. The second method is the most interesting for us to present here. The authors present a set of OCL constraints applicable on the UML meta-model, to adhere to the restrictions imposed by some ADLs in component-based architecture description. The authors have chosen three ADLs that they have considered suitably representative, which are C2 [30], Rapide [29] and Wright. More specifically, they have presented the UML profiles of these ADLs, because they have simultaneously introduced the constraints, stereotypes and tagged values which correspond to these ADLs. For the C2 language, for example, the authors introduce a stereotype named `C2Component` as an extension of the Class meta-class. An architecture constraint attached to this stereotype

indicates that the C2 components must implement exactly two interfaces, which must be stereotyped `C2Interface`, one of them positioned above the component and the other below. This constraint is expressed in OCL as follows : [32] :

```
1 self.interface->size = 2 and
2 self.interface->forAll(i | i.stereotype = C2Interface) and
3 self.interface->exists(i | i.c2pos = top) and
4 self.interface->exists(i | i.c2pos = bottom)
```

The authors introduced an enumeration beforehand (used in lines 3 and 4) for the position of interfaces (`top` and `bottom`) , and the stereotypes `C2Component`, which is the context of this constraint, and `C2Interface` used in line 2.

It should be noted here that the constraints defined in this way, at the meta-model level, apply to all instances of that meta-model, and therefore on all models (architecture descriptions) defined using this meta-model. These constraints thus represent quite strong conditions, being a part of the architecture description specification of the language. They will apply to every architecture description defined with this language. Therefore, these are not architecture constraints which will be verified on a particular architecture description (that of a specific application).

2.2.2 Expression of architecture constraints in the implementation phase

In the implementation phase, the developer is faced with two scenarios : i) manually writing the code (programs) corresponding to the architecture described in the design phase, or ii) automatically generating code skeletons from the architecture descriptions and then filling in the missing parts. Everything depends on the language used in the design phase and the tools associated therewith. In an ideal scenario, the constraints accompanying the architecture description are themselves transformed into code, or into constraints which can be verified on the code. Unfortunately, to the best of our knowledge, no work in the literature has been developed on this subject. Recently, we were inclined to answer this question (see the conclusion in section 2.6), but the work that we undertook does not fall within the subject matter of this chapter and therefore will not be detailed here.

In this section, we focus on the works which have suggested languages and tools for expressing constraints on programs. It should be noted here that architecture constraints sometimes involve quite a fine granularity in the architecture description, like attributes' access modifiers or method parameters. This is related to the architecture style underlying the programming language. For example, object-oriented programming languages offer an architecture style for applications constituted of concepts such as classes, prototypes, attributes, methods, etc. Architecture constraints on object-oriented applications must therefore necessarily involve this level of granularity (on these concepts and their properties). These constraints do not arise at the functional level (constraints on attribute values, for example). They arise rather at the structural, and therefore architectural, level of these applications.

Boris Bokowski in [5] suggested a *framework* called *CoffeeStrainer*. This *framework* facilitates constraints specification in Java and their static verification in Java programs. It enables condition expression in programs in order to implement good design practices, such as encapsulation or systematic invocation of a superclass method when this method is redefined in a sub-class. These constraints are defined in the form of particular comments (surrounded by the set of symbols `/*/`) which can be placed anywhere in a class. Within these comments, methods are defined for implementing the constraint using a reflexive layer (for meta-programming) provided by CoffeeStrainer. These methods are invoked whilst analyzing the syntactic tree representing the class in which the constraint has been defined, and therefore on which it is going to be verified. For example, the following constraint enables us to check that all the attributes of a class are private (data encapsulation principle in object-oriented programming).

```
1 interface PrivateFields {
2   /*/ public void checkField(Field f) {
3     if(!f.isPrivate()) {
4       reportError(f, ”field is not declared private”);
5     }
6   }
7 /*/
8 }
```

If a class implements the interface in the previous listing, the `checkField(Field f)` (line 2) method will be invoked as many times as there are attributes in the class, by passing it as an argument each time the `Field` object representing the attribute. This constraint reports an error by calling an inherited method known as `reportError(...)`, if one of the attributes is not private.

CoffeeStrainer can allow quite a fine level of detail to be considered when expressing constraints , e.g. in checking the type of instructions which are in methods (is it a method invocation, an assignment, etc. ?). This level of detail is not useful for expressing architecture constraints which involve aspects of a coarser granularity (declared attributes and methods, extended classes, implemented interfaces, etc.). To achieve this, `java.reflect` is quite sufficient. We will return to this point later in this chapter.

An older language, called CCEL (*C++ Constraint Expression Language*), is suggested in [10] for constraint specification involving the structure of C++ programs. Constraints expressible with this language only involve declarations in programs (declarations of classes, functions and variables), corresponding to the type of constraints that we would like to express. However, this language introduces a new syntax, inspired by C++, but which requires specific learning.

In [27], the authors suggest a language called CDL (*Category Description Language*), which enables to express architecture constraints in first-order logic as formulas involving trees representing program syntax (*parse trees*). This language allows constraint specification independently of all languages. However, in order to integrate this language into a concrete programming language, this language must be extended to enable the annotation (with the names of constraints) of the programs written with the target language on which the constraints must be verified.

DCL (*Dependency Constraint Language* [49]) is an architecture constraint specification language for object-oriented applications. This language enables the expression of constraints which are statically verified - i.e. checked, on the source code of object-oriented applications, before their execution. *dclchek* is the tool provided by the authors of this language for checking DCL constraints on Java code.

Firstly, this language allows to indicate the parts of the object-oriented application which represent modules (a module = a set of classes and of interfaces). Next, the constraints specify conditions that these modules must adhere to. The authors of this language indicate that two categories of constraints can be specified : “divergences” and “absences”. By “divergence”, the authors mean that the source code of an object-oriented application contains a dependency which does not adhere to the constraint. By “absence”, the authors mean that the source code does not contain a dependency. “Divergences” can be of two kinds :

1. constraints such as : “Only classes in module A can depend on the types in module B”. By the word “depend”, the authors mean, access (a class of module A can access the public members of a class of module B), declare, create, implement, inherit, etc. ;
2. constraints such as : “Classes of module A cannot depend on the types in module B”.

Some examples of constraints are given below :

```
1 only A can−access B
2 only A can−declare B
3 only A can−useannotation B
4 A cannot−create B
5 A cannot−implement B
6 A cannot−throw B
```

The constraint in line 2 indicates that only classes in module A can declare variables of types defined in module B. By contrast, the constraint in line 6 stipulates that the classes in module A cannot throw exceptions of the types defined in module B.

“Absences” are constraints such as : “Classes declared in module A must depend on types declared in module B”. Some examples of this category of constraints are shown below :

```
1 A must−extend B
2 A must−throw B
3 A must−useannotation B
```

The first constraint shows that classes declared in module A must extend a class declared in module B. The second constraint specifies that all the methods of the classes in module A must throw exceptions, the types of which are declared in module B . The last constraint imposes the use of at least one annotation declared in module B in all classes in module A.

Expressing architecture constraints is quite limited in DCL. Indeed, with this language we can impose conditions on the dependencies between types, but it is ,for example, impossible to write constraints limiting the number of architecture elements (e.g. number of instances, attributes or operations). This necessitates complex logical compositions or even requires a dynamic program analysis (e.g. to check if the value of an attribute does not correspond to a reference to an object of a certain “forbidden” type).

A number of works in the literature refer to architecture constraints as conditions on the structural dependencies between program elements. SCL (*Structural Constraint Language* [25]) is a constraint language for predicate specification in first-order logic. This language enables the analysis of the syntax of programs, which is represented as a graph, via a number of operations. For example, the operation `subclasses(class("X"))` returns the set of sub-classes of class “A”. The following example, taken from [25], introduces a constraint checking that the `equals` methods in Java classes have a correct signature :

```
1 def Object as class(”java.lang.Object”)
2   for p: packages, c: classes(p), m: methods(c)
3   (
4     (name(m)=”equals” & isPublic(m) & sizeof(params(m))=1)
5     =>
6     (returnType(m) = boolean & type(ith(params(m),0))=Object)
7   )
```

This constraint iterates on all packages, and for each of the methods in the classes of these packages (the `for` loop in line 2), if the method is public, is called `equals(...)` and accepts an argument (line 4), it must have `boolean` as the return type and its parameter must be of type `Object` (line 6).

The designers of this language suggested a sort of simplification of the OCL language, but given as a new language having its own syntax. For example, the `for` loop in line 2 of the above listing is written more verbosely in OCL by nesting `forAll` operations. They also suggested a direct adaptation of this language to the procedural and statically-typed object-oriented programming languages. To analyze the source code of programs, operations are directly integrated into the language, like `isPublic(...)` or `methods(...)`. These operations could have been introduced as navigation in the meta-model of the programming language that the analysis concerns, as we introduced them in the previous section on OCL language, or as will be introduced in the next section of this chapter. This has the benefit of rendering the constraint language independent of a particular programming language, and parameterizable by the latter. The OCL language provides this possibility and has the benefit of being a standardized language, well-known, having a strong tool support and easy to learn and to use [7].

Other languages proposed in the literature can be grouped together into one family ; that of the languages stemming from Prolog. For example, LogEn [14] (*Logical Ensembles*) is a language for expressing ensembles of elements, components of programs, and constraints on the dependencies between these sets. This language is based on a formalism *a la* Prolog, called DataLog [9], allowing programs to be represented in the form of relations as follows :

```
type(t1, 'bat.type.ObjectType')
type(t2, 'bat.type.IType')
interface(t1, t2)
```

In line 1, we have a type declaration whose identifier is `t1` and which is called `bat.type.ObjectType`. In line 3, the relation shows that t1 declares to implement the interface whose identifier is `t2` (declared in the line above).

Typically, constraints imposed by design patterns [19] are expressible using this language. This is done first by indicating the different roles in a pattern by sets, then specifying the constraints of belonging (or not) to these sets, relying on logic operators of conjunction or disjunction. For example, for the pattern *Factory*, the authors specify three sets representing :

- all the program elements of a certain application ;
- the `Factory` class (this set has the following identifier in the listing below : `TypesFlyweightFactory`) ;
- the class constructors whose objects will have to be created with the `Factory` class (this set has the following identifier : `TypesFlyweightCreation`).

This is done by using `part_of(...)` relations, as shown above. Next, the *Factory* pattern constraint imposes the creation of objects of a certain type, defined as follows through the `Factory` class :

```
violations(S, T, 'TypesFlyweight'): –
  part_of(T, 'TypesFlyweightCreation'),
  tnot(part_of(S, 'TypesFlyweightFactory')).
```

This constraint checks that all the S and T pairs are in `uses(S,T)` relation (i.e. dependent on one another), T belongs to the constructor set and S is not the `Factory` class. In other words, T (a constructor) must be used exclusively through T (the `Factory` class).

An interpreter of this language has been integrated into the incremental compilation/build process in Eclipse, to permanently check the constraints whenever the developers modify the source code of their programs.

The main benefits of this work are the interesting performances obtained during constraint interpretation (efficiency of set creation and verification of dependencies) and the incremental nature of the approach (its integration into Eclipse within its incremental compilation/build process). On the other hand, the authors focus exclusively on the structural dependencies between program elements (methods, attributes, super-classes, etc.). Constraints focus on the verification of the presence or absence of a dependency between these program elements. Nonetheless, this language suffers from a lack of expressiveness. Indeed, complex constraints involving complex navigations are expressed with difficulty in LogEn. *Law Governed Architecture* [33, 34] is a similar approach, based on Prolog, for expressing and checking constraints of dependencies between programs written in Eiffel.

In [4], the authors present an approach which proposes a separate language based on Prolog, called Spine, for writing design patterns in the form of constraints. As in LogEn, a constraint uses relations like `constructorsOf(...)` or `isStatic(...)`. The following example [4] shows the *Public Singleton* design pattern :

```
1  realizes('PublicSingleton', [C]) :-
2    exists(constructorsOf(C),true),
3    forAll(constructorsOf(C),
4      Cn.isPrivate(Cn)),
5    exists(fieldsOf(C),F.and([
6      isStatic(F),
7      isPublic(F),
8      isFinal(F),
9      typeOf(F,C),
10     nonNull(F)
11   ]))
```

In this constraint, we check that the `C` class has at least one constructor (line 2), that all its constructors are private (lines 3 and 4), and that it has at least one `final` attribute which is static and public whose type is class C, and its value is not null (lines 5 to 10).

This language exclusively focused on design patterns. Based on a formalism in Prolog, it has the same limitations as the LogEn language.

In practice, there are several static code analysis tools enabling the verification of architecture constraints. A non-exhaustive list of these tools includes : Sonar, Checkstyle, Lattix, Macker, Classycle, Architexa and JArchitect. These tools vary considerably in the functionalities that they implement. Some provide developers with the means to write constraints representing architecture styles or design patterns, by relying on a syntactic tree of programs (like Checkstyle). Others are limited to the specification of restrictions of dependencies between modules (like Lattix). They propose different notations, from program writing (as in Checkstyle) to the definition of specifications in a declarative (in XML, as in Macker) and/or graphic (as in Sonar) fashion. These tools allow constraints to be checked at different levels of the development process, for example, during programming (and therefore relying on just-in-time compilation in Eclipse among others), during *commit* in SVN version management systems (like Checkstyle), or during project building with tools like Ant and Maven (like Macker).

The last, but not least, family of languages is composed of languages which offer the possibility of writing meta-programs (also called reflexive languages). In these meta-programs, the developer has the possibility to access (introspection or intercession) the structure of programs which are reified in the form of objects (in reflexive object-oriented languages) or components in component-based languages, like Compo [46]). Java is an example of an object-oriented programming language which mainly offers the capability of introspection and Smalltalk is an example of an object-oriented programming language which offers introspection and intercession. For architecture constraint specification, we only need introspection, as architecture constraints analyze only the structure of programs, and have no side effects. Sometimes, architecture constraints should be dynamically evaluated, i.e. at the execution of programs on which the constraints are verified. This is unnecessary for some categories of architecture constraints, in which a static analysis of the architecture is enough. In some cases, e.g. the MVC pattern, shown in the next section), the constraint must verify the types of instances created at execution and their interconnections. It must therefore be dynamically evaluated.

In the next sections, we will present some studies that we have conducted in the last few years, on architecture constraint specification. We will show how we are able to exploit current languages, well-known by developers, such as OCL and Java, to express this type of constraints on applications written in various paradigms. We have chosen the three paradigms which are the most well-known in the software industry : the object, Componentcomponent and service paradigms.

2.3 Architecture constraints on object-oriented applications

Architecture constraints expressed on object-oriented applications enable the specification of (among other things) conditions imposed by architecture styles, design patterns, dependencies between types, or any kind of invariant involving the checking of the architecture of an application and not its state. In the rest of this section, we firstly show the specification of these architecture constraints on object-oriented applications in the design phase, then in the implementation phase.

2.3.1 Architecture constraints in the design phase

We have here chosen to show constraints expressed using the ACL language [52], on design models of object-oriented applications written in UML. ACL is a simplification of the OCL language. In ACL, only invariants can be expressed, i.e. it is not possible to define the other kinds of constraints `pre`, `post`-conditions, `init`, `derive` or `body`, which are possible in OCL. Moreover, in the context of the constraint, the used identifier represents the architecture description on which the constraint is applied. The identified element type is a meta-class in a meta-model. The navigation starts from this meta-class to analyze an architecture description in order to specify the constraint. We give some examples below to better illustrate our discussion.

Figure 2.3 shows an excerpt from the UML meta-model, obtained from the UMLlanguage superstructure specification, version 2.4.1 [40].

This meta-model focuses on describing classes, and more particularly packages, attributes, dependencies and profiles. A package is composed of a number of types (see in the center of Figure 2.3). By means of the `PackageableElement` and `NamedElement` meta-classes, classes inherit the ability to participate in dependencies (top-right of the figure). On the bottom-right of the figure, it is shown that a class can declare attributes which are `Property` instances. A class also inherits from `Classifier` the fact that it can have inherited or imported attributes (association between `Classifier` and `Property`). The left-most part of the figure illustrates the fact that we can apply a profile to a package, and that a profile is composed of a number of stereotypes.

We take the example of a constraint representing the MVC pattern. The dependencies between the different entities composing this pattern are illustrated in the diagram in Figure 2.4. To simplify, we have used UML packages to represent groupes of classes that play the three roles in this pattern (the model, the view and the controller). However, this does not necessarily have to reflect the application design (the classes of the view and those of

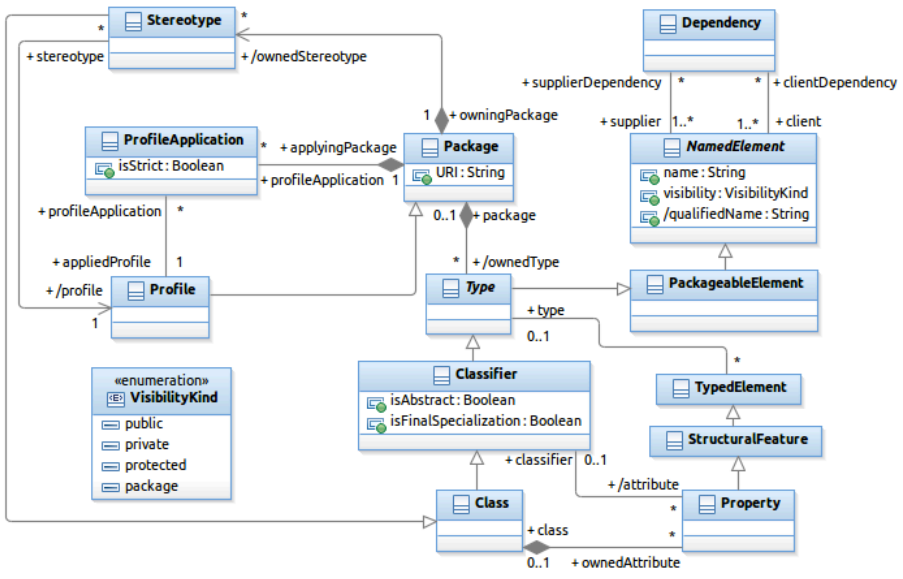

FIGURE 2.3 (Package, Property, Dependency and Profile) from the UML meta-model

the controller can be grouped in one package, or distributed over more than two packages). We assume thus that we have three stereotypes, allowing us to mark the classes in an application which represent the view (*View*), the model (*Model*) and the controller (*Controller*).

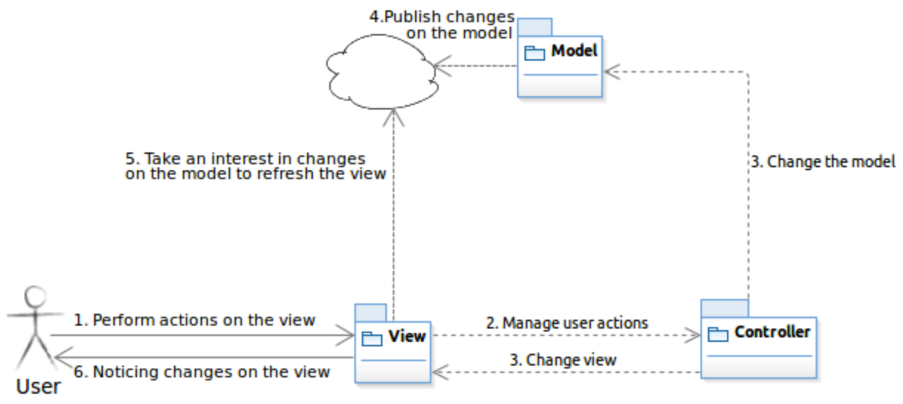
4.Publish changes on the model
Model
5. Take an interest in changes on the model to refresh the view
3. Change the model
1. Perform actions on the view
View
2. Manage user actions
Controller
6. Noticing changes on the view
3. Change view
User

FIGURE 2.4 MVC pattern illustration

This constraint is composed of three sub-constraints, which will check that :

– the classes stereotyped *Model* do not have to declare dependencies with the classes stereotyped *View*. This makes it possible to have, among other things, several views for the same model, and thus to uncouple them. In most MVC implementations, the model classes do not depend directly on the view classes, but rather declare an attribute having as a value a collection of objects that listen to model changes. The view can perform the role of listener for modifications which have been performed on the model, so that it can update itself (be refreshed) ;

– the classes stereotyped *Model* should not have dependencies with the classes stereotyped *Controller*. This makes it possible to have several possible controllers for the model ;
– the classes stereotyped *View* should not have dependencies with the classes stereotyped *Model*. The controllers will play the role of "intermediate" between the view and the model.

Using ACL, we obtain the following constraints :

```
 1 context MonApplication:Package inv:
 2 let model:Set(Type) = self.ownedType->select(t:Type |
 3   t.getAppliedStereotypes()->exists(name='Model'))
 4 in
 5 let view:Set(Type) = self.ownedType->select(t:Type |
 6   t.getAppliedStereotypes()->exists(name='View'))
 7 in
 8 let controller:Set(Type) = self.ownedType->select(t:Type |
 9   t.getAppliedStereotypes()->exists(name='Controller'))
10 in
11 self.ownedType->forAll(t : Type |
12   (model->includes(t)
13   implies
14   t.clientDependency.supplier->forAll(tt |
15     view->excludes(tt) and controller->excludes(tt)
16   ))
17   and
18   (view->includes(t)
19   implies
20   t.clientDependency.supplier->forAll(tt |
21     model->excludes(tt)
22   ))
23 )
```

The first line in the listing declares the context of the constraint. It indicates that the constraint is applied to the whole application package ; the meta-class Package (see figure 2.3) is then the starting point for all navigations in the rest of the constraint. Lines 2 to 9 serve to collect together the sets of classes representing the model, the view and the controller [7]. For example, in lines 2 and 3, we move on from the package to looking for the types defined in it. Next, we select only those which have `Model` as an applied stereotype, thanks to the operation `getAppliedStereotypes()` (not specified in UML/OCL, but implemented in IBM's RSA : *Rational Software Architect*).

Sub-constraints 1 and 2, textually specified in the previous enumeration, are formalized using ACL in the previous listing, between lines 11 and 16. In these constraints, we ensure that if a class is stereotyped as Model, then it must not have dependencies (by navigating to the `Dependency` meta-class) with classes stereotyped View or Controller. To test if a class is stereotyped as Model, we simply check its presence, thanks to the `includes(...)`) operation at line 12 in the set of objects of type `Class`, named `model`, defined in lines 2 and 3 of the listing. The last sub-constraint is formalized in lines 18 to 22.

Often, a dependency between two classes is translated as : i) the declaration in the first class of at least one attribute having as a type the second class ; ii) some parameters in operations of the first class, which have as a type the second class ; iii) in an operation of the first class, the return type is the second class. To simplify, we will take into account the first case : "at least one attribute in the first class has as a type the second class". It is,

7. We simplify the constraint here by assuming that the classes are found in one single package. If the classes are defined in sub-packages or sub-sub-packages, it will then be necessary to navigate recursively in these sub- or sub-sub- packages.

moreover, most often the case during the development of an application in accordance with the MVC pattern. The constraint previously given can be refined as follows, by supposing that the part from line 1 to 10 in the previous listing does not change :

```
...
self.ownedType->forAll(t : Type |
  (model->includes(t)
  implies
  if t.oclIsKindOf(Classifier)
  then
    t.oclAsType(Classifier).attribute->forAll(a |
      view->excludes(a.type) and controller->excludes(a.type))
  else true
  endif
  )
  and
  (view->includes(t)
  implies
  if t.oclIsKindOf(Classifier)
  then
    t.oclAsType(Classifier).attribute->forAll(a |
    model->excludes(a.type))
  else true
  endif
  )
)
```

In this constraint, the dependency is verified on all attributes defined in classes. Note the use of the `oclAsType(Classifier)` operation in this constraint to allow navigation by going through specialization relations between `Type` and `Classifier` (static type conversion of `Type` objects) for navigating then to `Property`.

We have not yet finished with the MVC pattern, which can only be partially verified in the design phase. Indeed, other checks will have to be run at execution. These are explained in the next section.

2.3.2 Architecture constraints in the implementation phase

In order to be able to check the previous architecture constraints in the implementation phase, and thus to check that the code produced in this phase always conforms to the constraints defined before, we will explain how to express these constraints in Java.

At first, we will show these constraints expressed in ACL, but this time on the Java meta-model, to ensure a smooth transition from the previous section. Next, we show how to specify these same constraints, by using the introspection mechanism provided by Java.

Figure 2.5 shows a simplified meta-model of the Java language. By simplified, we mean to say that this meta-model only shows entities in the Java language serving to write architecture constraints. We thus find classes, which have attributes (`Field` in Java terminology), methods and constructors. A class belongs to a package. This meta-association is navigable in only one direction. That is to say that from one package, we cannot know which types are defined there. All these elements can be annotated (a property inherited among others from the meta-class `AccessibleObject`). Except packages, the other meta-classes have `modifiers`, which can have different values listed in the enumeration named `Modifier`. An attribute can have a reference towards another object as its value for a particular object. It should be noted that the semantics of the meta-model are not precise enough on this point. Indeed, the association between `Field` and `ObjectReference` only has to be navigable when we go, in a constraint, from an object (instance of the meta-class `Object`), then towards its class (instance of `Class`), and finally towards the attribute (instance of `Field`). If the navigation starts from `Class`, then the access to

`Field` must not allow access to `ObjectReference`. This meta-model was constructed on the basis of classes defined in the `java.reflect` API whose methods enable the introspection of Java objects. The lack of semantics raised above is linked to the fact that on the one had, in Java there is no equivalent of UML's `Slot`. On the other hand, the `get(...)` method of the `Field` class returns the value of the attribute, but we have to pass it, as an argument, the object for which the value of the "attribute" (slot) must be returned. In reality, this is a more general problem. It is due to the fact that in Java, there is no true coupling between the objects and their meta-objects (instances de `Class`). Once we invoke `getClass()` on an object, we obtain the meta-object, but in this meta-object, there is no reference back to the object (we therefore lose access to the values of its "attributes").

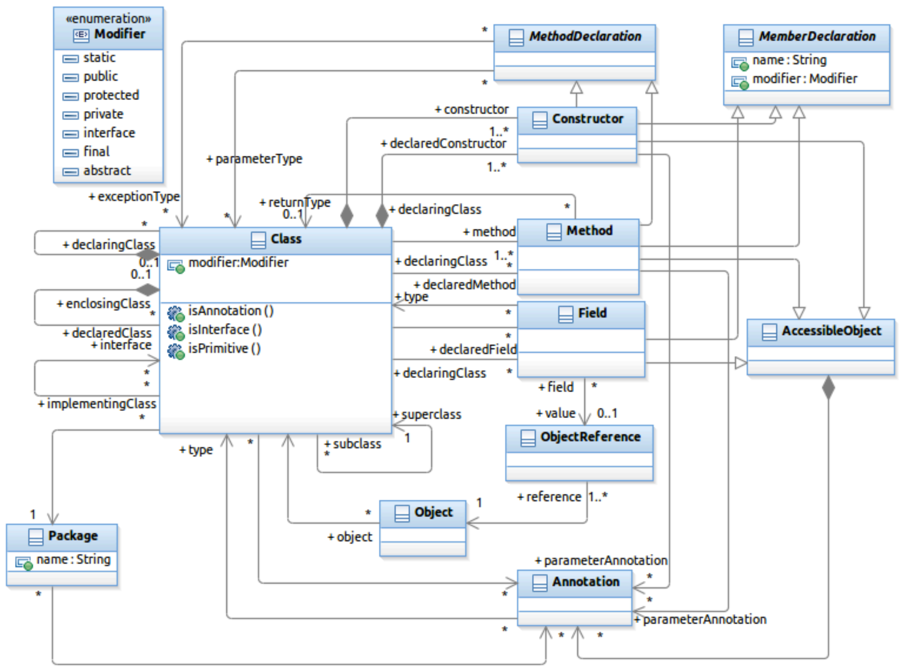

FIGURE 2.5 Excerpt from the Java meta-model

The sub-constraints of the MVC pattern introduced in the previous sub-section can be expressed on this meta-model in the following manner :

```
context Class inv:
self.annotation->exists(a:Annotation | a.type.name='Model')
implies
self.field->forAll(f : Field |
  not (f.type.annotation->exists(type.name='View')
  or f.type.annotation->exists(type.name='Controller')))
and
self.annotation->exists(type.name='View')
implies
self.field->forAll(not (type.annotation
  ->exists(type.name='Model')))
```

Here we assume the existence of three annotations, applicable on types (classes) and present at execution (`@Retention(RetentionPolicy.RUNTIME)`), corresponding to the three stereotypes previously presented. The context of the constraint cannot be the package of the entire application. As we specified previously, we unfortunately cannot navigate to the types which are defined in a package. We have therefore specified `Class` as the context of the constraint. The constraint must therefore be verified on all classes of the application. The navigation in the Java meta-model, shown in figure 2.5, is quite simple. We analyze the attributes here (objects of type `Field`) of the class, then their types and the annotations applied to them.

The constraint expressed in ACL above can be implemented in Java as follows :

```java
// We assume here that the classes of the application
// have been already loaded in a certain manner
public boolean invariant(Class<?>[] classesMyApplication) {
  for(Class<?> aClass : classesMyApplication) {
    if(aClass.isAnnotationPresent(Model.class)) {
      Field[] attributes = aClass.getDeclaredFields();
      for(Field anAttribute : attributes) {
        if(anAttribute.getType().isAnnotationPresent(View.class) ||
        anAttribute.getType().isAnnotationPresent(Controller.class))
          return false;
      }
    }
    if(aClass.isAnnotationPresent(View.class)) {
      Field[] attributes = aClass.getDeclaredFields();
      for(Field aAttribute : attributes) {
        if(anAttribute.getType().isAnnotationPresent(Model.class))
          return false;
      }
    }
  }
  return true;
}
```

The method `invariant(...)` accepts as parameters objects of type `Class`, representing each of the application classes [8]. Unfortunately, we will be unable to start navigation from the `Package` object representing the application package, because in java.reflect, this object does not enable to obtain references to the classes which are declared inside it. The `Package` object relates to a simple object containing information about the package (e.g. its name).

It should be noted here that unlike approaches for static code analysis, constraint verification and thus execution of this method, necessitates loading of the entire application by the class loader, in order to obtain the `Class` objects representing the different application classes, before putting them into an array which is passed as an argument during the invocation. In this chapter, we focus on architecture constraint specification and not on constraint verification.

Let us now return to a point raised during the introduction of the different sub-constraints, which formalize the restrictions on dependencies imposed by the MVC pattern. In the first sub-constraint, we have shown that the model classes do not have to declare dependencies with the view classes ; which makes it a constraint on the static types. However, according to the implementations of this pattern, we may find ourselves with a reference to a view object in a model object at execution. Indeed, what was illustrated by a cloud in Figure 2.4 can be implemented by the `Observer` pattern. In this case, a model object stores a collection of objects listening to changes on the model (the collection can be statically typed by an interface named, for example, `ModelModificationListener`). At execution, however, this collection will include view objects, whose classes implement

8. By “application classes”, we mean the classes which compose the application’ business domain. This excludes classes of the libraries used by the application.

ModelModificationListener, the previous interface. Therefore, dynamically we find ourselves with dependencies between the model classes and the view classes, whereas statically this dependency is not noticeable. This only needs to be accepted for the first sub-constraint. On the other hand, it does not need to be true for the other two sub-constraints, which do not have to declare dependencies, statically and dynamically. This is translated by the following Java code, which comes to complement the previous constraint :

```
 1 // We assume here the reception as an argument of an array
 2 // constituted of different objects which compose the application
 3 public boolean invariant(Object[] objectsMyApplication) {
 4   for(Object anObject : objectsMyApplication) {
 5     Class<?> aClass = anObject.getClass();
 6     Field[] attributes = aClass.getDeclaredFields();
 7     for(Field anAttribute : attributes) {
 8       // Verification of the previous listing ...
 9       boolean accessAttrModify = false;
10       if(! anAttribute.getType().isPrimitive()) {
11         try {
12          if(! anAttribute.isAccessible()) {
13            anAttribute.setAccessible(true);
14            accessAttrModify = true;
15          }
16          Class<?> cl = anAttribute.get(anObject).getClass();
17          if(cl.isAnnotationPresent(Controller.class)) return false;
18         }
19         catch(IllegalAccessException e) {
20           e.printStackTrace();
21         }
22         finally {
23           if(accessAttrModify) anAttribute.setAccessible(false);
24         }
25       }
26     }
27   }
28   return true;
29 }
```

This time in this constraint, we rely on the objects composing the application and not on Class objects representing the application classes. Here, we go further in the execution to obtain the objects making up the application and to seek the values of their *slots* (defined by the attributes declared in the classes of these objects), because we are assuming that the application has been loaded as well as launched. Obtaining these object slot values occurs in line 16. This is preceded by several checks to ensure that the object class attribute is not of a primitive type and is accessible (it has a public accessibility). It is the slot value type in question which is checked, to ensure that it does not relate to an annotated Controller class. The last sub-constraint formalizing the MVC pattern can be refined in the same way. This is not shown in the previous listing for reasons of simplicity and brevity.

The specification of this constraint at the design level is also possible, assuming that we have a model representing the instances making up the application. This constraint expressed in ACL will relate to the UML meta-model about instances. An excerpt from this is given in Figure 2.6. In this meta-model, an instance specification has a Classifier which defines it. It includes a number of slots, which have a StructuralFeature (e.g. a Property) that defines them. They have values designated by ValueSpecification. These can be of different types : InstanceValue (a reference to an instance), Literal-Specification (an integer, real number, etc.), etc. The meta-class which interests us is InstanceValue (shown in the meta-model in Figure 2.6). This is linked to Instance-Specification to designate the instance specification referenced in the slot.

The ACL constraint applied on this UML meta-model could be defined as follows :

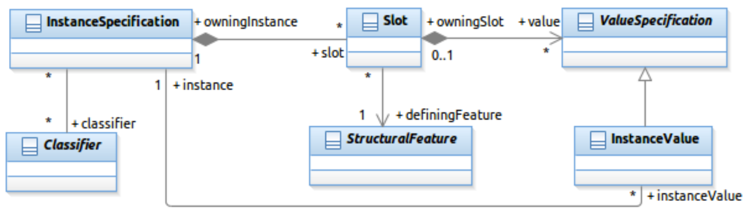

FIGURE 2.6 Excerpt from the UML meta-model (Instances)

```
 1 context InstanceSpecification inv:
 2 -- We assume here that we have already obtained
 3 -- the model, view and controller sets of classes
 4 -- as this had done previously
 5 if model->includes(self.classifier)
 6 then
 7   self.slot->forAll(s : Slot |
 8     -- Same verifications as previously
 9     -- using s.definingFeature to access
10     -- the attribute (StructuralFeature) which defines the slot
11     if s.value.isOclKindOf(InstanceValue)
12     then
13       controller->excludes(s.value.oclAsType(InstanceValue)
14         .instance.classifier)
15     else true
16     endif
17   )
18 else true
19 endif
```

In this constraint, `InstanceSpecification` is the constraint context. We therefore assume that the constraint must be verified on all instance specifications making up the application. In the constraint, we specified the fact that if the classifier of the instance specification is stereotyped `Model`, then we test if one of its slots contains a reference to an instance (line 11), i.e. the slot does not contain a primitive type value. In this case, we access the `Classifier` of the value stored in the slot. This does not have to be stereotyped `Controller`.

Here also, for reasons of simplicity and brevity, the last sub-constraint formalizing the MVC pattern is not refined.

The same constraint on the Java meta-model can be defined as follows :

```
 1 context Object inv:
 2 if self.class.annotation->exists(name='Model')
 3 then
 4   self.class.field->forAll(f:Field |
 5     -- Same verifications as previously
 6     -- using f.type to access
 7     -- the attribute type
 8     if f.value <> null
 9     then f.value.object.class.annotation = 'Controller'
10     else true
11     endif
12   )
13 else true
14 endif
```

In this constraint, the context is an object (`Object` instance) making up the application. We next navigate to its class, then to its attribute values (see lines 8 and 9). This ACL expression does not formalize the last MVC pattern sub-constraint.

We have in this section shown how architecture constraints can be written simply in Java during the implementation phase, without any extension of the language (apart from annotations, which must be defined, if necessary, for the formalized architecture patterns or styles. They can be simply defined as standard Java code, as shown in the example). We have not, however, explained how, and at what moment, these constraints are evaluated. Different solutions are possible, e.g. the automatic injection of verification code for these constraints at the end of the concerned class constructors. This is a work to which we will be turning our attention in the future. Moreover, we are in the process of developing a method, and a tool, for Java code generation (as in the listings above), from the architecture constraints expressed in ACL on the UML meta-model (like those given previously).

2.4 Architecture constraints on component-based applications

After having explored how architecture constraints can be specified on object-oriented applications, in this section, we are going to show how to express these constraints in component-based applications. Components are considered here as an evolution of the concept of object (or class - we will go into more detail later) to bring greater modularity to the architecture of these applications. Indeed, in component-based development, it is recommended that the provided, as well as the required functionalities of an application, should be explicitly declared. This allows an application's components to be uncoupled and gives their connection more flexibility. To construct a new application, instances of components will have to be created and their required functionalities satisfied, by connecting them to other instances which provide these functionalities.

We will proceed in two steps, as in the previous section. We will firstly present architecture constraint specification in the design phase, then show how this can be done in the implementation phase.

2.4.1 Architecture constraints in the design phase

We have chosen the UML standard as our modeling language in the design phase. We will explain how architecture constraints can be specified on component-based applications modeled with this language, widely-used in academia, but also in industry.

Figure 2.7 shows an excerpt from the UML meta-model, specifying the modeling elements around software components. The UML component model shown encompasses the specification of both components and composite structures in UML [40]. The `Component` meta-class is a specialization of `Class`. That gives it all the abilities of a class (participating in an inheritance relation, for example). Moreover, the `Class` meta-class henceforth (in such kinds of models) inherits from `EncapsulatedClassifier` and `Structured-Classifier`. This is to say that a component (specialization of `EncapsulatedClass-ifier`) may have `ports`, which can declare provided interfaces and required interfaces (see on the bottom-left in Figure 2.7). This permits components to encapsulate (and thus hide from their environment) the elements constituting them. They show their functionalities *via* these ports, which are communication points with the environment (other components). A component (specialization of `StructuredClassifier`) can declare `parts`, which are properties that reference instances constituting its internal structure (we would then say that the component has a composite structure). These instances can play `roles` in connections (being attached to connector ends). A component can have connectors which link "connectable" elements (`ConnectableElement` instances). That may be the port of a “part” in a component composite structure (association between `ConnectorEnd` and `Property` in the meta-model in Figure 2.7), or the encompassing component port[9] (with the composite structure). Lastly, a component can have one or several `Classifiers` (classes, for example), which “realize” it. By *Realization* of a component, the UML specification designates the set of classifiers that implement this component’s behavior. These may be other components (`Component` is a specialization of `Classifier`). This makes the UML component model hierarchical.

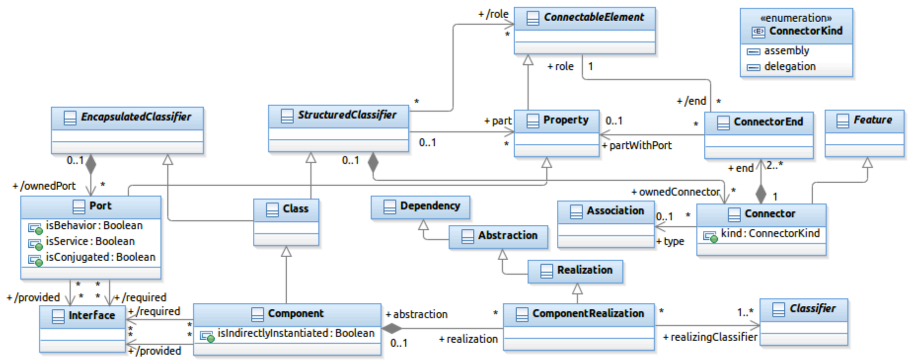

FIGURE 2.7 Excerpt from the UML meta-model (components and composite structures)

We introduce now informally (textually) the constraint that we would like to specify in this section. It relates to a constraint which formalizes the structural conditions imposed by the `Pipe & Filter` architecture style [44]. In this style, the following conditions must be formalized :

– there is only one component which defines one or more input ports (declaring provided interfaces only) connected to the encompassing component, or these ports are not connected at all (this is the left-most component in Figure 2.8). This same component must have at least one connected output port (declaring required interfaces only). The output ports of this component must all be connected to other components of the same hierarchical level as this component, or else not connected at all ;
– there is only one component which defines one or more output ports connected to the encompassing component or not connected at all (this is the right-hand component in figure 2.8). This component must have at least one connected input port. The input ports of this component must all be connected to other components of the same hierarchical level as this component, or else not connected at all ;
– the other components define input and output ports, of which at least one input and one output port are connected ;
– the connectors between each pair of components must go in the same direction. This means that there is no connector linking the input port of the first component to the output port of the second one, and another connector which links the output port of the first component to the input port of the second one ;
– connectors between all the components must go in the same direction. This means to say that for each pair of connected components, there is no third component, con-

9. These characteristics do not relate exclusively to components. Classes also can have ports and composite structures.

nected by its input ports to the first component and, by its output ports to the second component.

These constraints are illustrated in Figure 2.8. The components in this style are named filters and connectors are considered as pipes. In this chapter, we simplify the application architecture by considering the fact that it has been designed exclusively according to this style. Real-world applications are often constructed by combining various styles. In this case, the constraint will have a different formalization, which could for example rely on `Pipe` and `Filter` stereotypes applied on the components. The constraint must check that the conditions enumerated above are applied only to the stereotyped components, which participate thus in the implementation of the architecture style. Due to lack of space, we are unfortunately unable to develop this case here.

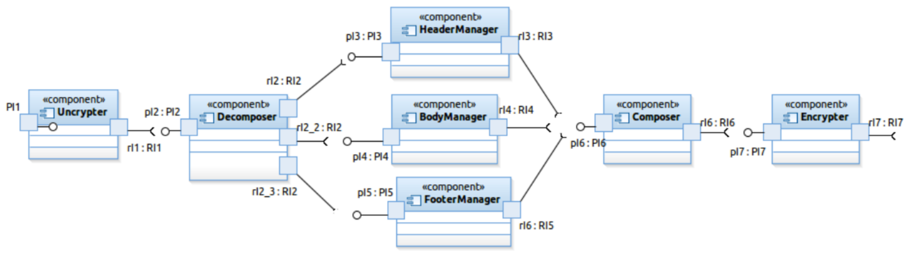

FIGURE **2.8** Pipe and Filter architecture style illustration

We will be able to write the different sub-constraints using ACL by relying on the meta-model in Figure 2.7. The first sub-constraint can be specified as follows :

```
-- We assume here that the components forming the application
-- are encapsulated in a component with a composite structure.
-- This is now shown in the figure illustrating the
-- Pipe and Filter style
context MyApplication : Component inv:
let internalComps : Set(Component) =
self.realization.realizingClassifier->select(c:Classifier |
  c.oclIsKindOf(Component))->oclAsType(Component)
in
internalComps->one(c : Component |
   self.ownedConnector->exists(con:Connector |
     con.end.role->exists(r1,r2 | r1.oclIsKindOf(Port)
        and (r1.oclAsType(Port) = c.ownedPort)
        and r1.oclAsType(Port).provided->notEmpty()
        and r1.oclAsType(Port).required->isEmpty()
        and r2.oclIsKindOf(Port)
        and self.ownedPort->includes(r2.oclAsType(Port))
     )
   )
   and
   self.ownedConnector->exists(con:Connector |
     con.end.role->exists(r1,r2 | r1.oclIsKindOf(Port)
        and (r1.oclAsType(Port) = c.ownedPort)
        and r1.oclAsType(Port).required->notEmpty()
        and r2.oclIsKindOf(Port)
        and self.ownedPort->excludes(r2.oclAsType(Port))
     )
   )
)
```

In this constraint, we firstly create a set constituted of components (`internalComps`) forming the internal structure of the component representing the application. Next, we check that there is only one component in this set, which has at least one connector linking it with the encompassing component, through a port with only provided (and not required) interfaces. It should be noted here that the verification of the presence of connectors between components undergoes the analysis of the application connector set (`self.ownedConnector`). Indeed, we cannot obtain the connectors attached to the port of a component by going from this component. This corresponds clearly to the uncoupling between components recommended in component-based development : a component does not recognize other components which it is connected to ; all it does is invoke operations on its required port ; it does not depend on a particular component. In the second part of the constraint (from line 20 onwards), we check that this single component is connected to the other application components through its output ports (declaring required interfaces).

The second sub-constraint previously enumerated can be formalized the same way, by inverting the required and provided interfaces.

The third sub-constraint is specified in ACL in the following listing :

```
...
and
internalComps->select(c |
  not self.ownedConnector->exists(con:Connector |
    con.end.role->exists(r1,r2 | r1.oclIsKindOf(Port)
      and (r1.oclAsType(Port) = c.ownedPort)
      and r2.oclIsKindOf(Port)
      and self.ownedPort->excludes(r2.oclAsType(Port))
    )
  )
  and c.ownedPort->select(provided->notEmpty()
    and required->notEmpty()))
->size() = internalComps->size() - 2
```

In this constraint, we count the number of components which participate in connections, but are not connected to the encompassing component. This number must be equal to the total number of application components minus the two components in the extremities.

Now, we see how the fourth sub-constraint can be defined in ACL :

```
...
and
internalComps->forAll(c1,c2 : Component |
  self.ownedConnector->select(con:Connector |
    con.end.role->oclIsKindOf(Port) and
    con.end.role->oclAsType(Port)->one(c1.ownedPort) and
    con.end.role->oclAsType(Port)->one(c2.ownedPort))
 ->forAll(con : Connector | con.end.role->select(oclIsKindOf(Port))
   ->oclAsType(Port)->exists(p1,p2:Port |
     (c1.provided->includesAll(p1.provided)
       and c2.required->includesAll(p2.required))
     xor (c2.provided->includesAll(p1.provided)
       and c1.required->includesAll(p2.required))))
)
```

Here, we check that in the connector set of the application, if a connector links two components c1 and c2, all other connectors between c1 and c2 must always link input ports (declaring required interfaces) from c1 to output ports (declaring provided interfaces) from c2, or inversely. There should not be connectors oriented from c1 to c2 and from c2 to c1.

The last, and undoubtedly most complex, sub-constraint, is defined below :

```
...
and
internalComps->forAll(c1,c2:Component |
  let conns : Set(Connector) =
  self.ownedConnector->select(con:Connector |
    con.end.role->oclIsKindOf(Port) and
    con.end.role->oclAsType(Port)->one(c1.ownedPort) and
    con.end.role->oclAsType(Port)->one(c2.ownedPort))
  in -- conns = {connectors between c1 and c2}
  conns->notEmpty()
  and
  internalComps->excludes(c3 : Component |
    c3 <> c1 and c3 <> c2 and
    if conns.end.role->oclAsType(Port)
      ->exists(p | c1.ownedPort.required->includesAll(p.required))
    then --Connector(s) c1 to c2
      not (
        self.ownedConnector.end.role->select(oclIsKindOf(Port))
        ->oclAsType(Port)->exists(p1,p2 |
          c3.ownedPort.required->includesAll(p1.required) and
          c2.ownedPort.provided->includesAll(p2.provided))
       or
        self.ownedConnector.end.role->select(oclIsKindOf(Port))
        ->oclAsType(Port)->exists(p1,p2 |
          c1.ownedPort.required->includesAll(p1.required) and
          c3.ownedPort.provided->includesAll(p2.provided))
      )
    else -- Connector(s) c2 to c1
      not (
        self.ownedConnector.end.role->select(oclIsKindOf(Port))
        ->oclAsType(Port)->exists(p1,p2 |
          c2.ownedPort.required->includesAll(p1.required) and
          c3.ownedPort.provided->includesAll(p2.provided))
       or
        self.ownedConnector.end.role->select(oclIsKindOf(Port))
        ->oclAsType(Port)->exists(p1,p2 |
          c3.ownedPort.required->includesAll(p1.required) and
          c1.ownedPort.provided->includesAll(p2.provided))
      )
    endif
  )
)
```

In this last part of the constraint, we check that for every pair of connected components (c1, c2) in the set of internal components, there is no third (c3) component connected *via* its input ports to c1, and *via* its output ports, to c2, if the connectors go from c1 to c2, and inversely if the connectors go from c2 to c1. This guarantees that all connectors in the application are oriented in the same direction.

We have simplified the use of the UML component model in this formalization. We are assuming that the developer has not defined a port declaring at the same time required and provided interfaces, which are attached to several connectors (input and output).

2.4.2 Architecture constraints in the implementation phase

In the literature, there are several component-based programming languages (such as ArchJava [1], ComponentJ [43] or SCL [16]), or *frameworks* (like Spring, OSGi or Fractal/Julia [8]). To write constraints we would need a language or *framework* offering the ability to realize introspection. Here too, we have several languages at our disposal. We have chosen a language developed in the thesis of Petr Spacek [45]. This language is called Compo. It was developed as part of a larger context than architecture constraint specification. The choice of this language was made for the following reasons :

1. it explicitly provides support for architecture constraint specification ;
2. it is reflexive, at almost all levels (everything is reified, except the service implementations) ;
3. the reification of entities of this language is realized using components, and not using objects, unlike in other component-based programming languages.

Thanks to this last point, we propose a homogeneous environment for developers, where they only define component descriptors ; they will never have to choose between objects or components to define this or that business domain entity in their applications.

An excerpt from the Compo meta-model is given in Figure 2.9. In this meta-model, there is a clear distinction between a component descriptor and a component instance (simply called : `Component`). In UML, `Component` inherits from `Class`, making it a component descriptor. The fact of inheriting from the `Class` meta-class gives it the ability to be instantiated (see the meta-association between `Classifier` and `InstanceSpecification` in the meta-model in Figure 2.6). In UML, a component instance has nothing particular comparatively with a class instance.

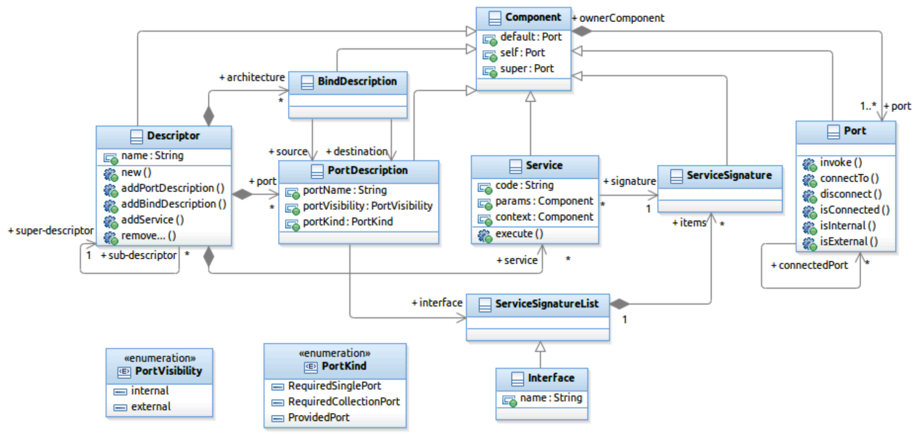

FIGURE **2.9** Excerpt from the Compo meta-model

In Compo, there is a generalized dichotomy between descriptors and their instances : component descriptor and component, port descriptor and port, etc. In a Compo component descriptor (meta-class `Descriptor`), the programmer can declare a number of port descriptors. The latter must show the list of service signatures (a service is the equivalent to an operation in UML), which can be grouped in a named or anonymous interface (`ServiceSignatureList`). These services have a signature. A connector (`BindDescription)`) links a source port descriptor to a destination port descriptor. A port realizes a port descriptor, and may be of kind Collection or Single. A port can, moreover, be provided or required. It can be internal or external. For example, an internal required port serves to connect component instances to their encompassing component. Lastly, we have integrated the inheritance in this language [46]. A component descriptor can inherit from another descriptor (its *super-descriptor* [10]).

Constraints imposed by the *Pipe and Filter* style can be programmed in Compo as follows :

```
Descriptor PipeAndFilter extends Constraint
{
    internally requires {
        scOne <: SubConstraintOne;
        scTwo <: SubConstraintTwo;
        scThree <: SubConstraintThree;
        scFour <: SubConstraintFour;
        scFive <: SubConstraintFive;
    }
    architecture {
        delegate contextscOne to context self;
```

10. This inheritance mechanism between component descriptors extends the classic inheritance between classes, especially by offering the option to extend the required ports [46]

```
12         delegate contextscTwo to context self ;
13         delegate contextscThree oneDelegReq to context self ;
14         delegate contextscFour to context self ;
15         delegate contextscFive to context self ;
16     }
17     service verify() {
18         |c1 c2 c3 c4 c5 |
19         c1 := scOne.verify();
20         c2 := scTwo.verify();
21         c3 := scThree.verify();
22         c4 := scFour.verify();
23         c5 := scFive.verify();
24
25         return (((c1.and([c2])).and([c3])).and([c4])).and([c5]);
26     }
27 }
```

In this listing, we rely on an architecture constraint specification model in the form of components initially introduced in [53]. Different sub-constraints imposed by the *Pipe and Filter* style are then defined by several component descriptors. In this listing, we declare the component descriptor, called `PipeAndFilter`, which contains the set of component instances verifying the five sub-constraints (declared in lines 4 to 8). These components are connected to their encompassing component (lines 11 to 15). Next, we start checks by invoking the `verify()` service on each port of the different internal components in the `PipeAndFilter` component `verify` service. The final result must correspond to the conjunction of different (Boolean) results returned by the internal component services (line 25).

For simplicity's sake, we give the listings of two component descriptors here. They involve the descriptors formalizing the first and fourth sub-constraints :

```
Descriptor SubConstraintOne extends Constraint
{
    service verify() {
        |retval|
        retval := true;
        intComps := context.getPorts().select([:p |
            &p.isRequired().and([&p.isInternal()]);
        ]);
        intComps.each([:ic |
            ic.getPorts().each([:x |
                if(&x.isProvided().and([&p.isExternal()])) {
                    | count |
                    &x.getConnectedPorts().each([:cp |
                        if(&cp.isProvided().and([&p.isExternal()])) {
                            if(&cp.getOwner() == context.yourself())
                            { retVal := retVal.and([true]); }
                            else
                            { retVal := retVal.and([false]); }
                        }
                    ]);

                }
                if(&x.isRequired().and([&p.isExternal()])) {
                    &x.getConnectedPorts().each([:cp |
                        if(&cp.isProvided().and([&p.isExternal()])) {
                          if(&cp.getOwner().getOwner() == context.yourself())
                            { retVal := retVal.and([true]); }
                            else
                            { retVal := retVal.and([false]); }
                        }
                    ]);
                }
            ]);
        ]);
        return retVal;
    }
}
```

In this listing, we have a component descriptor implementing the `verify()` service, which tests the first *Pipe and Filter* style sub-constraint. This service firstly identifies references to the internal components (rather than their ports), by relying on the internal required ports of the composite. Access to the meta-level is gained using the "&" operator (see line 7, for example). This makes it possible to know if a port is required or provided, or if it is internal or external. This also allows access to the connectors linking one port to the port of another component (line 13). The rest of the constraint corresponds to the ACL constraint defined in the first listing in this section, where we verify the presence of a single component connected to its encompassing component (providing the `context` port in the listing above) *via* its provided port(s) ; the other required ports of this internal component must be connected to the other internal components.

```
 1 Descriptor SubConstraintFour extends Constraint
 2 {
 3     service verify() {
 4         conns := context.getDescriptor().getDescribedConnections();
 5         conns.each([:conn |
 6             |dest source |
 7             source := conn.getSourcePortComponent();
 8             dest := conn.getDestinationPortComponent();
 9             conns.each([:conn2 |
10                 if((conn2.getSourcePortComponent() == dest).and([
11                     conn2.getDestinationPortComponent() == source]))
12                 { return false; }
13             ]);
14         ]);
15         return true;
16     }
17 }
```

The fourth sub-constraint programmed in Compo above checks the absence of the following scenario : if we have `conn` and `conn2` connectors between components, if `conn` has as its destination a component identical to the component representing the `conn2` source (see line 10), then `conn` has as its source the destination of `conn2` (line 11). This reflects the presence of connectors between a pair of components going in different directions, which is forbidden by the *Pipe and Filter* style.

We note here that architecture constraints expressed in Compo as components aim at improving reuse. These different "constraint-components" can be assembled to form a single constraint-component formalizing the *Pipe & Filter* style. Some amongst these constraint-components can be reused in the formalization of other styles, such as *Pipeline* or layered styles. These enable architects to avoid having to re-write parts of sub-constraints.

2.5 Architecture constraints on service-oriented applications

A service is a grouping of operations into one single "black box" entity, implemented in any language and deployed on any platform. By default, this entity does not maintain a state between the invocations of its different operations, coming from a given client program. When a service is accessible *via* the HTTP protocol (with SOAP or not), we have what is called a *web* service.

There is a profile with a number of standard stereotypes in the UML specification. Among these stereotypes, we find an extension of the `Component` meta-class, called `service`, which designates "functional stateless components" [40]. This corresponds to the notion of service discussed in this section.

Among architecture constraints expressible on service-oriented applications, we find the SOA patterns [15]. These patterns, such as *Service Façade*, impose restrictions at the structural level of an application. These restrictions must be formalized, in order to guarantee the maintenance of quality attributes associated with the pattern during development.

In this section, we will show how we express architecture constraints on service-oriented applications, designed and implemented using only one language, called BPEL (standardized by Oasis under the name WSBPEL : *Web Services Business Process Execution Language* [38]). Our choice of BPEL is driven by the fact that it is a standardized language, widely used by information systems developers to model and execute their business processes based on *web* services. Thus, in this chapter, a service-oriented application is assimilated into a BPEL process with several *web* service partners.

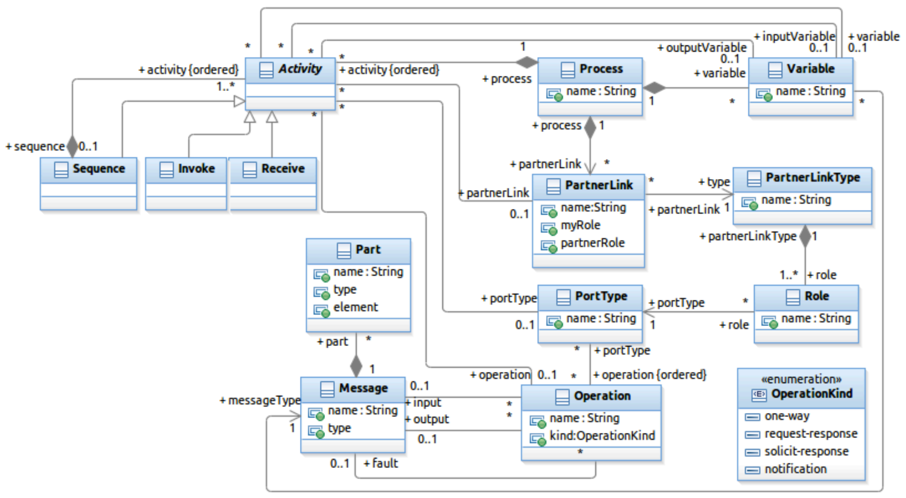

FIGURE 2.10 Excerpt from the BPEL and WSDL (Process, Activity, PartnerLink and PortType) meta-models

Figure 2.10 shows an excerpt from the meta-models of BPEL (upper part of the figure) and WSDL [11] (lower part). A process is composed of various ordered activities (`ordered` qualifier on the role `activity` in the association between `Process` and `Activity` in the figure). These activities can be of different kinds. We have listed only three of them in the meta-model (for the purposes of the constraints defined above) : `Invoke` to invoke the partner *web* service operations (providers), `Receive` to receive their responses or else to receive the requests of the partner *web* services (clients), and `Sequence` which is a composite activity (see composition with `Activity` in the meta-model). The latter serves to implement sequences of activities. A process can define a number of variables, which can contain request or response messages from *web* services. It declares links towards partner *web* services, (`PartnerLink`) which indicate the client or provider *web* services used by the process. Each *PartnerLink* designates roles for the process (`myRole`) and for the partner service (`partner role`). A role indicates a `PortType` (`Interface`, since WSDL 2.0) which represents the WSDL structure in which are described the provided or required operations, as well as the exchanged messages.

11. WSDL (*Web Service Description Language*) is the W3C standard for Web service interface description : www.w3.org/TR/wsdl20/.

There is a plethora of architecture patterns for service-oriented applications in the literature and in practice, known as SOA patterns or *Service-Oriented Architecture patterns* [15]). Of these patterns, we have chosen the *Service Façade* pattern [15]. This pattern recommends defining a single service for publishing/providing operations, realized by a BPEL process. The main objective of this pattern is to set aside the (sometimes complex) details of the services presented by a number of suppliers, by proposing a single service. A "toy" example of the BPEL process designed with this pattern is given in Figure 2.11.

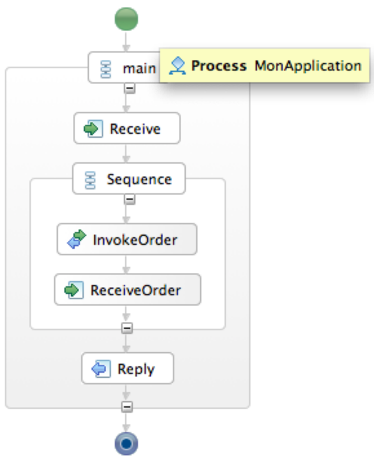

FIGURE **2.11** Basic service façade pattern illustration

In this figure, the Serviceservice facade is realized using the *partner link* attached to the first activity in the process (the `Receive` activity) and its corresponding `Reply` activity, which is found at the end of the process. In the middle of the process, we find a number of `Invoke` activities [12] to invoke service operations provided by third parties.

The architecture constraint imposed by this pattern consists of several sub-constraints :

– the process has as its first activity a `Receive` activity, linked to a *partner link* representing the client (of the process) ;

– the process has as its last activity a `Reply` activity, with the same `partner link`, the same `port type` and the same operation as the *Receive* activity of the previous sub-constraint ;

– the only `Receive` activities accepted in the middle of the process must be preceded by the corresponding `Invoke` activities. These are used to invoke partner service operations provided by third parties. The possible `Receive` activities here will then serve to receive messages returned by the invoked service operations.

Due to lack of space, we have deliberately simplified the constraint on the BPEL process. Indeed, if we want to be complete in the constraint specification imposed by the service façade Patternpattern, we would have to add other types of activities, which could interact with a process client, such as `Pick` and `OnEvent` activities. Here, we have only dealt with activities of type `Receive`.

This constraint is formalized in ACL on the meta-model of Figure 2.10 in the listings below. The first sub-constraint is specified as follows :

12. For simplicity's sake, a single `Invoke` is shown in Figure 2.11.

```
 1 context MonApplication : Process inv :
 2 let fst : Activity =
 3 if self.activity->first().activity = null -- is not a composite activity
 4 then self.activity->first()
 5 else if self.activity->first().oclIsTypeOf(Sequence)
 6   then self.activity->first().oclAsType(Sequence).activity->first()
 7   else null
 8   endif
 9 endif
10 in
11 if fst <> null
12 then fst.oclIsTypeOf(Receive)
13 else false
14 endif
```

In this first sub-constraint, we firstly identify what is the first activity declared in the process [13]. The different (`if`) tests enable cases where the first activity is a composite (`Sequence`s for example), rather than a simple activity, to be handled. In this case, the first activity within this sequence will have to be identified. The check undertaken in line 12 allows to ensure that this is a `Receive` activity.

Here also, we simplify the constraint specification by considering only `Sequence`s as possible composite activities within the process. In practice, other composite activities are possible. In this case, it suffices simply to add tests in the constraint, in order to take them into account (in the `else` part of the statement at line 7). In this constraint, we consider only one level of depth in composite services. If we want to take several levels of depth into account (for the first activity in the process, which is rare), a recursive navigation should be performed (see below for an example).

With the second sub-constraint, as with the first, it is sufficient to replace `first()` with `last()` to access the last activity and check that the activity type corresponds to `Reply`. What follows must be added, to check that the properties of this activity and the first activity (`Receive`) have identical names :

```
...
if lst <> null -- We assume that lst contains the last activity
then lst.oclIsTypeOf(Reply)
  and lst.oclAsType(Reply).partnerLink.name
    = fst.oclAsType(Receive).partnerLink.name
  and lst.oclAsType(Reply).portType.name
    = fst.oclAsType(Receive).portType.name
  and lst.oclAsType(Reply).operation.name
    = fst.oclAsType(Receive).operation.name
else false
endif
```

The third sub-constraint can be written in ACL as follows :

```
...
and
let activities : OrderedSet(Activity) =
self.activity->excluding(fst)->excluding(lst)->closure(activity)
in
activities->forAll(a : Activity |
if a.oclIsTypeOf(Receive)
then
  activities->exists(aa : Activity |
    activities->indexOf(aa) < activities->indexOf(a)
    and aa.oclIsTypeOf(Invoke)
    and aa.oclAsType(Invoke).partnerLink.name
```

13. Given that the activities in the process are ordered (see the meta-model), navigation to `Activity` from `Process` returns an `OrderedSet`, which allows operations like `first()` to be used to access the first activity in the process.

```
13        = a.oclAsType(Receive).partnerLink.name
14      -- and ... (same thing for the portType and operation)
15    )
16  else true
17  endif)
```

A BPEL process can be constituted of composite activities, which in their turn contain other composite activities, and so on, up to a certain depth. In the sub-constraint defined above, we thus recursively identify all activities in the process, by excluding the first (`fst`) and last (`lst`) activities. This recursivity is undertaken thanks to the OCL `closure(...)` operation in line 4. Next, we ensure that if there is a `Receive` activity among these activities, it is preceded by an `Invoke` activity corresponding to it, i.e. with the same `partner link`, `port type` and `operation` (lines 10 to 14).

We have shown in this section that simply combining OCL and a meta-model of BPEL and WSDL languages has allowed us to formalize architecture constraints imposed by an SOA pattern. We have done the same exercise on other patterns (results are in the process of being published), and we suspect that the difficulty resides in identifying the correct sub-constraints, which represent a given pattern (before their formalization). It relates to a classic problem in formal specifications. It is necessary to check that the set of these sub-constraints is complete and that each of these is rigorously defined. This guarantees that we know, as a result of their simple verification, if a service-oriented application conforms to an SOA pattern.

2.6 Conclusion

A software architecture defines the "coarse-grained" organization of a software system. These software systems have been growing in complexity which has driven, for many decades, the proposal of a multitude of works about software architecture documentation. This documentation has involved not only architects' product (the "what" : architecture description), but also the decisions taken during the development of this product and the reasons for these decisions (the "how" and the "why"). We focus on this second aspect in this chapter, where we argue that architecture description should be accompanied by a "formal" specification (formal, in the sense of being able to be automatically interpreted and verified), which describes the constraints imposed by architectural decisions. We have illustrated our remarks with a number of constraints formalizing some architecture styles and patterns (as examples of architectural decisions). We have shown how these constraints can be expressed using simple languages, which are even sometimes used to describe the architecture itself (in the case of Java and Compo). The goal of these architecture constraints is twofold. On the one hand, as supplementary documentation, they help in our understanding of the architecture of an application. On the other hand, as specifications able to be automatically verified, they enable reliable evolutions. This means we can know, among other things, if the new architecture (after evolution) consistently adheres to previously-taken decisions (work that we have conducted on component-based applications [50]).

Moreover, we have shown that the specification of these constraints is not limited to the architectures of component-based applications (work undertaken some years ago [52]). They have a use both in object- and service-oriented applications. Furthermore, we have shown their use, not only in the design phase, but also in the implementation phase (accompanying code).

We have developed several interpreters for the different languages presented in this chapter. All these interpreters perform static analysis of architecture descriptions (written in most cases in XML dialects). Moreover, we have previously been interested in the conversion of constraints written in ACL from one meta-model to another for component-based applications [51]. We have shown, for example, how we can automatically convert architecture constraints written with ACL on applications modeled with UML components into constraints written with ACL on applications implemented with Fractal.

We can formulate the following ideas as future perspectives for this concept of architecture constraints :

– constraint specification independently of a given paradigm, by using a meta-model of graphs (an architecture description being considered as a graph with nodes and edges), then their conversion to different paradigms ;
– proposal of a high-level language for the re-use of constraints by specialization (e.g. *Pipeline* style constraints are a specialization of *Pipe and Filter* style constraints) ;
– code generation from architecture constraints ;
– application of these constraints in an integrated development environment.

REFERENCES

1. Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava : Connecting Software Architecture to Implementation. In *Proceedings of the 22rd International Conference on Software Engineering (ICSE'02)*, pages 187–197. ACM, 2002.
2. Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 3rd Edition*. Addison-Wesley, 2012.
4. Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE'05)*, pages 224–232. ACM, 2005.
5. Boris Bokowsky. Coffeestrainer : Statically-checked constraints on the definition and use of types in java. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 355–374, Toulouse, France, 1999. Springer-Verlag.
6. Jan Bosch. Software Architecture : The Next Step. In *Proceedings of the 1st European Workshop on Software Architecture (EWSA'04)*, volume 3047 of *Lecture Notes in Computer Science*, pages 194–199. Springer, 2004.
7. Lionel C. Briand, Yvan Labiche, Massimiliano Di Penta, and Han (Daphne) Yan-Bondoc. An experimental investigation of formality in uml-based development. *IEEE Transactions on Software Engineering*, 31(10) :833–849, October 2005.
8. Eric Bruneton, Coupaye Thierry, Matthieu Leclercq, Vivien Quéma, and Stefani Jean-Bernard. An open component model and its support in java. In *Proceedings of the ACM SIGSOFT International Symposium on Component-based Software Engineering (CBSE'04). Held in conjunction with ICSE'04*, Edinburgh, Scotland, may 2004.

9. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1) :146–166, march 1989.

10. Anir Chowdhury and Scott Meyers. Facilitating software maintenance by automated detection of constraint violations. In *In Proceedings of the International Conference on Software Maintenance (ICSM'93)*, pages 262–271. IEEE, 1993.

11. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures, Views and Beyond, Second Edition*. Addison-Wesley, 2010.

12. Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures, Methods and Case Studies*. Addison-Wesley, 2002.

13. Remco C. de Boer and Rik Farenhorst. In search of 'architectural knowledge'. In *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, SHARK'08, pages 71–78. ACM, 2008.

14. Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, pages 391–400. ACM, 2008.

15. Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2009.

16. Luc Fabresse, Christophe Dony, and Marianne Huchard. Foundations of a Simple and Unified Component-Oriented Language. *Journal of Computer Languages, Systems & Structures*, 34/2-3 :130–149, 2008.

17. Davide Falessi, Giovanni Cantone, Rick Kazman, and Philippe Kruchten. Decision-making techniques for software architecture design : A comparative survey. *ACM Computing Surveys (CSUR)*, 43(4) :33 :1–33 :28, October 2011.

18. Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL : An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

19. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Sofware*. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc., 1995.

20. David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, 1994.

21. David Garlan, Robert T. Monroe, and David Wile. Acme : Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

22. Olivier Gilles and Jérôme Hugues. Expressing and enforcing user-defined constraints of aadl models. In *In Proceedings of the 5th UML and AADL Workshop (UML and AADL 2010)*, 2010.

23. N. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *Software, IEEE*, 24(4) :38–45, July-Aug. 2007.

24. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, August 1978.

25. Daqing Hou and H.J. Hoover. Using scl to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6) :404–423, 2006.

26. Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, 2005.

27. Nils Klarlund, Jari Koistinen, and Michael I. Schwartzbach. Formal design constraints. In *Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 370–383, San Jose, California, USA, 1996. ACM Press.

28. Philippe Kruchten, Rafael Capilla, and Juan Carlos Duenas. The decision view's role in software architecture practice. *IEEE Software*, 26(2) :36–42, 2009.

29. David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, 1995.

30. N Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural desing in the c2 style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'96)*, pages 24–32, San Francisco, California, USA, October 1996.

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

32. Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions On Software Engineering and Methodology*, 11(1) :2–57, 2002.

33. Naftaly H. Minsky. Law-governed regularities in object systems. part i : An abstract model. *Theory and Practice of Object Systems*, 2(4) :283–301, 1996.

34. Naftaly H. Minsky and Partha Pratim Pal. Law-governed regularities in object systems. part ii : a concrete implementation. *Theory and Practice of Object Systems*, 3(2) :87–101, 1997.

35. Robert T. Monroe. Capturing software architecture design expertise with armani. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2001.

36. Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4) :356–372, April 1995.

37. Mark Moriconi and R. A. Riemenschneider. Introduction to sadl 1.0 : A language for specifying software architecture hierarchies. Technical report, Computer Science Laboratory, SRI International, 1997.

38. OASIS. Web services business process execution language version 2.0. Oasis Website : http ://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, 2007.

39. OMG. Uml profile for corba and corba components (cccmp), version 1.0 specification, document formal/08-04-07. Object Management Group Web Site : http ://www.omg.org/spec/CCCMP/1.0/PDF, 2008.

40. OMG. Unified modeling language superstructure, version 2.4.1 specification, document formal/2011-08-06. Object Management Group Web Site : http ://www.omg.org/spec/UML/2.4.1/Superstructure/PDF, 2011.

41. OMG. Object constraint language specification, version 2.3.1, document formal/2012-01-01. Object Management Group Web Site : http ://www.omg.org/spec/OCL/2.3.1/PDF, 2012.

42. Trygve Reenskaug. Thing-model-view-editor an example from a planning system. Technical report, Xerox Parc, USA, May 1979.

43. J. C. Seco, Ricardo Silva, and Margarida Piriquito. Componentj : A component-based programming language with dynamic reconfiguration. *Computer Science and Information Systems*, 05(02) :65–86, 12 2008.

44. Mary Shaw and David Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

45. Petr Spacek. *Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language*. PhD thesis, Montpellier II University, France, 2013.

46. Petr Spacek, Christophe Dony, Chouki Tibermacine, and Luc Fabresse. An inheritance system for structural & behavioral reuse in component-based software programming. In *In proceedings*

of the 11th International Conference on Generative Programming and Component Engineering (GPCE'12), Dresden, Germany, September 2012. ACM Press.

47. Antony Tang, Muhammad Ali Babar, Ian Gorton, and Jun Han. A survey of the use and documentation of architecture design rationale. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, Pittsburgh, Pennsylvania, USA, November 2005.
48. Antony Tang, Jun Han, and Rajesh Vasa. Software Architecture Design Reasoning : A Case for Improved Methodology Support. *IEEE Software*, 26(2) :43–49, 2009.
49. Ricardo Terra and Marco Tulio de Oliveira Valente. A dependency constraint language to manage object-oriented software architectures. *Software Practice and Experience*, 39(12) :1073–1094, 2009.
50. Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. 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)*, pages 294–309, Vasteras, Sweden, June 2006. Springer LNCS.
51. Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Simplifying transformations of architectural constraints. In *Proceedings of the ACM Symposium on Applied Computing (SAC'06), Track on Model Transformation*, pages 1240–1244, Dijon, France, April 2006. ACM Press.
52. Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. A family of languages for architecture constraint specification. *In Journal of Systems and Software (JSS), Elsevier*, 83(1) :815–831, 2010.
53. Chouki Tibermacine, Salah Sadou, Christophe Dony, and Luc Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering (CBSE'11)*, pages 31–40. ACM, 2011.
54. Jeff Tyree and Art Akerman. Architecture decisions : Demystifying architecture. *IEEE Software*, 22(2) :19–27, March/April 2005.